Aspect Structure of Compilers

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Jeeva S. Paudel

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

Compilers are among the most widely-studied pieces of software; and, modularizing these valuable artifacts is a recurring theme in research. However, modularization of cross-cutting concerns in compilers is not yet well explored. Even today, implementation of one compiler concern scatters across and tangles with the implementation of several other concerns, thereby leading to a mismatch between different compiler modules and the operations they represent. Essentially, current compiler implementations fail to explicitly identify the control dependencies of different phases, and separately characterize the actions to execute during those phases. As a result, information about their program-execution path remains non-intuitive: it stays hidden within the program structure and cuts-across several phase implementations. Consequently, this makes compiler designs and artifacts difficult to comprehend, maintain and reuse. Such limitations occur primarily as a result of the inability of mainstream object-oriented languages, such as Java, to organize the cross-cutting concerns into clean modular units.

This thesis demonstrates how such modularity-issues in compilers can be addressed with the help of a relatively new, yet powerful programming paradigm called aspect-oriented programming.

Acknowledgements

I gladly take this opportunity to express my gratitude to different people who have contributed in upbringing this thesis:

1. My family for your warm love and invaluable inspiration.

2. SRL members, Dr. Brian de Alwis, Andrew Sutherland, David Vickers, Ian Hopkins, Andriy Hnativ, Qian Zhang, David Noete, and Yiqing Liu for your lively paper discussions and for your comments and suggestions on my talks.

3. Dr. Jose Nelson Amaral, for sharing your research ideas and for your suggestions on improving my talk during visit to the University of Alberta.

4. Dr. Karl Lieberherr, for showing me interesting research avenues and Dr. Mitchell Wand for your comments and suggestions on my talk during visit to the Northeastern University.

5. My committee members, Dr. Nathaniel Osgood and Dr. Kevin Schneider, for your enthusiastic support in completing and delivering this thesis.

6. Last and most important, my supervisor, Dr. Christopher Dutchyn, for your unwavering support and guidance throughout my master's studies. I have learned a lot from your questions, examples, critical comments and insights. Thank you for teaching me how to think, and learn.

To my parents.

Contents

Tables

Figures

Listings

Chapter 1

Introduction

Construction of large and complex software is a difficult and intricate task. As software evolves, it becomes difficult to maintain its reusable and comprehensible nature as a result of interdependent functions, interconnected design architecture, and large size. This leads to a crucial question of how to divide software into manageable pieces with simple mutual relationships. The resulting system should be relatively easy to update by replacing corresponding pieces with new ones. In 1974, Dijkstra[21] proposed decomposing a large programs into distinct units called *modules* in order to manage software complexity.

A module represents a set of related concerns. Examples of concerns include design rules, business logic, interaction patterns, services, and infrastructures. The act of factoring out large software into modules, called *modularization*[1], helps manage software complexity in several ways. First, by providing separate modules for distinct concerns that overlap in functionality to a minimal extent, it promotes clean separation of concerns. Separation of concerns enables us to break the complexity of a problem into loosely-coupled, easier to solve, subproblems. Further, this separation permits distinct and specialized groups of people to reason about and develop disparate pieces of software in isolation. Second, software features implemented in different modules can then be assembled to build a larger system. Third, by decoupling fast-changing design decisions and interaction rules, modules facilitate flexible maintenance of different parts of a software. Fourth, developing modules as pluggable features allows them to be tested independently and to be added or removed from the system. Consequently, it offers flexibility to upgrade or replace only certain

[1]We adopt Kiczales and Mezini [41, p.49]'s definition of *modularization*, "the implementation of the concern is: (i) localized, (ii) has a well-defined interface, and (iii) is amenable to separate development. It is also critical that such modularization be efficient. We avoid using general terms like modular or modularized to mean modular with respect to any particular programming technology, such as current type checking technology."

parts of the system, without having to consider the entire software. Thus, proper modularization of a large program allows different teams to design, build, assemble, analyze, deploy, and maintain different parts of a system in an independent manner.

Current programming languages such as Modula-2, Java, and C# offer a variety of modularity units including functions, classes, modules, and packages. However, there are other program features that do not correspond to these typical structures. These features are *scattered* across various program units and are *tangled* with the design and implementation of the entire system. Existing programming paradigms, such as Object Orientation (OO) and Component-based Software Engineering, fail to localize and group these *cross-cutting* software concerns.

The inability of mainstream languages to modularize cross-cutting concerns impairs software development and maintenance tasks in several ways. Tangled implementation of different software features makes it difficult to reason about, build, and test without requiring detailed knowledge of other parts of the system. Further, dependence of implementation of one feature upon others, called *coupling*, inhibits the ability to further reuse or modify an implementation.

In response, over the last few years, a new programming technique, Aspect Oriented Programming (AOP)[17, 42, 43, 54], has developed. It has helped improve the ability of programmers to separate the expression of different concerns[34, 69]. In particular, it provides increased support for managing cross-cutting concerns in software. It enhances modular and expressive capabilities of existing languages by offering additional units of *abstraction* and *encapsulation*. By abstraction, we mean the ability to hide the implementation details and provide a higher level view of the underlying logic and interaction patterns. Encapsulation here refers to the grouping of related behavior and associated contexts/attributes into a single entity. Essentially, AOP provides principled ways of defining and localizing the cross-cutting concerns into dedicated modules. By doing so, it promotes both separation of concerns and reusability of existing artifacts. Further, by enforcing logical boundaries between such concerns, it promotes maintainability of software components[5, 57].

One substantial and important category of software product, language systems, such as analyzers, compilers, translators, and interpreters do not avail themselves of

these new modularity constructs. Structural decomposition of such a system into different phases has become standard, and the components can be usually developed using well-known systematic techniques. The focus of this thesis, however, is another level of modularity: using this new AOP technology to extract, into local modules, the cross-cutting concerns that are scattered across different phase implementations.

1.1 Motivation

A compiler is a large, complex, and significant software system that must adhere to highest standards of quality. Because of this position, modularity assumes prime importance in compiler design and implementation. Demand for modular compilers has driven the development of software engineering and software modularity. Traditionally, the division of compilers into stages of lexical, syntactic, and semantic analyses followed by a generation phase has been a widely-used approach whose benefits are well known. However, modern techniques for improving modularity, such as aspects, have been largely unexplored in the domain of compilers. Further, no production compiler incorporating aspects has been built to-date. Therefore, execution paths related to different phases of a compiler, referred to as *control-flows*, still remain hidden deep within the compiler structure, making it difficult not only to understand but also to maintain and extend them.

Our task is to identify and modularize cross-cutting concerns inherent in compilers. This includes explicitly declaring the context and loci of control-flow of these concerns for improved comprehension. Further, we will characterize the actions, which will maintain or improve the comprehension and performance of such concerns. We carry out these endeavors in the context of an industrial strength compiler, the AspectJ compiler, ajc[36, 38, 54].

1.2 Thesis

We claim that *the logical structure of compilers does not align well with their current implementation because cross-cutting concerns are currently scattered and tangled with their phase structure; and, aspect-oriented modularization of such concerns im-*

proves their comprehensibility, provides better opportunities for reuse, and increases potential for flexible evolution.

1.3 Contributions

This research restructures an existing compiler for improved modularity by incorporating aspects. The goal is to write clearer and more concise code without any significant performance penalty. The improvements in expressiveness and structure will allow us to understand, reuse, and incrementally improve the compiler's components without incurring the expense of a complete rewrite. Further, this research makes its control-flow structure more explicit, for better comprehensibility.

The specific contributions are:

1. Identification of a number of points in the static program structure and dynamic execution graph of ajc corresponding to a variety of cross-cutting concerns; and, modular code changes that improve the control-flow structure and make it more explicit.

2. Example modularization of two different cross-cutting concerns:

   (a) peephole optimizer, and

   (b) error handler; and,

   identification of the difficulties for modularization of two other concerns:

   (a) separation of the Byte Code Engineering Library from the weaver, and

   (b) lazy evaluation of state dependencies.

3. Quantitative and qualitative assessment of the modified compiler in terms of

   (a) modularity,

   (b) reusability,

   (c) performance,

   (d) correctness, and

   (e) coupling and cohesion.

1.4 Outline

This thesis is organized as follows:

- First, we introduce background material related to the thesis in chapter 2. This includes placing this research in the larger context of programming languages, an overview of AOP, including relevant AspectJ constructs, and a description of existing alternate modularization techniques.
- We move on to describe a number of potential aspect candidates in compilers. In particular, chapter 3 discusses three standard candidates for modularization, and four other novel ones. In the subsequent chapters, we describe in depth the implementation and evaluation of the two most feasible candidates.
- Chapter 4 illustrates modularization of peephole optimizer in the context of ajc. Also, it presents a study that investigates the performance, comprehensibility, and modularity gains of this re-structuring.
- Chapter 5 is in similar vein to chapter 4. The candidate here is the error handling concern. Likewise, the evaluation metrices are comprehensibility, reusability, and modularity.
- Finally, in chapter 6, we summarize this work, and consider some avenues for additional research.

Chapter 2

Background

Considering compilers as the domain of interest, we begin this chapter by reviewing some background knowledge of compilers, and then proceed to lay the groundwork for our exploration of modularity issues in compiler design and implementation discussed in the remainder of the thesis. In particular, we first review the basics of compiler structure, followed by an overview of the programming paradigms and software engineering techniques currently employed for modularization. Next, we discuss modularity issues in compilers, and place this research in the larger context of programming languages. Then, we provide an insight into the ways in which Aspect Oriented Programming and its constructs assist us in addressing the modularity issues. This is followed by an overview of our candidate compiler – the AspectJ 1.6.4 compiler – from both structural and functional perspectives. Finally, we close this chapter with a literature review of prior efforts towards modularization of compiler design and implementation.

2.1 Compiler Organization

Compilers are programs that translate computer programs written in one language, the *source language*, into a semantically equivalent program written in another *target language*. The language that the compiler, itself a program, is written in is called the *meta language*. Generation of the target program from the source program involves several transformations. These transformations are called *phases*. To reflect the analyses performed between the phases, the input program is represented as an abstract syntax tree (AST). The AST represents the input program's phrase structure. Its sub-trees correspond to the phrases, such as statements, and expressions, of the source program. Its leaf nodes correspond to identifiers, literals, and operators

of the source program.

Good compiler design depends on a variety of factors, such as complexity of processing of input artifacts, availability of resources, and language facilities. A compiler for a relatively simple language used in an educational setting might be a single monolithic piece of software. However, modern compilers used in industrial settings to compile large and complex source languages entail restrictive design principles. Traditionally, the compilation process is divided into a number of phases, such as lexical analysis, parsing, semantic analysis, code generation, and optimization. This sequencing of compiler operations allows a language implementation to develop as a collection of separately compilable modules with consistent interaction patterns. This thesis focuses on such industrial-strength compilers.

Conventionally, compiler construction is viewed as a multi-stage process that generates output by following certain *compilation sequence*: the pre-defined order in which the phases are executed for any given input program. An artifact obtained as the output of one stage is passed on as an input to the succeeding stage in a pipeline, except symbol tables which remain for all the stages. This logical view leads us to understand and visualize compilers as shown in Figure 2.1.

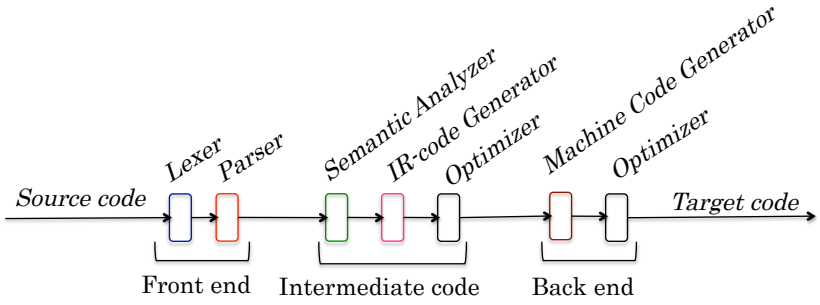

Figure 2.1: Logical structure of compiler phases.

From an implementation perspective, the front end of a compiler generates an abstract syntax tree (AST) to represent the input program, and the back end manages multiple traversals over the AST to implement various analyses, optimizations and finally, machine code generation.

The compiler phases are not actually realized in a clean modular structure as expected. An investigation of several popular compilers, namely: abc[4], ajc[36, 38,

54], gcc[32] , SML-NJ[2], and DemeterJ/F[47][1] has led to three major observations:

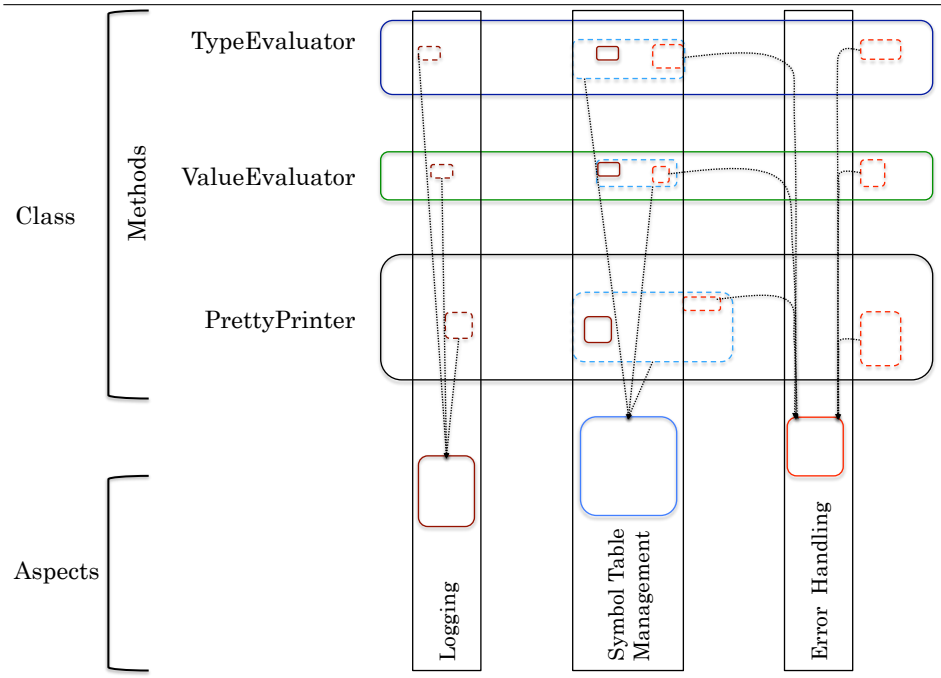

Figure 2.2: Cross-cutting nature of different concerns.

1. First, the phases of a compiler interact with each other in an intricate manner. Consider the semantic analysis and intermediate code generation phases. The semantic analyzer synthesizes and maintains type and value environments as part of type checking and symbol-table loading. Similarly, it also computes escape values as part of escape analysis. These values will be inherited by the intermediate code generator for managing static links and creating intermediate representation trees. Such intricate relationships engender tight coupling between type checking, escape analysis, symbol-table loading, and intermediate-code generation.

2. Second, these phases are of cross-cutting nature. Implementation of different concerns, such as symbol-table management, error handling, logging, and optimization, is often tangled and scattered across several units of modularization, as depicted in Figure 2.2. This scattering and tangling of code leads to poor structural and functional modularity. Although they are global design issues,

[1]Personal communication with Bryan Chadwick at Northeastern University.

which affect different methods and classes, they are handled locally within those methods and classes.

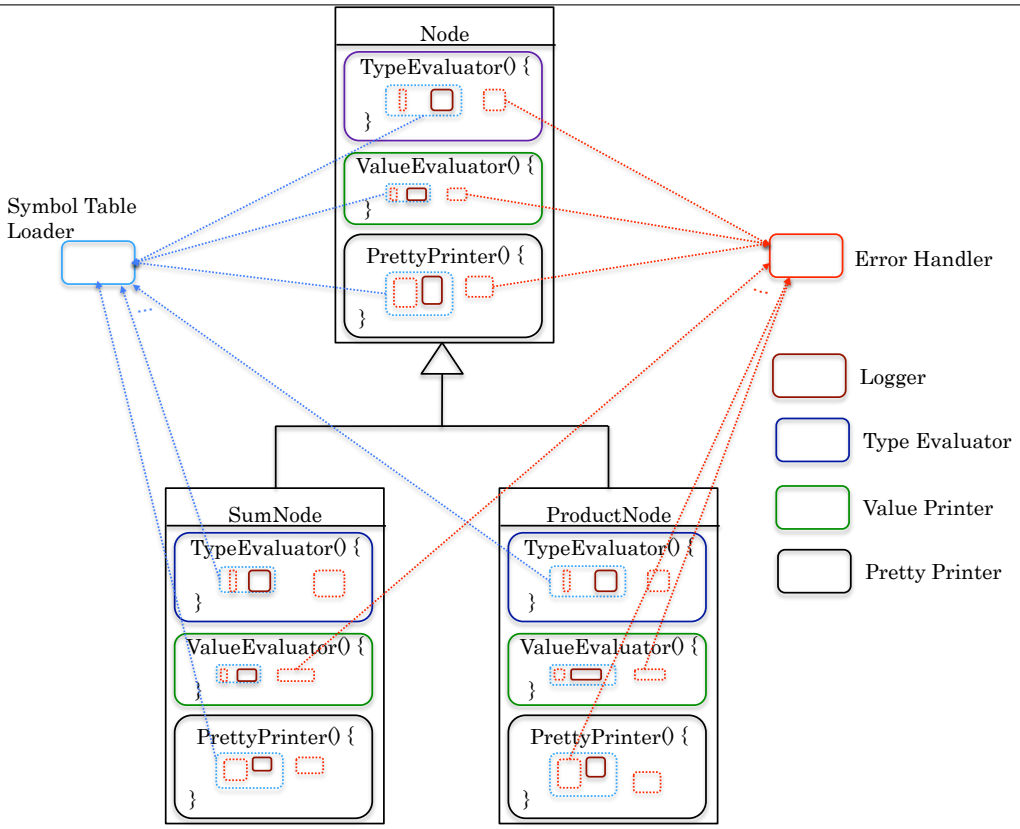

Figure 2.3: Implementation structure of compiler phases.

3. Third, control dependencies related to different concerns in compilers are not explicitly defined and localized into distinct modular units. As a result, context-sensitive execution paths related to different operations remain hidden within the compiler structure.

Currently, many compilers are structured as a small number of monolithic phases, which are either rigidly separated by different passes, or fused together by complex, unexpected interactions. The rigidly-separated passes obscure the meaningful transformations performed by individual passes. The complex interactions are mostly the result of a single phase performing several heterogeneous analyses, transformations and optimizations. Concerns, such as symbol table management, error handling, logging, and optimization, are orthogonal to traditional object-oriented hierarchy. They affect different units of modularity in the hierarchy, such as methods and classes, as

shown in Figure 2.3. However, they are still being implemented in an hierarchical fashion, which does not correspond to their natural decomposition. Combined, these factors make compilers difficult to understand and maintain.

Compilers with alternative structures such as nanopass[62] have been proposed to address the aforementioned problems. These alternatives facilitate development and maintenance of compilers by aligning actual implementations with their logical organizations. However, they incur additional overhead in terms of code duplication, additional passes, extra AST traversals and extra passing of intermediate artifacts through several passes of compilation.

To address these problems and challenges in the design and implementation of an optimizing compiler, this thesis describes a new approach to structuring the compiler code. Our goal is to attain modularity at both the structural and functional levels. The essential idea is to identify the cross-cutting functions in compilers and to implement them as stand-alone pluggable constructs. In order to define the activation order of these phases, compiler writers can implement a *compilation sequence manager* as an independent module. Further, to improve comprehension of control-flow information related to different features, we aim to define and express *control flow overlays* by leveraging the capabilities of existing language mechanisms. Control flow overlays declare the execution path corresponding to chosen *program overlays* – small segments of programs.

2.2 Intermediate Representation Trees

One common phase in the compilation sequence is creation of intermediate representation of the input code. It is an abstraction for machine language operations that is free of any machine-specific details. Also, it is independent of the source language. Although intermediate-code generation is not absolutely essential, it is preferred for reasons of increased portability, improved modularity, and greater optimization opportunities.

Intermediate code may be represented in different forms: linear sequence, expression trees, or pseudo-assembly. For the purpose of this thesis, we will consider an example of one in tree form. Table 2.1 summarizes the operators and operands of

the intermediate-representation (IR) tree that will appear in illustrative examples in this thesis.

Table 2.1: Intermediate representation trees.

	Operator	Operands	Description
Exp	`CONST`	int i	Integer constant i
	`TEMP`	Temp t	Temporary t; equivalent of machine-language register
	`BINOP`	int o, Exp e1, Exp e2	Apply binary operator o, such as PLUS and MINUS to e1 and e2
	`MEM`	Exp e	Contents of 1 word of memory starting at e
	`CALL`	Exp f, ExpList l	Apply function f to argument list l
	`ESEQ`	Stm s, Exp e	Evaluate statement s for side-effects, and expression e for a result
	`MEM`	Exp e	Contents of 1 word of memory starting at e
Stm	`MOVE`	Temp t, Exp e	Evaluate e and move its result to t
	`EXP`	Exp e	Evaluate e and discard the result
	`JUMP`	Exp e, LabelList l	Evaluate e and jump to one of the destinations from l
	`CJUMP`	int o, Exp e1, Exp e2 Label t, Label f	Evaluate e1 and e2 in order and compare result using o If the result is true, jump to t, else to f
	`SEQ`	Stm s1, Stm s2	Statement s1 followed by s2
	`LABEL`	Label l	Define name l to the current machine code address
	`ExpList`	Exp head, ExpList tail	List of expressions, where head is an Exp, and tail is an ExpList
	`StmList`	Stm head, StmList tail	List of statements where head is a Stm, and tail is a StmList

Note that expressions, represented as `Exp`, compute some values, possibly with side effects. Statements, represented as `Stm`, have side effects, and perform control flow, such as jumps and calls. We will provide additional description of intermediate representation when required.

2.3 Current Modularization Techniques

Existing modularization techniques do not adequately address behaviors spanning over different modules that are often unrelated. Since most systems include cross-cutting concerns, several techniques have emerged to modularize their implementation. Researchers have studied various ways to accomplish this task under a more general topic of "separation of concerns"[6, 7, 27, 30, 46] for elegant software design and evolution. Here, we will explore these existing alternative techniques that have been employed to modularize the implementation of cross-cutting concerns. Such techniques include mix-in classes, design patterns, adaptive programming, and domain-specific solutions.

2.3.1 Mix-in Classes

Mix-ins[6, 7, 27] are modular encapsulations for feature declarations and product definitions that can be shared in multiple modules. They are classes that are not used on their own; they are usually combined with other classes to extend the base class with desired properties.

Mix-in classes resemble Java classes except that a mix-in is parametrized over its superclass. Since a mix-in is not inextricably bound to any particular parent, a mix-in is regarded as being parametrized by a parent, which it is extending. Semantically mix-ins are, therefore, like functions from classes to classes, also called *functors* in the functional programming community. Consequently, they can be applied to each of multiple base classes to extend them into different derived classes with the same mixed-in behavior. By abstracting a class expression over an imported class, mix-ins enable the encapsulation of each extension in its own source code unit. While classes enable reuse because each class can be extended and refined by defining new subclasses, the reuse is one-sided; each class can be extended in many different ways, but each extension applies only to its superclass. A mix-in is parametrized with respect to its superclass, so it can add functionality to many different classes. Hence, a mix-in has a greater reuse potential than a class. Mix-ins enable us to uniformly extend a set of classes with a set of fields and methods. Thus, they improve the flexibility of class hierarchies, thereby improving the ability to modularize code and compose features.

2.3.2 Design Patterns

Another collection of techniques that is currently employed for modularization is *design patterns*. Design patterns describe superior and reusable solutions to commonly-occurring problems in software design. Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved[30]. This allows us to abstract the commonalities from the application, and to modularize separate behaviors or concerns into distinct units.

This section deals with one class of design patterns that is commonly found in

compilers: *behavioral* design patterns. This group of patterns help us identify common communication patterns between objects and concentrate on the assignment of responsibilities between them. Specifically, we mention two of them, namely the *visitor pattern*, and the *subject-observer pattern*[30] here.

Visitor Pattern

This behavioral design pattern promotes separation of concerns, and also provides a flexible design for extending existing functionality that are determined by types and multiple dispatch.

The visitor pattern is often used to separate the structure of an object collection from the operations performed on that collection. It defines two different class hierarchies: one for the *visitable elements* and the other for the *visitors*, as shown in Figure 2.4. The elements represent different data structures being operated on, and `Visitor`s represent different operations, algorithms, or behaviors related to these elements. Visitor pattern allows us to create a separate visitor concrete class for each type of operation. Every visitor has a method for every data structure element type. The data structure elements, however, only deal with the abstract visitor, and hence only have one method `accept()` that deals with it. That method is overridden in each concrete element, which performs the specialized operation for the visitor.

This pattern is also useful for adding new operations over existing data structures. This flexibility is the result of distinct separation of variant and invariant behavior in the visitor pattern. The variant behaviors are encapsulated in the concrete `Visitor`s. The programmer adds a new subclass to the visitor class hierarchy that defines the new operation. The invariant behaviors are represented by the data structure elements and the abstract `Visitor`. Thus, visitor pattern elegantly solves the problem of adding new operations to composite class hierarchies without modifying the text of either the composite class or its variants.

The visitor pattern, however, comes with several drawbacks:

1. The object-oriented paradigm describes a system as a collection of objects, where the supported operations are grouped with the objects themselves. The visitor pattern, on the other hand, forces us to create separate hierarchies for

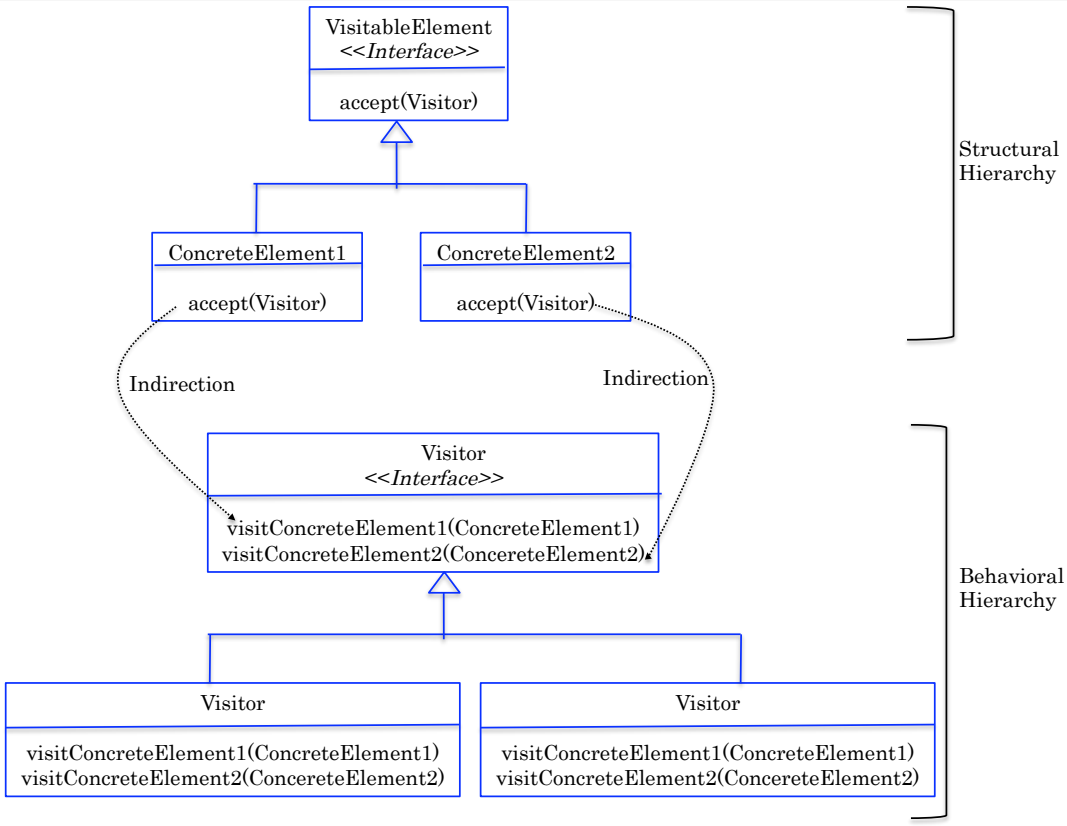

Figure 2.4: Object-oriented implementation of visitor pattern.

objects and for the operations on them. The visitor pattern violates the object-oriented viewpoint of the program.

2. With the flexibility to add new operations, the visitor pattern allows us to defer implementation of these operations. However, just as in case with mix-ins, the control-flow of the operation – invoking visiting logic – stays with the main class. Consequently, if a new visitable object is to be added to the operation structure, all the implemented `Visitor`s need to be modified. The separation of `Visitor`s from the objects they visit is only in one direction: `Visitor`s depend on visitable objects while visitable objects are not dependent on `Visitor`s[2].

3. Much code has to be written to prepare the use of `Visitor`s: the `Visitor` class with one abstract method per class, and a new `accept()` method in every class visited in the hierarchy. The extra code in the element classes makes it

[2]Part of the dependency problems can be solved by using reflection, with an attendant performance cost.

harder to understand and maintain the code. Furthermore, if a visitor pattern has not been incorporated from the beginning, the entire hierarchy has to be modified later to implement it. In the worst case, if the hierarchy cannot be modified, perhaps because the source is not available, the visitor pattern cannot be applied at all.

4. This pattern relies heavily on a double-dispatch mechanism to determine the concrete class of the element visited. Double-dispatch adds extra runtime method calls and could become problematic in performance-critical settings.

   Further, implementing redirection requires all the `Visitor`s to share the same interface as the abstract `Visitor`s. In other words, to support the double-dispatch mechanism, the abstract `Visitor`s forces the return types, arity, and parameter types of various visiting methods of a certain class to be the same. This requirement is difficult to meet because different operations usually have different computational needs and contexts. Also, this constraint tends to make programs less clear and introduces dependencies that can impede evolution.

Oliveira et al. have proposed a solution to ameliorate the problem relating to this structural rigidity imposed by the visitor pattern. They propose using generic and type-safe mechanisms to capture the essence of this pattern in a reusable manner. However, the problems resulting from the use of indirection still persist.

Lets consider a concrete example of the visitor pattern. Figure 2.5 shows a hierarchy of different nodes of an abstract syntax tree (AST) for an input program, with `Node` interface at its top. In the snippet of code shown, `VarNode` represents a variable-reference expression, `AssignNode` represents an assignment expression, and `ArithNode` represents an arithmetic expression. The `accept()` methods receive invocations from appropriate visitors with instances of appropriate types, upon which they can perform proper operations.

It also shows visitor hiearchy for operations over different nodes of the AST, with `NodeVisitor` interface at its top. The interface contains prototypes for visitor methods corresponding to different nodes. The `TypeCheckingVisitor` class, implements `NodeVisitor` and defines type checking mechanism for different kinds of expres-

```
/** Interface for Node hierarchy */
public interface Node {
    public void accept(NodeVisitor);
}
/** Variable reference Node */
public class VarNode implements Node {
    public void accept(NodeVisitor V) {
        //Implementation
    }
}
/** Assignment Node */
public class AssignNode implements Node {
    public void accept(NodeVisitor V) {
        //Implementation
    }
}
/** Arithmetic Node */
public class ArithNode implements Node {
    public void accept(NodeVisitor V) {
        //Implementation
    }

// /** Boolean Node */
// public class BoolNode implements Node {
//     public void accept(NodeVisitor V) {
//         //Implementation
//     }
}
```

(a) Node hierarchy.

```
/** Interface for visitors over different Node */
public interface NodeVisitor {
    public void visitVarRefNode(VarNode);
    public void visitAssignNode(AssignNode);
    public void visitArithNode(ArithNode);
    // public void visitBoolNode(BoolNode);
}
/** Type checker for different Node */
public class TypeCheckingVisitor
    implements NodeVisitor {
    public void visitVarRefNode(VarNode) {
        //Implementation
    }
    public void visitAssignNode(AssignNode) {
        //Implementation
    }
    public void visitArithNode(ArithNode) {
        //Implementation
    }
//  public void visitBoolNode(BoolNode) {
//      //Implementation
//  }
}
/** Code generator for different Node */
public class CodeGeneratingVisitor
    implements NodeVisitor {
    public void visitVarRefNode(VarNode) {
        //Implementation
    }
    public void visitAssignNode(AssignNode) {
        //Implementation
    }
    public void visitArithNode(ArithNode) {
        //Implementation
    }
//  public void visitBoolNode(BoolNode) {
//      //Implementation
//  }
}
```

(b) Visitor hierarchy.

Figure 2.5: Object-oriented implementation of visitor pattern.

sions, such as variable reference, assignment or arithmetic. Similarly, `CodeGeneratingVisitor` handles their code generation. Different `visit*` methods delegate to `accept()` methods in appropriate nodes when they have to type check, or generate code for the AST nodes.

If we wished to extend this implementation to include type checking and code generation for boolean expressions too, we need to implement the `Node` interface for booleans, update the `NodeVisitor` interfaces with visit methods for this new node. The required changes are shown as commented-out methods Figure 2.5. By requiring changes spanning across multiple classes and methods, such an implementation leads to possibilities of errors at multiple sites.

In subsection 2.5.3, we will investigate how aspect orientation overcomes such modularity problems associated with the visitor pattern, by providing an alternate implementation[31, 34, 68]. It should, however, be noted that the improvements in modularity are achieved at the cost of small performance overhead. Improved modularity is the primary motivation of this thesis, performance comes next.

Subject-Observer Pattern

Another behavioral design pattern that frequently appears in compiler implementations is the subject-observer pattern, see Figures 2.3.2, 2.6, and 2.3.2. The intent of this pattern is to "define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically" [30, p. 293].

This pattern is used to implement two abstractions that are dependent on one-another. For instance, consider implementation of the observer pattern in the context of a simple figure-editor system. The requirement is to update the screen after any changes to the underlying co-ordinates comprising a figure, such as points and lines. As shown in 2.3.2 too, `Subject` represents the core or independent abstraction, and `Observer` represents the variable or dependent abstraction.

```
1  public interface Observer {
       public void update (Subject s);
3  }

5  public interface Subject {
       protected List observers = new LinkedList();
7      public void addObserver (Observer o);
       public void remObserver (Observer o);
9  }
```

Listing 2.3: Interfaces for subject-observer pattern.

```
1   public class Point implements Subject {
      private int x = 0, y = 0;
3     int getX() { return x; }
      int getY() { return y; }
5     void setX(int x) {
          this.x = x;
7     }
      void setY(int y) {
9         this.y = y;
      }
11    public void add (SubjectFigElement) {
          // Implementation
13    }
      public void remove (SubjectFigElement) {
15        // Implementation
      }
17    public void addObserver (Observer o) {
          observers.add (o);
19    }
      public void remObserver (Observer o) {
21        observers.remove(o);
      }
23    public void notifyObservers() {
          // Notify each observer
25    }
    }
```

(a) Point class.

```
2   public class Line implements Subject {
      private Point p1, p2;

4     Point getP1() { return p1; }
      Point getP2() { return p2; }
6
      void setP1(Point p1) {
8         this.p1 = p1;
      }
10    void setP2(Point p2) {
      this.p2 = p2;
12    }
      public void add (Subject) {
14        // Implementation
      }
16    public void remove (Subject) {
          // Implementation
18    }
      public void addObserver (Observer o) {
20        observers.add (o);
      }
22    public void remObserver (Observer o) {
          observers.remove(o);
24    }
      public void notifyObservers() {
26        // Notify each observer
      }
28  }
```

(b) Line class.

Figure 2.6: Object-oriented implementation of subject-observer pattern.

There are two advantages to such an implementation. First, by encapsulating

```
public class ScreenUpdater implements Observer {
    public void update (Subject s) {
        // Screen update logic for different observers
    }
}
public class ViewPortUpdater implements Observer {
    public void update (Subject s) {
        // Color update logic for different observers
    }
}
```

Listing 2.6: Observer class for subject-observer pattern.

the dependent and independent abstractions into separate objects, it allows us to vary them independently. An `Observer` represents the dependent abstraction, and the `Subject` an independent abstraction. Next, it decouples the subject from the observers. The subject does not need to know anything special about its observers, other than to maintain their existence in a list. The observers simply subscribe to the subject and receive its event notifications. An event in our example is any change in the co-ordinate values. When the subject needs to inform its observers of an event, it sends a notification to each observer, through *notifyObservers()* method in our example. In this way, the observer pattern helps maintain consistency between related objects without making the two classes tightly coupled.

Object-oriented implementation of the observer pattern comes with three drawbacks however. First, addition or removal of a role from a class would entail changes in the corresponding classes. Second, any change in the notification mechanism would require changes in all the participating classes, `Subject` and `Observer`. Third, the pattern code, relating to `remObserver()`, `addObserver()`, and `notifyObservers()`, is scattered across different modularity units.

As we will see, aspect-orientation overcomes such problems associated with the observer pattern, by providing an alternate implementation[34].

2.3.3 Adaptive Programming

The visitor pattern and its variants have the disadvantage that programs are not easily adapted to changes in their inheritance and subsumption hierarchies[3]. Even

[3]A subsumption hierarchy refers to a collection, in which each entry in the hierarchy designates a set such that the previous entry is a strict superset, the *whole*, and the next entry is a strict subset, the *part*. For example, addition- and subtraction- expressions are part of arithmetic-expressions. Similarly, assignment-, arithmetic-, and declaration-expression are all part of an expression.

with separation of behavioral concerns from structural concerns, the visitor pattern is not amenable to evolution as we would like it to be. This is because the sequence of messages comprising operation depends on the underlying class hierarchy. Should the class hierarchy be changed because of system evolution, the sequence of messages usually must be changed as well, because some navigation paths have changed. As a result, evolution mandates that the operations must always be checked and usually reprogrammed. This is an expensive maintenance task.

Adaptive programming fills this void. This technique helps encapsulate class hierarchies using traversal strategies and visitors[46]. A traversal strategy describes a walk over data structures at a high level. It refers only to a minimal number of classes in a program's object model: the root of the traversal, the target classes, and waypoints and constraints in between to restrict the traversal to follow only the desired set of paths[46]. The visitors describe what to do at the specified points in traversal. In conjunction with the object model, a travel strategy helps to define methods that implement the traversal. An application behavior on the whole is defined by a collection of traversal strategies and visitors. Often, traversal strategies remain the same even when an object model changes slightly. In other cases, new traversal methods can be implemented in accordance with the new model, and the behavior can adapt to the new structure.

As an example, consider checking for undefined types in an XML schema containing a list elements and attributes. An element and an attribute declaration consists of names and types, which are defined as values of the corresponding attributes. Our task involves two traversals of the object structure representing the schema definition: one to collect all the types defined in the schema, and one to check each type referenced by a declaration to see if it is in the set of defined types. Implementation of these traversals in DJ[59], is shown in 2.3.3.

A DJ traversal is performed by calling the traverse method on an object. It takes three arguments: the root of the object structure to be traversed; a string specifying the traversal strategy to be used; and an adaptive visitor object describing what to do at points in the traversal.

In this example, the `getDefinedTypeNames` method collects the set of all type

```
static final ClassGraph cg = new ClassGraph();
public Set getDefinedTypeNames() {
    final Set def = new HashSet();
    cg.traverse(this, "from Schema via ->TypeDef,attrs,* to Attribute",
                new Visitor() { void before(Attribute host) {
                                    if (host.name.equals("name"))
                                        def.add(host.value); }});
    return def;
}

public Set getUndefinedTypeNames() {
    final Set def = getDefinedTypeNames(), undef = new HashSet();
    cg.traverse(this, "from Schema via ->Decl, attrs,* to Attribute",
                new Visitor() { void before(Attribute host) {
                                    if (host.name.equals("type")
                                        && !def.contains(host.value)
                                        undef.add(host.value); }});
    return undef;
}
```

Listing 2.7: DJ Implementation of traversal over XML schema.

definition names in a schema: it traverses the object structure rooted at the `Schema` object to every `Attribute` object reachable through the `attrs` field of a `TypeDef` object, and adds the attribute value if the attribute name is `name`'. Traversal `getUndefinedTypeNames` also traverses from `Schema` to `Attributes`, but it has different constraints: it collects at attributes of `Decl` objects.

As we see, a traversal strategy specifies the end points of the traversal, using the `from` keyword for the source and the `to` keyword for the target(s). In between, any number of constraints can be specified with `via` or `bypassing`. The benefit here is that the traversal strategy is oblivious to any specific association or hierarchies between the intermediate objects. Further, the traversal strategy will adapt itself if the root, target and the intermediate constraints remain the same, even if the object model changes. In this way, Adaptive Programming serves to separate the object structure from traversals upon them, thereby allowing independent changes to each.

Having looked at existing modularity techniques, we now proceed to investigate modularity problems in compilers from a broader perspective.

2.4 Modularity in Compilers

Conventional approaches to compiler construction view program source code as data, and different phases as functions operating on this data. Although, this is a generally accepted approach, it fails to meet the additional requirements of language specification, especially with respect to software engineering goals. Some of these

are:

- flexibility of compilation sequence,
- reuse of existing artifacts,
- extension of language models, and
- retargeting of compilers to different platforms.

Although the compilation process is decomposed into multiple phases, each phase itself is usually a large and complex entity with many inter-dependencies. A phase is often riddled with other smaller, but integral phases. For instance, a type evaluator must deal with other phases such as, symbol table maintenance, error handling, logging and pretty-printing. As a result, the type-evaluator module consists of code corresponding to these other operations, in addition to its own. Hence, the notion of re-usability has no concrete supporting mechanism in compiler construction. This means building a compiler often involves starting from scratch, even when similar applications are available. Hence, in addition to *functional* decomposition, there is need for another level of decomposition at the *structural* level for improved modularity, and its resulting benefits thereof.

From observation of several compilers, and literature reviews of language design and extension models, we have identified a number of impediments to modular language implementation. Here, first we identify such issues related to modularity in compilers. Then, we present a literature review of work related to aspect orientation applied to compilers for managing their complexity and modularity.

2.4.1 Modularity Issues

We explore the modularity issues in compilers at both the structural and semantic levels. Current compiler design and implementation techniques lead to three major classes of problems, which hinder their modularity, namely:

1. ***Tangling of semantics with syntax***

   The first modularity problem of language implementation resides at the structural level. In parser generators, such as YACC, syntax specifications and semantic actions are juxtaposed in the grammar specification file. The language

syntax is described by a formal specification, and semantics is implemented in a general purpose programming language. The semantics is usually attached to each grammar rule in the parser specification as action codes. The action codes are copied into the generated parser and are invoked when a production is reduced or derived. This tangling of action semantics and language syntax specification leads to coupling of two different concerns. This results in a large grammar specification file, polluted with action semantics. In summary, it leads to code that is hard to read and maintain.

This problem has been addressed to some extent with second generation parser generator tools. LL(k) parsers such as JavaCC+JJTree, and ANTLR employ visitor pattern for better separation of concerns. However, they still contain some traces of semantic actions in the grammar specifications. As discussed in subsection 2.3.2, modularity problems persist even with the use of visitor pattern. These problems can be addressed with AOP techniques.

2. ***Non-modular semantics***

   Although code is composable, compiler semantics is not compositional. Modular semantics would allow us to compose smaller semantics to infer broader meaning for better comprehension of the semantics that the compiler implements. There is a rich literature on modular semantics and compositional semantics[45, 63]. Still, none of these solutions achieve modular semantics convincingly and without restrictions[13, 74].

   As we can have a modular compiler, with non-modular semantics; we do not delve into the details of modular semantics and their composition here.

3. ***Intertwined phase implementations***

   The core of compiler development involves multiple passes over the abstract syntax tree (AST) that represents the source program. Typically, there are a number of analysis phases, each of which contains semantic operations cross-cutting the AST structure.

   Existing object-oriented approaches to compiler implementation encapsulate the semantic actions pertaining to one node as a method of that node class. This

approach leads to a problem where code related to one semantic phase scatters all over the AST node class hierarchy. For instance, operations such as optimization, error handling, and symbol table updates affect several other phases of a compiler and are present in every node. As with semantic behavior, their implementation is also scattered over the implementation of other functionality. This makes the resulting system hard to maintain and evolve.

A commonly employed solution to this problem is the visitor pattern. Semantic behaviors pertaining to a specific phase are encapsulated inside a single visitor class, and the actions are carried out using redirection. Each node in a visitor hierarchy has a general `accept()` method associated with it, which can redirect a semantic evaluation request to the appropriately specified method in the visitor class. We separate functional operations from the object structure. The benefit of using this pattern is that each traversal phase is isolated as a class that is independent of other nodes' classes and can be freely modified or extended. However, visitor pattern approach has a number of side effects, as discussed previously in subsection 2.3.2.

Some of these problems can be addressed by leveraging the abstraction, encapsulation, and modularization power of Aspect Oriented Programming.

2.5 Aspect Oriented Programming

Over the last decade, several language mechanisms have developed that claim to enhance expression, and modularization of cross-cutting concerns. Amongst them, Aspect Oriented Programming has become prominent, because:

- it is a powerful descriptor of control-flow overlays,
- it supports the notion of *open classes*, for enhanced modularity, and
- it blends well with the existing programming paradigms, such as Object Oriented Programming (OOP)[9].

A cross-cutting concern is one that is scattered and tangled with an existing implementation because of expressive limits of the language. These concerns do

not normally fit into a single program module, or even a small number of closely-related program modules. It is behavior and associated data that spreads across the scope of several features of a software. For instance, it may be a constraint that is characteristic of a piece of software, or shared behavior that every class must exhibit. Common examples of cross-cutting concerns are logging, profiling, context-sensitive error-handling, performance optimizations, and design pattern implementations.

Code and control flow pertaining to a cross-cutting concern often intertwine with those of other concerns. Therefore, modifying the implementation of a cross-cutting concern often involves numerous edits in many places. AOP complements OOP by facilitating another type of modularity that pulls together such widespread implementations of a cross-cutting concern into a single unit.

Among several implementations of AOP technology, AspectJ is the most richly-featured and popular one. In this section, we first give a primer of the AOP constructs, in the context of AspectJ language. In particular, we will look at those constructs that will be needed in comprehending the examples and descriptions in this thesis. Next, we will illustrate how aspect-oriented language constructs assist us in achieving improved modularity. Last, we compare and contrast AOP and other existing modularization techniques.

2.5.1 Aspect Oriented Programming Constructs

The ability of an AOP language to support modularization of cross-cutting concerns derives from the representational power of *join point model* (JPM), open-class support of *inter-type declarations*, and abstraction capability of *aspects*. We explore these language constructs here. They are shown in Figure 2.7.

Join Point Model

The JPM defines the "structure of dynamic cross-cutting concerns"[50, p. 2], and comprises concepts of :

1. ***join points*** - these are principled points in program execution or static locations in the source code. They enable access to the latent control structure of the language semantics. Examples of join points are method calls, method-body

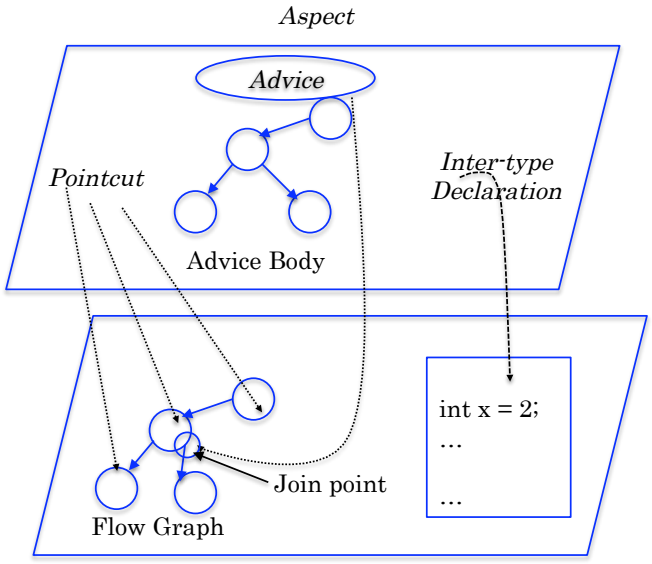

Figure 2.7: Aspect-oriented programming constructs.

evaluations, and class instantiations. AspectJ exposes only a small principled set of such events, identified by different *kinds* of join points. They are shown in Table 2.2.

Table 2.2: Description of join point kinds.

Join point kinds	Description
method or constructor call	represents execution points after method arguments are evaluated, but before the method or constructor is called.
method or constructor execution	represents execution points of a method or a constructor.
field access	represents read or write access to the field of a class.
exception handler	represents execution of exception handlers of specific types.
class-initialization	represents execution of static-class initialization of specified types.
lexical structure based	represents all join points inside a class or method's lexical structure. captures the code lexically inside a class, including an inner class.
control flow based	represents certain join points in a set of join points that occur in the control flow of some event.

Figure 2.8 summarizes different join points and their semantic descriptions. The numbered boxes in the left hand side represent different events, and the right hand side of the figure describes these events. For instance, the box labelled 1 represents an event before the execution of a method called `main` of the `Foo` class. An additional criterion is that the method should have a `void` return

type, and should take a `String` array as an argument.

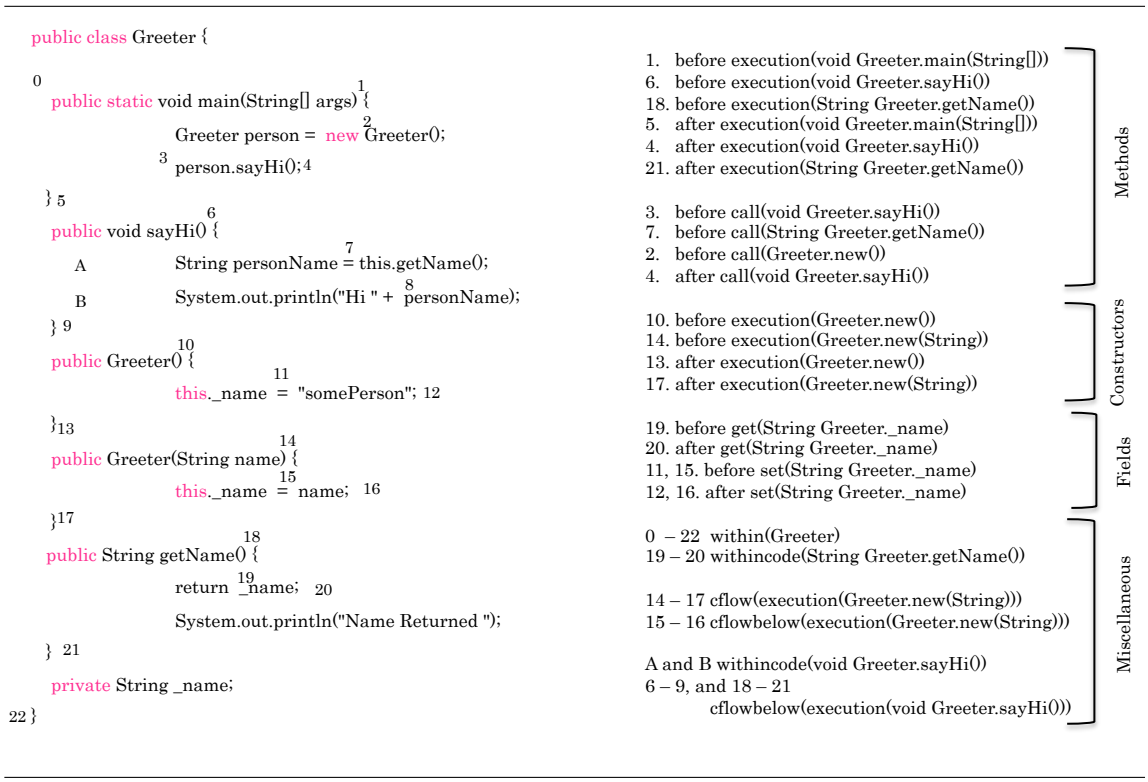

Figure 2.8: Examples of join points and their descriptions.

2. ***pointcut*** - it is a language construct that provides a means of specifying a set of join points, and their contexts. AspectJ includes several primitive *pointcuts*. These pointcuts can be composed to create compound pointcuts using `and`, `or`, and `not` operators ('&&', '||', and '!').

   Pointcuts help us define join points of interest. For instance, if we wish to identify dynamic join points based on lexical structure, we use the `within` and `withincode` pointcuts. In Figure 2.8, the pointcut `within(Foo)` represents join points within the `Foo` class, and `withincode(int Foo.bar())` represents those within the method `bar()` of the `Foo` class. These pointcuts are *anonymous*, because they do not have any identifiers. We can also create *named* pointcuts, as shown here:

```
1 pointcut FooScope(): within(Foo);
```

Listing 2.8: Example of a pointcut.

In this code, the keyword `pointcut` declares that what follows is a declaration of a pointcut, named `FooScope`, with trailing parentheses suggesting that the pointcut does not collect any context. Table 2.3 summarizes the primitive pointcuts along with their descriptions.

Table 2.3: Primitive pointcuts and their matching join points.

Pointcut	Description
`call(`*signature*`)`	join points at which a method or a constructor called matches the *method signature*: `retType rcvrType.method_id(paramType1,...)` or *constructor signature*: `classTypeName.new(paramType1, ...)`
`get(`*signature*`)`	join points matching execution of read or write access to a field with *field signature*: `fieldTypeName objectTypeName.fieldID`
`handler(`*typePattern*`)`	join points matching execution of an exception handler matching the *exception type-pattern*: `throwableTypeName`
`staticinitialization (`*typeName*`)`	join points matching execution of static blocks of a class matching *typeName*
`cflow(`*pointcut*`)` `cflowbelow(`*pointcut*`)`	join points in the control flow of events represented by *pointcut*, including/excluding those represented by the *pointcut* itself
`this(`*typeName*`)` `target(`*typeName*`)`	join points where the executing instance or the instance on which the method is called is of type *typeName*
`args(`*typeName1*`, ..., typeNameN)`	join points where the first argument is of type *typeName1*, contains any number and types of other arguments, and the last argument is of type *typeNameN*
`if(`*booleanExpression*`)`	join points where the *booleanExpression* evaluates to true
`within(`*typePattern*`)`	join points where the executing code is defined in a type matched by *typePattern*
`withincode(`*signature*`)`	join points where the executing code is defined in a method or constructor whose signature matches *signature*

3. ***advice*** - it is a means of affecting the semantics at the join points, by changing behavior `before`, `after`, or `around` the chosen join points. For instance, we wish to trace all the usages of the field `x` of the class `Foo` shown in Figure 2.8. First, we capture all the control points which access the value of `x`, with the pointcut `get(int Foo.x)`. Now that we have identified the join points of interest by means of pointcuts, we would like to execute some code before every access to value of `x`. We might write:

```
1 before(): get(int Foo.x) {
      System.out.println("Reference to field x, at "+
3                        thisJoinPoint.getSignature().toString());
  }
```

Listing 2.9: Simple before advice.

In this example, **thisJoinPoint** is a special reflective object that allows access to the run-time context in which the join point executes. Here, we use it to display the name of the event that triggers reference to the variable `x` within the objects of class `Foo`.

Construction of `after` advice is also similar. A change in decision to print the message after the field access, instead of before, involves replacing the keyword `before` with `after`. Additionally, there are two special cases of `after` advice, `after returning` and `after throwing`, corresponding to the two ways a sub-computation can return from a join point.

The third form of advice is the `around` advice, the most powerful form of all. It can not only read contextual information, but also change that information[4] and can even control whether the original join point computation should be done at all. It allows us to affect the join point at the time when the computation is imminent with the special `proceed()` construct. The `proceed()` expression takes a set of arguments and passes them on to the underlying join point. Consider the class `Foo` in Figure 2.8. If we wish to add a post-condition to the method `bar()` specifying that it should return an integer 1 in cases where it returns an integer 0, this requirement can be implemented with an `around` advice as shown in Listing 2.10.

```
int around(): execution (Foo.bar()) {
    int ret = proceed();
    if (ret == 0) {
        ret = 1;
    }
    return ret;
}
```

Listing 2.10: Simple around advice.

This advice executes the computation at the join point represented by `execution(Foo.bar())`. Based on its value, the advice either returns the original value or 1 if the calculated result is 0. Note that the `around` advice must be annotated with the return type, in contrast to the simpler `before` and `after` advices.

[4]Not entirely, cflow captured contextual information is immutable.

Inter-type Declarations

This is a concept AOP adopted from other techniques, such as mix-ins[7]. Inter-type declarations (ITDs) allow us to "statically introduce behavior to existing classes using static crosscutting techniques"[54, p.166].The concept of ITDs is also available in other languages, such as Modula-2 and Ada. They enable us to declare attributes, methods, and constructors on behalf of other existing classes. By doing so, they support the notion of open classes. Additionally, they allow classes to extend from new parents and change the inheritance relationship between classes using `declare parents` constructs. By providing facility of introduction, inter-type declarations also support concepts of mixin classes and multiple inheritance. Consider the `Point` class in Listing 2.11. It represents a point in a two-dimensional space.

```
1  class Point {
       int x, y;
3      Point (int x, int y) {
           this.x = x;
5          this.y = y;
       }
7      public void setX (int x) {
           this.x = x;
9      }
       public void setY (int y) {
11         this.y = y;
       }
13     public int getX (int x) {
           return this.x;
15     }
       public int getY (int y) {
17         return this.y;
       }
19 }
```

Listing 2.11: `Point` Class.

If we wish to extend its representation to use it in a three-dimensional space, we can implement this using inter-type declarations as shown in Listing 2.12.

```
1      int Point.z;
       public void Point.setZ (int z) {
3          Point.z = z;
       }
```

Listing 2.12: Inter-type declarations to make `Point` suitable for 3-D space.

Here, the first statement introduces an integer member named `z` into `Point` class. Likewise, the method `setZ()` (lines 2–4) adds a new method to this class. Thus, this class now represents a point with three coordinates.

Further, if wish to make the point cloneable, we can extend the class to im-

plement the `Cloneable` interface using the `declare parents` construct (line 1 in Listing 2.13). Listing 2.13 shows two inter-type declared methods, `slowClone()` and `fastClone()` that provide two different implementations of cloning operation.

```
1   declare parents: Point implements Cloneable;
2
3   public Point Point.slowClone() {
4       Point temp = new Point();
5       tempPoint.setX(this.getX());
6       tempPoint.setY(this.getY());
7       return tempPoint
8   }
9
10  public Point Point.fastClone() {
11      return new Point(this.getX(), this.getY());
12  }
```

Listing 2.13: Inter-type declarations to make `Point` class `Cloneable`.

At this point, we would like to introduce another `declare` statement, `declare precedence`, that is used in further discussions. AspectJ does not define the default order in which advices should be applied, when advices from multiple aspects match a common join point. In some situations, it is important to apply certain aspects in a chosen order. For instance, consider two aspects: one checking method invariants, and the other checking access permissions to that method. Essentially, the aspect dealing with access permission should be applied before the invariant-checking aspect. In such situations, the `declare precedence` statement is used to specify the desired order of application of the aspects. It is also defined inside an aspect, and is of the form:

`declare precedence:` *TypePatternList*;

Aspect

Finally, an aspect abstracts the modular implementation of cross-cutting concerns by encapsulating pointcuts, advices, and inter-type declarations for a single concern into a single module[40]. It provides a unit of modularity for a cross-cutting concern, enabling us to separate scattered logic from the classes in which it was previously tangled, and place it in a module of its own. Listing 2.14 shows a simple aspect that encapsulates the ITDs from Listing 2.12. An aspect resembles a class in many ways: it can contain methods and fields, extend other classes or aspects, and implement interfaces. However, aspects differ from classes in that they cannot be instantiated

```
aspect PointIn3D {
    int Point.z;
    public void Point.setZ (int z) {
        Point.z = z;
    }
}
```

Listing 2.14: A simple aspect.

using `new`. An aspect can be `abstract`, as can any named pointcut. Abstract pointcuts act similarly to a class's abstract methods: they let us defer details to derived aspects. A concrete aspect extending an abstract aspect must then provide concrete definitions of abstract pointcuts.

Aspect Instantiation Although users are not permitted to instantiate aspects, they can define the manner in which aspects should be instantiated. By default, AspectJ creates one instance of an aspect for an aspect type. Alternately, it can also be specified as follows:

`aspect issingleton()`

There are four other ways of instantiating aspects, using per-clause and specifying a pointcut as its argument:

- `perthis`– it creates an aspect-instance for each object bound to `this` at join points described by the argument pointcut.
- `pertarget`– it creates an aspect-instance for each object bound to `target` at join points described by the argument pointcut.
- `percflow`– it creates an aspect-instance for each object that exists in the control flow of the specified pointcut.
- `percflowbelow`– it creates an aspect-instance for each object that exists in the cflow-below of the specified pointcut.

Here is an example of its usage:

Development Aspect A development aspect is written during development of an application. Although it may not appear in the final product or application being shipped, it aids developers in various ways such as: limiting the scope of search,

```
aspect AspectPerShadowMunger perthis (shadowMungerCreation() {
    pointcut shadowMungerCreation():
        execution(ShadowMunger.new(..));
    ...
    ...
}
```

Listing 2.15: Aspect instantiation example.

enforcing development-policies, logging information, and tracing code segments. Appendix A shows some development aspects that we used during aspect-oriented modularization of ajc.

2.5.2 Aspect Oriented Programming and Modularity

AOP strives to cleanly separate concerns to overcome the problems discussed above. Consider snippet of code shown in Figure 2.9. It has two classes: *Point* and *Line*, intended for use in a graphics editor. A point represents co-ordinates in a two-dimensional space, and a line has two points, which denote its two ends. Now, consider the need to update the display every-time any coordinate changes. With a traditional OO approach, this entails inserting calls to `Display.update()`, a method responsible for updating various figure elements, after any coordinate is assigned a value. The resulting classes with these requirements are shown in Figure 2.9.

Although display updating is a concern distinct from setting the coordinate values, its implementation is tangled with the methods responsible for setting the coordinates. Further, the code related to display updating concern is not only scattered across several `setX()`, `setY()`, `setP1()`, and `setP2()` methods, but also across different classes. Thus, its implementation is not localized, thereby making its evolution cumbersome. Any need to change the display update concern would require us to track all the places where it occurs, and modify all the related classes. Further, this leads to the danger of inconsistent changes, if any one of these places is neglected or incorrectly modified.

Now, consider an aspect-oriented implementation of the display update concern. The base classes would remain unchanged, as shown in Figure 2.9. The additional functional requirement is now implemented in its own aspect, shown in Figure 2.10. By means of pointcuts, we first specify all the points in program control flow, where

```
/* Point class */
class Point {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;

  }
  void setY(int y) {
    this.y = y;

  }
}

/* Line class */
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;

  }
  void setP2(Point p2) {
    this.p2 = p2;

  }
}
```

(a) Simple `Point` and `Line` classes.

```
/* Point class */
class Point {
  private int x = 0, y = 0;

  int getX() { return x; }
  int getY() { return y; }

  void setX(int x) {
    this.x = x;
    Display.update(this);
  }
  void setY(int y) {
    this.y = y;
    Display.update(this);
  }
}

/* Line class */
class Line {
  private Point p1, p2;

  Point getP1() { return p1; }
  Point getP2() { return p2; }

  void setP1(Point p1) {
    this.p1 = p1;
    Display.update(this);
  }
  void setP2(Point p2) {
    this.p2 = p2;
    Display.update(this);
  }
}
```

(b) Display Update: OO implementation.

Figure 2.9: Object-oriented implementation of display-update concern.

display update should occur. Then, by means of the `after()` advice, we specify the display update behavior to instrument at the chosen points. Compared to the previous object-oriented implementation, this implementation localizes all the display-update-related code into a single module.

Any maintenance or update of the display concern would entail changes only in this aspect, thereby making its evolution simple. For instance, improving or adapting this aspect to implement display update in a three-dimensional co-ordinate system, might resemble in Figure 2.10. In this example, the pointcut `call (* *.set*(..))` contains a wild card pattern. It represents a call to any method whose name starts with "set", regardless of its containing type, return type and arity, and type of formals. In our example, since we are dealing with only two classes, namely `Point` and `Line`, it captures calls to all the `setX()`, `setY()`, `setZ()` (for 3-D), `setP1()`, and `setP2()` methods. This representation is more succinct than the previous one and is also more expressive in nature[5].

[5]But it does require discipline and standardization in naming; adding a method such as setupCache() would be

```
aspect DisplayUpdate {
  pointcut move(Figure fig):
    call(void Point.setX(int))    ||
    call(void Point.setY(int))    ||
    call(void Line.setP1(Point)) ||
    call(void Line.setP2(Point));

  after(Fig fig) returning: move(fig) {
    Display.update(fig);
  }
}
```

(a) Display update: aspect-oriented implementation.

```
aspect DisplayUpdateExtension {
  pointcut move(Figure fig):
    call(* *.set*(..));



  after(Figure fig) returning: move(fig) {
    Display.update(fig);
  }
}
```

(b) Extension of `DisplayUpdate` aspect.

Figure 2.10: Aspect-oriented implementation of display-update concern.

Further, as this concern is implemented as a pluggable construct, we could easily disable this implementation when not required. Likewise, as the concern implementation is free of any other concerns, we could easily adapt this aspect to implement similar concern in other applications also.

In these ways, aspect-orientation supports capturing of cross-cutting concerns, and enhances the modularization capability of existing programming paradigms. Note here that the cross-cutting display update concern that is scattered and tangled in a non-AOP implementation remains cross-cutting in nature, but it is no longer scattered and tangled in an AOP implementation.

2.5.3 Comparison of AOP and Existing Modularity Techniques

This section compares and contrasts aspect-oriented techniques in relation to other popular existing practices employed to deal with the cross-cutting concerns. From this, we will realize how AOP is naturally suited to handling the cross-cutting concerns, and recognize its benefits over other techniques in terms of expressiveness, re-usability and modularity.

problematic.

Mix-in Classes

As we saw in subsection 2.3.1, mix-ins enable us to uniformly extend a set of classes with a set of fields and methods. Although this can also be done with languages that support multiple inheritance, AOP allows us to do this with inter-type declarations and does not require us to directly set up/change an inheritance hierarchy. Further, in AOP, pointcuts help us expose runtime information to advices, whereas mixins only provide text substitution at the matched points.

Design Patterns

Here, we compare AOP with the two behavioral design patterns we described in subsection 2.3.2

1. **Visitor Pattern**

   The intent of the visitor pattern is to add behavior to an existing class hierarchy. AOP allows us to statically inject new operations into a class while keeping its original definition intact.

   For instance, consider the example shown in Figure 2.5. Adding support for boolean expressions would require us to make changes across several places in this hierarchy, as indicated by dispersed nature of the required code, shown as commented-out methods. With aspect-oriented programming, we can realize the equivalent behavior by means of an aspect, shown in Listing 2.20.

```
/** Interface for Node hierarchy */
privileged public aspect BooleanImplementor {
    /** Extend the Node hieachy with BoolNode */
    declare parents:
    public class BoolNode extends Node;
    /**
     * Implement for BoolNode, the accept method
     * derived from Node interface */
    public BoolNode.accept(Node) {
        //Implementation
    }
    /** Add typechecking visitor for BoolNode */
    public void TypeCheckingVisitor.visitBoolNode(BoolNode) {
        //Implementation
    }
    /** Add code generating visitor for BoolNode */
    public void CodeGeneratingVisitor.visitBoolNode(BoolNode) {
        //Implementation
    }
}
```

Listing 2.20: Aspect for type-checking and code generation of boolean nodes.

Hence, defining a hierarchy of new visitor classes and adding numerous `accept()` methods into the classes of the target hierarchy is not necessary. With AOP, adding a new behavior would mean creation of a concrete visitor aspect that introduces the corresponding method into all the classes, as shown in Figure 2.11. Likewise, removal of one such aspect would remove the associated behavior from the classes.

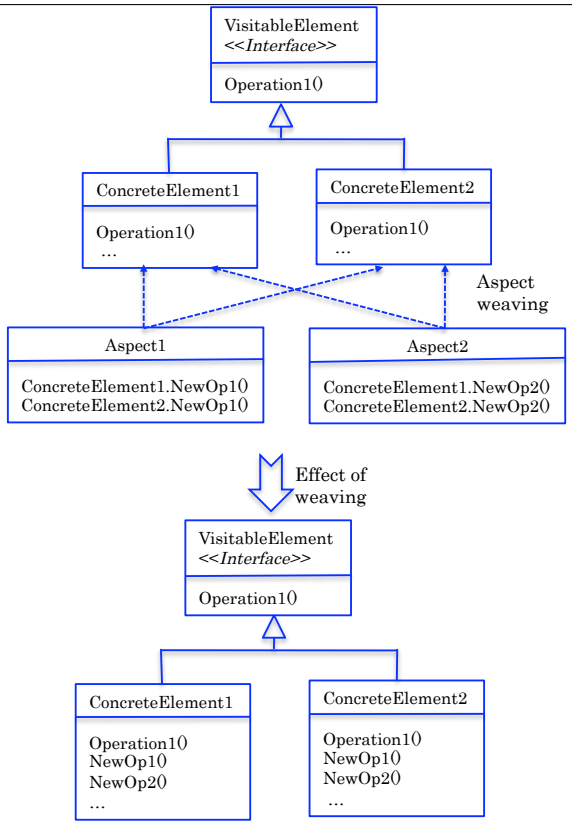

Figure 2.11: Aspect-oriented visitor pattern implementation.

Further, direct addition of operations to the object hierarchy in a static manner serves to remove the indirection operations and the associated problems. Instead of specifying the traversal code by means of visitors, they are now introduced by aspects. The outcome of this aspect oriented implementation is the class hierarchy, similar to the one shown in lower half of Figure 2.11. The design by itself is modular because particular operation for all relevant types is localized in one dedicated aspect. Further, the absence of double-dispatch leads to a simpler design. At run-time, the separation of the base system from the way in which

```
public abstract aspect ObserverProtocol {
  protected interface Subject { }
  protected interface Observer { }

  protected List getObservers (Subject s)
  {
      // Return list of observers
      // of Subject s.
  }
  public void addObserver (Subject s, Observer o)
  {
      // Add Observer o to the list of Observers
      // of Subject s.
  }
  public void remObserver (Subject s, Observer o)
  {
      // Remove Observer o from the list of
      // Observers of Subject s.
  }
  abstract protected pointcut
      subjectChange (Subject s);
  abstract protected void
      updateObserver (Subject s, Observer o);
  after (Subject s): subjectChange (s) {
      // call updateObserver() method on
      // each observer of Subject s.
  }
}
```

(a) Abstract observer aspect.

```
public aspect CoordinateObserver extends
 ObserverProtocol {
  declare parents: Point implements Subject;
  declare parents: Line implements Subject;
  declare parents: Screen implements Observer;

  protected pointcut subjectChange (Subject s):
      (call (void Point.setX(int))     ||
       call (void Point.setY(int))     ||
       call (void Line.setP1(Point))   ||
       call (void Line.setP2(Point))) &&
      target (s);

  protected void updateObserver (Subject s,
                                 Observer o) {
      // Update logic and implementation.
  }
}
```

(b) Concrete observer aspect.

Figure 2.12: Aspect-oriented subject-observer pattern implementation.

visitors visit the elements in the base systems is not present anymore, which eliminates the associated messaging overhead. In this way, aspect-orientation eliminates the need for visitor with its inherent support for open classes[14, 55], or introductions, and natural composition of textually separate modules into a logical whole.

2. **Subject-Observer Pattern**

   As discussed in 2.3.2 object-oriented implementation of the observer pattern is plagued by code scattering and tangling problem. Further, the pattern code is not easily reusable. Aspect orientation overcomes such problems associated with the observer pattern. Figure 2.12 shows aspect oriented implementation of this pattern. This implementation abstracts and localizes those concerns of the pattern that are common to all potential implementers of the pattern. Examples are: existence of subject and observer roles, publish-subscribe relation between the subjects and observers, and the event notification phenomenon. Additionally, it localizes the implementation of details specific to each instantiation of this pattern. Examples of such details are: specific classes which behave as subjects/observers, events that should trigger the update notifications to the observers; and the logic and mechanism of updates in the observers. Such details are implemented in a separate module. In our example, they are implemented

in the concrete aspect named `CoordinateObserver`.

In this way, aspect-oriented implementation of subject-observer pattern helps to extract and localize common details into a re-usable abstract aspect, by keeping intact its existing advantages.

Adaptive Programming

From a distance, Adaptive Programming appears to be a special case of AOP[46]. AOP, however, provides much broader support for separation of concerns. In addition, it provides us the ability to define and instrument behavior at chosen join points, unlike AP.

2.6 AJC: A Compiler for Aspects

Having decided to explore modularity issues in compilers, we have chosen to demonstrate our modularization endeavor in the AspectJ compiler (ajc). AspectJ is an aspect-oriented extension to Java, and has become the widely used, de-facto standard for Aspect Oriented Programming. It is written in Java, and like the Java compiler, it compiles programs written in Java to bytecodes. Although it is an aspect-oriented compiler, it surprisingly does not incorporate aspects to modularize the cross-cutting concerns within itself. Further, to support weaving from both source code and compiled code, often the compiler has to guarantee the correctness of a single piece of functionality at both these levels of weaving. Hence, the need and opportunity for handling the same concern at different levels in ajc provides a high level of complexity in modularization, over other mainstream compilers. A sizable number of committers, end-users, and its popularity were additional motivations that helped us establish the suitability of ajc for our purpose. What follows is a brief overview of ajc and its two passes.

2.6.1 Overview of the AspectJ 1.6.4 compiler

The ajc is primarily designed as a two-stage pipeline: an extended Java compiler and a binary weaver. We will first look at different components of ajc. Then, we provide a brief overview of its weaving and compilation process.

Components of ajc

1. ***Compiler*:** ajc (org.aspectj.ajdt.core) is an extension of the Eclipse Java Development Tool (JDT) compiler(org.-eclipse.jdt.core). Its parser has been extended in a recursive-decent manner to recognize and process AspectJ syntax. The extension acts as a pattern matcher for various AspectJ constructs, such as inter-type declarations, pointcuts, aspects, declare parents and declare soft. Also, it extends the Eclipse type resolution system to account for inter-type declarations. It produces plain Java class files as output.

2. ***Weaver*:** In addition to the normal front-end and back-end components of a compiler, ajc also contains a weaver. It provides bytecode weaving functionality both for inter-type declarations and advices. The weaver understands the attributes attached to aspects during compilation and performs the necessary pointcut matching and bytecode rewriting, or weaving.

3. ***Runtime*:** This module consists of classes that are used by generated code during runtime.

Overview of ajc passes

An overview of the compilation and weaving process of AspectJ 1.6.4[38], is depicted in Figure 2.13.

ajc first performs a *shallow parse* on all source files and generates the abstract syntax tree (AST). It collects and handles all AspectJ specific data and constructs, such as aspect names, hierarchy, and precedence, in a special way. For instance, an aspect declaration is parsed similarly as a type declaration. Likewise, pointcut declarations are implemented as a sub-type of abstract method declarations, and advice declarations are treated as method declarations. This also involves annotating the AST with additional aspect information attributes.

These compilation units[6] are then passed through `AjLookupManager` for *type binding*, and for generating the type hierarchy to be used with the AST. This helps to delineate connections between different parts of a program, and thus, enables clients to

[6]A compilation unit refers to the root node of the AST corresponding to a given Java source file.

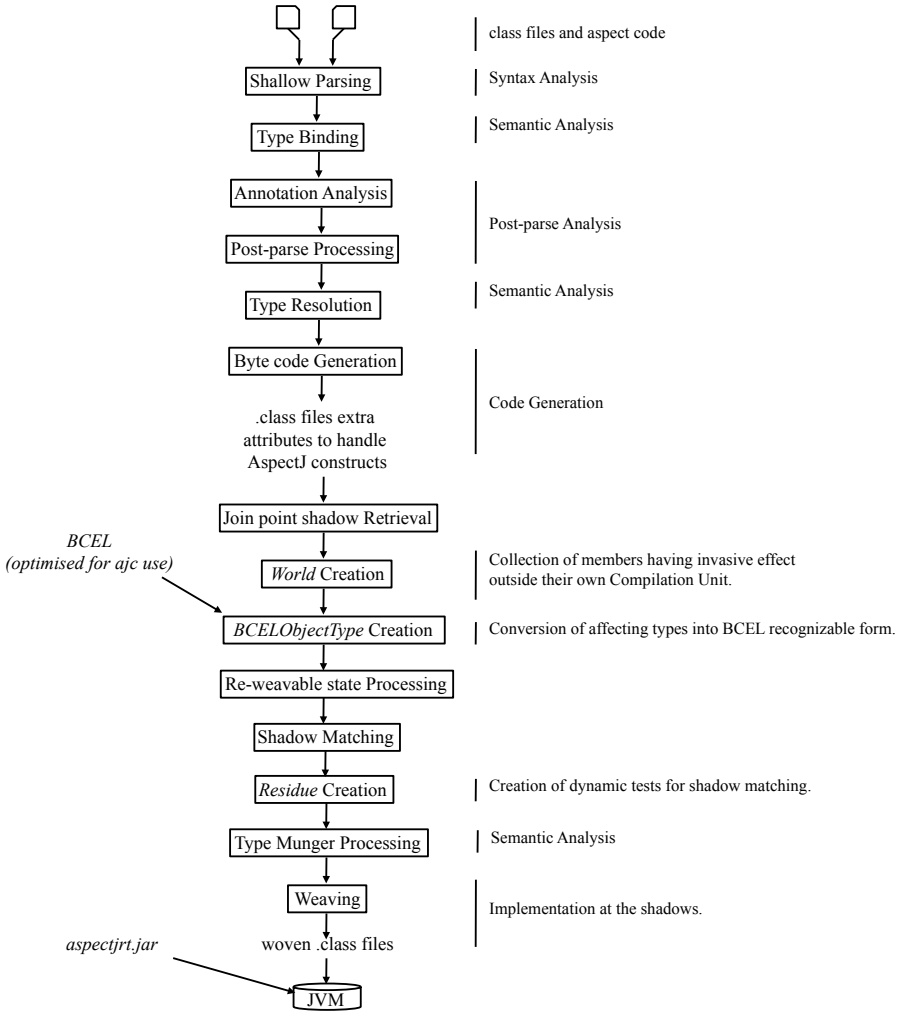

Figure 2.13: Overview of compilation and weaving of *ajc*.

analyze a program's structure more deeply. Type binding involves adding synthetic types to the type hierarchy to manage bindings resulting from AspectJ extensions. Examples are support for `EnclosingType`, `this`, `target`, and `declare` constructs. Synthetic types are empty/uninitialized fields, methods and type information such as super classes and super interfaces. They will be initialized correctly in a later stage. In addition to synthetic type creation, this stage generates bridge methods to override certain inherited methods, associated with aspects.

With respect to type-safety, the weaver-end of ajc contains certain typing-rules to which the aspect modules and its introductions should conform. Currently, the AspectJ language's type system ensures that advice code is executed at a join point

only if the advice signature is stronger than the signature of an intercepted join point[19, p.61][7].

This is followed by a *deeper parse* of the source files. During this phase, ajc delegates[8] to several other methods in the weaver to carry out *annotation insertions* and further analyses of the AST. The method calls construct annotated AST nodes with valid method and type declarations from the synthetic types, created earlier. These *postParse* methods, such as *PointcutDeclaration.postParse*, *AdviceDeclaration.postParse*, *DeclareDeclaration.postParse*, and *PseudoToken.postParse*, allow ajc to use the standard Eclipse code that analyzes a method body for analysing an advice body. Other postParse processing includes:

- generation of an aspect precedence relation from `declare precedence` statements,
- processing of inter-type declarations introduced by `declare parents`,
- name mangling of inter-type declaration members,
- completing the body of incomplete methods, classes and other types, which were created as part of shallow parsing. This involves annotating synthetic methods with additional attributes to indicate that they correspond to advice declarations, and also to store the pointcut referred to by the advice. Furthermore, additional arguments are added to the synthetic types to store additional information relevant to `thisJoinPoint` and `proceed` constructs, and
- removing of any ambiguities in the AST.

After postParse, the incomplete synthetic types generated during shallow parsing are, completed with all the required details.

[7] Here is a note on type-safety in ajc, from the paper: "the typing of pointcut and advice declarations is founded on this principle:

1. the body of an advice method must adhere to the advice signature (identical to how a regular methods should adhere to its signature),
2. the pointcut signature must be stronger than the signature of the join points that it selects, i.e. the selected join points must adhere to the pointcut signature, and
3. when an advice is bound to a pointcut, its signature must be stronger than that of the pointcut."

[8] this is a technical term involving the delegation design pattern and the delegate class.

To support incremental compilation, ajc associates the current compilation unit with this aspect-information-annotated AST. The AST is then compiled to produce class files annotated with AspectJ-specific information. For instance, a `before` advice in an aspect is represented as a special method in the class file with a Java attribute attached, capturing the fact that the method represents a `before` advice and what its pointcut was.

The weaver then retrieves *join point shadows* from the annotated class files. A join point shadow represents a bounded region of bytecode[36][9]. Semantically, it refers to dynamic points in program execution path, or static points in the source code. Every join point shadow is defined by a kind, a signature, and a region of bytecode. The source location of a shadow is given by the `SourceFile` attribute of the enclosing class file, and its line number is determined from the `LineNumberTable` attribute.

After retrieval of join point shadows, the weaver creates a **World** – the collection of all members that have an invasive effect outside their own compilation unit, before any weaving can take place.

Then, the weaver carries out *shadow matching* to find places in the source files and generated bytecode files where the code should be altered. This involves examining each join point shadow retrieved, to see if it is affected by any of the pieces of advice defined over the pointcuts. Advice and other advice-like entities, such as `declare` constructs, are represented by shadow-munger objects. A shadow munger performs transformation on join point shadow matched by its contained pointcut. During the weaving process, the pointcut for each shadow munger is matched against each join point shadow in the bytecode being processed. In some cases, the weaver needs additional information for shadow matching. As AspectJ join points are dynamic points in the call graph, the matching of shadow may not always be statically resolvable. When the pointcuts depend on the dynamic state at the join point, the mismatch is resolved by adding a dynamic test, also called the *residue*, that captures the dynamic part of the matching. The classes to be woven into are exposed to the

[9]There are a few exceptions, however. Initialization shadows, for example, require all constructors within a method to be inlined in order for their bytecode segment to be correct. In addition, exception-handler pointcuts do not have a clearly defined end-point. The description of this nature is beyond the scope of this dissertation, and hence, we do not delve into its details.

weaver as BCEL[39] representation forms.

Finally, the weaver rewrites the code of the system by inserting calls to the previously compiled advice methods. The basic weaving mechanism used in AspectJ is weave-time insertion of invocations of advice, or inlining of advice directly into base application code, namely at the join point shadow matching the corresponding pointcut. The weaver performs bytecode weaving using a derivative of BCEL, which is specially optimised for improving performance of ajc.

Upon completion of weaving, the bytecode is converted back to byte array form and emitted to .class files. These classes are then loaded into the Java Virtual Machine (JVM) in the presence of runtime library, *aspectjrt.jar*. The weaver also accepts pre-compiled class files, produced by an arbitrary Java source to bytecode compiler or other tool. It is capable of weaving the compiled aspects into these compiled bytecode jar files. Also, it can weave advice into classes dynamically as they are being loaded by the JVM.

2.7 Other Related Work

Others before us have explored the idea modularizing compilers[4, 72]. We proceed to review other projects related to aspect orientation in compilers that aimed at improving their modularity. We compare and contrast our design goals as well as underlying approach with this related work.

2.7.1 AspectBench Compiler

Avgustinov et al. [4] have built an extensible AspectJ compiler, called AspectBench Compiler (abc), with the basic objective of disentangling the code of the base compiler from that of extensions. The base version of abc implements full AspectJ language, while extensions contain additional language constructs. Extensions can be of different forms, requiring different levels of changes at different phases. For instance, *name pattern* scopes proposed by Colyer and Clement 15 that provides an abstraction mechanism for name patterns is fairly simple. It requires syntax extension and the addition of named patterns into the environment. Other extensions such as *parametric introductions*[33], *association aspects*[61], and *trace-based*

aspects[22] which are based on semantic properties require substantial alterations in several implementation modules.

The modularity structure of abc is primarily based on modularity of the Polyglot extensible compiler framework for the front end, the Soot analysis and transformation framework for the back end, and design patterns for other underlying implementations. Such implementations are prone to different limitations as discussed in section 2.3. But the concept of aspect-orientation for modularization still remains unexplored in abc.

The abc compiler also focuses on providing a software architecture to separate existing tools from aspect-oriented parts of the compiler. However, given the de-facto nature of ajc as a workbench for research into extensions of AspectJ and in view of the explosion of research into new features and analysis, this thesis concentrates on modularization of ajc itself. It aims to pave a way for easier, cost-effective and more comprehensive development of other systems – the belief is that modularization effort in this compiler will carry-forward to compiler implementations in general.

2.7.2 AspectJ Compiler

The core committers of AspectJ compiler development team have introduced three development aspects into the AspectJ 1.6.4 compiler (ajc):

1. CompilerAdapter.aj, which deals with ordering compiler passes,

2. OwningClassSupportForFieldBindings.aj, which deals with extending field locators and

3. OwningClassSupportForMethodBinding.aj, which deals with extending method locators.

They have also used a couple of developmental aspects to facilitate their compiler development and maintenance tasks. These aspects are available from the CVS repository at org.aspectj/shadows/org.eclipse.jdt.core/aspectj in the aj_v_785_R33x branch. In built form, they are also in HEAD/org.eclipse.jdt.core/jdtcore-for-aspectj-src.zip.

The significance of our work is even more evident when we look at the introduction of such aspects from the AspectJ development team. Although these concerns are close to the ones this thesis modularizes; they differ from the aspects that we have conceived.

2.7.3 Component Based Language Implementation

Wu [72] also investigates the idea of compiler modularization. His PhD dissertation presents a component-based language development framework with object-oriented syntax and aspect-oriented semantics. At the syntactic end, it provides a component-based LR parser that decouples grammar productions, and allows development of a large language driven by a set of smaller language parsers. It utilizes macros, templates, design patterns, and component-based software engineering techniques to separate syntax specifications from semantics in grammar files.

At the semantic end, it claims to encapsulate each semantic phase of a compiler as an aspect. However, it implements only two trivial compiler concerns – name analysis and pretty printing of the Java language.

The focus of this thesis is not aspect-orientation of all semantic phases. This work distinguishes itself from Wu's work in that it identifies the concerns that cross-cut the traditional phase-decomposition structure of compilers. We modularize such cross-cutting concerns by leveraging the capabilities of AOP language mechanisms.

2.8 Summary

The primary motivation for this work is the observation that existing trends in language implementations focus only on functional decomposition, to manage their complexity. Our belief is that this decomposition should be at the structural level too, so that it leads to implementation oriented thinking. Implementation aspects have had deep effect on the way we design and view programming languages.

Like other complex industrial tools, programming languages can be composed of relatively specialized parts that can be used as such in many kinds of system-building tools. This would give us the same benefits that are now regarded as self-evident in other engineering branches: new production (i.e., programming) systems

could be rapidly developed for different purposes using existing building blocks, old systems could be modernized by replacing certain parts with more advanced parts, and system maintenance would be eased because the system consists of small modules with clean interfaces.

Among other existing techniques, we examined how AOP constructs help us align practical implementation with logical decomposition by providing support for managing cross-cutting concerns, and expressing the control flow overlays. Moving on, we also considered prior efforts towards modularization of language implementations, and aspect orientation in compiler construction. Last, we showed how this work differs from these related works.

On the whole, the goal of this thesis is to restructure the core AspectJ compiler (ajc), implementing an aspect-oriented extension of Java, with the aspect oriented compiler AspectJ itself so that the concerns are better localized, extensions are composable, and control-flows are comprehensible.

Chapter 3

Candidates for Modularization

Compilers are programs built from a number of operations that interact with multiple other operations. In this chapter, we examine seven such operations. To cover the breadth of compilers' operations, we first look at three standard functions:

1. canonicalization,
2. compilation sequencing, and
3. register-allocation optimization.

In particular, we focus our discussion on their current implementation strategies and possibilities of their aspect-orientation for improved modularity. Next, we identify and investigate aspect-oriented modularization of four novel candidate implementations in compilers:

1. lazy evaluation of state dependencies,
2. separation of planning and usage of bytecode manipulation tools,
3. peephole optimization, and
4. error handling.

From a modularity perspective in compilers, identification of other similar candidate concerns and their implementation according to the principle of separation of concerns remains an active area of research. Here, we investigate only a fragment of this spectrum, which we now proceed to present for the sake of completeness. We provide an overview of the first two candidates in this chapter. The other two are the ones we have identified as suitable candidates for implementation. In chapters 4 and 5, we will describe in detail their analyses, implementations, and utilities.

3.1 Standard Candidates

For the purpose of our discussion, we characterize compiler operations that constitute a well-established notion of a distinct or separate module in a compilation sequence, such as canonicalization and register allocation, as standard candidates. In this category, we also include operations, such as compilation sequencing, whose current implementation conforms to different design patterns, and would result in improved modularity when realized using their aspect-oriented counterparts. We proceed to investigate them here.

3.1.1 Canonicalization

Our first standard candidate is canonicalization of the intermediate-representation tree.

During compilation of a source program, the semantic analyzer generates Intermediate Representation (IR) trees, which are then translated into assembly or machine language by the code generator. This conversion of IR trees to assembly code, however, cannot be done directly because there are certain concepts in the tree representation that do not correspond exactly to a machine language. One such difference is in the operational behavior of different instructions.

For instance, in IR trees, the `CJUMP` instruction can jump to either of its two labels. In machine-level conditional jumps, however, the instructions fall through to the next instruction if the condition evaluates to false.

To resolve such problems, resulting from mismatches between the IR code and machine code, compilers re-create the IR tree by moving `CALL`[1] nodes to the top level so that their parents are either `EXP()` or `MOVE(TEMP t, ...)`, and by replacing all `ESEQ`[2] and `SEQ` nodes with their equivalent forms. This process is called *canonicalization*. Snippet of Java code that creates canonical trees is shown in Figures 3.1

[1] The issue with `CALL`s is that multiple `CALL` nodes within a single expression could cause problems with call-related register usage, when mapped to lower-level code. For instance, nested calls, such as `BINOP(PLUS, CALL(...), CALL(...))`, will cause interference between register arguments and returned results. The `CALL` node is implemented in such a way that each function returns its result in the same dedicated return-value register `TEMP(RV)`. Thus, in an expression of such form, it is likely that the second call will overwrite the RV register before the PLUS operation executes.

[2] The issue with `ESEQ`s is that they impose strict order of evaluation by means of statements with side-effects.

and 3.2.

```
public interface ICanonvisitor{
    public EXP visit(EXP expNode);
    public STM visit(STM stmNode);
    public FUN visit(Fun funNode);
    ...
    ...
}
```

Listing 3.1: Interface for `CanonVisitor` class.

```
public class CanonVisitor implements ICanonvisitor {
    ...
    ...
    public EXP visit (CALL callNode) {
        EXP args = callNode._args.accept(this);
        STM s1 = getStm(args);
        TEMP t_new = new TEMP();
        STM s2 = new MOVE(t_new, new CALL(callNode.getFun(),
                                          (EXPList) getExp(args)));
        return new ESEQ(mergetStms(s1,s2), t_new);
    }
    public EXP visit (ESEQ eseqNode) {
        STM s1 = eseqNode._stmt.accept(this);
        EXP e1 = eseqNode._exp.accept(this);
        STM s2 = mergeStms(s1, getStm(e1));
        if (s2 != null)
            return new ESEQ(s2, getExp(e1));
        return eseqNode;
    }
    ...
    ...
    public EXP visit (EXP expNode) {
        ...
        ...
    }
}
```

Listing 3.2: `CanonVisitor` class.

In object-oriented paradigm, canonicalization is usually implemented as a visitor program over an IR tree. To handle `CALL` nodes, the visitor introduces a new temporary register, **temp**, to save the return value; and replaces the `CALL` node with an `ESEQ` node. The re-writing rule is implemented as follows:

> `CALL`(fun, args) is replaced by
>
> `ESEQ`(`MOVE`(`TEMP` t_new, `CALL`(fun, args), `TEMP` t_new)

The related piece of code is shown in Listing 3.2(b) (lines 4 – 11).

Similarly, to eliminate `ESEQ` nodes, the visitor program visits every expression node, extracts the statements out, and forwards them to an upper level, as shown in Listing 3.2 (lines 12 – 19). This process continues until the `ESEQ` nodes can be replaced by `SEQ` nodes. For instance, `ESEQ(S1, ESEQ(S2,e))` can be replaced by `ESEQ(SEQ(S1,S2),e)`.

Although the use of design patterns, visitor in this case, brings several benefits, it also hard codes the underlying system and makes it difficult to express changes in the code, as we discussed earlier in subsection 2.3.2.

Aspect-oriented implementation of this pattern provides better opportunity for reusing and evolving it. This support comes from the use of inter-type declarations for inserting visitor methods into an existing visitor pattern, as we saw in subsection 2.3.2.

3.1.2 Register Allocation Optimization

Our second example of the standard candidate is optimization of register allocation.

Local variables of methods, intermediate results of expression evaluations, and other similar values are usually stored in registers, rather than stack frames, to improve execution performance. Register access is faster, because instructions can directly use values stored in registers during computation[3]. Memory access, however, is slower because it involves extra memory cycles to execute *load* and *store* instructions, that are used to fetch values from the stack.

Registers are valuable resources, but they are available only in limited amounts. Usually, a machine has only one set of registers, which are used by several procedures and functions. Therefore, *register allocation*, the process of deciding which values to store in which registers over what period of program execution, entails careful planning and usage.

For instance, consider a function f, storing its local variable in a register r, calls procedure g, which also uses r to hold its temporaries. To avoid register-usage conflict in such a case, r must be stored into a stack frame, before g uses it; and, restored from the frame after g is done with the register. This store-and-fetch mechanism can be implemented either by f, the *caller*, or by g, the *callee*. Register r is called *caller-save* register if the caller implements the store-and-fetch mechanism, and *callee-save* register if the callee implements it.

Effective allocation of registers for locals and temporaries can minimize the number of store-and-fetch situations, or even avoid them entirely in some cases. For instance, if f knows that the value of a variable x is not needed after the call, it may

compute x in a caller-save register, but not save it when calling g. Conversely, if f has a local variable i that is needed before and after several function calls, it may save r_i just once (upon entry to f); and put i in some callee-save register r_i, and fetch it back just once (before returning from f).

Closer examination of this behavior reveals that there is an inherent publish-subscribe relation between the callers and the callees. Such a relationship can be represented using the subject-observer pattern. With this pattern, the calling function does not actually need to store all its arguments into its stack frame, to make room for those of the callee. When the callee function must write its arguments to the frame, the callee can notify its observer, which is the caller in this case. To receive such notifications, the caller can subscribe to the callee. Upon receiving a notice from the callee, the caller will alter its storage to save appropriate variables into stack frame, and to create room for callee's values in registers.

Burger et al. [10] have already proven this lazy-save register-allocation idea to be efficient. They investigate and implement such an optimization from a performance-improvement perspective. Our perspective is different– we focus on modularizing this implementation, so that we can isolate the optimization-related pieces of code from the compiler and implement them into a distinct module of its own. Specifically, aspect-oriented subject-observer pattern implementation would provide us additional benefits that we discussed in chapter 2. However, we chose not to implement this candidate as it is already considered for implementation.

3.1.3 Compilation Sequencing

Our third and final example of a standard candidate for aspect-oriented modularization in compilers is sequencing of compilation events.

Generation of a target program by a compiler involves traversals, called *passes*, over the source program or its internal representation – the abstract syntax tree (AST). Most modern compilers carry out multi-pass compilation – multiple traversals of AST to carry out different analyses. In addition to phase analyzers, such compilers consist of a compiler-driver or a compilation-sequence manager, which is responsible for invoking these phase analyzers in a desired order. It observes each phase in a

compilation sequence to decide about next phase invocation.

For instance, if the source program lacks a proper syntax structure, the parser generates error reports instead of constructing an AST. As a result of failed parsing, semantic analysis should not occur and similarly, other following phases should not occur either.

Intuitively, there is an inherent event-notify relationship between the compiler driver and different phases. Correct functioning of the compiler-driver depends on event notifications from a currently executing phase. If one phase executes successfully, the driver calls the next phase in the sequence, else it reports an error and stops further compilation. Such a one-to-many dependency relation between the driver and phases can be realized with the subject-observer pattern. As discussed in chapter 2, aspect-oriented implementation of this pattern, provides increased opportunities for code reuse, and better modularity.

The AspectJ compiler incorporates such an aspect, `CompilerAdapter`, that drives different compiler events. The interface and the aspect itself are shown in Listings 3.3, 3.4, and 3.5.

```
/** Interface for CompilerAdapterAspect */
public interface ICompilerAdapter {
        void afterDietParsing(CompilationUnitDeclaration[] units);
        void beforeCompiling(ICompilationUnit[] sourceUnits);
        void afterCompiling(CompilationUnitDeclaration[] units);
        ...
        ...
        void beforeGenerating(CompilationUnitDeclaration unit);
        void afterGenerating(CompilationUnitDeclaration unit);
}
```

Listing 3.3: Interface for CompilerAdapter aspect from ajc.

It does not incorporate aspect-oriented subject-observer pattern. It simply relies on the capabilities of the join point model for this. The sequencing and dependencies among phases of a compiler are realized by using the control-flow-expressivity capability of pointcuts and ordering among phases is realized through weaving rules. The weaving rules are achieved through different kinds of advices, such as before, after, after returning, and after throwing.

The `ICompilerAdapter` interface shows prototypes of methods that define the actions to trigger at various strategic points in compilation. Examples include events before and after diet parsing, compilation, type-analysis and -resolution, and code

```
/**
 * This aspect implements the necessary hooks around the JDT compiler
 * to allow AspectJ to do its job
 */
public privileged aspect CompilerAdapter {
   /** Default Adapter Factory */
   private static ICompilerAdapterFactory adapterFactory =
       new ICompilerAdapterFactory() {
           public ICompilerAdapter getAdapter(Compiler forCompiler) {
               return new DefaultCompilerAdapter(forCompiler);
           }};
   public static void setCompilerAdapterFactory(ICompilerAdapterFactory factory) {
       adapterFactory = factory;
   }
   /** Adapter to manage compilation event */
   private ICompilerAdapter compilerAdapter;
   pointcut dietParsing(Compiler compiler):
       execution(void Compiler.beginToCompile(ICompilationUnit[]))
       && this(compiler);
   pointcut compiling(Compiler compiler, ICompilationUnit[] sourceUnits) :
       execution(* Compiler.compile(..))
       && args(sourceUnits)
       && this(compiler);
   pointcut generating(CompilationUnitDeclaration unit) :
       call(* CompilationUnitDeclaration.generateCode(..))
       && target(unit)
       && within(Compiler);
   ...
   ...
```

Listing 3.4: Pointcuts and related code from CompilerAdapter aspect in ajc.

generation. Some of these are shown in Listing 3.3:

1. `afterDietParsing(...)`,
2. `beforeCompiling(...)`,
3. `afterCompiling(...)`,
4. `beforeGenerating(...)`, and
5. `afterGenerating(...)`.

The strategic points where the method invocations must occur are described by different kinds of advices, such as `before`, `after`, and `after() returning`. For instance, as shown in Listing 3.5, the body of `after` advice (line 22) contains a call to `afterDietParsing(...)` method on the `compilerAdapter` to handle postParse activities, discussed in subsection 2.6.1. Similarly, `before` (lines 42 -44) and `after` (lines 45 - 47) advices contain calls to pre- and post-code-generation phase of the compiler.

We chose not to implement any of these three candidates because one of them, compilation sequencing, is previously implemented, and the remaining two are fairly

```
    after(Compiler compiler) returning(): dietParsing(compiler){
        compilerAdapter.afterDietParsing(compiler.unitsToProcess);
    }
    before(Compiler compiler, ICompilationUnit[] sourceUnits) :
    compiling(compiler, sourceUnits) {
        compilerAdapter = adapterFactory.getAdapter(compiler);
        compilerAdapter.beforeCompiling(sourceUnits);
    }
    after(Compiler compiler) returning : compiling(compiler, ICompilationUnit[]) {
        try {
            compilerAdapter.afterCompiling(compiler.unitsToProcess);
        } catch (Exception e) {
            compiler.handleException(e);
            throw e; // rethrow
        } finally {
            compiler.reset();
            this.compilerAdapter = null;
        }
    }
    ...
    ...
    before(CompilationUnitDeclaration unit) : generating(unit) {
        compilerAdapter.beforeGenerating(unit);
    }
    after(CompilationUnitDeclaration unit) returning : generating(unit) {
        compilerAdapter.afterGenerating(unit);
    }
}
```

Listing 3.5: Advices from CompilerAdapter aspect in ajc.

straight-forward counterparts of different design patterns. Related prior work describes aspect-oriented implementation of such patterns [34]. In the next section, we will look at four novel candidates that we consider for implementation.

3.2 Novel Candidates

Although established compiler designs decompose the bulk of a compiler into different operations and try implementing them in distinct modules, there still remain other operations that are not yet properly isolated from others. In this thesis, we refer to such operations as novel candidates, and focus on their aspect-oriented modularization into separate modules of their own.

3.2.1 Lazy Evaluation of State Dependencies

The first novel candidate that we consider is lazy evaluation of state dependencies.

Traditionally, compilers have been internally organized into separate phases that are invoked in a fixed, predetermined order. We can improve this organization with a demand-driven ordering on the activation order of the phases. This flexibility would free the compiler writers from considering the phase activation order as integral to the design. The ordering of the phases is, however, constrained by two different kinds

of dependencies, namely: *phase dependencies*, which restrict how an input program can be passed through successive phases and *compilation-unit*[3] *dependencies*, which restrict how different parts of the input program can be passed through individual phases[44]. Lets look at examples of each of these.

The intermediate-code generation phase deals with translating abstract syntax into IR code. The IR code can be generated either along with or after completion of type-checking. In either case, intermediate code generation will take place only if the code is syntactically and semantically correct. Therefore, IR code generation depends on type-checking and context-dependent analysis. Likewise, when operations in an input program misuse their operands, the input program contains a type error. In such cases, the semantic analyzer should generate a type error and stop further steps of compilation. Thus, semantic analyzer depends on type checker and phases following type checking depend on the semantic analyzer.

Now, lets consider compilation-unit dependencies. A compiler performs type-checking on an AST. Therefore, the decision as to whether type-checking on a compilation unit can occur depends on whether all the required and dependent AST nodes have been generated or not. If the compiler tries to type-check an AST node, which depends on other AST nodes, it has to wait until the dependee node is available.

For a better understanding of the problem, consider the snippet of code written in SML programming language in Listing 3.6. This piece of code deals with type-checking the abstract syntax of an *if-expression* and also calls a method to generate its corresponding IR code.

The first three declarations(lines 4-6) in the `let` declaration sequence determine the types of `test, then` and `else` expressions that constitute an `if`-expression. The function `trExp` returns the intermediate representation of the entire expression, in addition to its type. The `checkIfInt()` function in the let-body tests whether the type of `test`-expression in the `if`-expression is of type integer[4]. Similarly, the `checkIfEqual()` function checks for the equality of the types of `then` and `else`

[3] A compilation unit refers to an AST node for an entire source file.
[4] Here, we assume that booleans are represented as integers.

```
let
fun trExp (Absyn.IfExp{testExp,thenExp,elseExp=SOME elseExp,pos}) =
        let
            val expTyT as {exp=expTest, ty=typeOftest} = trExp testExp
            val expTyN as {exp=expThen, ty=typeOfthen} = trExp thenExp
            val expTyE as {exp=expElse, ty=typeOfelse} = trExp elseExp
         in
            (checkIfInt (typeOftest, pos, "IF test")
            ; checkIfEqual (typeOfthen, typeOfelse, pos, "IF consequents")
            ; { exp = Translate.tranlsateIfThenElse (expTest, expThen, expElse)
            , ty = morePreciseTy (typeOfthen, typeOfelse) } )
         end
   ...
   ...
in
   ...
   ...
end
```

Listing 3.6: Type checking and intermediate-code generation for an `if`-expression.

expressions.

If either of these tests fails, the semantic analyzer reports errors and looks for other errors in this phase, but stops execution of other phases. If the tests pass, it invokes `translateIfThenElse()` function, which produces IR code corresponding to this `if`-expression. Here, we see that type checking and IR code generation are coupled into a single pass. A benefit of this approach is reduced number of compiler passes. A disadvantage is that two semantically different phases have been coupled together. Moreover, when IR code generation is done hand-in-hand with type checking, IR-code generation will occur for all AST nodes up to the point of type error. Clearly, there is extra work involved in generating IR code that will be discarded

Decoupling these phases would result in significant overhead, because the compiler must preserve the type, and value information of each expression or declaration across these phases. But there is an advantage to this approach. When IR code generation is done separately from type-checking, the compiler can issue an error message and stop further type-checking and other operations from upcoming phases. This prevents generation of intermediate code that will eventually be discarded.

As we see, there are distinct trade-offs between these two approaches to implementation of the semantic analyzer. A technique that aids us in overcoming the associated trade-offs is lazy evaluation of the state dependencies. Lazy evaluation allows compiler writers to focus on top-level logical structures of the compiler by avoiding the need to focus on partitioning them into separate passes[20].

On a different note, lazy calculation of state dependencies is beneficial from the

perspective of parallelization opportunities as well. In a strict compilation sequence, phase invocations on input programs occur in a fixed manner. Any subsequent phase in compilation cannot be invoked on parts of AST unless all compilation units have passed through the current phase. This is true even if the previous phase has completed its processing on certain compilation units. As a result, computing resources, such as processor and memory cycles, remain idle, for lack of processing tasks. Essentially, the processors must wait until the previous stage of compilation reaches a synchronization point, where the entire compilation units are finished processing through the phase in execution. This is shown in Figure 3.1.

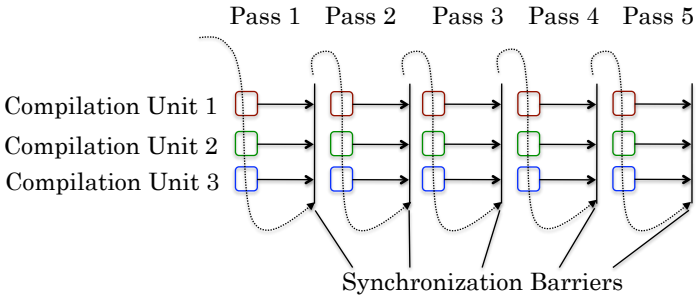

Figure 3.1: Current parallelization in compilation sequence.

Such a limitation is minimized when phase invocations occur in a lazy manner. Under lazy scheme, phase invocations on compilation units and their constituent nodes occur in a demand-driven fashion. This allows one compilation unit to proceed to next phase of processing, without having to wait for completion of processing of other compilation units. Noticeably, sometimes a compilation unit might have to wait for completion of other compilation-unit processing at some synchronization point, owing to its dependency upon other compilation units. Nevertheless, there is now increased opportunity for parallelization of phase processing because different cores or processors can work upon different compilation units, or even their member nodes. This is shown in Figure 3.2.

Using aspect-orientation to dovetail lazy evaluation of type checking with different phases helps modularize the implementation of this lazy state-calculation, which would otherwise have been scattered and tangled across different methods in the AST nodes.

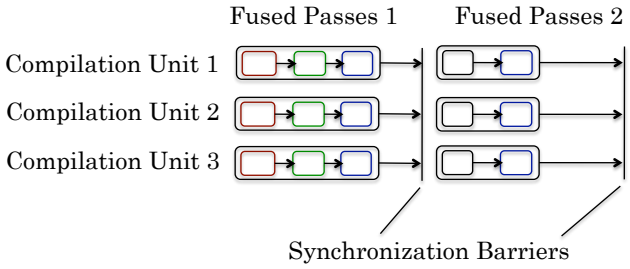

Figure 3.2: Desired parallelization in compilation sequence.

3.2.2 Peephole Optimization

Our second example of novel candidate for modularization is peephole optimization.

An important phase in compiler construction is optimization. A commonly-used optimization technique is *peephole optimization*(PPO)[1, 52]. It involves iterating over a small set of generated instructions, called the *window* or *peephole*, to identify redundancies and eliminate or merge them with a semantically-equivalent instructions.

Usually, PPO is done on machine-specific code. One downside of this is loss of portability. Less commonly, peephole optimization can be applied to intermediate code as well. However, such an implementation entails difficulty in modularizing this concern as a pluggable construct that can be integrated with phases before and after optimization. Listing 3.7 shows a pseudo aspect to carry out PPO. We defer discussion of real code until next chapter.

```
aspect DemoOptimizer {
    pointcut ppoOptimization (Exp AST):
          within (Compiler.*) &&
          call (*.iCodeGen (AST))  &&
          args (AST);

     optimIRCode around (Exp AST): ppoOptimization {
               Code IRCode = iCodeGen (AST);
               return (PeepHoleOptimizer (IRCode));
          }
}
```

Listing 3.7: Pseudo aspect for PPO.

In this example, we see that the pointcut `ppoOptimization(Exp AST)` matches the join points that involve any call to `iCodeGen()` method that takes an argument of type **Exp**, and is within the lexical scope of any class in the **Compiler** package. Here, `iCodeGen()` method refers to intermediate-code generation phase of compiler

construction. The advice code specifies that PPO behavior must be woven at the matching join point specified by the pointcut. The AST required for intermediate code generation and optimization is captured by the `args(AST)` pointcut.

If one wished to perform PPO on the machine code, the only predicate in **DemoOptimizer** aspect that needs updates is line 4. This involves replacing the pointcut `call(*.iCodeGen(AST))` with a new pointcut `call(*.machineCodeGen(IRTree))` describing our new places of interest.

Similarly, if the underlying implementation of the PPO is modified, this aspect would still continue to work; now it will weave additional behavior at the new places. This is yet another benefit of using aspects. If the language developer wished to analyze the compiler with or without PPO, he can do so by simply enabling or disabling this aspect, because it does not depend on any other implementations within the compiler, nor are any modules dependent on this aspect.

As a result, we see how aspects and the aspect components will fit nicely with the conceptual needs of the compilers. This validates aspect-oriented language as the right choice to meet our goals in restructuring compilers. We will more completely elaborate this example in chapter 4, where we provide details regarding implementation, evaluation and utility of a peephole optimizer, modularized as an aspect.

3.2.3 Separation of BCEL from Weaver

Our third candidate is using aspect-orientation to separate the planning and the use of Byte Code Engineering Library (BCEL)[39] in the back end of ajc.

Currently, ajc is built against its own optimized version of BCEL, with some bug-fixes and upgrades to support Java 5 and later versions. AspectJ's weaver has an interface layer into which BCEL implementations are plugged. This interface is rife with several BCEL dependencies. This precludes using the available weaver interface to investigate byte-code manipulation and related opportunities using alternate byte-code manipulation tools.

As it stands now, BCEL policies are entangled with their actual use on the weaver side, which results in an unclear separation between the BCEL and the weaver. In light of this problem, our goal of modularization would be to make the boundary

between BCEL and the weaver clearer and obtain a more reusable architecture by separating out the BCEL policies from usages.

A preliminary design analysis, however, shows that this problem is mainly the result of poor design and implementation. The interface could be clarified through simple refactorings, as described in Fowler [28]. Implementation of this concern would be a contribution to the user community, because it provides opportunities to use alternate byte code engineering tools, such as ASM[8]. From the perspective of research contribution however, this task lacks novelty. Hence, we decided against implementing this aspect candidate. Other factors contributing to this decision are summarized in Table 3.1, in our discussion of Candidate Selection, in section 3.3.

3.2.4 Error Handling

Our fourth, and final novel candidate for aspect-oriented modularization is error-handling, the process of defining how a compiler should behave when it encounters errors in an input program.

Currently, error-handling concern is tightly coupled to many different phases of a compiler, and is interleaved with compiler's underlying program logic in a complex manner. For example, error handling is integral to type and value evaluation, symbol table management, pretty printing, and other similar operations. A common approach to realizing these operations in object-oriented domain is as separate methods in different kinds of nodes. Typical examples of nodes are arithmetic-expression nodes such as sum and product nodes, initializer-expressions nodes such as variable and class initializers, and declaration nodes such as field and method declarations. Thus, error-handling implementation is scattered across different units of modularity (methods and classes) and tangled with other compiler operations.

Further, error-related control dependencies are not explicitly defined and localized into distinct modular units. Information regarding the program-execution path, source-code locations, and significant values about error-raising and -handling entities are non-intuitive: they are hidden within the static and dynamic compiler-implementation structure.

In this thesis, we consider and undertake the task of improving this error-handling

structure in compilers as another example of aspect structure of compilers. To meet this goal, we modularize many error handlers discovered in different phases of ajc into distinct units. This facilitates comprehension, modification and improvement of error-reporting and -patching processes. Details regarding implementation, evaluation and utility of modular error handling follow in chapter 5.

3.3 Candidate Selection

A clear and focused research candidate helps us develop solutions with appropriate designs, analyses and evaluation techniques. Therefore, before proceeding on towards the implementation details, we briefly describe the criteria we applied while selecting our candidates for implementation. We will survey four different novel operations and then use them to provide insight into why some of our choices have been good, and why others may not be reconsidered.

Table 3.1 summarizes assessment of four different novel candidates for implementation given in section 3.2 based on these criteria[5].

Table 3.1: Ranking of candidates based on different criteria.

Criterion	Candidates			
	Separation of BCEL	Lazy Evaluation	PPO	Error Handling
Novelty	-	✓	✓	✓
Interest	✓	✓	✓	✓
Time Feasibility	✓	-	✓	✓
Research Contribution	-	✓	✓	✓
User Contribution	✓	✓	✓	✓
Total Weightage	✓ (3)	✓ (4)	✓ (5)	✓ (5)

The first criterion – *novelty* – identifies the newness of an idea. An idea is considered new if it is has not yet been proposed, or has not been implemented even if it has already been proposed. The second criterion – *interest* – refers to our personal interest on implementation of an idea. The third criterion – *time feasibility* – refers to how feasible the implementation of a candidate will be within the given time frame for a master's thesis. Similarly, the fourth criterion – *research contribution*

[5]These are borrowed from FINER criteria defined by Thabane and Ye [66]. We modified one criterion – *ethics*– because it is not as important for computer science as for anesthesiology. Further, we added one more criterion - User contribution, to suit our needs.

– identifies if a candidate-idea contributes to the existing research in related field by solving a problem, or by improving certain features. The final criterion – *user contribution* – refers to the usefulness of the idea to the general user-community.

From a pilot implementation, we found that separating planning and use of BCEL lacked novelty from research perspective, despite being novel from user perspective. It is clear from talks in the user and development community that this work would prove significant for the user community by providing increased opportunities for testing alternate byte-code manipulation tools besides BCEL. It does not, however, involve fundamentally challenging avenues for research in modularization. Hence, we dropped this candidate despite being interested in it.

We decided against implementation of lazy evaluation of state dependencies for reasons of greater time-requirements, and the need for more comprehensive analysis of ajc.

The remaining two candidates, peephole optimization and error handling, ranked well in all our decision criteria. Hence, we decided to implement them to support our thesis about aspect structure of compilers.

3.4 Summary

In this chapter, we examined seven different compiler operations whose implementations are scattered and tangled across various units of modularity. We also estimated that their aspect-oriented implementation would serve to improve the modular structure of compilers. Further, by using well-founded decision criteria[66], we performed a strategic assessment of these clinical modularization opportunities, which helped us select research candidates for demonstrating the modularization possible in compilers.

In the next two chapters, we examine our two candidates, peephole optimization and error handling, in detail, and show the benefits of aspect-oriented compiler design.

Chapter 4

Peephole Optimization

Modern compilers perform optimizations to improve the execution-time and -space requirements of generated code. In this chapter, we discuss from modularity perspective, the implementation of a common optimizer, a *peephole optimizer*[1, 52, 53, 65] at the bytecode level. This provides an example of separating an entire compiler-pass into a distinct pluggable aspect.

We begin this chapter with an introduction to peephole optimization, followed by our design goals and contributions from its implementation. Next, we describe the architecture of intermediate code in ajc, and also discuss its code-generation strategy. Then we describe in detail a table-driven peephole optimizer that improves this intermediate code. Next, we present performance results for our optimizer. Finally, we describe the related works and close with a summary of the chapter.

4.1 Introduction

Code generated by compilers is rarely in optimal form. Even if the code generated for each source statement is optimal in isolation, it is likely that the optimal fragments become suboptimal when juxtaposed, for example when creating a method body. The output code is, therefore, amenable to further improvements so that it executes faster, or takes less space, or both. This improvement is achieved by means of program transformations called *optimizations*. Compilers that employ such optimizing transformations are called *optimizing compilers*[2, 32, 64].

In fact, separating optimization into a distinct module makes writing a compiler easier: one can write a naïve code-generator and leave the optimization of the poor-quality code-fragments for the later phase. This simplifies the code-generator since it requires little or no context, without discarding optimizations that depend on only

a small amount of context. Unfortunately, in modern compilers, optimization is an integrated phase. The focus of this chapter is on modularizing compiler passes as pluggable constructs, using peephole optimization as a working example.

4.1.1 Java Virtual Machine Basics

Before investigating peephole optimization, motivation for its implementation, and related design goals, we present some background information on the Java Virtual Machine (JVM) that is fundamental to understanding the further discussions.

All Java and AspectJ programs are compiled into class files that contain bytecode, the machine language of the JVM. Bytecode can be executed by interpretation, Just-In-Time (JIT) compilation, or other technique chosen by the designer of a particular JVM. Each instruction consists of a one-byte *opcode* followed by zero or more *operand*s. The opcode indicates the action to take, while any other information required for bytecode execution is encoded in the operands following the opcode.

All computation in the JVM centers on the stack. Because the JVM has no registers for storing values, everything must be pushed onto the stack before it can be used during computation. Bytecode instructions, therefore, operate primarily on the stack. Table 4.1 shows a list of bytecode mnemonics that appear in the discussion of this thesis.

For each mnemonic-opcode shown, we describe the operation that it represents, and also show how the operand stack changes as result of execution of the operation indicated. For instance, consider the first opcode `SWAP`. It represents the operation of swapping the top two values on the stack. Before execution of this instruction, stack contained *value1* at the top and *value2* immediately below it. After the bytecode execution, the order of operands in the stack has changed, and the resulting new order is shown after the $\Longrightarrow$ symbol: *value2* is at the top and *value1* immediately below it.

4.1.2 Peephole Optimization

Given this understanding of bytecode form, the manner in which the JVM handles bytecodes, and the mnemonics used to represent different bytecode instructions, we

Table 4.1: JVM Instructions.

Mnemonic	Operation	Operand Stack		
SWAP	Swap the top two operand stack values	..., *value2, value1*	⟹	..., *value1, value2*
DUP	Duplicate the top operand stack value	..., *value*	⟹	..., *value , value*
ILOAD A	Load an int from local variable A onto stack	...	⟹	..., *value*
POP	Pop the top operand stack value	..., *value*	⟹	...
ISTORE A	Store an int from stack into local variable A	..., *value*	⟹	...
IRETURN	Return int from method	..., *value*	⟹	*[empty]*
INEG	Negate int	..., *value*	⟹	..., *result*
IADD	Add int	..., *value1, value2*	⟹	..., *result*
ISUB	Subtract int	..., *value1, value2*	⟹	..., *result*
LDC	Load a constant onto stack	...	⟹	..., *value*
IF_ICMPEQ	Branch if int comparison value1 == value2 true	..., *value1, value2*	⟹	...
IF_ICMPNE	Branch if int comparison value1 ! = value2 true	..., *value1, value2*	⟹	...
IF_ICMPLT	Branch if int comparison value1 < value2 true	..., *value1, value2*	⟹	...
IF_ICMPGE	Branch if int comparison value1 >= value2 true	..., *value1, value2*	⟹	...
IF_ICMPGT	Branch if int comparison value1 > value2 true	..., *value1, value2*	⟹	...
DUP_X1	Duplicate the top operand stack value and insert two values down	..., *value2, value1*	⟹	..., *value1 , value2, value1*

proceed to look at peephole optimization in some depth. Peephole optimization (PPO) is a simple but effective technique for locally improving the generated code. To improve the performance of the program, a peephole optimizer[65]:

- examines sequences of instructions in a small window of code, called the *peephole*, and
- replaces inefficient sequences of instructions with more efficient but semantically equivalent sequences.

It applies a set of pattern-matching rules to the code, in order to identify inefficient sequences of instructions. For instance, consider the sample replacement-suite shown in Table 4.2.

The two consecutive SWAP instructions, shown in line (1), simply swap the top two words of the stack twice. Essentially, this leaves the contents of the stack unchanged. A SWAP followed by a SWAP can, therefore, be safely eliminated. Accordingly, the corresponding optimized replacement for this pattern is empty.

The second example shows that duplicating the top word of the stack onto the top of the stack, and then swapping their contents can be optimized by simply

Table 4.2: Sample replacement suite for peephole patterns.

Pattern			Replacement	
(1) SWAP	SWAP	$\Longrightarrow$	[*empty*]	
(2) DUP	SWAP	$\Longrightarrow$	DUP	
(3) INEG	IADD	$\Longrightarrow$	ISUB	
(4) STORE A	STORE A	$\Longrightarrow$	POP	STORE A
(5) ICONST 0	IADD	$\Longrightarrow$	[*empty*]	
(6) ILOAD A	ISTORE A	$\Longrightarrow$	[*empty*]	

getting rid of the SWAP instruction. Essentially, a SWAP following a DUP is unnecessary, for it means swapping two equal values. Accordingly, the corresponding optimized replacement for this pattern contains only the DUP instruction.

The third example shows that negating the sign of the integer on the top of the stack followed by an addition operation can be replaced with a subtraction operation. Here, it is worth noting that we are replacing a set of instructions with a single instruction that is less costly.

Similarly, the fourth pattern shows that a sequence of STORE – STORE to the same location, A in this case, can be replaced with a sequence of POP and STORE instructions.

For brevity, we do not delve into the details of other patterns and their optimized replacements. At this point, this information suffices for an understanding of the way in which peephole optimization is performed. The approach is to take a list of instructions and slide through the window consisting of a small number of instructions until the end of the list and replace them with instructions that lead to improved performance in terms of time and space.

As simple as it is, peephole optimization is effective too. As instructions are replaced in the peephole window, new opportunities for further optimizations arise. For instance, consider the following sequence of bytecodes. For instance, consider the following bytecode sequence for the statement: `A = A + 0;`

The first column shows the original sequence of bytecodes. The instructions (2) and (3) push a constant 0 to the stack and then add it to the top of the stack. This computation does not change the value of the stack, hence, the instruction pair (2) and (3) can be safely eliminated during first round of optimization; the resulting

	Original	1^{st} Opt.	2^{nd} Opt.
(1)	`ILOAD A`	`ILOAD A`	[*empty*]
(2)	`ICONST 0`	[*empty*]	
(3)	`IADD`	[*empty*]	
(4)	`ISTORE A`	`ISTORE A`	[*empty*]

code is shown in the second column. This code is again amenable to peephole optimization. Load and store to the same location can be safely eliminated during second phase of optimization because it leaves the operand stack unchanged. The resulting bytecode is thus empty, since all the instructions have now been eliminated.

4.2 Motivation and Design Goals

So far, we looked at the basics leading to the discussion of PPO. In this section, we describe our motivation for implementing a peephole optimizer as an illustrative example for demonstrating the aspect structure of compilers. We end this section with a list of design goals pertinent to our implementation, along with the resulting contributions.

An important question that arises regarding PPO is about the phase where it should be incorporated. Conventionally, it is carried out at the machine-code level because it provides better opportunities for machine-dependent optimizations. Another possibility is to use it at the intermediate-code level. Because intermediate code may not represent all machine-specific details, it offers fewer opportunities for optimization. For instance, some machines have special-purpose instructions, such as `INC`[1], which are cheaper than their more general counterparts like `ADD`. Optimization opportunities dependent on such machine-specific details are lost when it is performed at the intermediate-code level, unless these details are also encoded in the intermediate code.

We perform this optimization at intermediate code when compiling AspectJ source to machine code. This decision is motivated by three major findings:

[1] `INC` is an assembly instruction that increments the contents of the source register. It is considered to be faster than `ADD`, which adds the contents of a source register to that of the destination and stores it in the destination register.

1. First, our candidate compiler, ajc, generates intermediate code at many different places across several modules. To improve the performance of generated-code, ajc applies some pattern-replacements, which are locally hardwired into the places of interest within the compiler source. Identifying and explicitly expressing such sites by means of AspectJ's join point model serves to make the aspect structure of the compiler more explicit. This work will emphasize locality in design of compilers by more explicitly encapsulating PPO-concern as a separate pluggable module. The resulting benefits are flexibility in testing and configuring optimization without the need for hacking its internals.

2. Second, decreased optimization opportunities are not a big issue in the context of ajc, because the intermediate code of ajc is the bytecode. The benefit comes from opportunities to use semantic knowledge pertinent to bytecodes, such as instruction equivalence, for improving the intermediate code. Consequently, optimization can be applied to even special-purpose instructions in the intermediate code, whose bytecode equivalents contain several components, because these components are visible in the intermediate representation as well.

3. Third and final reason is the desire to emit compact blocks of bytecode. Currently, the AspectJ compiler throws an exception if a method becomes too large during weaving, and does not fit into the maximum method code size, less than 65536 bytes. In fact, this is a well-documented limitation of Java. One way of reducing its occurrence is to pack an optimized version of method body. Peephole optimization at intermediate level is one opportune way to do this.

In light of these problems, the design goals for our implementation of peephole optimization are:

1. **Explicit control-flow information**

   Much legacy code for PPO is written in high-level languages containing sequences, loops, branches, methods, and classes. Sequences describe the ordering in which instructions execute. Loops repeatedly execute a sequence of instructions in their bodies until some conditions are met. Branches describe different instructions to execute depending on their test-conditions. Methods

collect a sequence of related operations into a single abstract higher-level operation. Classes group related operations and their associated attributes into a single, even higher-level unit.

Essentially, each of these constructs abstract some control-flow information. For instance, every instruction in a sequence is control dependent on execution of its preceding instruction[2]. Similarly, every instruction after a branch is control dependent on the branch and must wait for the branch to execute before it can execute.

Although control constructs such as sequences, loops and branches are considered to abstract some kind of control-flow information, others such as methods give great power but can obscure other significant properties of the program, such as thread safety. We believe that explicitly reifying information related to sites, contexts and ordering of method invocations facilitates expressing much higher-level control-flow abstractions than those expressed by sequences, loops, branches and method implementations.

For instance, lets consider a method performing type munging in ajc. An initial investigation of the ajc source shows that there is a call to `munge(...)` method in the body of `weaveInterTypeDeclarations(...)` method, which is a member of `AjLookupEnvironment` type. However, it is not clear whether this is the only source file containing code related to type munging. Further navigation and inspection of ajc reveals that the `munge(...)` method that deals with type munging is invoked in eight different source files scattered across two different packages namely `org.aspectj.ajdt.internal.compiler.lookup` and `org.aspectj.weaver.bcel`. It would be beneficial for compiler developers if we could explicitly define:

- the control-flow in which the `munge(...)` method is invoked. This would facilitate reasoning about the overall type-munging action. Examples of control-flow information related to munging are munging in the control flow of processing of type mungers, addition of inter-type declarations, addition

[2]This is not entirely true, however. Sequence of instructions that do not have side-effects and are not dependent on each other can be executed in parallel.

of aspects, and determination of super and sub types.

- the types which contain code related to the `munge(...)` method. Such information would be something of following nature: types dealing with munging are `AjLookupEnvironment`, `EclipseTypeMunger`, `BcelAccessForInlineMunger`, `BcelCflowCounterFieldAdder`, `BcelClassWeaver`, `BcelPerClauseAspectAdder`, and `BcelTypeMunger`.
- the packages that deal with munging. Examples include name of the packages, as we saw earlier, containing the appropriate types.

Such information would allow compiler writers focus on only a smaller section of the compiler, and make it easier to locate the places of interest. The benefit of such information portrayal arises from reduced time and effort involved in understanding the compiler implementation for maintenance or extension. This has led us to aim for explicitly defining these higher-level control-flow structure in explicating the code.

2. **Pluggable implementation**

   Currently, peephole optimizers are implemented as core compiler code. This complicates several facilities: configuration of the compiler to operate with or without the optimizer, and assessments of its performance impacts. This has led us to consider as design goal the benefit of seamless integration of the optimizer into the compiler for supporting easier customization and testing.

With these design goals in mind, our specific contributions resulting from modularized implementation of the optimizer are:

- identification of points in the static program structure and dynamic execution graph of the ajc compiler that are affected by our optimizer,
- an example optimizer that can be enabled or disabled in a pluggable manner, and
- improvements in the performance of the code generated by ajc.

The next two sections prepare us for understanding the optimizer, its integration into ajc, and its pluggable nature.

4.3 Architecture of Intermediate code in AJC

The intermediate code generated by the front end of ajc and accepted by the back end for bytecode generation is the `Instruction`-based language. It is a stack-based representation that resembles JVM instructions. The `Instruction`s exactly match the mnemonics of bytecodes. They are modeled as objects, which enables programmers to obtain a high-level view upon control flow without handling details like concrete bytecode offsets. `Instruction`s consist of an opcode (sometimes called the tag), a length in bytes and an offset (or index) within the bytecode. In addition to emitting bytecodes, bytecode input-streams can be read in and converted to `Instruction` types.

`Instructions` are wrapped into `InstructionHandle`s, objects that are returned from append and insert operations. A list of `Instruction`s, called an `InstructionList`, is implemented as a list of `InstructionHandle`s, which mediate read-only traversal of the list. It should be noted that, `Instruction` is a misnomer in ajc – references to `Instruction`s in the list are not implemented by `Instruction`s but by `InstructionHandle`s. In ajc, bytecode instruction equivalents are `InstructionHandle`s. This makes appending, inserting and deleting areas of code easier and also allows us to reuse immutable instruction objects.

Since this form uses symbolic references, computation of concrete bytecode-offsets does not need to occur until the compiler has finished the process of generating or transforming code. When an `InstructionList` is ready to be emitted as pure bytecode, all symbolic references are mapped to actual bytecode offsets. The intermediate form provides a method, `getByteCode()`, to do this easily.

4.4 Code Generation Strategy in AJC

Ajc follows a linear statement-by-statement code-generation strategy. It generates `Instruction`s and `InstructionList`s corresponding to field initializers, methods, static class initializers, advices, aspects and other similar constructs. Further, it does the same for parts of different actions, such as initializations, method inlinings, bridge-method creations, method dispatches, load-time and compile-time weaving,

and managing `thisJoinPoint` arguments.

Although code generation in ajc occurs in several different places, it can not perform any sophisticated optimizations, such as whole-program analysis, to improve code generated at those sites. This is because it supports incremental weaving. It, however, does some simple optimizations, such as removal of `NOP` instructions, and instruction-pattern replacements. Even these implementation are local to the places where the optimized behavior are desired. Therefore, their implementation is scattered across different methods in different classes

Now that we have familiarized ourselves with the form of intermediate code in ajc, and discussed how it is generated, we will describe the actual implementation details in the next section.

4.5 Implementation

Our design goal is to yield a pluggable model of peephole optimization. The first step involved is identification of control points dealing with the generation and use of `Instruction`s and `InstructionList`s. Once we capture the sites of interest, we can specialize their behavior for optimization. In this section, we identify the related join points in ajc and the advices with regard to the optimising effect they express.

4.5.1 Development Aspects

We used two development aspects in order to facilitate our task of identifying potential join points in ajc, where we can specify peephole optimization. Listing 4.1 shows one such aspect. The other is given in Appendix A.

This aspect is declared private so that it has access to all members including private or protected resources of other types. Let's examine the aspect in more detail:

- The first pointcut `includeScope()` acts as a guard for other pointcuts. It serves to confine our search to only those types which are within the `bcel` package.
- The second pointcut `accessSites()` captures any access or assignment of values of types `Instruction` and `InstructionList`. It is composed of `get` and `set`

```
/**
 * Aspect to identify places where instructions and instruction lists are
 * accessed, or initialized.
 */
public privileged aspect SeekOptimizationSites {
    /**
     * Limit the scope of our search to a package
     */
    pointcut includeScope():
                within(org.aspectj.weaver.bcel..*);

    /**
     * Places where instructions and instructionlists
     * are accessed or initialized
     */
    pointcut accessSites():
                get (InstructionList *)
                || set (InstructionList *)
                || get (Instruction *)
                || set (Instruction *);

    /**
     * Exclude join points in test suites
     */
   pointcut excludeScope():
                !withincode(* *.suite(..));

    /**
     * Composition of all pointcuts of interest
     */
    pointcut potentialSites1():
                includeScope()
                && excludeScope()
                && accessSites();

    /**
     * Simple advice to examine the affected places
     */
    after():
                potentialSites1(){
                    System.out.println("Potential site for peephole optimization");
                }
}
```

Listing 4.1: A development aspect.

pointcuts.

- The third pointcut `creationSites()` captures places where new instances of `Instruction` or `InstructionList` are created. The constituent pointcuts refer to constructor calls for each type.

- The fourth pointcut `excludeTestSuite()` excludes consideration of any matching join points that occur in the lexical scope of `suite()` methods, which are the test methods.

- The fifth pointcut `potentialSites()` combines the `accessSites()` and `creationSites()` to capture all places where `Instructions` or `InstructionList` types are accessed for their values, assigned some vaues, or instantiated. The `includeScope()` pointcut limits the join points to within the `bcel` package.

Similarly, the `excludeTestSuite()` pointcut precludes matching join points within test cases.

- The `after` advice applied to the `potentialSites()` pointcut simply prints a message after the matching join points.

By using AJDT, we can now limit our investigation of the ajc compiler to these places. Obviously, this aspect does not represent all sites of potential interest to us. The other development aspect is shown in Appendix A.

4.5.2 Optimization Aspects

In this section, we provide three examples of optimization aspects, along with a description of different sets of pointcuts and advices inserting optimization behavior into the compiler at those points. In some cases, we show abbreviated advices; their details are provided in Appendix C.

Optimizing inlined instructions

The first example is about specializing method bodies that will be inlined with peephole optimization. Let's examine the related pointcuts and advices in Listing 4.2.

The first pointcut `callToGenInlineInstructions()` captures calls to `genInlineInstructions()` method of the `BcelClassWeaver` class. This method takes `InstructionList` of the method to be inlined, and inlines them to the recipient method. Carrying out peephole optimization on this list would reduce the size of instructions that will be inlined.

The second pointcut `lexicalScopeAddAjcInitializers()` identifies join points within the lexical scope of`addAjcInitializers()` method within `LazyClassGen` class.

The third pointcut `callToinitializeAllTjpsWithinLazyClassGenAddAjcInitializers()` captures calls to method `initializeAllTjps()` of the `LazyClassGen` class. Within the `addAjcInitializers()` method of this class, there is a call to `getStaticInitializer().getBody().insert(il)`, which inserts a list of instructions for static initialization of classes. Here, we aim to optimize the list of in-

```
/**
 * Pointcuts identifying inlined instructions
 * And, advice applying peephole optimization to them
 */
pointcut callToGenInlineInstructions():
            call (* BcelClassWeaver.genInlineInstructions(..))
         && !within(PPOptimizer);

pointcut lexicalScopeAddAjcInitializers():
            withincode (* LazyClassGen.addAjcInitializers());

pointcut callToinitializeAllTjpsWithinLazyClassGenAddAjcInitalizers():
            call (* LazyClassGen.initializeAllTjps())
         && lexicalScopeAddAjcInitializers();

pointcut callToGetAdviceInstructions():
            call (* BcelAdvice.getAdviceInstructions(BcelShadow,
                                                     BcelVar,
                                                     InstructionHandle))
         && !within(PPOptimizer);

pointcut methodsReturningInstructionList():
            callToGenInlineInstructions()
         || callToinitializeAllTjpsWithinLazyClassGenAddAjcInitalizers()
         || callToGetAdviceInstructions();

Object around(): methodsReturningInstructionList() {
    return (optimizeInstList ((InstructionList) proceed()));
}
```

Listing 4.2: Example-1: Peephole optimization of inlined instructions.

structions, `il`, responsible for static initialization of classes; so, we capture the calls to `intializeAllTjps()` that are only within the `addAjcInitializers()` method. We use the second pointcut, `lexicalScopeAddAjcInitializers()` for this purpose. Optimization at this join point would be highly useful when there is a need to create many static initializers as a result of aspect instantiation on a `perObject` basis.

The fourth pointcut `methodsReturningInstructionList` combines all of the above three pointcuts so that we can refer to all the join points with a single identifier.

Having identified this set of join points for PPO, we now present an advice to incorporate peephole optimization. Listing 4.2 shows an `around` advice applied to the `methodsReturningInstructionList` pointcut. This advice captures the list of instructions returned, passes them to the peephole optimizer, i.e., `optimizeInstList()` and returns the optimized list of instructions. Complete implementation details of the `optimizeInstList()` method are shown in Appendix A. This advice has crosscutting effect across 19 different locations in ajc; but, its implementation appears just in this one place.

Optimizing packed method-bodies

This example is about optimization of instructions generated as part of method-body creation. Related code for our second example is shown in Listing 4.3

```
/**
 * Pointcuts identifying packed method bodies
 * And, advice applying peephole optimization to them
 */
pointcut callToOptimizedPackBody(MethodGen mgen ):
            (call (public void LazyMethodGen.optimizedPackBody(MethodGen))
            || call (public void LazyMethodGen.packBody(MethodGen)))
        && args (mgen);

Object around(MethodGen mgen): callToOptimizedPackBody(mgen){
    mgen.setInstructionList(optimizeInstList(((LazyMethodGen)
                                        (thisJoinPoint.getTarget())).getBody()));
    return proceed(mgen);
}
```

Listing 4.3: Example-2: Peephole optimization of packed method-bodies.

The pointcut `callToOptimizedPackBody(MethodGen mgen)` captures calls to methods that pack a list of instructions into a method body. Here, we intercept the calls to these methods and capture the list of instructions that are to be packed into the method body, run peephole optimizer on this list and then pack the method with this optimized list of instructions as body. The advantage of doing so is that we would have a method-body with a reduced size. It is because our peephole optimization pattern always results in a replacement that has the same or fewer number of instructions, but not more. We will look at these patterns in subsection 4.5.3

The `around` advice shown in Listing 4.3 captures the list of instructions of the target object where the specified join points match, then performs peephole optimization on the list and finally, fills the newly created instance (in case of `packBody()`) or a local copy (in case of `optimizedPackBody()`) of type `MethodGen` with this optimized list by means of `proceed` advice. The effect of this advice is over 6 different places.

Removing `NOP` instructions

Now, we consider our third and final example shown in Listing 4.4.

The pointcut `methodBodyCapture(Object)` captures executions of methods `run()` or `pack()` within the `LazyMethodGen` class. We run the peephole optimizer on the method body before it is actually packed into the `LazyMethod`. We do the same

```
/**
 * Pointcuts identifying places with NOP
 * And, advice applying peephole optimization to them
 */
pointcut methodBodyCapture(Object callee):
            (execution ( *void LazyMethodGen.BodyPrinter.run())
            || execution (public MethodGen LazyMethodGen.pack()))
         && target(callee)
         && if(callee != null);

before(Object callee): methodBodyCapture(callee){
    try {
        ((LazyMethodGen)callee).stripNops();
    } catch (ClassCastException cce) {
        cce.printStackTrace();
    } catch (NullPointerException npe){
        npe.printStackTrace();
    }
}
```

Listing 4.4: Example-3: Peephole optimization to remove `NOP` instructions.

before execution of the method `run()`.

The `before` advice in Listing 4.4 advises the join points captured by the pointcut `methodBodyCapture(Object)` to remove `NOP` instructions from the body of the `LazyMethodGen` types. The target types on which this optimization is to be carried out are captured as a pointcut context. Hence, `NOP` instructions will be removed from the list of instructions (i.e., body) of `LazyMethodGen` types before any method body is packed.

It is worth noting here that, by encapsulating these pointcuts and and advices, aspects facilitate the design of software with a clear separation of concerns. Our concern, peephole optimization, is clearly modularized in an aspect and can be easily hooked into the base compiler. In addition, owing to its clean modular structure and stand alone nature, it is possible to assess the performance of the compiler in the presence and absence of this optimization. Furthermore, the control flow dependencies relating to this optimization have become explicit through the pointcut and advice specifications.

4.5.3 Optimization Patterns

So far, we investigated the sites in the ajc compiler where we can apply PPO. In this section, we will describe several optimizing transformations, among those we have we implemented.

1. Unnecessary-code elimination

Consider the instruction sequence:

(1) `ILOAD A`

(2) `ISTORE A`

This sequence of load and store to the same location can be safely eliminated because executing these operations in sequence leaves the stack unchanged. However, note that if instruction (2) had a label, we could not be sure that (1) was always executed immediately before (2) and so we could not remove the instruction pair.

2. Instruction-strength reduction

   Consider the instruction sequence:

   (1) `ISTORE A`

   (2) `ISTORE A`

   Lets consider that the stack contained three different values before execution of these instructions: *value1* at the top, *value2* immediately below it, and *value3* below *value3*. When the first `ISTORE A` instruction executes, it removes *value1* from stack and stores it in the local variable A. Similarly, the second instruction retrieves *value2*, which is now at the top of the stack, and stores it in variable A, replacing the old value. As we see, the first store to the variable becomes useless because of the second `STORE` instruction. This sequence of store and store to the same location can, therefore, be safely replaced with the following sequence:

   (1) `POP`

   (2) `ISTORE A`

   The new instruction pair is more efficient than the previous one, because `POP` is cheaper than `ISTORE`, as it does not have to store the retrieved value.

Table 4.3 shows a representative list of other replacement patterns and their optimizations that we have implemented.

Our example patterns are not novel, but comprise a number of typical peephole optimizations, in order to exercise the novel modularity that aspects let us express.

Table 4.3: Replacement patterns and peephole optimizations.

Pattern			Replacement	
SWAP	SWAP	$\Longrightarrow$	[*empty*]	
DUP	SWAP	$\Longrightarrow$	DUP	
ILOAD A	POP	$\Longrightarrow$	[*empty*]	
DUP	POP	$\Longrightarrow$	[*empty*]	
DUP_X1	POP	$\Longrightarrow$	SWAP	
ILOAD A	ISTORE A	$\Longrightarrow$	[*empty*]	
ISTORE A	IRETURN	$\Longrightarrow$	IRETURN	
INEG	IADD	$\Longrightarrow$	ISUB	
INEG	ISUB	$\Longrightarrow$	IADD	
LDC 0	IF_ICMPEQ	$\Longrightarrow$	IFEQ	
LDC 0	IF_ICMPNE	$\Longrightarrow$	IFNE	
LDC 0	IF_ICMPLT	$\Longrightarrow$	IFLT	
LDC 0	IF_ICMPGE	$\Longrightarrow$	IFGE	
LDC 0	IF_ICMPGT	$\Longrightarrow$	IFGT	
LDC	IF_ACMPNE	$\Longrightarrow$	IFNONULL	
ISTORE A	ISTORE A	$\Longrightarrow$	POP	ISTORE A
ISTORE A	ILOAD A	$\Longrightarrow$	DUP	ISTORE A
ILOAD A	ILOAD A	$\Longrightarrow$	ILOAD A	DUP

4.6 Evaluation

In earlier sections, we described our peephole optimizer, and illustrated with examples how aspect-oriented implementation helps to make the control-flow information explicit, and to obtain a better localized structure of the optimizer in a pluggable manner.

In this section, we state the results of our assessments. The presentation is divided into three parts. In the first two parts, we consider the correctness and performance impacts of our implementation. In the final part, we consider the global question of modularity, and see how our thesis is supported by this work.

4.6.1 Correctness Assessment

In doing optimization, we attempt to be as aggressive as possible in improving code, but never at the expense of making it incorrect. An optimizer is said to be *safe* or *conservative* if it guarantees that it does not make a correct program incorrect. In order to verify this property of our candidate compiler, we ensure that it passed all the existing JUnit tests. Currently, the ajc consists of more than 3700 test cases which verify correctness of different compiler functions at different levels of weaving, from the level of individual functions to end-to-end compilation of large programs.

Some of the JUnit tests had to be modified to account for optimization. These are the ones which checked for the ordering and number of different method calls and output print streams, and the ones which performed comparisons at the level of `Instruction`s and `InstructionList`s.

Further, we also validated our modified compiler against 13 popular test suites, recommended by the developers of ajc. They are distributed along with the ajc.

4.6.2 Performance Assessment

Although it is formally undecidable whether an optimization improves code performance in all situations[56, chap. 11], we assess the performance impacts of our peephole optimizer on a number of commonly used AspectJ specific benchmark suites, and test cases to ensure that it does not significantly slow down the performance, if not improve it.

Table 4.4 summarizes different metrics used for performance evaluation. On the

Table 4.4: Performance assessment of the peephole optimizer.

Subjects	Compilation							Execution
	Time(ms)	Heap Memory(MB)		Non-heap Memory(MB)		Garbage Collection		Time(ms)
		Allocated	Used	Allocated	Used	Number	Time	
tracelib	31.25 (+1.1)3.4%	3.2 (+0.0)0%	1.6 (+0.3)18.75%	11 (+0.2)18%	11 (+0.0)0%	85 (+0)	5% (+0%)	16.32 (-1.0)6%
tracev1	28.75 (+1.9)6.6%	2.6 (+0.4)15%	2.2 (+0.4)18%	11 (+0.2)18%	11 (+0.0)0%	40 (+0)	3% (+0%)	14.69 (-0.9)6.12%
tracev2	2.75 (+0.2)7.2%	2.3 (+0.4)17%	1.8 (+0.4)22%	9.6 (+1.4)14.5%	9.5 (+0.9)9.47%	57 (+6)	5% (+0.7%)	8.04 (-0.5)6.21%
tracev3	3.00 (+0.2)6.7%	2.4 (+0.2)8.3%	1.7 (+0.2)11%	9.8 (+1.4)14.2%	9.8 (+0.8)8%	65 (+11)	5% (+ 0.8%)	10.5 (-0.2)2%
tjp	16.25 (+0.8)4.9%	3.0 (+0.0)0%	1.9 (+0.1)5%	11 (+2)18%	11 (+0.0)0%	88 (+0)	5% (+0%)	13.74 (-1.0)7.3%
basic	3.00 (+0.1)3.3%	2.3 (+0.4)17%	1.9 (+0.4)21%	9.8 (+1.8)18%	9.6 (+0.1)10%	69 (+6)	5% (+0.7%)	6.23 (-0.1)1.6%
billing	2.80 (+0.2)7.1%	2.3 (+0.0)0%	1.4 (+0.2)14%	9.6 (+1.6)16%	9.5 (+0.5)5%	58 (+7)	4% (+1%)	8.9 (-0.8)8.9%
timing	3.00 (+0.3)10%	2.4 (+0.3)12.5%	1.6 (+0.3)18%	9.8 (+1.8)18%	9.6 (+0.1)10%	70 (+3)	5% (+0.8%)	12.8 (-1.4)10.9%
spacewar	2.93 (+0.2)6.8%	2.4 (+0.1)4.1%	1.9 (+0.3)15%	9.9 (+1.3)13%	9.6 (+0.0)0%	70 (+0)	5% (+0%)	7.3 (-0.7)9.5%
introduction	3.06 (+0.2)6.5%	2.4 (+0.1)4.1%	1.9 (+0.3)15%	11 (+2.0)18%	11 (+1)9%	84 (+6)	5% (+1%)	9.2 (-0.3)3.2%
coordination	2.53 (+0.2)7.9%	1.9 (+0.0)0%	1.3 (+0.0)0%	9.0 (+0.0)0%	8.9 (+0.0)0%	44 (+0)	1% (+0%)	6.7 (-0.1)1.5%
beam	1.31 (+0.1)7.6%	1.9 (+0.0)0%	1.6 (+0.0)0%	8.7 (+0.3)3.4%	8.2 (+0.1)1.2%	35 (+8)	0% (+1%)	5.9 (-0.0)0%
raceraj	2.00 (+0.6)30%	1.9 (+0.6)31%	1.2 (+0.4)33%	8.7 (+1.0)11%	8.1 (+0.5)6%	35 (+4)	1% (+.9)%	14.9 (-1.2)8.0%
Weighted Avg.	+5.6%	+7.9%	+14.19%	+14.18%	+4.8%	(+4)	+.5%	-6.1%

left are 13 different benchmark applications. These benchmarks were chosen because

they include comprehensive test cases, are easily accessible, well defined, and open source. Across the top are 5 different evaluation criteria for measurement of performance. These are compilation and execution times of the test cases, heap and non-heap memory allocated during compilation, and number of garbage collections and percentage of time spent on them.The tests were performed on Mac OS 10.5.6 running on an Intel core 2 Duo MacBook 2.1 with 3 GB memory, 4MB L2 cache, 2.16 GHz processor speed and 667MHz bus speed.

The entries in the first rows corresponding to each subject show the absolute values of measurements of different evaluation criteria. The entries in the second row for each subject in the table show the differences between the values for the original and the re-factored compiler. These are the mean readings obtained from 10 measurements of each evaluation metric.

These numbers were obtained using the Yourkit Java profiling tool[73]. The positive numbers indicate increased values. Our measurements show that while the compile time performance increases marginally, the run-time performance improves marginally as well. The improvements range from 0 to 10%, which is the result of application of our optimizer. During compilation, time is spent on identifying the matching patterns, on applying new patterns, and on weaving the aspects. Hence, there is increase compilation times for the test cases. As aspects are applied, the compiler creates additional objects, hence, there is an increase in heap and non-heap memory allocation and use. This has led to an increase in the number of garbage collections performed by the compiler, and time spent on garbage collection too.

In summary, there is a modest performance improvement resulting from the implemented optimization. It should, however, be noted that our primary goal for exercising PPO is improved modularity, so any gains in performance simply demonstrate the effectiveness of the new modularity.

4.6.3 Modularity Assessment

Figure 4.1 is a SeeSoft Eick et al. [24] visualization showing modularity of peephole optimizer. Each block in the diagram represents a java source file in the compiler and the length of each block is proportional to the size of the file. The red stripes in the

blocks represent locations in the source files that are involved in optimization. This figure was generated using the standard aspect visualization tool, available with the AspectJ compiler. We found that the effect is pervasive across at least 76 different locations within 17 different methods of 11 classes.

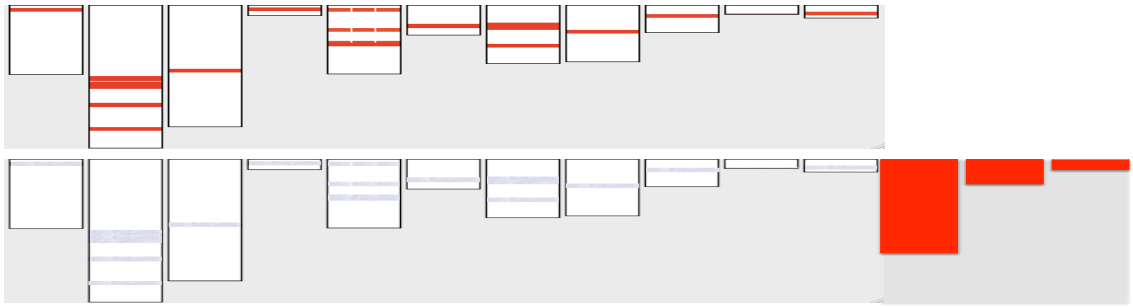

Figure 4.1: Nature of peephole optimization code – Before and after aspect-oriented modularization.

One should note that although the peephole behavior affects several modules, the implementation is well localized into its own distinct module as an aspect. This is indicated by lighter stripes in the lower SeeSoft representation in Figure 4.1; the code is not actually present at those places, but its effect is. The optimizing behavior is encapsulated into separate modules, represented by red-color bars. Further, this behavior is isolated from the rest of the compiler concerns in such a way that one can readily plug in or remove this facility from compilation without the need for any kind of change, either to the compiler or the aspect itself.

Aspect-oriented modularization of peephole optimization is, therefore, well localized and isolated from the rest of the compiler. As we discussed in chapter 2, our aspect-oriented implementation better captures such cross-cutting concerns.

On the basis of these results, we believe peephole optimizing the intermediate code is worthwhile, since the optimizer need only be written once and supports additional, more sophisticated optimizers.

4.7 Related and Future Work

Several prior works describe peephole optimization in compilers[29, 52, 53, 65], but this work is the first one to explore it from modularity perspective, with aspect-orientation.

McKeeman [52] describes how redundant instructions may be discarded during the final stage of compilation by using peephole-optimizing techniques. We employ his techniques in this paper during pattern-matching and their replacements with efficient instruction-sequences.

Another related work is by Tanenbaum et al. [65], which describes peephole optimization of intermediate code, in contrast to conventional approaches that focus on object code or machine code. They describe a table-driven peephole optimizer that improves the performance of a stack-machine-based intermediate code by about 15% over other techniques. Our work also focuses on intermediate code, but is of a different form than theirs. Further, our focus is on improving comprehensibility and reusability, rather than significantly improving performance.

Fraser and Wendt [29] also deal with peephole optimization at the intermediate-code level. This paper describes a compiler with a code generator and machine-directed peephole optimizer that are tightly integrated. Both functions are performed by a single rule-based rewriting system that matches and replaces patterns. They claim that such an organization helps make the compiler simple, fast, and retargetable, and also corrects certain phase-ordering problems. In contrast, our implementation decouples the optimizer from the code generator as well as the entire compiler. We claim and demonstrate that such a decoupling helps to seamlessly integrate or remove the optimizer from the base compiler without the need for any integration-related changes.

Another related work on peephole optimization is by McKenzie [53]. Their paper describes techniques for increasing the throughput of a peephole optimizer for intermediate code. The focus of our work is not on increasing the efficiency of our optimizer in terms of performance, but in terms of explicitness and pluggability.

Existing prior work, as we saw, mostly deal with improving the throughput and efficiency of the optimizer. None of these works consider modularity in implementation as their goal. We rely on the descriptions and results of these papers while implementing the pattern-matching and -replacement rules, but with a primary focus on making the implementation clear and easily configurable.

4.8 Summary

We have described a way to implement peephole optimization for improved modularity. Our mechanism builds on the well-understood principles of compiler peephole optimization, and uses a typical optimizer with minimal changes.

While it is possible to perform peephole transformations using traditional object-oriented style, this would result in scattering of the code across methods within different classes, and tangling with other compiler operations. This is primary the result of method invocations locally from places chosen for peephole optimization. Consequently, this would result in a compiler riddled with optimization-related code in many places. With our approach, the optimizer sits on top of the undisturbed compiler. It can be run immediately by activating the aspect implementing the optimization. Similarly, it can also be deactivated by simply removing the aspect from the compiler build path, or from the load-time weaving process.

The significance of our peephole optimizer is not that it improves the code performance, but that it demonstrates how to improve the state of the art compiler design by isolating semantically independent behaviors from the base compiler, into their own distinct modules. Peephole optimization is a simple and widely incorporated concept, the challenge was to render its implementation equally comprehensible, and maintainable. Our implementation is more comprehensible because we have explicitly specified the sites where pattern replacements occur. We also capture the contexts that are required during replacement of inefficient instruction sequences with more efficient ones. Similarly, our optimizer is more amenable to maintenance because its implementation is free from any unrelated pieces of code dealing with other compiler operations. As a result, there is no need for investigating and modifying other pieces of the compiler, except the one dealing with optimization.

Chapter 5

Error Handling

In this chapter, we discuss aspect-oriented re-factoring of error-handling concerns in compilers to achieve improved modularity. We claim that such modularization provides opportunities for better comprehension and increased reuse of error-handling artifacts compared to conventional approaches to implementation.

We begin with a discussion of error-handling and -recovery concepts, then outline our intended contributions and related design goals, and follow this with an overview of the scattered and tangled nature of current error-handling mechanisms. Next, we present four different examples of our modularization approach. In each example, we also demonstrate how the modularized units can be evolved to implement error-recovery actions. Next, we present results of our measurements assessing the costs and benefits of the implementation from different perspectives. Finally, we describe related prior work and possible future work emanating from this endeavor.

5.1 Introduction

Recall that a compiler can successfully compile only well-formed programs. Quite often, the compiler must deal with incorrect input-programs: those containing a variety of errors arising during any phase of compilation. For instance, it detects misspelled keywords during lexical analysis, ill-formed expressions (missing parenthesis or a semicolon) during syntactic analysis, and application of an operator to incompatible types during semantic analysis.

A compiler that halts on finding a single error does not effectively help users in detecting and correcting the program.

5.1.1 Error Handling

After detecting an error during compilation, most commercial compilers report the location of additional errors in that phase, along with some meaningful explanations. This process is called *error handling*.

Planning error handling from early stages of compiler development can both simplify the structure of compilers and improve their response to errors in the program to be compiled[67]. A good error handler for input program errors should:

- allow compiler designers to quickly identify the source and associated context of errors – in order to emit informative messages to the compiler user
- be amenable to improvements,
- maintain compilation performance for correct programs, and
- attempt recovery from simple errors whenever possible.

This is an ideal view of error handlers. In practice, however, the error-handling concern is not as well implemented as we would like it to be. Its implementation is spread across several modules, and the control-flow of error-raising and -handling sites are not explicitly defined.

Although error handling is commonplace and forms a global design issue, few languages have been designed with error handling in mind [60, 67]. Most programming-language specifications do not describe how a compiler should respond to errors. They only define well-formed programs, but leave the decision about handling of ill-formed programs to compiler designers[71, chap. 9] [67, chap. 4]. As a result, error handling is often compromised to make compiler development easier.

5.1.2 Error Recovery

In the event of errors, most compilers emit error-related information about the input program, report failures and stop execution of other phases in compilation. This behavior can be improved to be more user-friendly. A preferable approach from users' perspective is to detect further errors, and better yet some kind of adjustment to allow the compilation to proceed. Some modern compilers attempt this approach.

They attempt to adjust certain inputs or values at the point of error in a way that allows compilation to proceed further. This process is called *error recovery*.

Error recovery[18] is a four-step process, which consists of:

- detection, which means detecting the presence of an error in the input-program
- diagnosis, which means identifying the location and nature of error, such as lexical and type errors
- reporting, which means providing useful information to the programmer so that he/she can identify and correct them, and
- patching, which means modifying the state of the compiler, so that compilation can proceed

With current compiler implementation techniques, error recovery demands a significant amount of time and cognitive effort. The artifacts related to error-handling[1]– source, context and control flow – are scattered across different program elements and tangled with the implementation of other operations. Moreover, the task of identifying such error-related artifacts needs be repeated every time a new error-reporting or -patching facility is to be introduced.

5.2 Contributions and Design Goals

Error handling is a crucial part of compiler construction. Effective modularization of error-handling concern, however, is a challenging task.

First, the error concern is tightly coupled to different phases of a compiler, and is interleaved with the underlying program logic in a complex manner. Consider the semantic analysis and symbol-table loading phases. The semantic analysis phase synthesizes and maintains type and value environments as part of type-checking and symbol-table loading. As a result, error-handling code related to symbol-table loading and type-checking tend to get tangled with both of these operations. In addition, since the error-handling code is integrated into methods responsible for

[1] We will look at them shortly.

such operations, it leads to tight coupling of error-handling concern with other core compiler-concerns.

Second, error-related control dependencies are not explicitly defined and localized into distinct modular units. In fact, they are handled in an inelegant manner. For instance, some of them are wrapped and thrown as exceptions, while most of them are handled locally within different methods by guarding possible error-conditions with a series of `if-else` statements. As a result, compiler writers must navigate and understand most of the compiler source for identifying these locations to change or evolve existing error-handling in compilers.

In light of these problems of non-modular and hidden structure of error handling in compilers, specific contributions resulting from our modularization are:

1. example identification of points in the static program structure and dynamic execution graph of the compiler that are responsible for error handling in ajc
2. a diverse set of examples for modularization of distinct kinds of scattered error-situations.
3. explicit extension points for error recovery in ajc

Our design plan is to identify and explicitly define error handling artifacts:

- *sites*- locations in the program source code where errors need to be detected and handled,
- *contexts*- values synthesized at error sites, calling or executing types, and argument and return types of error-raising methods, and
- *control flows*- execution paths in the compiler.

Modularization of these artifacts increases the value of the system in terms of comprehensibility, reusability, extendibility of error recovery and estimability of new error-recovery strategies.

5.3 Error Handling in *ajc*

So far, we identified error-handling-related modularity problems in compilers, and then outlined our contributions and related design plans. We will now investigate error-handling scenario in ajc.

Although ajc[2] is a compiler for aspect-oriented programs, it does not incorporate aspects to modularize the error handlers within itself. Hence, there are plenty of opportunities for aspectization of error handling concerns. Further, because of its support for weaving from source code and pre-compiled code, the compiler has to guarantee the correctness of a single functionality at both these levels. Hence, the need and opportunity for handling the same error at different levels in ajc provides a fair level of complexity in modularization of error-handling concerns.

This study primarily targets the weaver end of ajc. The weaver alone generates more than 150 different error messages. References to these errors, and their handlers are scattered across different units of code such as methods, classes, and packages. Examples of errors handled by the weaver end include those related to parsing, argument binding, type resolution, missing types, compilation environment, and weaver states.

Any change in an error handling policy, thus, requires a variety of changes, possibly touching many components of the compiler, including the front-end scanner and parser, the type checker, the matcher and weaver; and, potentially requiring relatively sophisticated program analysis to ensure correctness and efficiency.

5.3.1 Limitations of existing design of error handling in ajc

We will now explore the downsides of current design and implementation of error handling in the context of ajc. Consider Listing 5.1, which shows a snippet of code from `MissingResolvedTypeWithKnownSignatures` class.

While attempting to resolve a required type in the `World`[3], if the weaver fails to find any required dependent type, it logs an error report and then returns an instance

[2] Although the design decisions were conceived with AspectJ in mind, they are equally applicable to other compilers in general.

[3] ajc collects all members that have an invasive effect outside their own compilation unit into a `World` before any weaving can take place.

```
/** This class helps defer --cannot find type-- errors */
public class MissingResolvedTypeWithKnownSignature {
    public ResolvedMember[] getDeclaredFields() {
        raiseCantFindType(WeaverMessages.CANT_FIND_TYPE_FIELDS);
        return NO_MEMBERS;
    }
    public ResolvedMember[] getDeclaredMethods() {
        raiseCantFindType(WeaverMessages.CANT_FIND_TYPE_METHODS);
        return NO_MEMBERS;
    }
    ...
    public int getModifiers() {
        raiseCantFindType(WeaverMessages.CANT_FIND_TYPE_MODIFIERS);
        return 0;
    }
}
```

Listing 5.1: Error reporting for MissingTypes

of this class, instead of simply throwing an error. By assigning this special type, `MissingResolvedTypeWithKnownSignatures`, to any missing type or an ill-typed expression, the compiler allows the type checker to ignore the missing or ill-formed types whenever it is subsequently encountered. This class defers the production of the *cannot find type* error until some code requires such information which cannot be determined from the type `Signature` alone. This enables the weaver to be more tolerant to missing types and thus, delays the compilation failure to a certain extent.

This class has a facility to report errors upon attempts to access various missing types: fields, methods, interfaces, pointcuts, super-classes and modifiers. Although, they are neatly modularized in a single class, this implementation has four major problems:

1. First, there is clear tangling of two different concerns here: the *functional* and the *error handling* requirements. The functional requirement is that these methods should return some default value of appropriate types. The error-handling concern is that an error should be reported upon any attempt to access non-existent attributes of `MissingResolvedTypeWithKnownSignatures` type. Such tangling inhibits possibilities of creating and reusing abstractions of error handling and functional operation.

2. Second, it lacks a clear information about program-execution path through which the errors are reported. A careful examination of the above implementation reveals that the error reports are generated before these methods return values, through invocations of `raiseCantFindType(..)` methods. However,

this is not visible outright in the current implementation. This would have been even more cumbersome in situations where the error-handlers were interspersed across different methods in different classes; because, the loci of such handlers are not explicitly identified and defined in the compiler source.

3. Third, it lacks an explicit way to situate these error reporters. If a developer wished to extend existing error reporting, he would have to inspect the code again to find them. Thus, it lacks a facility to properly define and localize the context in which these reports should be generated. Further, by looking at this implementation, it is difficult for compiler writers to ensure if these are the only methods that call `raiseCantFindType(..)` methods. Aspects helps us to do so, an example is given in Appendix B, as Listing B.3. The idea is that upon any attempts to handle errors from modules other than those solely responsible for error handling, this aspect generates a warning message.

4. Fourth, this implementation is brittle to changes. For error diagnosis or handling, we might to need to carry out data-oriented or control-oriented changes or both[47]. Design patterns[30] such as *Subject-Observer* and *Visitor* will suffice for a decently modular implementation, if any one of these changes is required. In cases where both of them are required, it is difficult to do so without code repetition and tangling using traditional programming paradigms.

These are the problems we are trying to address by modularization of error-handling concern.

5.4 Design of Modular Error Handlers

In this section, we describe a general design for modularizing error recovery to support better comprehension, reduced redundancy, and increased reuse opportunities for recovery. In short, we identify the points in program execution where error checks should happen, and explicitly move them into separate modular units, along with the actions to be taken in case of such errors. Here, we do not propose any efficient error recovery schemes, but only try to localize and encapsulate such error-related concerns into clean modular units. Hence, existing error recovery in ajc is not effec-

tively changed, save for the example implementations depicting the usability of our design.

To locate declarations and references of error handlers, we followed two main approaches to code inspection. First, we employed development aspects to identify the *error print streams, loggers, exception-throwers* and *-handlers*. Listing A.3 shows an example of development aspect employed during modularization of error-handling. The aspect finds all places in the ajc, where error messages are generated, and exceptions are thrown. By reaching those sites, we were able to locate only a few error handling sites.

To identify other customized error reporters, we had to manually inspect the code. A major finding of this was that the compiler is riddled with error handlers, and it involved significant effort to locate them. So, here we try to reduce the economic burden associated with identifying such join points, by encapsulating them in dedicated modules.

We have implemented a set of aspects in ajc to modularize error-handling pertaining to:

1. ill-formed AspectJ constructs such as type patterns, pointcuts, advices and inter-type declarations,
2. failure to create initialization and pre-initialization shadows for type mungers,[4]
3. failure to type-check and resolve an input-program,
4. violation of type-munging rules during weaving, and
5. incorrect context-bindings in pointcuts.

We supply four examples, and their subsequent reuse and improvement for enhanced reporting and recovery. One of these is a syntactice error, while the other three are semantic ones.

5.4.1 Modularizing `MissingType` Error Handler

This semantic example is about missing types, first introduced in section 5.3.

[4]Recall that a munger is a representation for advice and advice-like entities that weaves required behavior at the join point shadows. See chapter 2 for details.

We begin by extracting the error-raisers and -throwers related to missing types from Listing 5.1. Using AspectJ, we extract and modularize handlers related to missing type error into aspects as shown in Listing 5.2 and 5.3.

```
/**
 * Abstract aspect for handling missing and incompatible types
 */
public abstract privileged aspect MissingAndIncompatibleTypes {
    abstract pointcut currentType(MissingResolvedTypeWithKnownSignature aType);
    abstract pointcut contextForMissingTypes();
    abstract pointcut contextForIncorrectTypeAssignability(ResolvedType otherType);
    pointcut missingResolvedTypes(MissingResolvedTypeWithKnownSignature aType):
        currentType(aType)
        && contextForMissingTypes();
    pointcut incompatibleTypeAssignability(MissingResolvedTypeWithKnownSignature aType,
                                           ResolvedType otherType):
        currentType(aType)
        && contextForIncorrectTypeAssignability(otherType);
}
```

Listing 5.2: Abstract aspect for reporting `MissingType` error.

```
/**
 * Concrete aspect for reporting missing type error
 */
privileged aspect MissingResolvedTypeErrorReporter extends MissingAndIncompatibleTypes {
    protected  pointcut contextForMissingTypes():
        execution(public ResolvedMember[] *.getDeclaredFields())
        ||execution(public ResolvedMember[] *.getDeclaredMethods())
        ||execution(public ResolvedType[] *.getDeclaredInterfaces())
        ||execution(public ResolvedMember[] *.getDeclaredPointcuts())
        ||execution(public ResolvedType *.getSuperclass())
        ||execution(public int *.getModifiers())
        ||execution(public boolean *.hasAnnotation(UnresolvedType));

    protected  pointcut contextForIncorrectTypeAssignability(ResolvedType otherType);

    protected pointcut currentType(MissingResolvedTypeWithKnownSignature aType):
        this(aType);
}
```

Listing 5.3: Concrete aspect for reporting `MissingTypes` error.

After separation of error handlers, the re-factored class now looks as shown in Listing 5.4, in contrast to Listing 5.1. We will assess the benefits of such modularization later.

The advice shown in Listing 5.5 is part of an abstract aspect `MissingAndIncompatibleTypes`, and defines the action to take in the event of `missingResolvedTypes`. An abstract aspect helps us define a set of events that is left unspecified, but allows us to give the advice that should apply. Then, in the concrete aspect `MissingResolvedTypeErrorReporter`, shown in Listing 5.3, we bind the methods to the join points to which the advice should be hooked. This way, we are separating location (points in static program and dynamic execution graph) where errors might be handled

```
/**
 * MissingTypeWithKnownSignature class after extraction of error handlers
 */
public class MissingTypeWithKnownSignature {
    public ResolvedMember[] getDeclaredFields() {
        return NO_MEMBERS;
    }
    public ResolvedMember[] getDeclaredMethods() {
        return NO_MEMBERS;
    }
    ...
    ...
    public int getModifiers() {
        return 0;
    }
}
```

Listing 5.4: `MissingTypeWithKnownSignature` class after extraction of error-handling concern.

```
/**
 * An improved advice for reporting missing type error
 */
before(MissingResolvedTypWithKnownSignature aType):
        missingResolvedTypes(aType){
        String typeName = thisJoinPointStaticPart.getSignature().getName();
        String weaverMsg = WeaverMessages.CANT_FIND_TYPE;
        if (typeName.endsWith("Methods"))
                weaverMsg = WeaverMessages.CANT_FIND_TYPE_METHODS;
        else if (typeName.endsWith("Fields"))
                weaverMsg = WeaverMessages.CANT_FIND_TYPE_FIELDS;
        ...
        ...
        else if (typeName.endsWith("Modifiers"))
                weaverMsg = WeaverMessages.CANT_FIND_TYPE_MODIFIERS;
        aType.raiseCantFindType(weaverMsg);}
```

Listing 5.5: Improved advice for more informative reporting of `MissingType` error.

from the actions that should be taken to handle them. Further, by looking at the context associated with the advices, we can infer end-to-end data flow related to error handlers. In our examples, it is easy to spot that *missing type* errors are raised only by instances of `MissingResolvedTypeWithKnownSignature` types.

Design Evolution Yet another benefit of this modularization is flexibility in accommodating design decisions. For instance, suppose we decide to change the design decision to report input-program errors only after returning default values in response to the getters in Listing 5.1. We can easily implement this change by changing the kind of advice to `after`.

At present, our error reporter is naïve in that it says nothing other than *missing type* error. If we wish to inform precisely the name of missing type, this change needs to occur only in the advice body as shown in Listing 5.5. It is, however, reasonable that in some cases we might have to change the pointcut as well to bind arguments

and to make our places of interest more evident. Note that by decoupling the error-handling concern from the functional code, we enable changes to each of them in relative isolation.

5.4.2 Modularizing `ParserException` Error Handler

Our first syntactic example is handling errors while parsing. Specifically, here we will look at errors raised as a result of ill-formed type-patterns and pointcut-expressions.

In order to modularize error handlers related to parsing, we first identify the join points that lead to `ParserException`s, and capture them as pointcuts. Then, as part of advice implementation, we surround computation under these join points with try-catch blocks. The catch block is engineered to perform appropriate error reporting whenever a `ParserException` is raised. First consider handling errors when parsing a given type pattern. Listing 5.6 shows pointcut specification for capturing join points which might raise `ParserException` when parsing invalid type-patterns.

```
/** Pointcut to capture invalid type pattern */
pointcut captureInvalidTypePattern(String patternString, AjAttributeStruct location):
    cflow(execution(private static TypePattern parseTypePattern(String,
                                                    AjAttributeStruct))
          && args(patternString, location))
    && (call(public TypePattern PatternParser.parseTypePattern())
        ||call(public void *.setLocation(ISourceContext, int, int)));
```

Listing 5.6: Pointcut to capture invalid type pattern in AspectJ attributes.

From the pointcut definition, we see that invalid type-patterns are detected when `parseTypePattern(..)` method is called within `parseTypePattern(..)` method itself, in the control flow of execution of the outer `parseTypePattern(..)` method. For generating error messages, the pointcut also captures the context using the `args()` pointcut.

Similarly, Listing 5.7 depicts pointcuts for capturing join points that raise `ParserException` when parsing pointcuts. In this case, the invalid pointcuts are detected during calls to `parsePointcut(..)` method in the control flow of execution of an outer `parsePointcut(..)` method.

From these pointcut definitions, it is clear that parsing errors occur in the control flow of top-level or recursive excecution of `parseTypePattern` or `parsePointcut` methods. Since the behavior to identify join points is quite similar, we can abstract

```
/** Pointcut to capture invalid pointcut */
pointcut captureInvalidPointcut(String pointcutString, AjAttributeStruct location):
    cflow(execution(private static Pointcut parsePointcut(String,
                                                           AjAttributeStruct,
                                                           boolean))
          && args(pointcutString, location, *))
    && (call(public Pointcut PatternParser.parsePointcut())
        ||call(public void *.setLocation(ISourceContext, int, int)));
```

Listing 5.7: Pointcut to capture invalid pointcut definition.

a single advice to report errors at both of these kinds of sites. The advice is shown in Listing 5.8.

```
/* Report error on capturing invalid Type Pattern or Pointcut */
Object around(String patternOrPointcutString, AjAttributeStruct location):

    /* Pointcuts where similar behavior has to be implemented are composed together */
    captureInvalidTypePattern(patternOrPointcutString, location)
    ||captureInvalidPointcut(patternOrPointcutString, location) {
      try{
        return proceed(patternOrPointcutString, location);
      } catch (ParserException e){
          AtAjAttributes.
              reportError("Invalid "
                          + thisJoinPointStaticPart.getSignature()
                            .getName().replace("", "parse")
                          + patternOrPointcutString + "' : "
                          + e.toString()
                          + (e.getLocation() == null ? "" : " at position "
                             + e.getLocation().getStart()), location);
          return null;
      }
    }
```

Listing 5.8: Advice to report error on finding invalid type pattern or pointcut.

It reports an invalid pointcut or type pattern error, along with its position in the type pattern or pointcut expression, and also the location in the file.

Design Evolution Besides these, the compiler has to deal with several other errors that occur during parsing, such as: ill-formed per-clauses, annotation pointcuts, annotation aspects and other AspectJ constructs. Such closely related pointcuts are good candidates to be localized into a single aspect that handles parsing errors across different AspectJ attributes. Further, composing them provides an opportunity to handle errors through a single advice.

5.4.3 Modularizing `InCorrectReturnType` Error Handler

Our second semantic example provides recovery from errors that occur when new types violate type-munging rules. Specifically, when an existing type hierarchy is changed or new type is introduced, the weaver must ensure that the new types do

not violate pre-defined type-munging rules. Here, we will look at error handling related to method overriding in sub-types.

Ajc enforces a covariance of return types in overriding methods[12]. Our example shows how to capture into single module, this behavior for inter-type declarations. We have identified and extracted the artifacts related to this error in aspects. The pointcuts and associated advices are shown in Listing 5.9 and 5.10.

```
/** Aspect handling incorrect return type error during binary weaving of declare parents */
pointcut checkCompatibilityOfReturnTypes(BcelClassWeaver weaver,
                                         ResolvedMember superMethod,
                                         LazyMethodGen subMethod):
    execution(private boolean BcelTypeMunger.
              enforceDecpRule4_compatibleReturnTypes(BcelClassWeaver,
                                                     ResolvedMember,
                                                     LazyMethodGen))
    && args(weaver, superMethod, subMethod);
Object around(BcelClassWeaver weaver, ResolvedMember superMethod, LazyMethodGen subMethod):
    checkCompatibilityOfReturnTypes(weaver, superMethod, subMethod){
      if (!superMethod.getTypeSignature().replace('.', '/').
      equals(subMethod.getTypeSignature().replace('.', '/'))) {
        ResolvedType subType = weaver.getWorld().resolve(subMethod.getReturnType());
        ResolvedType superType = weaver.getWorld().resolve(superMethod.getReturnType());

        if (!superType.isAssignableFrom(subType)) {
          weaver.getWorld().
               getMessageHandler().
               handleMessage(MessageUtil.
                             error("Return type is incompatible with "
                                   + superMethod.getDeclaringType()
                                   + "." + superMethod.getName()
                                   + superMethod.getParameterSignature()),
                              subMethod.getSourceLocation());
      }
      }
      return proceed(weaver, superMethod, subMethod);
    }
```

Listing 5.9: Pointcut to capture overriding methods with incorrect return types.

The pointcut `checkCompatibilityOfReturnTypes()` identifies sites where type-munging rules are violated. It describes the situation in which `InCorrectReturn` is handled is while enforcing type compatibility rules as part of type-munging rules for `declare parents` as indicated by `execution (private boolean BcelMunger.enforceDecpRule4compatibleReturn(..))`.

The advice then checks to see if the super type is assignable from the sub type. If the types are not assignable, it reports an incorrect return type error, along with information about types involved and their source locations.

Listing 5.10 shows yet another site which could raise an incorrect return type error, along with the advice for reporting this error.

The pointcut `checkMungerToAdd()` identifies incorrect return type errors that are handled in the control flow of adding inter-type mungers as indicated by cflow point-

```
1  /** Aspect handling incorrect return type error while weaving from source */
   pointcut checkMungerToAdd(ResolvedType resType,
3                            ResolvedMember parent,
                             ResolvedMember child):
5      cflow(execution(public void ResolvedType.addInterTypeMunger(ConcreteTypeMunger))
           && this(resType))
7      && (execution(public boolean ResolvedType.checkLegalOverride(ResolvedMember,
                                                                    ResolvedMember))
           && args(parent, child))
9      && if(!Modifier.isFinal(parent.getModifiers()));
11
   Object around(ResolvedType resType, ResolvedMember parent, ResolvedMember child):
13 checkMungerToAdd(resType, parent, child){
       if (!(resType.world.isInJava5Mode() && parent.getKind() == Member.METHOD)) {
15         if (!parent.getReturnType().equals(child.getReturnType())) {
               resType.world.showMessage(IMessage.ERROR,
17                                       WeaverMessages.
                                         format(WeaverMessages.ITD_RETURN_TYPE_MISMATCH,
19                                                parent, child),
                                         child.getSourceLocation(),
21                                       parent.getSourceLocation());
           }
23     }
       return proceed(resType, parent, child);
25 }
```

Listing 5.10: Pointcuts to capture inconsistent method overriding and advice to handle the error.

cut `cflow(execution(public void ResolvedType.addInterTypeMunger(ConcreteMunger))`. The exact point where this happens is while checking the type of overriding method (resulting from new type munger to be added) as indicated by `execution(public boolean Resolved.checkLegalOverride(...))`. Further, we also have access to the context of this error through the `args(..)` pointcut.

The advice then check to see if the return types of the parent and overridden methods are same. If they are not compatible, it reports an error along with other useful information: the types and methods involved and the exact source locations.

Design Evolution We can improve error handling capability of the compiler by trying to recover from this error. A simple recovery scheme is to change the return type of the overriding method so that it is same as that of the overridden method. In doing so, we need to keep other properties of the overriding method intact.

We have already modularized the related control dependencies, and identified the associated context and action to trigger in the event of occurrence of this error. Hence, we can easily reuse this information while implementing the error recovery action. For example, while weaving from source, this recovery-attempt involves a replacement of error reporter from the advice with a call to `proceed`. Essentially, we continue the current computation with a corrected `subMethod` with all other argu-

ments remaining the same. Here, `subMethod` is the internal name for the overridden method. The resulting advice is shown in Listing 5.11.

```
/**
 * Error recovery advice applicable to the existing pointcut describing sites of
 * incorrect return type error
 */
Object around(ResolvedType resType, ResolvedMember parent, ResolvedMember child):
    checkMungerToAdd(resType, parent, child){
        if (!(resType.world.isInJava5Mode() && parent.getKind() == Member.METHOD)) {
            if (!parent.getReturnType().equals(child.getReturnType())) {
                ResolvedMember newChild = correctChild(parent, child);
                return proceed(resType, parent, newChild);
            }
        }
        else return proceed(resType, parent, child);
    }
```

Listing 5.11: Advice to recover from inconsistent method overriding while weaving from source.

Note that the advice must first replace execution of `addInterTypeMunger(ConcreteMunger munger)` with corrected munger instead of the old faulty munger. The new `subMethod` is created using the modified `munger`. Also, note that creating a new munger with the corrected return type is an expensive operation, because we must discard some intermediate states obtained after execution of `addMunger(..)` and create them all again.

Since our primary focus is on improving modularity in implementation with reusability as one of the goals, performance issues are neglected in favor of clarity.

This concept of reusability of pointcuts is more visible in Figure 5.1. $EHAspect_n$ represents an indexed family of error-handling aspects where pointcuts are defined as part of error-handler modularization. Essentially, they intercept methods raising errors (such as the node labelled **C**) to generate error reports and make a safe exit from the program. If it is an exception condition, these aspects can access control flow contexts and can perform appropriate exception-handling, as indicated by arrows from nodes **C** to **B** to **A** and then to **Error**.

As part of error recovery, aspects $ERAspect_n$ will reuse pointcuts defined in $EHAspect_n$. At this point, if there is need for additional context besides those captured with `args()` pointcut, we have reflective access to it through `thisJoinPoint`. These aspects will also instrument the methods leading to the input-program error, and try to change the context at the point of error so that compilation could proceed

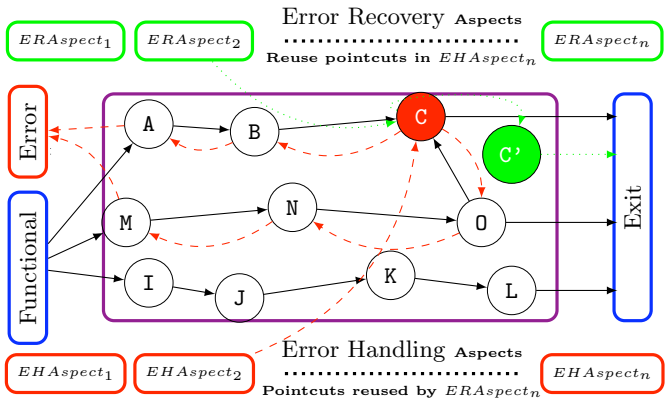

Figure 5.1: Artifacts of modular error handling.

further. This is shown by green dotted-arrows in Figure 5.1. The recovery algorithm implementation for the remaining pointcut is also similar to this. Hence, we skip discussion of the same. If any new errors stem from recovery attempts, we issue a warning message so as to apprise users of the attempted error recovery.

5.4.4 Modularizing `IncorrectPattern` Error Handler

The third semantic error relates to use of un-supported contexts and type-patterns in pointcuts, such as a `cflow` context with `declare` pointcut, and also semantically incorrect pointcuts, such as a circular pointcuts[5].

As there are numerous cases of such errors in pointcuts, we first create an abstract aspect, and then extend it to handle each particular case. With an abstract aspect, we can create a reusable unit of crosscutting by deferring some of the implementation details to the concrete sub-aspects.

Consider the aspect shown in Listing 5.12. In this aspect, the pointcut `exactConcretizeSites()` (line 16) is declared abstract to allow sub-aspects to provide its definition. The pointcut `commonConcretizeSites()` identifies all places where different kinds of`Pointcuts` are concretized[6]. These pointcuts are, then, composed to create `incorrectConcretizeSites` pointcut. The net effect is that it helps us define a

[5]Circular pointcut is a pointcut that refers to itself in its definition.

[6]Recall that all pointcuts are made concrete from their abstract forms after all the required information are filled in, following the deep-parsing phase. See subsection 2.6.1 for details.

```
/**
 * Report errors, when type-munging rules are violated
 */
public privileged abstract aspect IncorrectPointcutHandler {
        protected String _ErrorName = null;
    /**
     * Locate places pointcuts are concretized
     */
    pointcut commonConcretizeSites(ResolvedType inAspect, .., IntMap bindings):
        execution(public Pointcut *.concretize1(ResolvedType, ResolvedType, IntMap));
    /**
     * Locate exact places where specific pointcuts are concretized
     */
    pointcut abstract exactConcretizeSites();
    /**
     * Locate places which create incorrect pointcuts
     */
    pointcut incorrectConcretizeSites(ResolvedType inAspect, .., IntMap bindings):
        commonConcretizeSites(ResolvedType, .., IntMap))
        && exactConcretizeSites();
    /**
     * Generate error-reports on attempts to create incorrect pointcuts
     */
    Pointcut around(ResolvedType inAspect, .., IntMap bindings):
        inConcretizeSites(inAspect, bindings){
            MessageUtil.error(WeaverMessages.format(this._ErrorName,thisPointcut),
                              thisPointcut.getSourceLocation()));
        return Pointcut.makeMatchesNothing(Pointcut.CONCRETE);
    }
}
```

Listing 5.12: Abstract aspect to report incorrect pointcuts.

subset of join points represented by concretizing pointcuts, by narrowing the scope with the abstract pointcut that will be defined in sub-aspects.

The `around` advice (lines 28 – 33) applied to `incorrectConcretizeSites()` pointcut, defines the error-handling action to take upon detecting incorrect pointcut-patterns and -contexts. The way of handling both these errors is same; the difference lies in their names. This information is held by the `ErrorName` string in the base aspect.

Listing 5.13 depicts a concrete aspect that extends the `IncorrectPointcutHandler` aspect. It specializes the abstract aspect with additional information describing the sites and situations in which the advice defined in the base-aspect must be applied. One additional requirement is that the executing instance should be of type `CflowPointcut`. Next, the pointcut should be a `declare` kinded one. Essentially, these aspects define error-handling behavior when a `cflow` context occurs in a `declare` pointcut.

Design Evolution Now, consider handling errors related to circular pointcut definitions. For this, we will reuse the abstract aspect defined previously. The concrete aspect implementing our desired error-handling behavior is shown in Listing 5.14.

```
/**
 * Report errors, when cflow context is attempted in declare constructs
 */
privileged aspect CflowPointcutErrorHandler extends IncorrectPointcutHandler {

    ErrorName = "Cflow in declare error";

    pointcut exactConcretizeSites(ResolvedType searchStart,
                                  IntMap bindings,
                                  CflowPointcut thisPointcut):
          && this(thisPointcut)
          && if(thisPointcut.isDeclare(bindings.getEnclosingAdvice()));
}
```

Listing 5.13: Concrete aspect to report `cflow` in `declare` pointcut.

This implementation involves specialization of the abstract aspect with additional information. The first is that the abstract pointcut is now defined to check if the pointcut in consideration is of type `ReferencePointcut`, and check if it is concretizing itself. The other is to initialize the `ErrorName` field with appropriate string, to include in the error report.

```
/**
 * Report errors, when pointcuts are circular in nature
 */
privileged aspect CircularPointcutErrorHandler extends IncorrectPointcutHandler {

    ErrorName = "Circular pointcut error";

    /**
     * Locate places which create incorrect pointcuts
     */
    pointcut exactConcretizeSites(ResolvedType searchStart,
                                  IntMap bindings,
                                  ReferencePointcut thisPointcut):
        && this(thisPointcut)
        && if (thisPointcut.concretizing);
}
```

Listing 5.14: Concrete aspect to report circularity in pointcut.

Thus, we see that by extracting commonalities and localizing them into a separate module, an abstract aspect provides opportunities for reuse. Our abstract aspect, containing an abstract pointcut, allows base aspects to implement the crosscutting logic without needing the exact details. Note that an abstract aspect by itself does not cause any weaving to occur; therefore, we provide concrete subaspects to do so.

5.5 Evaluation

As we contemplate modular error-handling, the first question that arises is the associated costs and benefits. In compilers, this means that we have to vouch for correctness and performance guarantees of our remodularized implementation. We

begin this section by supporting these guarantees. Next, we evaluate our implementation from modularity perspective to assess its impact on coupling, its support for re-usability, and any obstacles to which it might lead.

5.5.1 Correctness Assessment

A fundamental property of compilers is correctness. In order to verify this property of our candidate compiler, we made sure that it passed all the existing JUnit tests, and the new ones that tested for correctness of error recovery.

5.5.2 Performance Assessment

For performance assessment, we compare elapsed time to complete the JUnit tests. Table 5.1 shows the mean results of ten different readings, along with standard deviations of the results, for both the original and the restructured ajc. The tests

Table 5.1: Assessment of modularization using performance metrics.

Platform	Compiler	Elapsed Time(sec)	σ(sec)
Mac OS X	Original	966.4	9.23
	Restructured	(+5.2)0.5%	(+0.33)3.5%
Ubuntu	Original	959.9	8.11
	Restructured	(+4.3)0.4%	(+0.6)7.3%
Windows XP	Original	959.9	8.11
	Restructured	(+20)2.0%	(+2.33)27%

were performed across three different platforms. First, is the Mac OS 10.5.6 running on an Intel Core^{TM} 2 Duo MacBook 2.1 with 3GB memory, 4MB L2 cache, 2.16GHz processor speed and 667MHz system bus speed. The second is the Windows XP operating system running on Sony Intel Pentium dual core T3200 Laptop with 2GB memory, 1MB L2 cache, 2GHz processor speed and 667MHz system bus speed. The third is the Mandriva Linux OS running on an Intel Core 2 CPU with 2.4GB memory and 4MB cache and 2.4GHz processor speed.

We found that modularization of error handlers with aspects resulted in 0.5% – 2% increase in compilation time of the JUnit test. This small difference between elapsed time for completion of JUnit tests in the original compiler and error-modularized compiler demonstrate that time overhead of aspectization of error handlers is minimal, usually within the standard deviation. Note that the major source of this performance overhead is attributable to expensive operations involved in recovering from the *incorrect return type* error from an inter-type declared method.

5.5.3 LOC Assessment

To examine the way our modularization affects code size, we first measured the size of the original and re-factored compilers, making a distinction between the aspect code and the base code. Their overall size are comparable, as shown in Table 5.2. These numbers were obtained, using the SLOCount tool.

Table 5.2: Assessment of modularization using LOC metric.

Package	LOC in implementation			Concerns Modularized		
(org.aspectj)	*OO*	*AO*		Detection	Reporting	Recovery
weaver	13548	13159	(195 in aspects)	✓	✓	✓
bcel	16005	15702	(677 in aspects)	✓	✓	✓
Total	29553	29733	(872 in aspects)			

A notable point in the above table is the reduced size of the `weaver` package. This is the result of removing redundant pieces of code that were previously scattered across multiple methods within different classes of the `weaver` package. In the `bcel` package, however, there is an increase in overall size – 677 lines of code (LOC) comes from aspect-orientation (AO). This increase in LOC primarily results from our error recovery implementation in `bcel` package. Overall, we see that aspect-orientation re-factoring has increased the code size by a small number. Another reason for increased code size is because our implementation contains a number of tiny aspects, and requires a fair amount of boilerplate for creating the abstract aspects. Our belief is that aspect-oriented modularization of code related to remaining partially-

completed error-handlers will lead to a decrease in code size by making use of already available abstract aspects, thereby removing the redundant pieces of code.

We now examine in some detail how well scattering and tangling of error handling is managed.

5.5.4 Modularity Assessment

Figure 5.2 shows SeeSoft representations[24] of error handlers in the weaver end of ajc before and after aspect re-factoring. Colored stripes represent different kinds of errors and are proportional to the size of the classes represented by the vertical bars. These figures were generated using the standard aspect visualization tool, available with the AspectJ compiler.

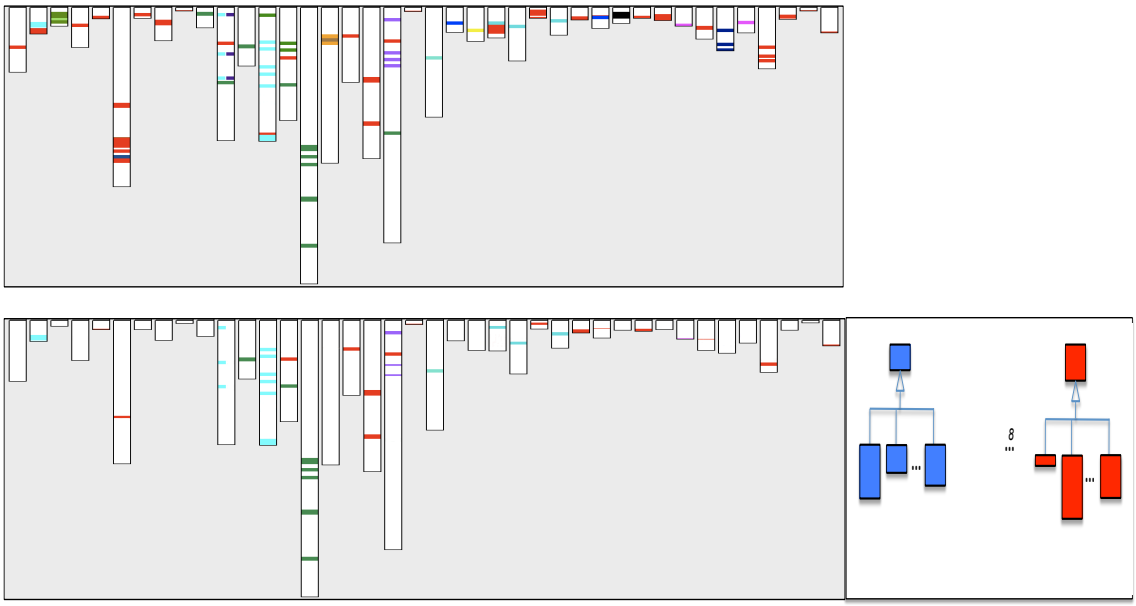

Figure 5.2: Nature of error handling before and after modularization.

Before modularization, the error handlers were scattered across methods in 40 different classes as indicated by dispersed lines within the white blocks. After re-factoring error-handling, we reduced the scattering of error handlers down to 21 classes. Further, we were able to decrease scattering among different methods within a class. We modularized 8 different groups of related errors, and we believe additional 8 of them could be modularized using the same technique. There are few errors, how-

ever, which require code modification to expose the join points for modularization[7]. This is an option for future work.

Benefits of Modular Error Handling

Based upon our modularization endeavor in ajc, we provide a list of benefits of separating error handling from the program and implementing it as a separate pluggable construct. These benefits come in two different forms: one, directly resulting from aspect-oriented modularization. They are improved specification, reduced dependencies, and cleanly separated concerns. The other benefits come in derived form, they result from the direct ones. They are: improved comprehensibility, increased reusability, and improved flexibility.

Direct Benefits We first investigate the direct benefits.

1. **Simplified error-handling specification**: The join point model serves to define the locations and control flows associated with error handlers. This makes it easy to identify error recognition loci, because they are now explicitly defined in their dedicated place: the pointcuts.

2. **Reduced Dependencies**: In its present state, there is tight coupling between several classes in ajc because of dependencies relating to error handlers. This is shown by means of a dependency structure matrix (DSM) in Figure 5.3. Here, we consider the dependencies arising only due to error concern. With the modularization we have implemented, the dependencies among classes (represented by $C_1, ..., C_n$) resulting from error-handling modularization will now look like one shown in Figure 5.4. This is simply an illustration of the nature of coupling between the classes whose error-handling codes are completely extracted into aspects. For a complete picture of the dependency relationship between member classes of ajc, please refer to Appendix D.

   After aspect-oriented modularization of the pieces of code related to error handling, the dependency picture changes dramatically. Dependencies among classes owing to error-handling concerns are now completely removed for the

[7]See chapter 6 for details.

concerns we dealt with. As a result, the classes no longer depend on the error-handling implementations in other classes. Further, the dependency direction is now changed such that aspects (represented by $A_1, ..., A_n$) responsible for error handling will now depend upon the existence of these classes to collect required contexts. This new dependency is visible in the lower part of the DSM in Figure 5.4.

org.aspectj.bcel	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35
AnnotationAccessFieldVar		x																																	x
AnnotationAccessVar		x	x									x									x														x
AtAjAttributes				x																															
BcelAccessForInlineMunger					x													x	x	x															
BcelAdvice						x	x																						x	x					
BcelAttributes							x	x																											
BcelCflowAccessVar							x	x											x	x	x														
BcelCflowCounterFieldAdder								x	x																										
BcelCflowStackFieldAdder										x																									
BcelClassWeaver											x												x		x										
BcelField		x										x									x		x												
BcelFieldRef			x										x										x												
BcelSignatureToTypeConverter														x									x												
BcelMethod															x								x												
BcelObjectType																x							x												
BcelPerClauseAspectAdder																	x																		
BcelRenderer																		x	x										x						
BcelShadow																		x	x										x						
BcelTypeMunger																				x			x										x		
BcelVar												x	x	x	x						x		x												
BcelWeaver																						x													
BcelWorld					x																	x	x												
ClassPathManager																								x											
ExceptionRange																									x										
IfFinder									x																	x									
LazyClassGen											x																x			x					
LazyMethodGen																												x							
PoliceExtensionUse		x														x				x									x						
Range						x																								x					
ShadowRange																								x						x	x				
TypeAnnotationAccessVar																							x	x						x		x			
UnwovenClassFile											x																						x		
UnwovenClassWithByteCode											x																							x	
Utility																																			x

Figure 5.3: DSM for `bcel` package before error-handling modularization.

3. **Cleaner separation of concerns**: Separation of error-handling behavior of classes from normal functional behavior makes it easier to understand what those two behaviors really are, because their implementations do not collide in the same text. Additionally, as shown in Figure 5.1, the control flows related to error handling are separated from other functional control flows. With traditional object-oriented implementations, one would need to propagate back along the control flow path in which the error was raised in order to notify the enclosing types about the error. In Figure 5.1, if the node labelled **C** is a method that raises an error, then as indicated by the dashed-arrows, the compiler would have to propagate back along the path of its control flow to inform the enclosing type about the error. This leads to contaminating the functional

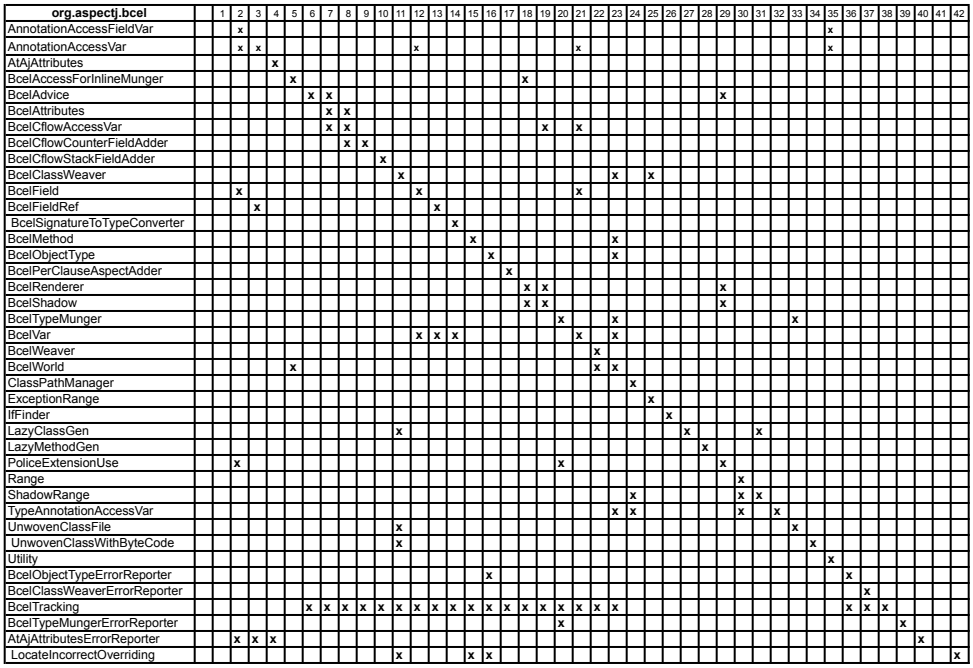

Figure 5.4: DSM for `bcel` package after error-handling modularization.

control with error-handling control flow. However, with aspect-oriented modular implementations of error handlers, this is not necessary. As the control in which the input-program error was recognized is already precisely captured by means of pointcuts, one can use this information for appropriate error handling.

Derived Benefits We now investigate the benefits resulting from the direct ones.

1. **Increased comprehensibility of error handlers**: In addition to control dependencies, pointcuts also expose the data dependencies associated with the error handlers. From the list of formals and return values in the pointcut definition, one can easily infer the data dependencies of error handlers. This serves to improve their comprehensibility.

2. **Increased re-usability**: From examples presented in section 5.4, we see that the error-recovery aspects reuse the pointcuts defined for error reporting. Correct error recovery schemes need to know the precise sites of error handling, the type of exception being handled, and the control flow of its occurrence. All such information is encapsulated by error reporting aspects ($ERAspect_n$). Later, the error recovery aspects ($EHAspect_n$) reuse these pointcuts to garner such infor-

mation. Then, they replace the erroneous AST-node in the control flow path with a corrected one. Other approaches to error recovery might involve resuming execution with changed contexts. This will then allow computation to proceed further, as indicated by dashed arrows in the Figure 5.1.

During modularization, context and operations that are similar to different error handlers are abstracted out as abstract aspects. These are shown by the top-level colored boxes in Figure 5.2. Later, these commonalities are reused by concrete aspects to incorporate operations distinct to each kind of error.

3. **Increased flexibility in design decisions regarding errors**: Design decisions such as when, where and how to do error handling are easily modifiable. This benefit comes from the expressive power of the join point model.

These benefits, however, come at the expense of slightly reduced performance, attributable to byte-code weaving and error recovery computations. Overall, modular error handling engenders key benefits at the cost of modest performance overhead.

Issues in Modularizing Error Recovery

Aspects introduce additional modules into a program; this increases the range of potential interactions and may have negative impacts on the understandability and the re-usability of the code. Some examples are:

1. **Confounded ordering of error handling aspects**: AspectJ does not provide a simple means of controlling order of application of multiple aspects at the same join point. AspectJ constructs, such as `declare precedence` require global knowledge of all existing aspects to explicitly define the order in which they should be woven. Thus, any unintended order of aspect composition, resulting from incomplete knowledge of existing aspects, may affect error handling behavior in ways unanticipated by the developers.

2. **Masking of error-handling advices by foreign aspects**: It is possible that an unbounded foreign aspect completely replaces our desired computation by introducing new advice around `adviceexecutions` related to error handlers.

3. **Overly-broad application of error handlers**: Unbounded pointcuts may result in advice weaving at unintended points. However, strictly bounded pointcuts may prove too restrictive at times [51]. It is, therefore, difficult to correctly identify the circumstances in which foreign pointcuts need bounded and unbounded pointcuts.

In addition, we found that there is an overall increase in code size because of aspect-oriented modularization. However, we believe that the code size will eventually decrease upon modularization of other remaining error handlers that need some additional work for exposing their join points.

Furthermore, we are not perfectly confident about identification of error handling pieces of code in ajc in entirety. However, this is an issue which could be overcome with better design decisions early on – examples include: consistent naming patterns, annotating each error handling piece of code, and handling errors across the whole compiler with consistently customized methods.

Overall, we believe our aspect modularization appears to provide a net positive benefit to the structure and understandability of ajc.

5.6 Related and Future Work

Others [11, 16, 25, 26, 37, 48] have attempted to modularize error-handling concerns. Lippert and Lopes [48] carried out a study to assess the suitability of aspects for separation of exception-handling code from normal application code. Likewise, Colyer et al. [16] carried out a similar study to separate exception handlers from middleware. Lippert and Lopes's study was later complemented by Filho et al. [26] through aspect-oriented refactoring of exceptions in different real-world applications. Their study concluded that although aspectization of exception handlers created increased opportunities for reuse and decreased interference in the program texts, they are not always sufficient to encapsulate all possible exceptions. Further, they concluded that it is difficult to reuse the exception-related artifacts in practice. Our work builds on theirs and shows that in the domain of error handling in compilers there is benefit to aspect orientation.

Another related work is by Hogstedt [37]. It focuses on restoring the safe state of objects, after a method throws an exception and terminates abruptly. Our goals of error recovery attempt the same thing, but in the compiler domain.

Fernando et al. [25] have carried out an interesting study of the interplay between aspects and exceptions. They provide a catalog of exception-handling scenarios to guide aspectization of exception handling. Their paper does not describe how the guidelines would change in situations where developers wanted to capture the context of the exceptions.

Another extension to Fernando et al.'s work is *EJFlow*[11], a syntactic extension to the AspectJ language for tracking exception flows. It provides *an explicit exception channel* that encapsulates the location of exception raisers, handlers, other intermediate sites for exceptions that must be signalled to the enclosing context. The major limitation of the EJFlow tool is that it lacks the power of `cflow` pointcuts. For instance, consider an exception that can occur within a single static location. Depending upon the control flow and context associated with this exception, the way errors are reported and handled could be totally different. This also applies to exceptions that are raised from the same static locations in the program code. EJFlow fails to handle such contexts associated with exceptions, in its present form. Our work is differs in that we consider handling errors rather than just exceptions in input-program. Further, our efforts are more complex because we use `cflow` context too, which is crucial to error-handling.

Most of these efforts attempt to modularize exception handlers to achieve different goals. None of these studies have, however, identified the potential for artifact and knowledge reuse for error recovery that comes from carefully-implemented modular error-handlers. Our work extends the findings and conclusions of the aforementioned papers to modularize error recovery.

5.7 Summary

We have shown how error recovery can be modularized by leveraging the expressive and encapsulative powers of AOP. This is the first published description of implementing modular error recovery in compilers. In contrast to other proposals, this

study identifies how modularizing error handling provides opportunities for reuse of existing software artifacts and knowledge to create and add new error-recovery schemes.

Error handling and recovery often involves comprehending and modifying an unfamiliar and complex code base. Our approach makes it easier to quickly identify the source and location of input-program error situations, understand their behavior, and test new recovery schemes. To facilitate this process, this work addresses queries such as

- what is the control flow in which this error occurs?
  - `cflow` pointcuts, along with advice-weaving rules (`before, after,` and `around`) give this information.
- what is the context when the error is detected?
  - the contexts are captured using `args, this,` and `target` pointcuts.
- which part of code accommodates a new error recovery flow path?
  - evident from pointcut descriptions.
- how to capture this error-situation in another site?
  - by composing existing pointcuts with new pointcuts describing the additional sites of interest.

Answers to such queries provide compiler writers with a broader perspective of error concerns in the system – such as structural, relational and behavioral – by the use of static and dynamic information. These sources of information help the compiler implementers make better informed decisions about error recovery.

We experimentally demonstrate that, in exchange for modest run-time overhead, error recovery modularization leads to net benefits, deriving from a clean separation of error concerns from functional concerns.

Chapter 6

Conclusion

This chapter summarizes our work in relation to the aspect structure of compilers. We begin with a brief overview of the solution. Then, we outline the contributions of our work. Finally, we conclude by identifying some open research questions arising from our implementation and results.

6.1 Summary

Current compiler implementations fail to maintain a clean and direct mapping between the modules of the implementation and the semantic operations they provide. A semantic operation is not isolated into a distinct module of its own and often one module contains scattered and tangled code-segments dealing with many other operations.

In chapter 3, we saw a number of example concerns that do not have such mappings. In three cases, we found that aspects provided a clearer way to modularize the cross-cutting concerns. They are:

- canonicalization,
- register allocation optimization, and
- compilation sequencing.

Then, we presented four other compiler operations that would form novel and effective aspect-candidates for modularization. They are:

- lazy evaluation of state dependencies,
- separation of planning and usage of bytecode manipulation tools,
- peephole optimization, and

- error handling.

After assessing their overall feasibility on the basis of different criteria, we chose the last two for further examinations, and to test our modularization thesis.

We discussed about the first implementation, peephole optimization in chapter 4. We capture the essential abstraction of places in the ajc compiler that receive computational effects of the optimizer, and also develop an abstraction of advices with regard to the effects they express. We identify the points of interest for optimization in entirety by means of developmental aspects. They helped us identify all those places in ajc which generate or group instruction sequences. Furthermore, the modularized peephole optimizer is designed in such a way that its inclusion into or exclusion from the core ajc is easy, and does not require any code changes.

We investigated the second candidate, error handling, in chapter 4. The error handling implementation also creates similar abstractions. In addition, it refines the aspect classifications, yielding reusable abstract aspects for different groups of related errors. Furthermore, the encapsulated concerns are now available for others to improve with efficient error recovery schemes.

We thus demonstrate that aspects provide an effective structuring mechanism for better modularizing the cross-cutting concerns. Our approach to modularization helps make control dependencies and associated context explicit. This is important because control flow information is fundamental to understanding the compiler. In addition, our approach leads to a more reusable compiler-design by isolating distinct concerns into their own modules.

In summary, the logical structure of compilers does not align well with their current implementation because cross-cutting concerns are scattered and tangled with their phase structure; and, aspect-oriented modularization of such concerns improves their comprehensibility, provides better opportunities for reuse, and increases potential for flexible evolution.

6.2 Contributions

The specific contributions of this work are:

1. Identification of seven different concerns that pervade the compiler and cannot be easily localized into clean modular units with mainstream languages. In one case, separation of Bytecode Engineering Library's planning and usage in the weaver, we found that its implementation would offer negligible research contributions.

2. Code changes that yield an elegant model of modularization of compiler operations, where

   (a) principled points in program execution are represented as dynamic join points

   (b) principled points in program source are represented as static join points

   (c) description of these points are denoted by means of pointcuts, and

   (d) specializers of the behavior at these control points are denoted by means of advice.

3. Example modularization of two different cross-cutting concerns:

   (a) peephole optimization, demonstrating overall compiler control flow (as phases) as pluggable units, and

   (b) error handling and recovery, as modular extendable units.

4. Identification of the difficulties for modularization of two other cross-cutting concerns:

   (a) separation of the Byte Code Engineering Library from the weaver and

   (b) lazy evaluation of state dependencies.

Overall, we found that the logical structure of compilers does not align well with their current implementation because cross-cutting concerns are currently scattered and tangled with their phase structure. We demonstrate that aspect-oriented modularization of such concerns improves their comprehensibility, provides better opportunities for reuse, and increases potential for flexible evolution.

A significant portion of this thesis involved refactoring of an existing compiler in order to uncover its aspect structure. We made changes to the internal structure of

ajc to make it easier to understand and simpler to modify and reuse. We stressed that one should structure compilers not only for computation, but also for understanding so that it is easy to reason about them by looking at modules from a much higher perspective. We used aspect-orientation for this purpose and found that there is clear benefit to it.

It is important to realize that although we demonstrate our modularization in one particular compiler, the cross-cutting concerns we isolate are inherent in compilers in general. We feel that our solution strategy has relevance well beyond the AspectJ compiler, and thus will be of interest to the general compiler community.

6.3 Future Work

As aspect-oriented modularization still remains largely unexplored in the domain of compilers, many questions remain yet unanswered. These open research questions pertaining to our work and leading to further investigations come in three groups: one related to directly extending this work and implementing the remaining candidates; one related to mining other aspects in ajc and implementing them, and one related to concerns requiring code changes for exposing the join point.

6.3.1 Extending and Implementing

The first possibility is to implement the remaining viable candidate – lazy evaluation of state dependencies. Others have begun this work: Warth [70] has implemented a lazy type system for Java. His implementation is tightly coupled to the base Java compiler, hence, does not support pluggable integration. A related question to consider is:

- Can aspect-oriented implementation isolate this lazy behavior into a set of related modules that can be readily replaced or removed for analysis and extension purposes?

6.3.2 Aspect Mining in AJC

Another related avenue of future work in ajc is *aspect mining* – identifying the scattered and tangled pieces of code implementing a cross-cutting concern. There

are several tools that aid in mining aspects, such as Aspect Mining Tool[35], AspectJ Development Tool[23], and FINT[49]. Here is an example of a potential aspect in ajc.

Shadow-matching Aspect In ajc, entities like advice are represented by shadow-munger objects. A shadow munger performs transformations on join point shadows matched by its contained pointcut.

Currently, for every shadow munger defined, ajc examines each different kind of join points for a given type, to see if it matches the description of the pointcut to which the shadow munger is bound[1]. This clearly involves extra work. A better approach would be to dis-regard all other join point shadows that are irrelevant to the pointcut in consideration. For instance, if we are dealing with a method-execution pointcut, it would be desirable to refrain from examining all other kinds of join points, such as calls and initializations. Questions that need to be addressed are:

- Can we create join point shadows, and manage the values required at those points in an incremental manner?
- How modular will this implementation be with aspect-orientation?

6.3.3 Exposing Join Points

The second question relates to code changes for exposing join points. One difficulty we encountered while modularizing error-handling concern was extraction of error-related control points that were not identifiable by means of existing pointcuts.

Although AspectJ supports the notion of open-classes, it does not allow access to the local variables of methods. In several places, ajc generates error messages which depend upon the values computed locally within nested if-statements that are part of nested for-loops. For instance, consider Listing 6.1.

[1]This is not entirely true. For pointcuts guarded with `within` pointcut, ajc uses a *fast-match* algorithm. For such pointcuts, ajc does not examine every single type in order to locate the matching join point shadow. In fact, it does not even consider the type signature at this point; it quickly tries to find the affected types specified by the `within` pointcut. When it finds a match, it returns true, indicating that this is a matching type, but further analyses are required to determine whether it contains any desired join points, specified by the pointcut with type-signature.

```
/**
 * Extract class level annotations and turn them into AjAttributes.
 */
public List readAj5ClassAttributes(JavaClass javaClass,
                                   ReferenceType type,
                                   ISourceContext context,
                                   IMessageHandler msgHandler,
                                   boolean isCodeStyleAspect) {
  ...
  ...
  Constant[] cpool = javaClass.getConstantPool().getConstantPool();
  for (int i = 0; i < cpool.length; i++) {
      Constant constant = cpool[i];
      if (constant != null && constant.getTag() == Constants.CONSTANT_Utf8) {
          String constantValue = ((ConstantUtf8) constant).getBytes();
          if (constantValue.length() > 28 && constantValue.charAt(1) == 'o') {
              if (constantValue.startsWith("Lorg/aspectj/lang/annotation")) {
                  containsAnnotationClassReference = true;
                  if ("Lorg/../annotation/DeclareAnnotation;".equals(constantValue)) {
                      msgHandler.
                          handleErr(new Msg("Found unsupported @DeclareAnnotation (see '"
                                            + type.getName() + "')",
                                            IMessage.WARNING, null,
                                            type.getSourceLocation()));
                  }
                  if ("Lorg/../annotation/Pointcut;".equals(constantValue)) {
                      containsPointcut = true;
                  }
                  ...
                  ...
                  // further operations use containsPointcut

  return theComputedList;
}
```

Listing 6.1: Error reporting from nested expressions.

It shows a snippet of code from `readAj5ClassAttributes()` method, which contains more than 100 lines of code and generates 6 different kinds of error messages. The method extracts deals with class-level annotations. To generate the error message (lines 20 – 25) shown, the compiler traverses the constant pool and checks for certain conditions expressed by test-conditionals in the if-expressions. As AspectJ does not provide pointcuts to select such join points, extracting this kind of error handler would lead to repetition of code related to constant pool traversal and condition-tests. Although this approach supports isolation of error concern from annotation-extraction concern, we believe this is not an efficient way of realizing modularity.

Exposing such join points is another avenue for future work. The pertinent questions to consider are:

- What kinds of code changes are sufficient for realizing our modularity goal?
- Are any new pointcut facilities required to identify such points in program execution?

References

[1] V. H. Allan. Peephole Optimization as a Targeting and Coupling Tool. In *MICRO 22*, pages 112–121. ACM Press, 1989.

[2] A. Appel, L. George, D. MacQueen, J. Reppy, Z. Shao, L. Huelsbergen, E. Gansner, and M. Blume. *Standard ML of New Jersey*. `http://www.smlnj.org`, Last accessed: June, 2009.

[3] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, Second edition, 2002.

[4] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc : An Extensible AspectJ Compiler. In *AOSD '05*, pages 87–98. ACM Press, 2005.

[5] E. L. A. Baniassad, G. C. Murphy, C. Schwanninger, and M. Kircher. Managing Cross-cutting Concerns during Software Evolution Tasks: An Inquisitive Study. In *AOSD '02*, pages 120–126. ACM Press, 2002.

[6] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, 1992.

[7] G. Bracha and W. Cook. Mixin-based inheritance. In *ECOOP '90*, pages 303–311. ACM Press, 1990.

[8] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A Code Manipulation Tool to Implement Adaptable Systems. In *ASF '02*, 2002.

[9] T. Budd. *Introduction to Object Oriented Programming*. Addison-Wesley, Third edition, 2001.

[10] R. G. Burger, O. Waddell, and R. K. Dybvig. Register Allocation Using Lazy Saves, Eager Restores, and Greedy Shuffling. In *PLDI '95*, pages 130–138. ACM Press, 1995.

[11] N. Cacho, F. C. Filho, A. Garcia, and E. Figueiredo. EJFlow: Taming Exceptional Control Flows in Aspect-Oriented Programming. In *AOSD '08*, pages 72–83. ACM Press, 2008.

[12] G. Castagna. Covariance and Contravariance: Conflict without a Cause. *TOPLAS '95*, 17(3): 431–447, 1995.

[13] K. Chen, J. Sztipanovits, and S. Neema. Compositional Specification of Behavioral Semantics. In *Conference on Design, Automation and Test in Europe*, pages 906–911. EDA Consortium, 2007.

[14] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *OOPSLA '00*, pages 130–145. ACM Press, 2000.

[15] A. Colyer and A. Clement. Large-scale AOSD for Middleware. In *Aspect-Oriented Software Development*, pages 56–65. ACM Press, 2004.

[16] A. Colyer, A. Clement, R. Bodkin, and J. Hugunin. Using AspectJ for Component Integration in Middleware. In *OOPSLA '03*, pages 339–344. ACM Press, 2003.

[17] A. Colyer, A. Clement, G. Harley, and M. Webster. *Eclipse AspectJ*. Addison-Wesley, 2004.

[18] R. P. Corbett. *Static Semantics and Compiler Error Recovery*. PhD thesis, University of California, 1985.

[19] B. De Fraine, M. Südholt, and V. Jonckers. StrongAspectJ: Flexible and Safe Pointcut/Advice Bindings. In *AOSD '08*, pages 60–71. ACM Press, 2008.

[20] O. de Moor, S. L. P. Jones, and E. V. Wyk. Aspect-Oriented Compilers. In *GCSE '99*, pages 121–133. Springer-Verlag, 2000.

[21] E. W. Dijkstra. On the Role of Scientific Thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

[22] R. Douence, P. Fradet, and M. Südholt. Composition, Reuse and Interaction Analysis of Stateful Aspects. In *AOSD '04*, pages 141–150. ACM Press, 2004.

[23] Eclipse team. *AJDT: AspectJ Development Tools*. `http://www.eclipse.org/ajdt/`, Last accessed: June, 2009.

[24] S. G. Eick, J. L. Steffen, and E. E. Sumner. Seesoft – A Tool for Visualizing Line Oriented Software Statistics. *Transactions on Software Engineering*, 18(11):957–968, 1992.

[25] Fernando, Castor, A. Filho, C. Garcia, Mary, F, and Rubira. Extracting Error Handling to Aspects: A Cookbook. In *ICSM '07*, pages 134–143. IEEE Computer Society, 2007.

[26] F. C. Filho, N. Cacho, E. Figueiredo, R. Maranh, A. Garcia, and C. M. F. Rubira. Exceptions and Aspects: The Devil is in the Details. In *FSE '06*, pages 152–162. ACM Press, 2006.

[27] R. B. Findler and M. Flatt. Modular Object-Oriented Programming with Units and Mixins. In *ICFP '98*, pages 94–104. ACM Press, 1998.

[28] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2005.

[29] C. W. Fraser and A. L. Wendt. Integrating Code Generation and Optimization. In *CC '86*, pages 242–248. ACM Press, 1986.

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

[31] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing Design Patterns with Aspects: A Quantitative Study. In *AOSD '05*, pages 3–14. ACM Press, 2005.

[32] GCC team. *The GNU Compiler Collection*. `http://gcc.gnu.org/onlinedocs`, Last accessed: June, 2009.

[33] S. Hanenberg and R. Unland. Parametric Introductions. In *AOSD '03*, pages 80–89. ACM Press, 2003.

[34] J. Hannemann and G. Kiczales. Design Pattern Implementation in Java and AspectJ. In *OOPSLA '02*, pages 161–173. ACM Press, 2002.

[35] J. Hannenmann. *Aspect Mining Tool*. `http://hannemann.pbworks.com/Aspect+Mining+Tool`, Last accessed: July, 2009.

[36] E. Hilsdale and J. Hugunin. Advice Weaving in AspectJ. In *AOSD '04*, pages 26–35. ACM Press, 2004.

[37] K. Hogstedt. Automatic Detection and Masking of Nonatomic Exception Handling. *IEEE Transactions on Software Engineering*, 30(8):547–560, 2004.

[38] J. Hugunin. *Guide for developers of the AspectJ compiler and weaver, 2004.* `http://dev.eclipse.org/viewcvs/index.cgi/org.aspectj/modules/docs/developer/compiler-weaver/index.html?revision=1.2&root=Tools_Project`, Last accessed: June, 2009.

[39] Jakarta community. *BCEL Manual.* `http://jakarta.apache.org/bcel/manual.html`, Last accessed: May, 2008.

[40] G. Kiczales. *AOP: The Fun Has Just Begun.* Keynote talk at AOSD, 2003.

[41] G. Kiczales and M. Mezini. Aspect-Oriented Programming and Modular Reasoning. In *ICSE '05*, pages 49–58. ACM Press, 2005.

[42] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect Oriented Programming. In M. Akşit and S. Matsuoka, editors, *ECOOP '97*, pages 220–242. Springer-Verlag, 1997.

[43] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications, 2003.

[44] W. R. LaLonde and J. d. Rivieres. A Flexible Compiler Structure that Allows Dynamic Phase Ordering. In *Compiler Construction*, pages 134–139. ACM Press, 1982.

[45] S. Liang, P. Hudak, and M. Jones. Monad Transformers and Modular Interpreters. In *POPL '95*, pages 333–343. ACM Press, 1995.

[46] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS, 1995.

[47] K. J. Lieberherr and D. Orleans. Preventive Program Maintenance in Demeter/Java. In *ICSE '97*. ACM Press, 2004.

[48] M. Lippert and C. V. Lopes. A Study on Exception Detection and Handling Using Aspect-Oriented Programming. In *ICSE '00*, pages 418–427. ACM Press, 2000.

[49] M. Marin, L. Moonen, and A. van Deursen. FINT: Tool Support for Aspect Mining. In *WCRE '06*, pages 299–300. IEEE Computer Society, 2006.

[50] H. Masuhara, G. Kiczales, and C. Dutchyn. A Compilation and Optimization Model for Aspect-Oriented Programs. *Compiler Construction*, pages 46–60, 2003.

[51] N. McEachen and R. T. Alexander. Distributing Classes with Woven Concerns: An Exploration of Potential Fault Scenarios. In *AOSD '05*, pages 192–200. ACM Press, 2005.

[52] W. M. McKeeman. Peephole Optimization. *Communications of the ACM*, 8(7):443–444, 1965.

[53] B. J. McKenzie. Fast Peephole Optimization Techniques. *Software Practice and Experience*, 19(12):1151–1162, 1989.

[54] R. Miles. *AspectJ Cookbook.* O'Reilly Media, First edition, 2004.

[55] T. D. Millstein and C. Chambers. Modular Statically Typed Multimethods. In *ECOOP '99*, pages 279–303. Springer-Verlag, 1999.

[56] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, 1997.

[57] G. C. Murphy, R. J. Walker, and E. L. Baniassad. Evaluating Emerging Software Development Technologies: Lessons Learned from Assessing Aspect-Oriented Programming. *IEEE Transactions on Software Engineering*, pages 438–455, 1999.

[58] B. C. Oliveira, M. Wang, and J. Gibbons. The Visitor Pattern as a Reusable, Generic, Type-safe Component. In *OOPSLA '08*, pages 439–456. ACM Press, 2008.

[59] D. Orleans and K. J. Lieberherr. DJ: Dynamic Adaptive Programming in Java. In *REFLECTION '01*, pages 73–80. Springer-Verlag, 2001.

[60] M. P. Robillard and G. C. Murphy. Designing Robust Java Programs with Exceptions. In *FSE '00*, pages 2–10. ACM Press, 2000.

[61] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association Aspects. In *AOSD '04*, pages 16–25. ACM Press, 2004.

[62] D. Sarkar, O. Waddell, and R. K. Dybvig. A Nanopass Infrastructure for Compiler Education. In *ICFP '04*, pages 201–212. ACM Press, 2004.

[63] G. L. Steele, Jr. Building Interpreters by Composing Monads. In *POPL '94*, pages 472–492. ACM Press, 1994.

[64] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-In-Time Compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[65] A. S. Tanenbaum, H. van Staveren, and J. W. Stevenson. Using Peephole Optimization on Intermediate Code. *Transactions on Programming Languages and Systems*, 4(1):21–36, 1982.

[66] L. Thabane and C. Ye. Posing the Research Question: Not So Simple. *Canadian Journal of Anesthesiology*, 56(6):71–79, 2009.

[67] A. V.Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Second edition, 2007.

[68] V. Vranic. AspectJ Paradigm Model: A Basis for Multi-paradigm Design for AspectJ. *Generative and Component-Based Software Engineering*, pages 48–57, 2001.

[69] R. J. Walker, E. L. A. Baniassad, and G. C. Murphy. An Initial Assessment of Aspect-Oriented Programming. In *ICSE '99*, pages 120–130. ACM Press, 1999.

[70] A. Warth. LazyJ: Seamless Lazy Evaluation in Java. In *FOOL/WOOD '07*, pages 472–492. ACM Press, 2007.

[71] D. Watt, D. Brown, and R. W. Sebesta. *Programming Language Processors in Java: Compilers and Interpreters AND Concepts of Programming Languages*. Prentice Hall Press, 2007.

[72] X. Wu. *Component Based Language Implementation With Object Oriented Syntax and Aspect Oriented Semantics*. PhD thesis, University of Alabama, 2007.

[73] Yourkit Inc. *Java Profiler*. `http://www.yourkit.com/overview/index.jsp`, Last accessed: July, 2009.

[74] W. Zadrozny. On Compositional Semantics. In *Conference on Computational Linguistics*, pages 260–266. Association for Computational Linguistics, 1992.

Appendix A

Development Aspects

This appendix shows development aspects used during aspect-oriented modularization of peephole optimzation and error handling.

A.1 Optimization Category

This aspect is used to identify places where `Instructions` and `InstructionLists` are accessed, or initialized.

```
/**
 * Aspect to identify places where instructions and instruction lists are
 * accessed, or initialized.
 */
public privileged aspect SeekOptimizationSites {
    /**
     * Limit the scope of our search to a package
     */
    pointcut includeScope():
                within(org.aspectj.weaver.bcel..*);

    /**
     * Places where instructions and instructionlists
     * are accessed or initialized
     */
    pointcut accessSites():
                get (InstructionList *)
                || set (InstructionList *)
                || get (Instruction *)
                || set (Instruction *);

    /**
     * Exclude join points in test suites
     */
   pointcut excludeScope():
                !withincode(* *.suite(..));

    /**
     * Composition of all pointcuts of interest
     */
    pointcut potentialSites1():
                includeScope()
                && excludeScope()
                && accessSites();

    /**
     * Simple advice to examine the affected places
     */
    after():
                potentialSites1(){
                    System.out.println("Potential site for peephole optimization");
                }
}
```

Listing A.1: Development aspect to locate initialization and access of `Instructions`.

This aspect is used to identify places where `Instructions` and `InstructionLists` are created. We exclude from consideration those join points which are part of test suites and java langauge and utility packages.

```
/**
 * Aspect to identify places where instructions and instruction lists are
 * created, excluding those in test suites and java core
 */
public privileged aspect DevelopmentAspectSeekSites {
    /**
     * Limit the scope of our search to a package
     */

    pointcut includeScope():
                within(org.aspectj.weaver.bcel..*);

    /**
     * Places where instructions and instructionlists
     * are created
     */
    pointcut creationSites():
                   call (InstructionList.new(..))
                || call (Instruction.new(..));

    /**
     * Exclude join points in test suites
     */
    pointcut excludeScope():
                !within(SeekOptimizationSites)
             && !withincode(* *.suite(..))
             && !call (* java.util..*.*(..))
             && !call (* java.lang..*.*(..))
             && !call (* org.aspectj.weaver.TestUtils.*(..))
             && !call (* org.aspectj..*.asm..*.*(..));

    /**
     * Composition of all pointcuts of interest
     */
    pointcut potentialSites2():
                includeScope()
             && excludeScope()
             && creationSites();
    /**
     * Simple advice to examine the affected places
     */
    after():
                potentialSites2(){
                    System.out.println("Potential site for peephole optimization");
                }
}
```

Listing A.2: Development aspect to locate creation of `Instructions`.

A.2 Error Handling Category

This aspect identifies places in ajc that throw or handle exceptions, and access error print streams.

```
/**
 * An aspect to identify potential places of error- and exception-handling
 */
aspect SeekAppExceptions {

    /**
     * Limit the scope of our search to a package
     */
    pointcut withinScope() : within(org.aspectj.weaver..*);

    /**
     * Find places from where Exceptions are raised
     */
    declare warning : withinScope() &&
        (call(* *(..) throws Exception+)
         || call(new(..) throws Exception+)) :
        "Any method or constructor call throwing an exception";

    /**
     * Find catch clauses handling Exception
     */
    declare warning : withinScope() && handler(Exception+):
        "Exception handling code";

    /**
     * Find any acess or use of an error stream
     */
        pointcut accessErrorStreams():
                set(* java.lang.System.err)
                || get (* java.lang.System.err)
                || set (* java.lang.System.out)
                || get (* java.lang.System.out)
                || set (* Message.ERROR)
                || get (* Message.ERROR);

        after(): accessErrorStreams()
                 && withinScope() {
                     System.out.println("Error message from: "
                                        + thisJoinPoint.toShortString());
                 }
    /**
     * Find calls to customized error handlers
     */
        pointcut customizedErrorHandlers():
                call (* reportError(..))
                || call (* reportWarning(..)
                || call (* handleMessage(..));

        after(): customizedErrorHandlers()
                 && withinScope() {
                     System.out.println("Customized Error handler at:"
                                        + thisJoinPointStaticPart.getSourceLocation());
        }
}
```

Listing A.3: Development aspect to locate error handling sites.

Appendix B

Policy Enforcement Aspects

In this appendix, we present those aspects which enforce ordering of different aspects, and act as safety checks for future extensions, when new aspects are added.

B.1 Optimization Category

This aspect describes rules to ensure that calls to various optimizers are done safely within intended modules only. By generating error reports when optimization is done locally, outside of optimizing aspects, this aspect acts as a safety check for implementation of the optimizer as a separate unit. This is crucial because all sets of optimization are intended to be pluggable.

```
/**
 * Rules to ensure that calls to optimizers are done safely
 * Acts as safety check for future extensions
 */
privileged aspect CompilationAdviceRecipe issingleton()
{
    /**
     * NOP removal must not be local
     * Globally handled by RemoveNops module
     */
    declare error:
              call (* *.stripNops())
              && !within(RemoveNops):
                        "Nops must be removed only from RemoveNops aspect";
    /**
     * Peephole optimization must be done from PPOptimizer module
     */
    declare error:
              call (* *.filter())
              && !within(PPOptimizer):
                        "PPO must be done by PPOptimizer aspect";
    /**
     * Remove fall through GOTOs from UpdateGoToLabels aspect
     */
    declare error:
              call (* *.stripFallThroughGotos())
              && !within(UpdateGoToLabels):
                        "Unnecessary GoTo labels should only be removed from method body";
}
```

Listing B.1: Aspect to enforce isolation of optimizations from core compiler.

This aspect defines the order in which different optimizers should be applied.

```
/**
 * Aspect defining the order of our optimizations
 */
public aspect OptimizationOrder issingleton() {

        /**
         * We decide to carry out optimization in following order
         */

        declare precedence :
                    RemoveUnreachableCode,
                    PPOptimizer,
                    RemoveNops,
                    UpdateGoToLabels,
                    *;

}
```

Listing B.2: Aspect defining precedence between various optimizations.

B.2 Error Handling Category

In this section, we present aspects enforcing our desired behavior regarding error handling.

The first aspect `RestrictErrorHandlingToAspects` warns compiler writers upon any attempts to perform error handling locally within different modules, responsible for other core functions. It ensures that error handling is done only by aspects dedicated for this purpose.

```
/**
 * An aspect to enforce that error-handling is not performed locally
 */
aspect RestrictErrorHandlingToAspects {

    /**
     * Limit error- and exception-handling to these aspects.
     */
    pointcut allowedScope() :
             within(org.aspectj.weaver.ParserErrorHandlingAspect)
             || within(org.aspectj.weaver.TypeErrorHandlingAspect)
             || within(org.aspectj.weaver.WeavingErrorHandlingAspect)
             || within(org.aspectj.weaver.MungingErrorHandlingAspect);

    /**
     * Emit warning if exception throws occur out of permitted modules
     */
    declare warning :
             !allowedScope()
             && (call(* *(..) throws Exception+)
                 || call(new(..) throws Exception+)) :
                     "Error handling should be done globally";

    /**
     * Find any acess or use of an error stream
     */
        pointcut accessErrorStreams():
                 set(* java.lang.System.err)
                 || get (* java.lang.System.err)
                 || set (* java.lang.System.out)
                 || get (* java.lang.System.out)
                 || set (* Message.ERROR)
                 || get (* Message.ERROR);

    /**
     * Find calls to customized error handlers
     */
        pointcut customizedErrorHandlers():
                 call (* reportError(..))
                 || call (* reportWarning(..)
                 || call (* handleMessage(..));

    /**
     * Emit warning if error reports are generated outside of permitted modules
     */
    declare warning :
              !allowedScope()
              && accessErrorStreams()
              && customizedErrorHandlers():
                      "Local error reporting is not permitted";

}
```

Listing B.3: Aspect to generate warning at the sites of local error-handling.

Appendix C

Optimization Methods in Detail

Here, we present in detail the methods related to different kinds of optimizations that we performed.

The `optimizeInstList(..)` walks over `Instruction`s and `InstructionList`s and tries to find any potential places for peephole optimization.

C.1 Optimization of `InstructionLists`

```
/**
 * Slide through window of two consecutive instructions and look for
 * potential patterns for optimization
 *
 * Filter: static class holding information about new instructions
 * that replace peephole patterns
 *
 * This method is part of PPOOptimizer aspect
 */

public InstructionList optimizeInstList(InstructionList body) {
    InstructionHandle curr, next;
    curr = body.getStart();
    while (curr != null) {
        next = curr.getNext();
        if (next == null)
            break;
        next = curr.getNext();
        Instruction currinst = curr.getInstruction();
        Instruction nextinst = next.getInstruction();
        Filter peep = null;
        peep = filter(currinst, nextinst);
        if (peep != null) {
            if (peep.replace.length == 0) {
                System.out.println("Eliminate"+ currinst + " to"+ nextinst);
            } else {

                System.out.println("Eliminate"+ currinst + " to"+ nextinst);
                System.out.println(" with");

                for (int j = 0; j < peep.replace.length; j++) {
                    System.out.println(" " + peep.replace[j]);
                }
            }
            try {
                body.delete(next);
            } catch (TargetLostException e) {
                System.out.println("Invalid deletion attempt");
            }
            try {
                body.delete(curr);
            } catch (TargetLostException e) {
                System.out.println("Invalid deletion attempt");
            }

            for (int k = peep.replace.length - 1; k >= 0; k--) {
                body.insert(next, peep.replace[k]);
            }
        }
        curr = next;
    }
    return body;
}
```

Listing C.1: Method managing optimization of instruction lists.

C.2 Removal of NOPs

This method is responsible for removing NOPs and then, managing the InstructionHandles appropriately.

```
/**
 * Remove NOPs and update the oldTargeters to target the next instruction
 */
public void LazyMethodGen.stripNops() {

    InstructionHandle curr = body.getStart();
    while (true) {
        if (curr == null)
            break;
        InstructionHandle next = curr.getNext();
        if (next == null)
            break;
        if (curr.getInstruction().getOpcode() == Constants.NOP) {
            body.delete(curr); // curr.removeAllTargeters();
            InstructionTargeter[] targeters = curr.getTargetersArray();
            if (targeters != null) {
                for (int i = 0, len = targeters.length; i < len; i++) {
                    InstructionTargeter targeter = targeters[i];
                    targeter.updateTarget(curr, next);
                }
            }

        }
        curr = next;
    }
}
}
```

Listing C.2: Method for removing NOP instructions.

C.3 Removal of unreachable code

This method removed unreachable pieces of code, and then manages the `InstructionHandles` appropriately.

```
/**
 * Inter-type declared method for removing unreachable code
 */
public void LazyMethodGen.stripUnreachableCode() {
    InstructionList mutableBody = body.copy();
    final HashMap<Object, Integer> labelPos = new HashMap<Object, Integer>();
    InstructionHandle currentih = body.getStart();
    final Instruction currentinst = currentih.getInstruction();
    int i = 1;
    Iterator iIter = body.iterator();
    while (iIter.hasNext()) {
        final Object ce = iIter.next();
        InstructionHandle insthan = (InstructionHandle)ce;
        if (insthan.hasTargeters()) {
            labelPos.put(ce, new Integer(i));
            i++;
        }
    }

    /**
     * Visit the blocks depth-first.
     * Stack of Labels that begin blocks that have been visited
     */
    final Set<Instruction> visited =
        Collections.synchronizedSet(new HashSet<Instruction>());

    /**
     * Stack of Labels that begin blocks that have not been visited.
     */
    Stack<Instruction> stack = new Stack<Instruction>();
    if (! body.isEmpty()) {
        visited.add(currentinst);
        stack.push(currentinst);
    }
    while(! stack.isEmpty()){
        Instruction currentlabel = null;
        try {
            currentlabel = (Instruction)stack.pop();
        } catch (ClassCastException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        Integer labelIndex = (Integer)labelPos.get(currentlabel);
        if(labelIndex == null)
            break;
        try {
            i = labelIndex.intValue();
        } catch (NullPointerException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        for (InstructionHandle ih = body.getStart(); ih != null; ih = ih.getNext()){
            Instruction inst = ih.getInstruction();
            if (!ih.hasTargeters()){ //ADDED NOW
                if(inst.isReturnInstruction() ||
                   inst.getOpcode() == Constants.ATHROW){
                    break;
                }
                    ... //continued in next page
```

Listing C.3: Inter-type declared method for removing unreachable code.

```
60                  ... //continued from previous page}
                    else if (inst.isJsrInstruction() ||
62                          (Constants.instFlags[inst.getOpcode()] & Constants.IF_INST) != 0){
                        if(!visited.contains(currentlabel)){
64                          visited.add(currentlabel);
                            stack.push(currentlabel);
66                      }
                    else if (inst.isGoto()){
68                      currentlabel = inst;
                        if(!visited.contains(currentlabel)){
70                          visited.add(currentlabel);
                            stack.push(currentlabel);
72                      }
                        break;
74                  }
                    else if(inst.isReturnInstruction()){
76                      break;
                    }
78
                    else if (inst.getOpcode() == Constants.TABLESWITCH
80                          || inst.getOpcode() == Constants.LOOKUPSWITCH){
                        final InstructionHandle[] switchtargets =
82                          ((InstructionSelect)inst).getTargets();
                        for (int j = 0, slen = switchtargets.length; j < slen; j++){
84                          currentlabel = switchtargets[j].getInstruction();

86                          if(!visited.contains(currentlabel)){
                                visited.add(currentlabel);
88                              stack.push(currentlabel);
                            }
90                      }
                        break;
92                  }
                } else if (ih.hasTargeters()){
94                  currentlabel = inst;
                    visited.add(currentlabel);
96              }
            }
98      }
    boolean reachable = false;
100     Iterator instIter = mutableBody.iterator();
        while (instIter.hasNext()) {
102         InstructionHandle inhToObj = (InstructionHandle)instIter.next();
            if (inhToObj.hasTargeters()) {
104             Instruction forLookUpInVisited = inhToObj.getInstruction();
                if (!visited.isEmpty()) { //HashSet
106                 reachable = visited.contains(forLookUpInVisited);
                }
108         }
            else if (!reachable){
110             if(body.contains(inhToObj)){
                    try {
112                     body.delete(inhToObj);
                    } catch (TargetLostException e) {
114                     throw new BCException("should not happen");
                    }
116             }
            }
118     }
```

Listing C.4: Inter-type declared method for removing unreachable code (contd.).

```
/** Filter represents a set of instructions that result from a peephole optimisations. */
static class Filter {
  Instruction[] replace;
  Filter() {
      replace = new Instruction[0];}
  Filter(Instruction replace) {
      this.replace = new Instruction[] { replace };}
  Filter(Instruction replace1, Instruction replace2) {
      replace = new Instruction[] { replace1, replace2 };}
}
public static Filter filter(Instruction first, Instruction second){
  switch (second.getOpcode()){
  case Constants.SWAP:
      if(first.getOpcode() == Constants.SWAP){ //Eliminate swap-swap
          return new Filter();
      }
      if(first.getOpcode() == Constants.DUP){    //swap means nothing if it's after a dup.
          return new Filter(first);
      }
      break;
      ...
  case Constants.POP2:
      switch(first.getOpcode()){
      case Constants.LDC:
          Assert.isTrue((first.getType() == Type.LONG) || (first.getType() == Type.DOUBLE),
                        "Unable to pop2 a 1-word operand");
      case Constants.LLOAD: // Fall through
          return new Filter();  // Eliminate push and pop
      }
      break;
      ...
  case Constants.IF_ICMPNE:
      // Replace ldc 0-if_icmpne with ifne
      if (first.getOpcode() == Constants.LDC) {
          if (first.isConstantInstruction()) {
              if (first.getValue().intValue() == 0) {
                  return new Filter(new InstructionBranch(Constants.IFNE,
                                                          second.getValue().intValue()));
              }
          }
      }
      break;
      ...
          switch (second.getOpcode()) {
              // Replace store-load with dup-store, load-load with load-dup
          case Constants.ILOAD:
              if (first.getOpcode() == Constants.ISTORE) {
                  if (first.equals(second)) {
                      return new Filter(new Instruction(Constants.DUP),first);
                  }
              }
              if (first.getOpcode() == Constants.ILOAD) {
                  if (first.equals(second)) {
                      return new Filter(first, new Instruction(Constants.DUP));
                  }
              }
              break;
              return null;
          }
  }
```

Listing C.5: Implementation of pattern matching and replacement for peephole optimization.

Appendix D

Design Structure Matrices

In this chapter, we show the Design Structure Matrices for the ajc in order to demonstrate the impact of aspect-oriented modularization on dependencies between different classes. This figure is automatically generated by Lattix Design Structure Matrix tool.

This DSM depicts coupling between classes of `weaver` package before modularization of error-handling concerns.

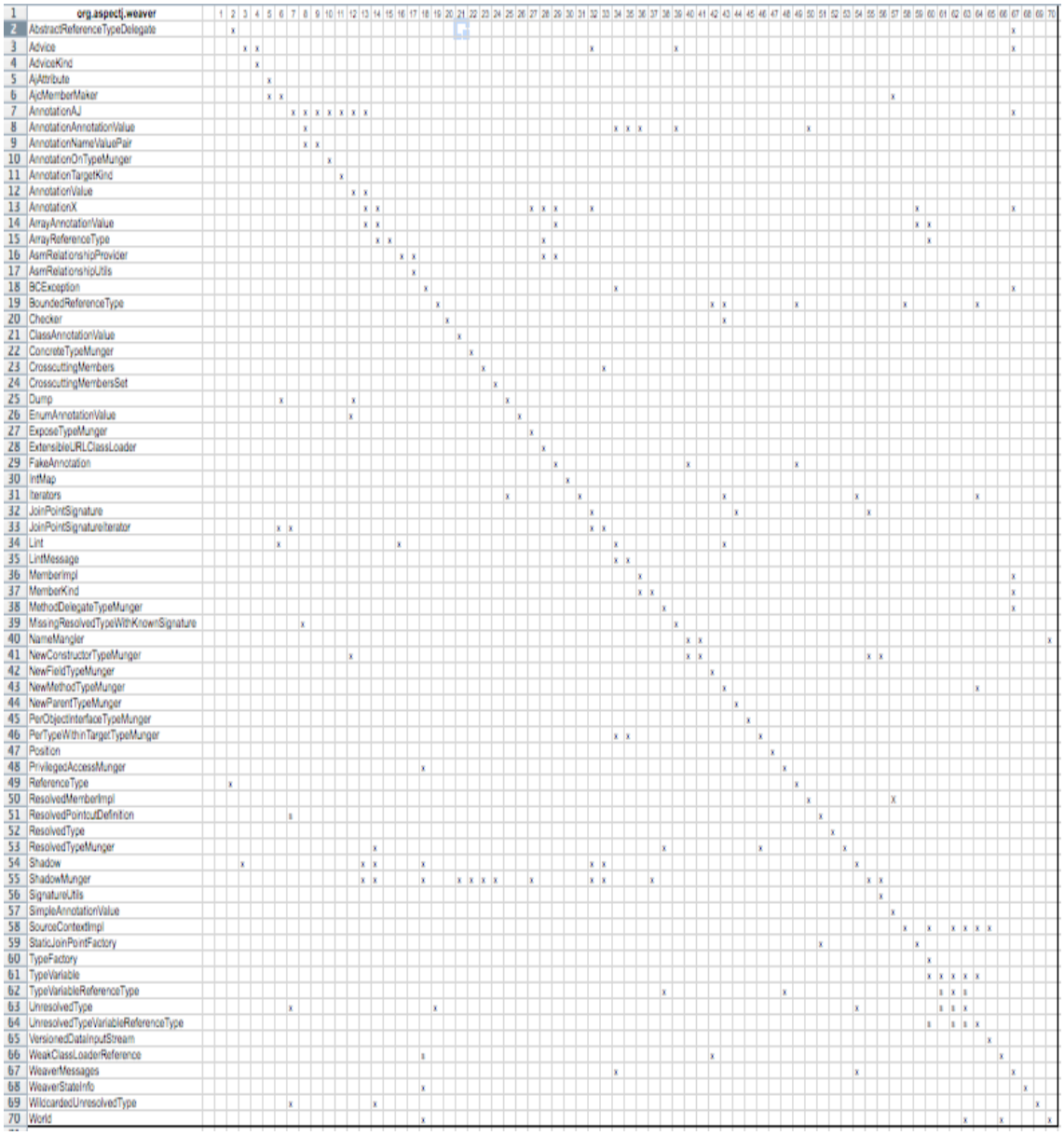

Figure D.1: DSM for `weaver` package before error-handling modularization.

This DSM depicts coupling between classes of `weaver` package after modularization of error-handling concerns.

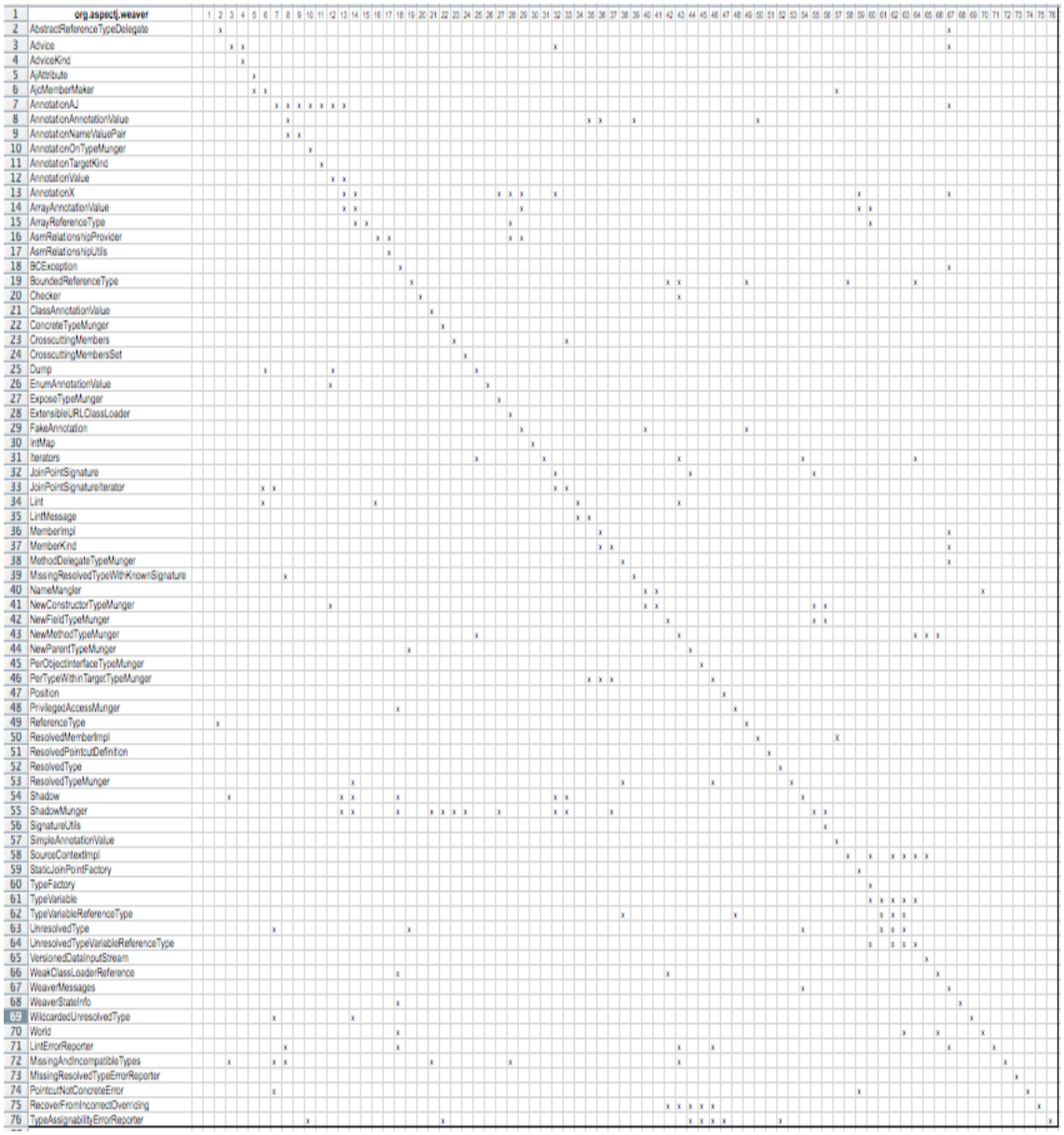

Figure D.2: DSM for `weaver` package after error-handling modularization.