Tool Support For Understanding and Diagnosing Pointcut Expressions

by

Lingdong Ye

B.Eng., Southwest University of Finance and Economics, 1998
B.Sc., McMaster University, 2005

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University Of British Columbia

August 31, 2007

Abstract

Writing correct AspectJ pointcuts is hard. This is partly because of the complexity of the pointcut language and partly because it requires understanding how a pointcut matches across the entire code base.

In this thesis, we present algorithms that compute two kinds of useful information that can help AspectJ developers diagnose and fix potential problems with their pointcuts. First, we present an algorithm to compute almost matched join points. Second we present algorithms to compute explanations of *why* a pointcut does not match (or does match) a specific join point.

We implemented two tools using these algorithms. The first is an offline tool that analyzes a code base and produces a comprehensive report. Using this tool, we were able to find several real problems in existing, medium-sized AspectJ code bases.

The second tool is an Eclipse plugin called PointcutDoctor. PointcutDoctor is a natural extension of AJDT, the mainstream IDE for AspectJ. It provides developers easy access to the same information from within their already familiar development environment.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I thank my supervisors Kris De Volder and Gregor Kiczales for their guidance, support and encouragement that make this work possible. Thanks to them for being supportive for me to work on the thesis in another city. Thanks to them for all the insightful discussions that will continue to guide my whole career life.

I thank Gail Murphy for being my second reader and giving me invaluable comments.

Thanks to all the great friends I've met in Vancouver for making it such an amazing experience.

Chapter 1

Introduction

1.1 Thesis Statement

In this thesis, we present algorithms to compute two kinds of information about AspectJ [14] pointcuts: almost matched join points and explanation on why a pointcut does not match (or does match) a given join point. We claim that this information can help a developer find problems in existing pointcuts. We also claim that this information can be added to an existing integrated development environment (IDE) in a logical and clean way.

In the following sections, we first introduce the background information about AspectJ, Aspect-Oriented Programming, pointcuts and current IDE support for writing pointcuts. We then illustrate that pointcuts are hard to write, and that current AspectJ IDE tools do not provide sufficient support for writing AspectJ pointcuts. Finally, we explain how we will prove our thesis and outline the key contributions of this thesis.

1.2 AspectJ and Aspect Oriented Programming

AspectJ is an extension to the Java programming language that provides support for Aspect-Oriented Programming (AOP) in a Java-like syntax. Aspect-Oriented Programming [15] is a new programming paradigm that attempts to improve modularity of software systems. It provides support for modularizing crosscutting concerns using a new language construct, aspect. Crosscutting concerns refer to concerns that cut across other concerns, and thus, conflict with them in terms of system decomposition. Crosscutting concerns cannot be well modularized using mainstream programming methodologies, such as procedural programming and object-oriented programming. Using mainstream programming methodologies, if we try to modularize crosscutting concerns, other concerns will be scattered across many modules.

We next illustrate the concepts of crosscutting concern and pointcut using the code of a simple figure editor. In the following code, class `Shape` is

the common super class for all shapes in the system. There are two kinds of shapes in the system, `Point` and `Line`. Class `Display` manages the shapes drawn on the screen.

```
abstract class Shape {
  Color color;
  public void setColor(Color c) {
    this.color = c;
  }
  ...
}

class Point extends Shape {
  int x, y;
  public void setX(int x) {
    this.x = x;
  }
  public void setY(int y) {
    this.y = y;
  }
  ...
}

class Line extends Shape {
  Point p1, p2;
  public void setP1(Point p) {
    this.p1 = p;
  }
  public void setP2(Point p) {
    this.p2 = p;
  }
  ...
}

class Display {
  public static void refresh() {
    ...
  }
  ...
}
```

Let's consider the implementation of a display-refresh concern: to refresh the display whenever any type of shape changes. Its implementation in Java requires adding a line `Display.refresh()` in *every* method that renders a change of any type of shape. These methods include `Shape.setColor(Color)`, `Point.setX(int)`, `Point.setY(int)`, `Line.setP1(Point)` and `Line.setP2(Point)`. We say this display-refresh concern is a crosscutting concern because it cuts across the existing system decomposition. As a result, its implementation in Java scatters across multiple modules, i.e. classes and methods.

In AspectJ, we can modularize this concern using the following *aspect* without the need to modify any class above:

```
1.  aspect DisplayRefreshing {
2.    // The Pointcut
3.    pointcut shapeChange(): execution(void Shape.setColor(Color)) ||
4.      execution(void Point.setX(int)) ||
5.      execution(void Point.setY(int)) ||
6.      execution(void Line.setP1(Point)) ||
7.      execution(void Line.setP2(Point));

8.    // The Advice
9.    after():shapeChange() {
10.      Display.refresh();
11.   }
12. }
```

In the above code, line 3-7 declares a *pointcut* that captures execution of all methods that will change the state of a shape. Pointcut is an AspectJ language construct that is used to pick out *dynamic join points*, well-defined events in the execution of a program, such as method execution, object instantiation and field access. Every dynamic join point has a corresponding static shadow in the source code or byte code of the program, which is called *join point shadow*[13]. For simplicity, in this thesis, when we say that a pointcut matches a join point shadow, we mean that the pointcut matches the join points that correspond to the join point shadow. Primitive pointcuts such as `call`, `execution` and `set` specify the kind and other characteristics of the expected join points. Primitive pointcuts can be combined using logical operators including `&&` (AND), `||` (OR) and `!` (NOT). The example pointcut above has the semantics "to pick up join points at the execution of method `Shape.setColor(Color)`, or execution of method `Point.setX(int)` or ...".

Line 9-11 is an *advice* that defines the behavior after the state of a shape is changed: to refresh the display. Advice is an AspectJ language construct that defines the behavior of the aspect at the the join points specified by the pointcut.

1.3 Existing IDE Support for Writing AspectJ Pointcuts

It is important to note that tool support is essential for effectively working with AspectJ code and pointcuts in particular. First, aspects potentially affect the entire code base. The AspectJ pointcut language is powerful and is designed to be capable of picking up join points arising from the entire code base. For instance, in the previous example we can write a pointcut `execution(* *.set*(..))` and an advice that refreshes the display when *every* method whose name starts with string "set" gets executed, no matter in which class those methods are declared. Second, there is nothing in the source code of those affected methods that describes or refers to the pointcut or advice. This absence of information has been called obliviousness [11][12][16][8] by the AOP community. The obliviousness characteristic of AOP is necessary for modularizing crosscutting concerns, because it allows crosscutting concerns to be expressed in separate modules without modifying the code that they crosscut with. But controversially, it also makes the code harder to read and maintain. Tools that explicitly reveal and display the correlation between aspects and the code affected by them help to improve the readability and maintainability of AOP code.

AspectJ Development Tools (AJDT) [1], a plug-in of the Eclipse platform [3], is a mainstream IDE for AspectJ. AJDT provides support for writing pointcuts by showing a list of advisees for the selected advice (i.e. matches for the pointcut) in a Cross References View. A screenshot of AJDT with its Cross References View open is shown in Figure 1.1. AJDT also issues some compiler warning messages for several cases of suspicious code, for example, when the pointcut of an advice does not match any join points (as shown in Figure 1.2).

In the next section, we are going to see why such information is insufficient for a developer to ascertain the correctness of her pointcuts, as well as to diagnose the problems in her pointcuts.

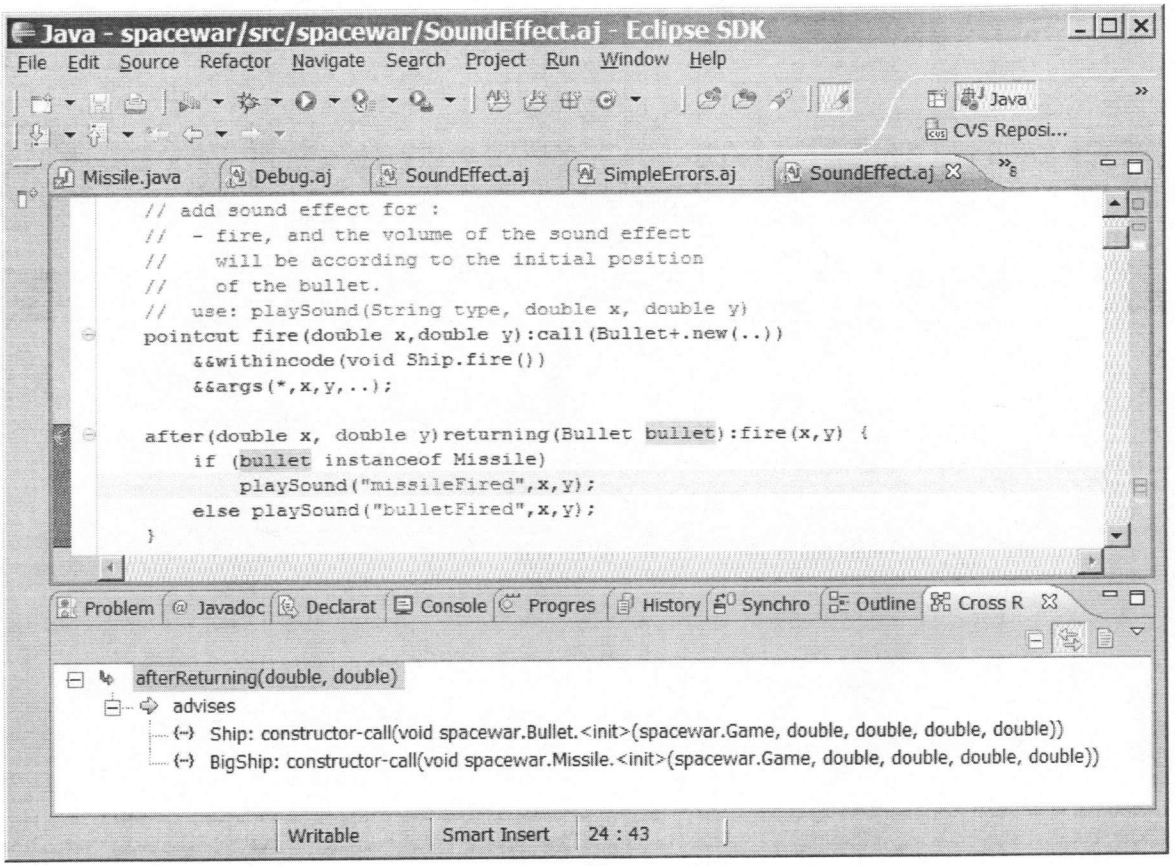

Figure 1.1: AJDT's Cross References View showing a list of matches

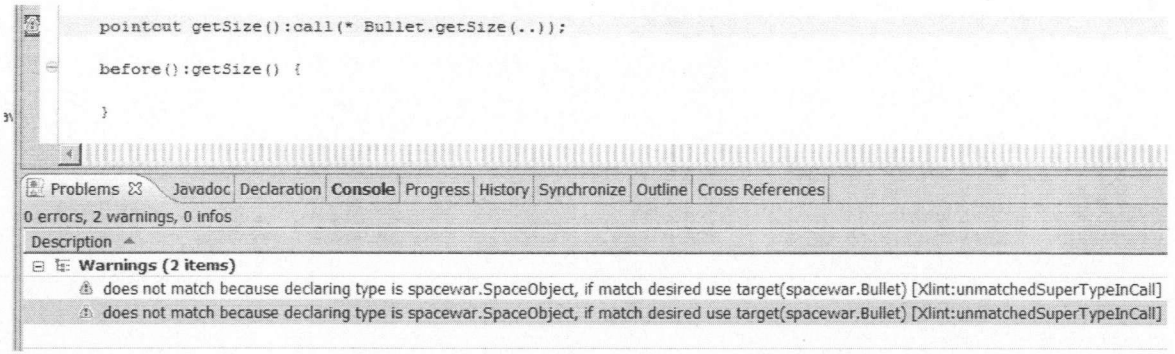

Figure 1.2: One of AJDT's warning messages if no match

1.4 Motivating Examples

To write a correct pointcut, a developer needs to understand how the pointcut matches across her entire code base, as well as the subtleties of the pointcut language. In this section we provide two examples to illustrate these two requirements respectively.

1.4.1 Example 1

Ascertaining that a pointcut is correct in a given code base requires a global understanding of the code at a level of detail that is not easy to obtain or remember for developers. To assist developers, AJDT provides an explicit representation of the join points a pointcut matches in a given code base, i.e., a list of matches in its Cross References View. We believe that this information is insufficient for a developer to determine a pointcut's correctness. We illustrate this problem with a concrete example which was taken from [17]:

```
pointcut connectionCreation(String url,
  String username,String password)
     : call(public static Connection
          DriverManager.getConnection(String,
             String, String))
          && args(url, username, password);
```

This pointcut matches the calls to method `getConnection` of `DriverManager` that have three `String` parameters. The `args` pointcut binds the three parameters of the method call to variables so that they can be used later. Assume that a developer formulated this pointcut to capture the creation of database connections by calling related methods in `DriverManager` class. How would the developer know that this pointcut is correct? Since the intention is to capture *all* creations, verifying correctness means ascertaining that there are no accidentally missed creation sites. AJDT's cross references view shows which join point shadows produce join points that are matched by the pointcut, but it does not show any explicit information about join points that are *not* being matched. Consequently the view does not help to discover unintended misses. Indeed, it is hard for a developer to scan a list of matches and realize something that should be there is not. However this kind of determination is often critical in understanding whether a pointcut is correct. For example, there are three `getConnection` methods declared in `DriverManager` that can be used to create database connections:

```
DriverManager.getConnection(String,String,String)
DriverManager.getConnection(String, Properties)
DriverManager.getConnection(String)
```

Only calls to the first of these 3 methods are matched by the example pointcut. The pointcut is therefore incorrect, since it fails to match calls to the other two. Because AJDT's cross references view provides no explicit information about non-matched join points, it provides little help to discover this important fact. As a result the bug in the pointcut is likely to go unnoticed. A tool that shows explicitly information about unmatched join points could help developers avoid this kind of unintended misses.

1.4.2 Example 2

Another difficulty with regard to pointcut writing is the complexity of the pointcut language semantics itself. Consider for example the following pointcut:

```
pointcut fire():
    call(Bullet.new(..))
    && withincode(void Ship.fire())
```

Will the following constructor calls be matched by this pointcut?

```
class BigShip extends Ship {
  public void fire() {
    Bullet b1 = new Bullet(null, 0, 0);          //(1)
    Bullet b3 = new Missile(null, 0, 0);         //(2)
  }
}
```

Intuitively, this pointcut matches calls to the constructor of `Bullet` in the lexical scope of `Ship.fire()`. However, the reality of how the pointcut actually matches in this eample might be surprising to some developers: the join point corresponding to (1) is matched, while the join point corresponding to (2) is not. First of all, both calls pass the test `withincode(void Ship.fire())`, though neither of them are in `Ship.fire`. This is because `withincode` not only matches join point shadows within the specified method, but also *implicitly* matches join points shadows within all the methods that override it, so (1) is matched. Second, although `Missile` is a subclass of `Bullet`), (2) is not matched. This is because the `call` pointcut does not match any calls to constructors for subtypes of the type specified in

the pointcut. These and other complexities of the AspectJ pointcut language and the Java language make writing correct pointcuts difficult. AJDT only provides limited support for explaining a pointcut, e.g. the warning message shown in Figure 1.2. We believe that a tool that offers an explanation of why a certain join point is matched or not matched could help developers diagnose problems with their pointcuts.

1.5 Algorithms to Compute Useful Information

The examples in the previous section motivate that it would be useful to show information about non-matched join points to developers as well as explanations about why a pointcut matches or does not match certain join points.

This thesis contributes algorithms to compute such information. Our first algorithm computes almost matched join points. We use a technique we call *pointcut relaxation* that uses simple heuristics to provide a more nuanced notion of "almost matched" than existing algorithms. Our algorithm also computes *virtual join point shadows* to handle possible evolutions of the code base.

We also contribute several algorithms that compute two kinds of explanations to help a developer understand and diagnose problems in her pointcuts. The first computes a color-coded highlighting of a pointcut expression that shows which parts of the pointcut expression are responsible for the non-match(or match). The second produces a text message explaining why a highlighted part matches (or does not match) the given join point.

We claim that the information discussed above can help a developer find problems in her pointcuts. We also claim that this information can be added to existing IDE in a logical and clean way.

To prove our first claim, we developed an offline tool that produces a comprehensive report about all the pointcuts in a given code base. We applied this tool to the code bases of two medium-sized AspectJ projects in different domains (AspectJ refactored jEdit [9] and ORBacus [18]). We found that even in a relatively stable code base developed by experienced programmers there are still problems in the pointcuts due to programmers' incomplete knowledge of the code base or confusion about the AspectJ pointcut language. This shows that the information computed by our algorithms can help a developer find problems in her pointcuts.

To prove our second claim, we developed an Eclipse plugin called Point-

cutDoctor[1]. PointcutDoctor extends AJDT seamlessly to provide developers easy access to the same information from within their familiar development environment. This shows that this information can be added to existing IDE in a logical and clean way.

1.6 Contributions

This thesis makes the following contributions:

- We present our pointcut relaxation algorithm to compute almost matched join point shadows. Our algorithm is more sophisticated and provides a more nuanced notion of "almost matched" than the *boundary join points* technique previously proposed by Anbalagan and Xie [5]. We will discuss these differences in detail in Chapter 3.
- We present algorithms that explain *why* a pointcut does not match (or does match) join points produced by a given join point shadow and provide improvement suggestions.
- We developed a user interface that is a natural extension to AJDT on Eclipse and shows this information to developers from within their familiar development environment.
- We show that the information on almost matched join points and the explanations computed by our algorithms can be used to find problems with pointcuts in real AspectJ code.

1.7 Overview of the Thesis

The remainder of this thesis is organized as follows: Chapter 2 presents PointcutDoctor, our IDE tool implemented as an extension of AJDT. This shows how the information our algorithms compute can be made accessible from within the IDE, and thus supports our second claim. We choose to discuss about PointcutDoctor first because it will give readers a more intuitive overview of the idea behind this thesis by seeing how the information our algorithms compute is used in a concrete IDE. Chapter 3 discusses our algorithm and heuristics to compute almost matched join point shadows. Chapter 4 gives the pointcut explanation algorithm as well as the heuristics for providing the detailed textual explanation. Chapter 5 presents our case

[1]PointcutDoctor: http://pointcutdoctor.cs.ubc.ca

studies on two medium sized code base. Chapter 6 presents some of the limitations of our algorithms, of their current implementation and of the evaluation presented in this thesis, and some ideas on how these could lead to future work. Chapter 7 presents our conclusions.

Chapter 2

PointcutDoctor: An Extension of AJDT

In this chapter, we demonstrate PointcutDoctor, an extension to AJDT that provides developers two kinds of information about AspectJ pointcuts: almost matched join points and an explanation on why a pointcut does not match (or does match) a given join point. We can see how PointcutDoctor adds the information to existing IDE in a logical and clean way. This proves our second claim of our thesis. We choose to discuss the IDE tool first because it provides an intuitive overview of the idea behind this thesis by seeing how the information our algorithms compute is used in a concrete IDE.

2.1 PointcutDoctor

The user interface of PointcutDoctor is designed to be non-disruptive to the users who are already familiar with AJDT. All features are integrated into the current AJDT user interface without introducing any new views. The existing AJDT interface is preserved and extended with some additional behavior in a clean and logical way.

The main point where PointcutDoctor functionality has been added is in the cross references view. The two major features are discussed below.

2.1.1 Almost matched join point shadows

First, as discussed in Chapter 1, the original AJDT cross references view only displays information about matched join point shadows (see Figure 1.1). As shown in Figure 2.1, PointcutDoctor extends this with an additional list of almost matches. Notice also that some of the almost matched shadows are marked as "virtual". Our algorithm produces virtual shadows for calls to methods that have been declared but for which no actual calls exist within the code base. The rationale behind this is that even though such calls do not exist in the current code base it is possible, and perhaps likely, that they

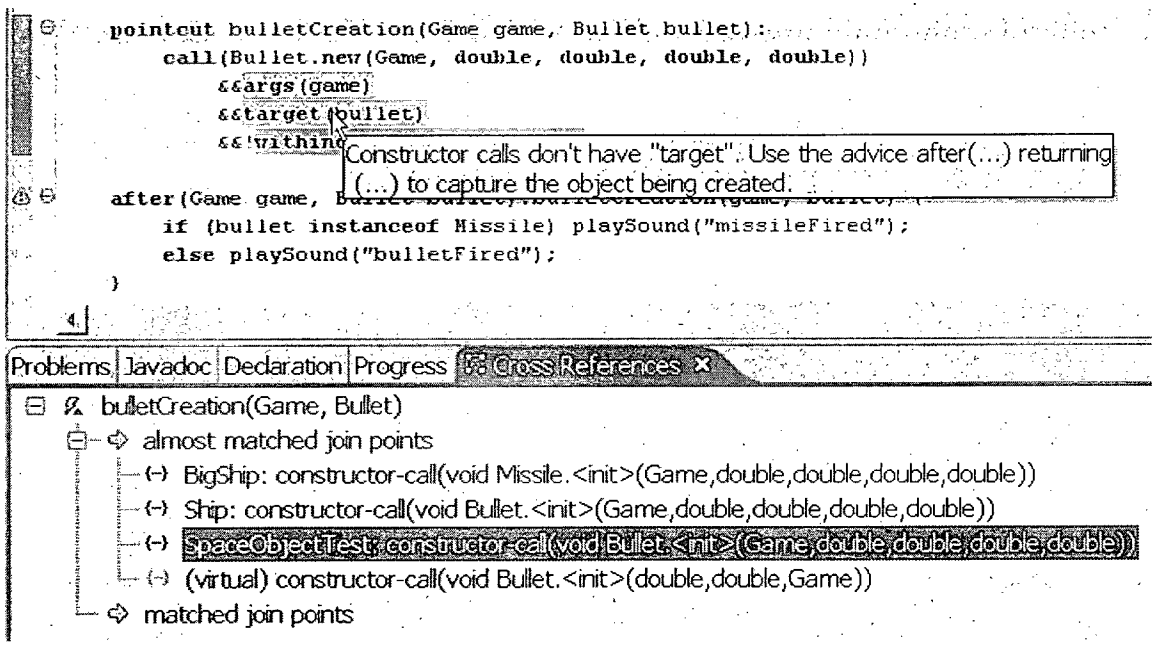

Figure 2.1: Screenshot of PointcutDoctor

will be added in future versions and a developer should take such potential join points into account when trying to write "robust" pointcuts.

2.1.2 Pointcut explainer

The second extension is a *pointcut explanation* feature that is closely integrated into AJDT's cross references view and editor. As shown in Figure 2.1, when the developer clicks on a shadow in the cross references view PointcutDoctor provides an explanation by highlighting specific parts of the pointcut right in the editor. The highlighted parts are those that are responsible for matching or not matching this join point shadow. The highlighting uses a color-coding scheme based on the matching status of the pointcut fragment. Three highlighting colors are chosen for "Matching" (colored green), "Not matching" (colored red) and "Maybe" (colored yellow), where "Maybe" indicates that matching requires runtime determination.

If the colored highlighting alone is not a sufficient explanation, the developer may elicit more information by hovering over the highlighted part of the pointcut to request an explanation why this particular pointcut fragment is colored the way it is. The textual explanation elaborates the causes and tries to educate the user on the subtleties of AspectJ and provides suggestions for improving the pointcut. The explanations are tailored to the specific context of the user's code.

2.2 A Use Case

Figure 2.1 shows the screenshot of PointcutDoctor in a typical scenario. A user writes a pointcut and finds an expected join point shadow appear in the list of almost matches. She clicks on that join point shadow in the cross references view of AJDT. The responsible fragments are highlighted in two different colors: `args(game)` and `target(bullet)` are in red indicating that they do not match the selected join point shadow, and `withincode` are in green indicating that it matches. Because she is not sure why `target(bullet)` does not match the join point shadow, she moves her mouse over that fragment. A tooltip shows up explaining why this fragment does not match and providing suggestion to include this join point shadow.

2.3 Summary

In this chapter, we showed PointcutDoctor, an extension to AJDT that provides developers two kinds of information about AspectJ pointcuts: almost matched join points and an explanation on why a pointcut does not match (or does match) a given join point. PointcutDoctor provides developers easy access to the information from within their already familiar development environment. This chapter proves the second claim of our thesis, the information computed by our algorithm can be added to existing IDE in a clean and logical way.

Chapter 3

Computing Almost Matched Join Points

In this chapter we present our algorithm to compute almost matched join point shadows. We use a technique we call *pointcut relaxation.* We also propose *virtual join point shadows* to handle the possible evolution of the code base. Finally, we will discuss how our algorithms are different from some existing work.

3.1 Pointcut Relaxation

Pointcut relaxation is the process of slightly modifying the original pointcut to produce *relaxed pointcuts* so that they match more join points than the original one. We can compute almost matched join point shadows from the difference of the sets of matches for the relaxed pointcuts and the original pointcut. As previously noted, pointcut relaxation is similar to Anbalagan and Xie's method of computing boundary join points [5]. However, pointcut relaxation provides a more nuanced notion of "almost matched" and provides support for virtual join point shadows in potential future versions of the code. A more detailed comparison of the two techniques can be found in Section 3.3.

A pointcut is relaxed in two steps—preprocessing and conjunction relaxation. The overall process is as shown below:

```
Set<Pointcut> relax(Pointcut pointcut)
{
  /* Step 1: preprocessing */
  Pointcut ptcDNF = convertToDNF(
                        dropNegation(pointcut));

  /* Step 2: relaxing conjunctions */
  Set<Pointcut> relaxedPointcuts = φ;
  for each conjunction p in ptcDNF
```

```
  {
    Set<Pointcut> ptcs = relaxConjunction(p);
    relaxedPointcuts = relaxedPointcuts ⋃ ptcs;
  }
  return relaxedPointcuts;
}
```

3.1.1 Step 1: Preprocessing

In the preprocessing step, all negation pointcuts (pointcuts proceeded by a `!`) are dropped and the resulting pointcut is then converted into Disjunctive Normal Form (DNF)[2]. Dropping negations relaxes a pointcut allowing it to match join points excluded by the negated expression. Dropping negations also has the advantage that it greatly reduces the complexity of the resulting DNF. Dropping negated parts of a pointcut is straightforward and conversion to DNF is a standard technique [7], so we will not elaborate on the preprocessing step any further. Pointcut relaxation is simplified by this preprocessing step because we can just relax the conjunctions in the DNF separately in the following steps.

3.1.2 Step 2: Relaxing Conjunctions

In the second step, each conjunction (primitive pointcuts connected by `&&`) of the resulting DNF will be relaxed separately, and the results are recombined. This step is driven by a set of heuristic rules. We used instructional books ([17], [10]), web resources ([4]), and the result of a study on the AspectJ-user mailing list [2], as guidelines to devise these heuristics. In our mailing list study, we identified and classified 67 discussion topics about pointcut writing in the topics posted from May 2005 to December 2006.

Relaxers are defined for different parts of a pointcut. Specifically, for primitive pointcuts that have method/field signatures (`call`, `execution`, `get`, `set`, `withincode`, `pre/initialization`), a relaxer is assigned to each component of their signature patterns, such as the return type pattern, the declaring type pattern and the parameter list pattern. For other primitive pointcuts, a relaxer is assigned to the whole pointcut. Some relaxers are capable of applying different relaxation methods based on heuristics that analyze both the pointcut and the code base to which it is being matched.

[2]DNF is a standard notion in logic as described in [7], e.g., $(a \wedge b \wedge c) \vee (d \wedge e)$ is in DNF, while $(a \vee b) \wedge c$ is not.

Table 3.1 elaborates the different relaxers used in PointcutDoctor and our reporting tool.

The pseudo code for relaxing conjunctions is given below:

```
Set<Pointcut> relaxConjunction(Pointcut ptcConjunction)
{
  List<Relaxer> candidates = createRelaxers(
                                  ptcConjunction);
  sortRelaxersByPrecedence(candidates);
  List<Relaxer> selectedRelaxers = selectFirstN(
                                  candidates, N);
  Set<Pointcut> relaxedPointcuts = φ;
  applyAllRelaxers(ptcConjunction, 0,
      selectedRelaxers, relaxedPointcuts);
  return relaxedPointcuts;
}

void applyAllRelaxers(Pointcut pointcut, int index,
  List<Relaxer> selectedRelaxers,
  Set<Pointcut> relaxedPointcuts)
{
  if (index<selectedRelaxers.size()) {
    Set<Pointcut> relaxedByOne =
        selectedRelaxers[index].relax(pointcut);
    for each relaxedPointcut in relaxedByOne
      applyAllRelaxers(relaxedPointcut, index+1,
            selectedRelaxers, relaxedPointcuts);
  } else
    relaxedPointcuts.add(pointcut);
}
```

First, a list of candidate relaxers are generated for the non-wildcard parts[3] in the original pointcut. For example, for pointcut `call(* Foo.bar(int)) && target(f)`, the candidate list consists of `DeclaringTypeRelaxer`, `NameRelaxer`, `ParamsRelaxer` and `ThisOrTargetRelaxer`. Next, the list of candidate relaxers is sorted using a heuristic rule based on our intuition determining their precedence. The precedence of relaxers used in PointcutDoctor and our reporting tool is defined as follows:

[3] A non-wildcard part refers to a pointcut part that is neither a * nor ..

```
ParamsRelaxer > ArgsRelaxer > AnnotationRelaxer >
ModifierRelaxer > ThrowRelaxer > ReturnTypeRelaxer >
HandlerRelaxer > DeclaringTypeRelaxer >
ThisOrTargetRelaxer > WithinRelaxer > NameRelaxer
```

The first N most significant relaxers will be selected for the next step, where N is a configurable parameter that can be changed to produce more or less almost matched join point shadows. In our current implementation, an experimental value of 6 is initially assigned to N. In addition, we restrict N to be less than the number of non-wildcard parts in the pointcut to avoid creating an overly-broad relaxed pointcut (one that matches every join point). The effect of this constraint on N is to relax specific pointcuts more than generic ones. This approach seems to work well in practice.

Finally, the selected relaxers are executed to relax all applicable parts in the pointcut depending on the conditions defined in each relaxer.

Note that it is possible that the conditions of multiple relaxation methods in a relaxer (e.g. DeclaringTypeRelaxer) are satisfied simultaneously. In such cases, multiple relaxed pointcuts will be created.

We use the relaxed pointcuts and a modified AspectJ weaver to compute almost matched join point shadows. To compute almost matched join point shadows, we augmented the original AspectJ weaver to match the relaxed pointcuts in parallel with the original pointcut.

3.2 Virtual Shadows

In some cases, even when a join point does not exist in the current code base, it is very likely these join points will occur in future versions of the code, e.g. if a method is declared but not used (this is often the case when using code libraries). Our tool is able to detect this case and identify these *virtual* join point shadows as matched/almost matched join point shadows.

This technique is implemented by augmenting the weaving process of AspectJ compilation. The AspectJ compiler produces instances of different subclasses of its abstract `Shadow` class when visiting each join point shadow in the code base and matches these shadows against each applicable pointcut. We introduce a group of virtual shadow classes that extend the `Shadow` class and that are instantiated right after the related code element (e.g. a method declaration) is visited. These virtual shadows act almost the same as other shadows in the matching process, except that they are not matched against the standard shadow mungers, such as advices, declares etc.

Table 3.1: Relaxers

Relaxer		Condition	Relax Operation
Signature Relaxers	AnnotationRelaxer		change to *
	ModifierRelaxer		change to *
	ReturnTypeRelaxer	if the return type pattern is primitive type (`int`, `void`, ...)	change to *
		if the return type pattern can be resolved to a single type	change to its super type
		if the return type pattern has no +	add +
	DeclaringTypeRelaxer	if the declaring type pattern has no +	add +
		if the pointcut is not about constructors and can be resolved to a single type	change to its super type
		if the declaring type does not match types in all packages	add *.. at the beginning
	NameRelaxer	if the name pattern does not start with *	add * at the beginning
		if the name pattern does not end with *	add * at the end
	ParamsRelaxer		change to ..
	ThrowRelaxer		change to *
Pointcut Relaxers	ArgsRelaxer		drop
	HandlerRelaxer	if the exception type pattern has no +	add +
		if the exception type pattern can be resolved to a single type	change to its super type
	ThisOrTargetRelaxer		drop
	WithinRelaxer	if the type pattern has no +	add +

3.3 Related Work

We next discuss how our algorithm is different from some existing work. Anbalagan and Xie propose boundary join points [5] in the context of AspectJ pointcut testing. Boundary join points are join points that are not matched by the pointcut but have close textual representations to the matched ones. Our algorithm for computing almost matched join point shadows is more sophisticated. Firstly, in contrast to boundary join points, pointcut relaxation uses important structural information from the code base to inform relaxation. For example we consider two classes as similar if they are related through inheritance whereas boundary join points only considers textual similarity. Secondly, pointcut relaxation attaches a different significance to different parts of a pointcut. For example a variation in the name of a method signature is considered more significant than a variation in the return type. In contrast boundary join points treat all parts of a pointcut as having the same significance. Thirdly, we propose the notion of *virtual join point shadows* which allows showing information about almost matched join points that do not exist in the current code base but are likely to occur in future versions of the code.

Anbalagan and Xie also propose mutant pointcuts [6]. The way to generate mutant pointcuts is similar to our pointcut relaxation in that the original pointcuts are slightly modified. However, this technique is used for verifying if a test suite is able to detect common errors in pointcuts, rather than computing almost matched join points information that helps developers write pointcuts.

3.4 Summary

In this chapter, we presented our algorithm, a technique called *pointcut relaxation*, to compute almost matched join point shadows. We also proposed *virtual join point shadows* to handle the possible evolution of the code base. Finally, we discussed how our algorithms are different from Anbalagan and Xie's work on boundary join point and mutant pointcuts.

Chapter 4

Pointcut Explanation

In order to explain *why* a pointcut does not match (or does match) a join point, we present two kinds of explanations in PointcutDoctor. The first is a color-coded highlighting of a pointcut expression that shows which parts of the pointcut expression are responsible for the non-match(or match). The second is a text message explaining why a highlighted part matches (or does not match) the given join point.

In the following two sections, we will discuss the algorithms that compute these two kinds of explanation.

4.1 Computing Color-coded Highlighting Explanation

Intuitively, if a pointcut unintentionally does not match a join point, this is because there is something wrong with one or more parts of the pointcut, i.e. these parts do not match (or do match[4]) the given join point. PointcutDoctor highlights these *responsible parts* in the pointcut using a color indicating whether they match or not.

In this section we present our algorithm to identify these responsible parts. First we show that any pointcut can be converted into a propositional formula, so that the following definitions and algorithms can be discussed using standard propositional logical notations. We then develop the key definitions and the algorithm to compute the explanation. Finally we discuss some variations of the given algorithm.

4.1.1 Converting Pointcuts into Propositional Formulas

Any pointcut can be converted into a propositional formula. First, any primitive pointcut can be converted into a propositional formula that has a conjunction of variables. Each variable corresponds to a fragment of the

[4]When a negated pointcut matches a join point, it could cause the entire pointcut not to match.

pointcut (e.g. the declaring type) and its value is determined by the given join point and the pointcut. Secondly, since any compound pointcut consists of primitive pointcuts connected by $\wedge(AND)$, $\vee(OR)$ and/or $\neg(NOT)$ operators, we can recursively convert a compound pointcut into a propositional formula.

For example, the pointcut `call(* Foo+.foo(int,..)) && target(SubFoo)` can be converted into $u_1 \wedge u_2 \wedge u_3 \wedge u_4 \wedge u_5$. where the variable $u_1, ..., u_5$ represent the matching status of different parts of the pointcut for a given join point:

- u_1: does the return type in the signature of the join point match the pattern $*$?
- u_2: does the declaring type in the signature of the join point match the pattern `Foo+`?
- u_3: does the name in the signature of the join point match the pattern `foo`?
- u_4: does the arguments of the given join point match the pattern $(int, ..)$?
- u_5: does the target of the given join point match the pattern `SubFoo`?

Notice that we convert each component of a pointcut into a unique variable. Each variable represents a fragment of the pointcut and is treated as an independent piece of the cause, even though intuitively these variables might not be independent. For example, in `call(* Foo.foo(int)) && args(int)`, two distinct variables are introduced to represent the two occurrences of `(int)` in the pointcut. As a result when a join point does not have an `(int)` parameter lists both occurrences of `(int)` in the pointcut are considered as independent causes for the non-match. This results in the desired behavior for generating coloring explanations, i.e. each occurrence of `(int)` in the pointcut will be independently marked as responsible for the non-match and colored accordingly.

4.1.2 Key Definitions

We can now present formal definitions to allow us to identify the *responsible parts* in a pointcut, that is, the parts of the pointcut that are responsible for a particular (non)match. We mainly use the terminologies and mathematical notations from [7].

For the propositional formula L consisting of variables u_1, ..., u_n and the logical operators $\wedge(AND)$,$\vee(OR)$ and $\neg(NOT)$, we represent its interpretation I as a set of tuples: $I = \{(u_1, z_1), ..., (u_n, z_n)\}$, where $z_i \in \{T, F\}$ and (u_i, z_i) stands for assigning value z_i to variable u_i. $L(I)$ denotes the truth value of L given the interpretation I. A subset of an interpretation is called a *partial interpretation*. We now define the following terms:

Definition 4.1.1. [sufficient falsifying condition] *The partial interpretation I^p is a* sufficient falsifying condition *for L if and only if $\forall$ interpretation $I \supseteq I^p$, it follows that $L(I) = False$*

Definition 4.1.2. [minimal sufficient falsifying condition] *The partial interpretation I^p is a* minimal sufficient falsifying condition *for L if and only if both of the following conditions are met:*

1. *I^p is a sufficient falsifying condition for L*
2. *$\forall I^{p\prime} \subset I^p, \exists$ an interpretation $I \supset I^{p\prime}$ such that $L(I) = True$*

Definition 4.1.3. $[L_F^I]$ *For the formula L and an interpretation I, we define L_F^I to be the set of all minimal sufficient falsifying conditions that are subsets of I.*

Similarly, we can define L_T^I to be the set of all minimal sufficient true conditions for L.

Our tool uses the definitions of L_F^I and L_T^I to explain why a pointcut does not match (or does match) a given join point. They represent *all* the possible ways that cause a non-match (or match). We use the value in the assignment tuple to determine the color code for the part of the pointcut that corresponds to the respective variable.

4.2 The Algorithm

We can use the definitions developed above to derive our explanation algorithm based on a recursive traversal of the given propositional formula.

Algorithm 4.2.1. *Let L denote a propositional formula. Let I an interpretation for that formula and V a boolean value. We inductively define*

cause(L,I,V) by the following equations:

$$cause(M \wedge N, I, T) = cause(M, I, T) \bigotimes cause(N, I, T); \quad (4.1)$$

$$cause(M \wedge N, I, F) = cause(M, I, F) \bigcup cause(N, I, F); \quad (4.2)$$

$$cause(M \vee N, I, T) = cause(M, I, T) \bigcup cause(N, I, T); \quad (4.3)$$

$$cause(M \vee N, I, F) = cause(M, I, F) \bigotimes cause(N, I, F); \quad (4.4)$$

$$cause(\neg M, I, T) = cause(M, I, F); \quad (4.5)$$

$$cause(\neg M, I, F) = cause(M, I, T); \quad (4.6)$$

$$cause(u, I, T) = \begin{cases} \{\{(u, T)\}\} & if (u, T) \in I \\ \phi & otherwise \end{cases}; \quad (4.7)$$

$$cause(u, I, F) = \begin{cases} \{\{(u, F)\}\} & if (u, F) \in I \\ \phi & otherwise \end{cases} \quad (4.8)$$

Where M and N denote propositional formulae, u denotes a propositional variable, and $\bigotimes$ is an operator similar to cartesian product such that for power sets A and B (set of sets), $A \bigotimes B = \{x \bigcup y \mid x \in A, y \in B\}$, for example,
$\{\{a\}, \{b, c\}\} \bigotimes \{\{b\}, \{d\}\} = \{\{a, b\}, \{a, d\}, \{b, c\}, \{b, c, d\}\}$.

Theorem 4.2.1. *Let L be a propositional formula where every variable only occurs at most once. Let I be an interpretation for L. Algorithm 4.2.1 will compute L^I_F and L^I_T, i.e. cause(L,I,True)=L^I_T and cause(L,I,False)=L^I_F.*

Proof: By structural induction on the propositional formula. The majority of the proof is straightforward. The only interesting part of the proof is that, in order to prove each element in the set $cause(M \wedge N, I, False)$ is a minimal falsifying condition for $M \wedge N$, we need to assume that in the propositional formula L every variable only occurs once. As discussed in Section 4.1.1 we can assume that this is so because of the way the formula is derived.

(Base Case) When L is a single variable, that is $L = u$, if $I = \{(u, T)\}$, then trivially $\{(u, T)\}$ is the only minimal sufficient true condition for L, so we have $u^I_T = \{\{(u, T)\}\}$ according to the definition of u_T, and $u_F = \phi$ since there is no minimal sufficient falsifying condition exists for L. On the other hand, according to Algorithm 4.2.1, $cause(u, I, T) = \{\{(u, T)\}\}$ and $cause(u, I, F) = \phi$. So we have $cause(u, I, T) = u^I_T$, and $cause(u, I, F) = u^I_F$.

Similarly, if $I = \{(u, F)\}$, we can prove $cause(u, I, T) = u^I_T$, and $cause(u, I, F) = u^I_F$.

(Induction Step) Suppose $M_F^I, M_T^I, N_F^I, N_T^I$ are minimal sufficient falsifying/true conditions for Boolean formulas M and N given the interpretation I, and the following hold:

- $cause(M, I, T) = M_T^I$
- $cause(M, I, F) = M_F^I$
- $cause(N, I, T) = N_T^I$
- $cause(N, I, F) = N_F^I$

We need to prove:

1. $cause(M \wedge N, I, T) = (M \wedge N)_T^I$
2. $cause(M \wedge N, I, F) = (M \wedge N)_F^I$
3. $cause(M \vee N, I, T) = (M \vee N)_T^I$
4. $cause(M \vee N, I, F) = (M \vee N)_F^I$
5. $cause(\neg M, I, T) = (\neg M)_T^I$
6. $cause(\neg M, I, F) = (\neg M)_F^I$

According to Algorithm 4.2.1 and the induction hypothesis, the proof of the above is equivalent to the proof of:

$$(M \wedge N)_T^I = M_T^I \bigotimes N_T^I \tag{4.9}$$

$$(M \wedge N)_F^I = M_F^I \bigcup N_F^I \tag{4.10}$$

$$(M \vee N)_T^I = M_T^I \bigcup N_T^I \tag{4.11}$$

$$(M \vee N)_F^I = M_T^I \bigotimes N_T^I \tag{4.12}$$

$$(\neg M)_T^I = M_F^I \tag{4.13}$$

$$(\neg M)_F^I = M_T^I \tag{4.14}$$

To avoid redundancy, we only prove (4.10), (4.12) and (4.13). The other three equations can be proved in the same fashion.

Let I_m and I_n be interpretations such that $M(I_m) = F$ and $N(I_n) = F$. Let I_m^p and I_n^p denote any element in $M_F^{I_m}$ and $N_F^{I_n}$, that is, $I_m^p \in M_F^{I_m}$ and $I_n^p \in N_F^{I_n}$.

From Definition 4.1.3 we know that:

$$I_m \supseteq I_m^P \text{ and } I_n \supseteq I_n^P \tag{4.15}$$

$$\forall I'_m \subset I_m^p, \exists \text{ interpretation } I_m^S \supset I'_m \text{ such that } M(I_m^S) = T \tag{4.16}$$

Let I'_m and I'_n be *any* subset of I_m^P and I_n^P respectively. From (4.16), we know that we can find interpretations I_m^S and I_n^S such that $I_m^S \supset I'_m$ and $M(I_m^S) = T$, $I_n^S \supset I'_n$ and $N(I_n^S) = T$.

Proof for (4.10) $(M \wedge N)_F^I = M_F^I \bigcup N_F^I$:

- (Sufficient) Since $M(I_m) = F$, we have $\forall I \supseteq I_m \supseteq I_m^p, (M \wedge N)(I) = M(I) \wedge N(I) = M(I_m) \wedge N(I) = F \wedge N(I) = F$
- (Minimal) We could select an I such that $I \supseteq I_m^S$ and $N(I) = T$. So we have $I \supseteq I_m^S \supset I'_m$ and $(M \wedge N)(I) = M(I_m^S) \wedge N(I) = T$.

That is, I_m^P is a minimal sufficient falsifying condition for $M \wedge N$. Similarly, we can prove I_n^P is also a minimal sufficient falsifying condition for $M \wedge N = F$. Thus, $(M \wedge N)_F^I = M_F^I \bigcup N_F^I$.

Proof for (4.12) $(M \vee N)_F^I = M_T^I \bigotimes N_T^I$:

- (Sufficient) From (4.15), we know that $I_m \bigcup I_n \supseteq I_m^P \bigcup I_n^P$. Since $M(I_m) = F$ and $N(I_n) = F$, we have $\forall$ interpretation $I \supseteq I_m \bigcup I_n \supseteq I_m^P \bigcup I_n^P$, $(M \vee N)(I) = M(I_m) \vee N(I_n) = F$.
- (Minimal) Since $M(I_m^S) = N(I_n^S) = T$, let $I' = I'_m \bigcup I'_n$, $I = I_m^S \bigcup I_n^S$, we have $I' \subset I_m^P \bigcup I_n^P$, $I \supset I'$ and $(M \vee N)(I) = M(I_m^S) \vee N(I_n^S) = T$.

That is, $I_m^P \bigcup I_n^P$ is a minimal sufficient falsifying condition for $M \vee N$. Since I_m^P and I_n^P could be any element in $M_F^{I_m}$ and $N_F^{I_n}$ respectively, according to the definition of the operator $\bigotimes$ (See Section 4.2), $(M \vee N)_F^I = M_F^I \bigotimes N_F^I$.

Proof for (4.13) $(\neg M)_T^I = M_F^I$:

- (Sufficient) Since $M(I_m) = F$, we have $(\neg M)(I_m) = \neg M(I_m) = T$.
- (Minimal) Since $M(I_m^S) = T$, we have $(\neg M)(I_m^S) = \neg M(I_m^S) = F$

So, I_m^P is a minimal sufficient true condition for $(\neg M)$. That is, $(\neg M)_T^I = M_F^I$.

(End of proof)

4.3 Algorithm Variants

Three-value Variant: In AspectJ, there is a notion of $Maybe$ that represents the result of matching in case runtime determination is required. If we want to explain why a pointcut matching is evaluated as $Maybe$, we can define the cause L_M^I similar to previous definitions of L_F^I and L_T^I. We can augment Algorithm 4.2.1 by adding:

$$\begin{aligned} &cause(M \vee N, I, Maybe) \\ &\quad = (cause(M, I, Maybe) \bigotimes cause(N, I, Maybe)) \bigcup \\ &\quad\quad (cause(M, I, Maybe) \bigotimes cause(N, I, True)) \bigcup \\ &\quad\quad (cause(M, I, True) \bigotimes cause(N, I, Maybe)); \end{aligned}$$

$$\begin{aligned} &cause(M \vee N, I, Maybe) \\ &\quad = (cause(M, I, Maybe) \bigotimes cause(N, I, Maybe)) \bigcup \\ &\quad\quad (cause(M, I, Maybe) \bigotimes cause(N, I, False)) \bigcup \\ &\quad\quad (cause(M, I, False) \bigotimes cause(N, I, Maybe)); \end{aligned}$$

$$cause(\neg M, I, Maybe) = cause(M, I, Maybe);$$

$$cause(u, I, Maybe) = \begin{cases} \{\{(u, Maybe)\}\} & if (u, Maybe) \in I \\ \phi & otherwise \end{cases}$$

Approximation Variant: The algorithm described in Theorem 4.2.1 not only finds the variables that are responsible for the non-match (or match), but also provides some structural information in the cause. For example, for $L = u \vee (v \wedge w)$ with the interpretation $u = v = w = F$, we have $L_F = \{\{(u, F), (v, F)\}, \{(u, F), (w, F)\}\}$. This result not only says "u, v, w are all responsible variables", but also indicates "u and v (or u and w) *alone* can sufficiently make $L = F$". However, this information introduces exponential time and space complexity to the algorithm. The complexity comes from the cartesian product operation involved in (1) and (4).

In practice, we believe the information of responsible variables alone is adequate for users to diagnose the problems in their pointcuts. In order to improve the performance of PointcutDoctor, we chose to discard this structural information in the cause by replacing (1) with:

$$cause(L \wedge N, I, T) = cause(L, I, T) \bigcup cause(N, I, T);$$

and replacing (4) with:

$$cause(L \vee N, I, F) = cause(L, I, F) \bigcup cause(N, I, F);$$

The modified algorithm has linear complexity in the number of variables in the formula, and still provides the responsible variables in the results.

4.4 Computing Textual Explanation

The textual explanation aims to provide explanations in the context of the code base and educate users about language subtleties and best practices.

Similar to our pointcut relaxation algorithm (See 3.1), we use the results of our study of the aspectj-users mailing list, instructional books [17][10], and other web resources [4] as general guidelines to create a catalog of explanation heuristics.

The heuristic rules are categorized by the kind of pointcut components and the type of join point. A heuristic rule consists of a group of conditions and a message to be shown for the user when these conditions are satisfied. The conditions describe the testing on the given join point and the patterns in the pointcut. The messages are customized by the information in the code base, e.g. the actual declaring type of the given join point. There are 35 rules implemented in PointcutDoctor at the time of this writing. Table 4.1 elaborates the heuristic rules to be used when the pointcut does not match the join point, and Table 4.2 presents the heuristic rules for matches. Table 4.4 explains the code used in Table 4.1 and Table 4.2. Table 4.3 lists the actual messages being presented to the user when the condition is satisfied.

For example, let's consider pointcut `call(Foo.new(..)) &&target(foo)` and join point shadow `constructor-call(Foo.new())`. The `target(foo)` sub-expression causes the pointcut not to match the join point. Our tool will produce the message MSGTarget0: "Constructor calls do not have target. Use the advice after(...) returning(...) to capture the object being created.". This message is determined by looking up rows corresponding to `target` pointcut in Table 4.1, and then looking up join point kind `constructor-call` in the found rows.

Table 4.3: Explanation Messages

Code	Message
MSGAnno0	Annotations are not inherited by default, though the method [methodName] is overridden in [jp.dt]
MSGRT0	The return type [jp.rt] is not matched even though it's a subtype of [ptn.rt]. Add a "+" to include subtypes of [ptn.rt] in the return type.
MSGDT0	Calls to constructors of [ptn.dt]'s subtypes (e.g. [jp.dt]) won't be matched. Use [ptn.dt]+ to include calls to constructors of its subtypes.
MSGDT1	Constructors of [ptn.dt]'s subtypes (e.g. [jp.dt]) won't be matched. Use [ptn.dt]+ to include constructors of its subtypes.
MSGDT2	The declaring type of [private/static] methods has to be matched exactly by the pattern "[ptn.dt]", i.e. methods with the same signatures in subtypes are not matched.
MSGDT3	The method [methodSig] is not applicable to the type [ptn.dt], i.e. it is declared in [jp.dt] but not in [ptn.dt] or any of [ptn.dt]'s super types. Use [ptn.dt]+ to include all qualifying methods declared in its subtypes.
MSGDT4	Call pointcut only matches against the static target type, but the static target type of this call ([jp.dt]) is neither [ptn.dt] nor subtype of [ptn.dt]. Use [call(* set*(..))&&target(SubFoo)] to include the join point with the runtime target type being [ptn.dt].
MSGDT5	[jp.dt] cannot be matched by pattern "[ptn.dt]"
MSGDT6	A "[jp.dt]" is not a "[ptn.dt]". Use [execution(* set*(..))&&this(SubFoo)] to include the join point arising here with the runtime type being [ptn.dt].
MSGDT7	The declaring type is not matched because field [fieldName] is re-declared in class [jp.dt]. Use [jp.dt] to pick this join point only, and [ptn.dt]+ to pick fields declared in [ptn.dt]'s subtypes
MSGArgs0	[ptn.param] does not match join points with [param-num] arguments
MSGArgs1	[jp.param] cannot be matched by pattern [ptn.param].
MSGParam0	Unlike args(...), the parameter pattern here only matches against the STATIC types of the join point's parameters, so the parameter types of this join point ([jp.param]) cannot be matched by the pattern [ptn.param]
MSGThrow0	The method does not declare "throws [ptn.throws]". By Java convention, a method does not need to explicitly declare unchecked exceptions, though any method could throw such exceptions.
MSGThis0	There is no "this" in a static context
MSGTarget0	Constructor calls don't have "target". Use the advice after(...) returning(...) to capture the object being created.
MSGTarget1	Calls to static methods don't have "target".

Continued on next page

Table 4.3 – continued from previous page

Code	Message
MSGTarget2	The "target" could not be of type [ptn.type]
MSGWithincode0	This is not matched because method [jp.enclosingMethod] is declared in [declaring type of jp.enclosingMethod] but not in its super type [ptn.dt]
MSGWithin0	This is not matched because within only cares about static lexical scope. Use [ptn.type]+ to include join points within subtypes of Foo.
MSGRef0	[ptn.refName] does not match this join point, see the definition of [ptn.refName] for detailed reason.
MSGHandler0	The handler pointcut does not implicitly match subtypes. Use handler([ptn.et]+) to include this join point.
MSG0	The [partType] of the given join point "[partValue]" cannot be matched by the pattern "[pattern]"
M_MSGDT0	It is matched because: 1) The static target type of the call ([jp.dt]) is a subtype of [ptn.dt]; 2) method [methodSig] is [declared/inherited] in [ptn.dt]
M_MSGParam0	It is matched because: [paramT] is a subtype of [paramPattern]
M_MSGWithincode0	The withincode pointcut matches method overrides, and [SubFoo.bar()] overrides method [Foo.bar()].
M_MSGArgs0	[ptn.param] is a subtype of [jp.param]. Runtime test is needed because the runtime type of the argument could, but not necessarily, be [ptn.param].
M_MSGArgs1	[jp.param] is a subtype of [ptn.param], and the runtime type of the argument will always be of type [ptn.param].
M_MSGHandler0	It is matched because [jp.param] is a subtype of [ptn.param]
M_MSG0	

4.5 Summary

In this chapter, we discussed the algorithms that computes two kinds of explanations about why a pointcut does not match (or does match) a join point. The first is a color-coded highlighting of a pointcut expression that shows which parts of the pointcut expression are responsible for the non-match(or match). The second is a text message explaining why a highlighted part matches (or does not match) the given join point.

In the next chapter, we will present a case study as the evaluation of the utility of the information computed by our algorithms in this chapter and Chapter 3.

Table 4.1: Not-match Explanation Heuristics

Pattern		Condition		Message
Patterns in signature pattern	Annotation Pattern	[jp.dt] isSubTypeOf [ptn.dt] & [jp.method] declared in both [jp.dt] and [ptn.dt] & [ptn.dt].method has annotation [ptn.anno]& [jp.method] hasn't annotation [ptn.anno]		MSGAnno0
	Return type pattern	[jp.rt] isSubTypeOf [ptn.rt] & [ptn.rt] does not have "+"		MSGRT0
		[jp.kind]=constructor-call & [jp.dt] isSubTypeOf [ptn.dt]		MSGDT0
	Declaring type pattern	[jp.kind]=constructor-execution/(pre)initialization & [jp.dt] isSubTypeOf [ptn.dt]		MSGDT1
		[jp.kind]=method-call &	[jp.method] is private or static & [jp.dt] isSubTypeOf [ptn.dt]	MSGDT2
			[jp.dt] isSubTypeOf [ptn.dt] & [jp.method] is declared in [jp.dt] but not in [ptn.dt]	MSGDT3
			[ptn.dt] isSubTypeOf [jp.dt]	MSGDT4
			otherwise	MSGDT5
		[jp.kind]=method-execution &	[jp.method] is private or static & [jp.dt] isSubTypeOf [ptn.dt]	MSGDT2
			[jp.dt] isSubTypeOf [ptn.dt] & [jp.method] declared in [jp.dt] but not in [ptn.dt]	MSGDT3
			[ptn.dt] isSubTypeOf [jp.dt]	MSGDT6
			otherwise	MSGDT5
		[jp.kind]=get/set & [jp.dt] isSubTypeOf [ptn.dt] & [jp.field] is declared in both [jp.dt] and [ptn.dt]		MSGDT7
	Parameter pattern	the number of [jp.args] is not matched by [ptn.args]		MSGArgs0
		the number of [jp.args] is matched by [ptn.args] & one of [ptn.args] isSubTypeOf one of [jp.args]		MSGParam0
	Throws Pattern	[jp.method] does not declare [ptn.et] & [ptn.et] is an unchecked Exception		MSGThrow0
Primitive Pointcuts	args	the number of [jp.args] is not matched by [ptn.args]		MSGArgs0
		the number of [jp.args] is matched by [ptn.args] & [jp.argType] !sameOrSubTypeOf [ptn.argType] & [ptn.argType] !sameOrSubTypeOf [jp.argType]		MSGArgs1
	this	the join point is in static context		MSGThis0
	target	[jp.kind]=constructor-call		MSGTarget0
		[jp.kind]=method-call & [jp.method] is static		MSGTarget1
		[jp.kind]=field get/set & [jp.field] is static		MSGTarget2
		[jp.kind]=initialization/preinitialization		MSGTarget3
		[jp.kind]=exception-handler		MSGTarget4
	withincode	[jp.enclosingClass] isSubTypeOf [ptn.dt] & [jp.enclosingMethod] is not declared in [ptn.dt] & [ptn.dt] does not have a "+"		MSGWithincode0
	within	[jp.enclosingClass] isSubTypeOf [ptn.type]		MSGWithin0
	handler	[jp.et] isSubTypeOf [ptn.et]		MSGHandler0
otherwise				MSG0

Table 4.2: Match Explanation Heuristics

Pattern		Condition	Message
Patterns in signature pattern	Declaring type	[jp.kind]=method-call & [jp.dt] isSubTypeOf [ptn.dt] & [jp.method] declared/inherited in [jp.dt] and [ptn.dt]	M_MSGDT0
	pattern	[jp.kind]=method-execution & [jp.dt] isSubTypeOf [ptn.dt] & method declared in [ptn.dt] or its super type	M_MSGDT0
	Parameter pattern	[jp.param] isSubTypeOf [ptn.param] & [ptn.param] includeSubtypes	M_MSGParam0
Primitive pointcuts	withincode	[jp.enclosingMethod] overrides [ptn.method]	M_MSGWithincode0
	args	[ptn.param] isSubTypeOf [jp.param]	M_MSGArgs0
		[jp.param] isSubTypeOf [ptn.param]	M_MSGArgs1
otherwise			M_MSG0

Table 4.4: Conventions used in Table 4.1 and Table 4.2

[jp.dt]	The declaring type in the signature of the join point
[ptn.dt]	The declaring type pattern of the signature in the pointcut
[ptn.anno]	The annotation pattern of the signature in the pointcut
[jp.method]	The corresponding method at the join point (for method execution and method call join point only)
[ptn.rt]	The return type pattern of the signature in the pointcut
[jp.kind]	The kind of the join point
[jp.field]	The corresponding field at the join point (for field get/set join point only)
[jp.args]	The arguments at the join point
[ptn.args]	The argument patter in the pointcut
[ptn.throws]	The throws pattern in the pointcut
[jp.enclosingClass]	The enclosing class of the join point
[ptn.type]	The type in `within` pointcut
[jp.et]	The exception type in the exception handler join point
[ptn.et]	The exception type in the `handler` pointcut

Chapter 5

Evaluation

We implemented an offline analysis tool that generates a comprehensive report about all pointcuts in a given code base. Using this tool, we were able to find several real problems in existing, medium sized AspectJ code base. This supports our first claim of our thesis that the information computed by our algorithm can help a developer find problems in her pointcuts.

The report generated by our tool consists of almost matched and matched join point shadow information for all pointcuts in the code base, and the corresponding textual explanation for each listed join point shadow. A sample of the report is shown in Figure 5.1.

The code bases we used in our case study are Aspect-oriented ORBacus[18] and Aspect-oriented jEdit[9]. Aspect-oriented ORBacus is an AspectJ refactoring of a CORBA middleware [5] consisting of 310 pointcuts, 1297 classes, and 122,110 lines of code. Aspect-oriented jEdit is an AspectJ refactoring of a mature text editor [6] consisting of 354 pointcuts, 428 classes, and 35,890 lines of code.

5.1 Case Study Procedures

Prior to our case studies, we had no knowledge or experience with the selected code bases. We wanted to determine whether it is possible to discover bugs (i.e. unintended misses) in pointcuts by looking through the list of almost matched join point shadows and their corresponding explanations. We skipped finding bugs that result in unintended matches because AJDT already provides a list of matches and evaluation of the utility of such list is outside the scope of this thesis.

First, we ran our tool against both code bases to generate the reports. Second, we read through the reports to try to identify unintended misses. Since we were not the authors of the code and we did not directly know the intention of these pointcuts, we made assumptions based on the reports and

[5] ORBacus: http://www.iona.com/
[6] jEdit: http://www.jedit.org

```
after(com.ooc.OBCORBA.ORB_impl orb_impl, com.ooc.OB.Properties properties,
  String key) returning(String value):
call(* com.ooc.OB.Properties.getProperty(..)) && this(orb_impl) &&
    args(key) && target(properties)

...\orbacus\src\aop\codeset\aspects\Codec.aj:42::1253
* almost matched (21):
  1. ORBSupport: method-call(java.lang.String java.util.Properties.
       getProperty(java.lang.String))
     Not matched because:
      - Call pointcut only matches against the static target type, but the
        static target type of this call (java.util.Properties) is neither
        com.ooc.OB.Properties nor subtype of com.ooc.OB.Properties.
        Use call(* getProperty(..))&&target(com.ooc.OB.Properties) to include
        the join point with its runtime target type com.ooc.OB.Properties.
      - The "this" of the given join point (ORBSupport or its subtype) cannot
        be matched by the pattern "this(com.ooc.OBCORBA.ORB_impl)"
  ...
* matched (3):
  1. ORB_impl: method-call(java.lang.String com.ooc.OB.Properties.
                 getProperty(java.lang.String))
  2. ORB_impl: method-call(java.lang.String com.ooc.OB.Properties.
                 getProperty(java.lang.String))
  3. ORB_impl: method-call(java.lang.String com.ooc.OB.Properties.
                 getProperty(java.lang.String))
```

Figure 5.1: A Sample of the Report Used in Case Study

quick investigations of the code base (e.g. we looked at whether a class is subclass of another, whether a method is overridden in a subclasses etc.) Finally, we sent the list of possible bugs to the authors of both code bases seeking to determine the validity of those bugs.

Out of the 310 and 354 pointcuts in ORBacus and jEdit code bases, there are respectively 56 and 71 pointcuts that have at least one almost matched join point shadows. It took us roughly 2 hours to go through the ORBacus report, and 1.5 hours to go through the jEdit report.

5.2 Results

Table 5.1 shows the numbers of potential bugs we identified , and those the original developers confirmed to be valid.

In the table, the first two columns lists the type of the bug and the code of the explanatory message produced by our tool. As for the bug numbers, column `I` refers to the numbers of bugs that were identified as potential bugs when we investigated the report; column `R` refers to the numbers of real bugs confirmed by the developers; column `V` refers to the number of "virtual" bugs: the developers claimed that they were not bugs since the missed virtual and potential join points does not affect the correctness of the **current** code base, however, they agreed including the missing potential join points in the pointcut would be better style.

We next explain each bug type in detail:

declaring-type-call this error is caused by misunderstanding of the static semantics of `call` pointcut: the declaring type in `call` pointcut only matches against the static type of the call target, not its runtime type[7]. However, developers often think of the runtime type when they write pointcuts. For example, in one of the code bases,
`call(* com.ooc.OB.Properties.getProperty(..))` is used intending to capture calls to `getProperty` method for instances of `com.ooc.OB.Properties` only. Here, `com.ooc.OB.Properties` is a sub-class of `java.util.Properties`. This pointcut does not match the join point below (in the second statement):

```
java.util.Properties props =
        new com.ooc.OB.Properties();
...
String p = props.getProperty("key");
```

[7]This semantics applies to the declaring type of `execution` and `withincode` too.

Bug Type	Message	ORBacus			jEdit		
		I	C		I	C	
			R	V		R	V
declaring-type-call	MSGDT4	2	2		3		3
param-num-not-match	MSGArgs0	13	2	3	1	1	
param-type-not-match	MSGParam0				1	1	
subtype-constructor-call	MSGDT0				3		3
modifier-not-match	MSG0	2	2				
return-type-not-match	MSG0	1	1				
handler-subtype	MSGHandler0				1	1	
Total		**18**	**7**	**3**	**9**	**3**	**6**

Table 5.1: Messages Issued and Number of Bugs Found in Case Study (I:Identified, C:Confirmed, R:Real bugs, V:"Virtual" bugs)

param-num-not-match The developer was unaware of some overloaded constructors/methods in the code base. Some expected join points are ruled out by either the parameter pattern or `args` pointcut because the number of parameters in the join point cannot be matched. Note that we found 13 potential bugs of this type but most of them are false positives. This is because most overloaded methods call each other, so it's the expected behavior to only match one "root" method in the pointcut. The "virtual" bugs in this category are mostly misses of virtual calls to overloaded constructors.

param-type-not-match The join point is not matched because the type of one of the parameters cannot be matched. The developer wrote `call(void add*Listener(EventListener))` intending to match `method-call(void JButton.addActionListener(ActionListener))`. But it cannot be matched even though `ActionListener` is a subtype of `EventListener`.

subtype-constructor-call If "+" is not used in the declaring type of `call` pointcut, it will not match calls to the constructors of the declaring type's subtypes. All of the identified bugs are classified as virtual since the developer agreed including all subtypes might be better style, as we discussed earlier.

modifier-not-match The developer was unaware of the different modifiers for some of the expected join points.

return-type-not-match Similar to "modifier-not-match", the developer was not aware of the different return type. It seems that developers often ignore the difference in modifier and return type when they write pointcuts.

handler-subtype Similar to the subtype-constructor-call case, `handler` pointcut does not match exception handlers with sub-exception types, if "+" is not used in the pointcut. The developer used `handler(Throwable)` hoping to capture all exception handlers.

In summary, we believe that the result of this study is quite positive: Several real bugs and style problems were discovered in real code bases. Given the sizes of both code bases, our unfamiliarity with them, and the relatively short time we spent on interpreting the reports, we believe it demonstrates that the process of identifying problems is facilitated by the information available in the report.

Anecdotally, even when an almost matched join point shadow is not intended to be matched, the information can still be useful, as one of the developers of the case study code bases said, "these warnings indeed make me think about whether it could be problematic for other subtypes.".

5.3 Performance Impact

In our development environment(Intel Core2 Duo Processor 2.33GHz CPU, 2G memory, Windows XP Professional), it took 50 seconds and 68 seconds for our tool to generate the reports for the Aspect-oriented jEdit and OR-Bacus code base, respectively. The memory usage of the javaw.exe process is 350M and 397M bytes for the two code bases. Compared to the numbers with the unmodified AspectJ compiler(20 seconds/250M and 30 seconds/290M), these numbers indicate that our extension to AspectJ compiler introduces performance overhead. We believe the running time overhead is caused by the large number of relaxed pointcuts being tested in the matching process. When a compound pointcut with many primitive pointcuts connected by `||` and `&&` gets relaxed, potentially a large number of duplication will be produced by the DNF conversion (see Section 3.1.1). The memory usage overhead is because we produce extra helper objects in the process of pointcut relaxation and explanation, compared to the unmodified AspectJ compiler.

5.4 Summary

In this chapter, we presented the result of our case study in order to validate the usefulness of the information produced by the algorithms presented in this thesis. Based on this result, we believe that the information computed by our algorithms can help a developer find problems in her pointcuts. This supports our first claim in this thesis. We also evaluated and analyzed the performance impact of our algorithms.

Chapter 6

Limitations and Future Work

In this chapter we discuss some of the limitations of our algorithms, of their current implementation and of the evaluation presented in this thesis. We also present some ideas on how these could lead to future work.

6.1 Limitation of Implementation

A limitation of our current implementation of pointcut relaxation is its performance overhead compared to the standard AspectJ compiler. It should be noted that performance was not the focus of this thesis. We rather focused on providing information that helps developers to diagnose and fix problems with their pointcuts. Having established that the information our algorithms compute is useful, future work could focus on more efficient implementations. For example, the run-time overhead could be reduced by optimizing the preprocessing and relaxation algorithms to produce fewer relaxed pointcuts. The memory usage would also be improved by this optimization because fewer helper objects would be generated.

Another limitation of the current implementation is that it relies on a fixed set of heuristics. Our evaluation shows that the current set of heuristics provides useful results. However we believe that they can still be improved and extended. This is a possible topic for future work.

A third limitation of the current implementation presented is that we have not directly considered `cflow`, `cflowbelow` and `if` pointcuts. However, a user can indeed write the enclosed pointcut in `cflow` or `cflowbelow` as a named pointcut, and have it explained by our tool. The direct support for these three pointcuts is a possible topic for future work.

6.2 Limitation of Evaluation

The evaluation presented in this thesis focused on determining whether information on almost matched join points and explanations of why they do or do not match is useful to find problems in pointcut expressions. We decided to

separate the question of whether this information is useful by itself from the question of how to present it in the IDE. Therefore, we used an offline report to perform our case-studies rather than the PointcutDoctor IDE. Because of the "raw" presentation format used, we believe the results of the case studies are relatively independent from presentation issues. Presently, we have not directly evaluated the design of the PointcutDoctor GUI. Nevertheless, we have faith in our GUI design because a) the presented information has been shown to be independently useful and b) our GUI design naturally extends the state-of-the-practice AJDT UI. Future work could focus independently on the GUI design and determine whether it is effective in making this useful information readily available to developers.

Another limitation of the evaluation is that we did not separately evaluate the utility of the explanations versus the lists of matches and almost matches. What our results show is that in combination this information can be useful to find problems with pointcuts. However, our results provide no solid conclusions about the usefulness of these types of information considered in isolation.

6.3 Summary

In this chapter, we discussed some of the limitations of our algorithms, of their current implementation and of the evaluation presented in this thesis. We also presented some ideas on how these could lead to future work.

Chapter 7

Conclusion

In this thesis, we proposed to help AspectJ developers diagnose and fix potential problems in their pointcuts by extending existing IDE to provide two kinds of information: almost matched join point shadows and explanation on why a pointcut matches (or does not match) a given join point. We presented our algorithms to compute this information. We elaborated the heuristics used in our algorithms extracted from our previous empirical study on Aspectj user mailing list and other resources.

We make two claims in this thesis. First, we claim that the information computed by our algorithms can help a developer find problems in her pointcuts. Second, we claim that this information can be added into existing IDE tools in a clean and logical way.

To prove our first claim, we evaluated the utility of the information produced by our algorithm by conducting case studies on two existing medium-sized AspectJ code bases. By going through the report generated by our tool, we were able to identify real problems in the code bases in a couple of hours, despite that we did not have any previous knowledge about the code bases before the study. Based on the results of the case study, we believe that the information computed by our algorithms can help a developer find problems in her pointcuts.

To prove our second claim, we developed an Eclipse plugin called PointcutDoctor. PointcutDoctor is a natural extension of AJDT that provides developers easy access to the information discussed above from within their familiar development environment.

Bibliography

[1] Ajdt: Aspectj development tools. http://www.eclipse.org/ajdt.

[2] Aspectj users mailing list archive. http://dev.eclipse.org/mhonarc/lists/aspectj-users/maillist.html.

[3] Eclipse - an open development platform. http://www.eclipse.org.

[4] Ibm developerworks: Aop@work. http://www-128.ibm.com/developerworks/views/java/libraryview.jsp ?search by=AOP@work.

[5] Prasanth Anbalagan and Tao Xie. Apte: automated pointcut testing for aspectj programs. In *WTAOP '06: Proceedings of the 2nd workshop on Testing aspect-oriented programs*, pages 27–32, New York, NY, USA, 2006. ACM Press.

[6] Prasanth Anbalagan and Tao Xie. Efficient mutant generation for mutation testing of pointcuts in aspect-oriented programs. In *Proceedings of the 2nd Workshop on Mutation Analysis (MUTATION 2006)*, pages 51–56, November 2006.

[7] M. Ben-Ari. *Mathematical logic for computer science.* Prentice-Hall, Inc., Upper Saddle River, NJ, 1993.

[8] C. Clifton and G. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. In *Iowa State University Technical Report, TR 03-15.*

[9] J. Collins-Unruh and G. Murphy. Aspect-oriented jedit. unpublished.

[10] Adrian Colyer, Andy Clement, George Harley, and Matthew Webster. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ development tools.* Addison-Wesley, 2004.

[11] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA 2000*, October 2000.

[12] Robert Filman. What is aspect-oriented programming, revisited. In *Workshop on Advanced Separation of Concerns, 15th European Conference on Object-Oriented Programming*, June 2001.

[13] Erik Hilsdale and Jim Hugunin. Advice weaving in aspectj. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35, New York, NY, USA, 2004. ACM Press.

[14] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[15] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[16] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, New York, NY, USA, 2005. ACM Press.

[17] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications Co., Greenwich, CT, USA, 2003.

[18] Charles Zhang and Hans-Arno. Jacobsen. Quantifying aspects in middleware platforms. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 130–139, New York, NY, USA, 2003. ACM Press.