Pointcuts by Example

by

Edward J. McCormick

B.Sc., Northeastern University, 2002

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

December 16, 2005

Abstract

The thesis of this dissertation is that it is possible to construct IDE-based tools that allow the editing of AspectJ pointcut expressions through direct manipulation of effective representations of join points in the code to which a pointcut applies.

There are two reasons why providing tool support for managing pointcut expressions is important. The first reason is that the pointcut-advice mechanism obscures exactly what a given piece of code does. This is because the execution of any piece of code may trigger additional advice of which the developer may be unaware. The second reason is that the creation and maintenance of pointcut expressions that capture the intended set of join points is a difficult task. Pointcut expressions are typically based on assumed naming and structural patterns in the code. We argue that without proper tool support, it is difficult for the developer to properly verify that the conventions are followed strictly enough for a pointcut to remain robust

Current tool support addresses these issues to some extent by alerting the developer to where the local join points exist in the code, and the advice that applies at that point. However, if the developer finds that the join points are incorrect he must open the aspect and edit the pointcut expression outside the context of the code where the join point should exist.

In this dissertation we present the notion of *Pointcuts by Example*. Using this technique, a developer is able to specify examples in the code of join points that a pointcut expression *should* or *should-not* match. As a constructive proof of existence, we developed prototype tool support for editing pointcuts by example called the *Pointcut Wizard*. The PW presents a GUI interface for editing pointcut expressions by selecting **add** or **remove** operations at join point shadow sites.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Thesis Statement

The thesis of this dissertation is that it is possible to construct IDE-based tools that allow editing of AspectJ pointcut expressions through direct manipulation of effective representations of join points in the code to which a pointcut applies.

This chapter consists of two sections. The first section explains and motivates the thesis statement. The second section describes our validation of the thesis statement.

1.2 Background and Motivation for the Thesis

1.2.1 Aspect-Oriented Programming

Programming languages that support Aspect-Oriented Programming rely on the notion of a *join point* mechanism to specify when, where and how an aspect should alter program execution. In this dissertation we focus on AspectJ [13], which is probably the most well known and mature aspect-oriented language today. With AspectJ, a developer can create *pointcut expressions* to describe a set of join points at which to apply some aspect functionality. Other aspect languages that offer similar support are Ceasar [17], DemeterJ [9] and Aspectual Collaborations [15].

All these aspect-oriented programming languages have one thing in common: the goal to provide novel language mechanisms to allow modularization of so-called crosscutting concerns. A concern is defined very generally as any kind of issue or problem that a developer would want to reason about in relative isolation. Two concerns are said to be crosscutting if a decomposition of the system that modularizes one concern tends to cut across a decomposition along the lines of the other concern. The existence of crosscutting concerns in a software system generally leads to *tangling* and *scattering* [19]. Tangling means that one module contains a mix of code addressing several concerns. Scattering means that code addressing a single concern is scattered across multiple modules of the system. Tangling and scattering greatly increase the complexity of software understanding, evolution and maintenance.

An example of crosscutting concerns is given in [14]. The authors present a hypothetical situation of a developer creating a robotics application. The core concerns during his design process are motion management and path computation. The application is successfully developed, but later it becomes clear that path optimizations would greatly improve run-time results. Since optimization was not a high-priority in the initial design of the system, the system is mostly decomposed along the lines of the functional concerns which are well modularized. The path optimization concern crosscuts the functional concerns of motion management and path computation. Therefore, the implementation of the optimization concern ends up scattered across many different modules, as well as tangled with the other concerns addressed by those modules. Future updates to the optimization code thereby become an exercise with much greater difficulty.

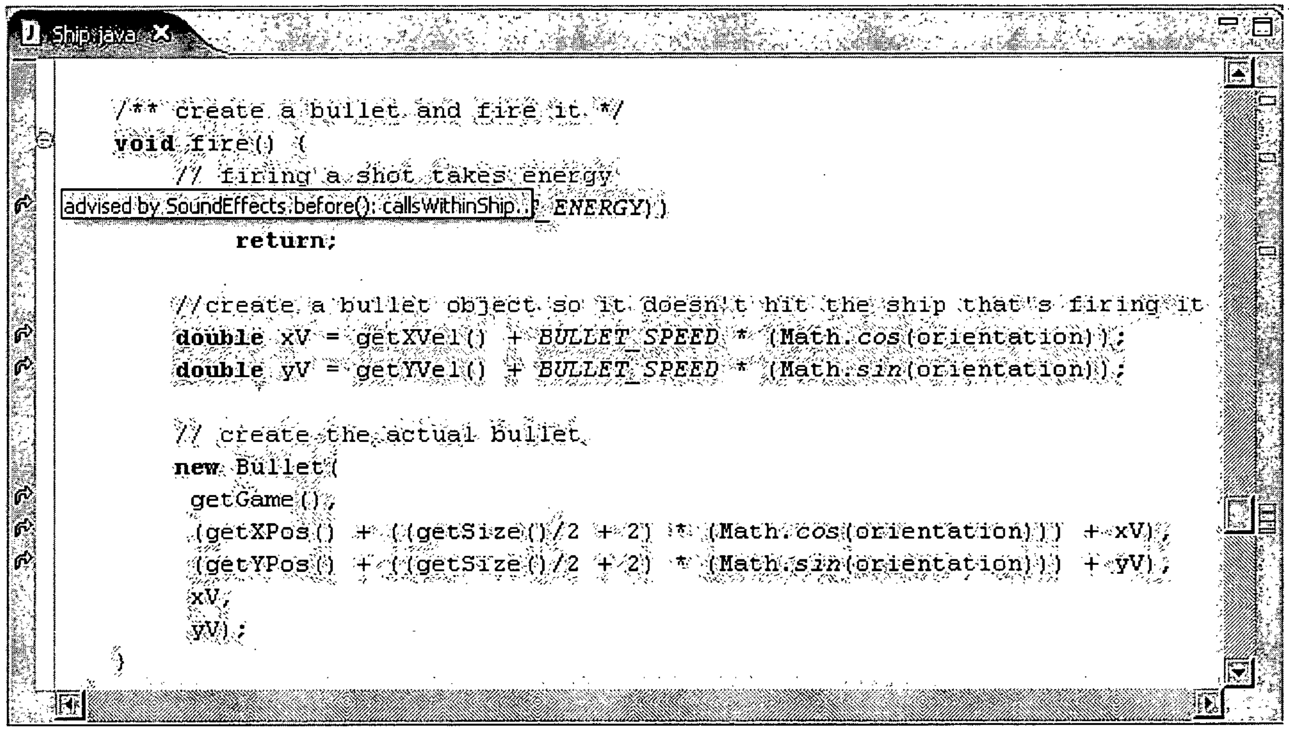
The goal of AspectJ is to introduce a novel kind of modular unit, called an *aspect*, which provides the developer with the means to modularize crosscutting concerns such as the one in this example. If the modularization of the path optimization concern as an aspect is successful, all the concern-specific code will be localized within a small number of related aspects and classes and can be viewed and edited in relative isolation.

An AspectJ developer can write code in terms of AspectJ's *join point model*, in which join points are principled points in the execution of a program. Using a syntactic element of AspectJ called a pointcut designator, the developer is able to describe a set of join points at which a particular piece of concern-specific code should execute. The code that the developer wishes to execute at the join points matched by a particular pointcut expression are contained in a modular structure similar to a Java method called an advice. Put simply, the advice of an aspect describes *what* should happen and the pointcut designator describes *when*.

1.2.2 Tool Support For Managing Pointcuts

In this dissertation we will be presenting a new technique for working with AspectJ pointcut expressions. There are two reasons why providing tool support for managing pointcut expressions is important.

The first reason is that the pointcut-advice mechanism obscures exactly what a given piece of code does. This is because the execution of any piece of code may trigger additional advice of which the developer may be unaware. This problem is addressed by tools such as AJDT[5], which provide a visual representation of where in the code an AspectJ advice will be executed. The static representation of a dynamic join point is referred to as the join points *shadow*. Figure 1.1 shows the gutter annotation mechanism AJDT uses to denote a join point shadow. By hovering over the annotation the developer can view more specific information pertaining to the shadow, such as its signature and the pointcut expression which it is matched by. Additionally, a menu option at the shadow site allows the developer to hyperlink directly to the piece of advice which will be executed at the join point during the program's execution. Tools of this nature ease the burden on the developer of having to think both in terms of the base code and any advice that might be executed.



Ship.java

```
/** create a bullet and fire it */
void fire() {
    // firing a shot takes energy
advised by SoundEffects.before(): callsWithinShip.. _ENERGY))
        return;

    //create a bullet object so it doesn't hit the ship that's firing it
    double xV = getXVel() + BULLET_SPEED * (Math.cos(orientation));
    double yV = getYVel() + BULLET_SPEED * (Math.sin(orientation));

    // create the actual bullet
    new Bullet(
     getGame(),
     (getXPos() + ((getSize()/2 + 2) * (Math.cos(orientation))) + xV),
     (getYPos() + ((getSize()/2 + 2) * (Math.sin(orientation))) + yV),
     xV,
     yV);
}
```

Figure 1.1: Gutter annotations mark shadows in AJDT

The second reason is that the creation and maintenance of pointcut expressions that capture the intended set of join points is a difficult task. Pointcut expressions are typically based on assumed naming and structural patterns in the code. We argue that without proper tool support, it is difficult for the developer to properly verify that the conventions are followed strictly enough for a pointcut to remain robust. If they are not, a pointcut constructed by the developer may result in false-positive or false-negative join points occurring at run-time. This problem is also partially addressed by placing, as in AJDT, gutter annotations at join point shadow sites. Additionally, AJDT provides a *cross-references* view, as shown in Figure 1.2, with which the developer can look up any advice associated with a join point shadow in the code, or alternatively any join point shadows associated with an aspect or advice.

The drawback with these tools is that although they present information concerning the location and existence of join point shadows, they offer little help in correcting a malformed pointcut expression. It is left as a task for the developer to inspect the shadow sites and infer the error in the pointcut expression. In the following section we present another technique to support the AspectJ developer: effective views.

1.2.3 Effective versus non-effective representations

In this section we will explain what we mean with the word *effective* in the thesis statement. We will illustrate the usefulness of having an effective tool by illustrating how a developer may use the powerful but non-effective AspectJ tool AJDT to edit a pointcut expression.

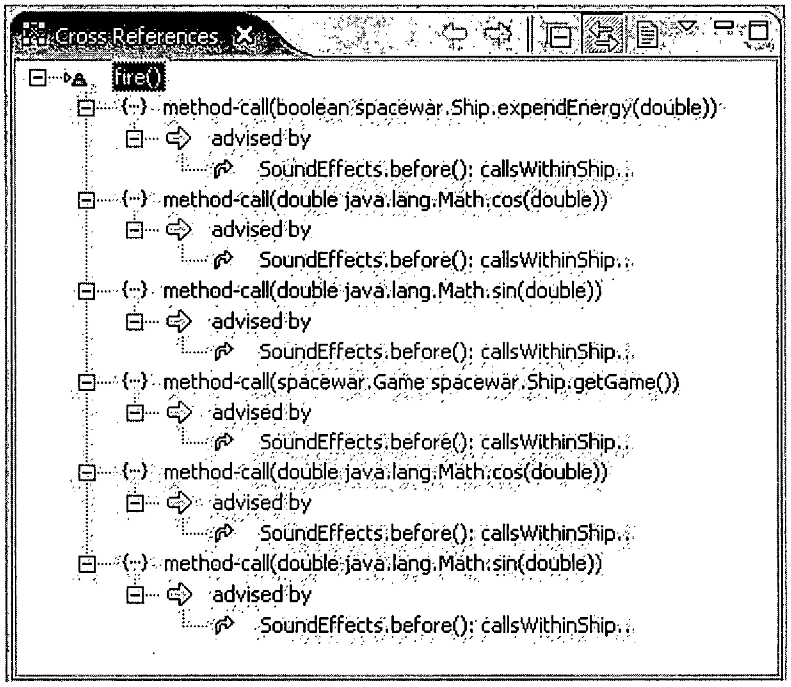
As discussed in the previous section, creating and maintaining correct pointcut expressions can be an onerous and error prone activity. The most common problem with pointcut expressions is that they either capture too many join points or too few.

AJDT's gutter annotations provide some help by allowing developers to visit and inspect the join point sites in the code and spot sites that are incorrectly matched (included or excluded) by a pointcut. However, when such points are discovered AJDT provides no support to rectify the situation. To illustrate our point, we present a typical scenario based on the section of code shown in Figure 1.3, which was taken from an open source video game called Spacewar [1].

The object of Spacewar is for the player to shoot down enemy ships before being shot down. In this scenario, the developer would like to create a version of the game with sound effects. He decides that this feature can be added most easily through an aspect. As a first task, he creates an advice to play a gunshot sound. The pointcut for this advice must match every point during the execution of the program when a ship fires a bullet at another ship. To begin, he creates an Aspect and some advice to play the sound file. He then explores the code base until he finds a method (shown in Figure 1.3) called `fire` in a class called `Ship`. He returns to the Aspect and creates the following pointcut to capture the execution of this method:

```
pointcut gunShot(): execution(void Ship.fire())
```

AJDT's cross references view confirms that a join point will be created for each execution of `fire`. He then runs the program, but soon finds that his new sound feature contains a bug: the sound is played every time the player attempts to fire the



```
Cross References
fire()
  {-} method-call(boolean spacewar.Ship.expendEnergy(double))
    advised by
      SoundEffects.before(): callsWithinShip..
  {-} method-call(double java.lang.Math.cos(double))
    advised by
      SoundEffects.before(): callsWithinShip..
  {-} method-call(double java.lang.Math.sin(double))
    advised by
      SoundEffects.before(): callsWithinShip..
  {-} method-call(spacewar.Game spacewar.Ship.getGame())
    advised by
      SoundEffects.before(): callsWithinShip..
  {-} method-call(double java.lang.Math.cos(double))
    advised by
      SoundEffects.before(): callsWithinShip..
  {-} method-call(double java.lang.Math.sin(double))
    advised by
      SoundEffects.before(): callsWithinShip..
```

Figure 1.2: The Cross-References view of AJDT

```
class Ship {

 void fire() {
  // firing a shot takes energy
  if (!expendEnergy(BULLET_ENERGY))
      return;

  double xV = getXVel() + BULLET_SPEED * (Math.cos(orientation));
  double yV = getYVel() + BULLET_SPEED * (Math.sin(orientation));

  // create the actual bullet
  new Bullet(
    getGame(),
    (getXPos() + ((getSize()/2 + 2) * (Math.cos(orientation))) + xV),
    (getYPos() + ((getSize()/2 + 2) * (Math.sin(orientation))) + yV),
    xV,
    yV);
 }
}
```

Figure 1.3: method Ship.fire()

gun, even if he is out of bullets and no shot is actually fired. In order to fix the problem, he decides that he must refine his pointcut to capture only those join points where a bullet is actually fired. By re-inspecting the Ship.fire method's code, the developer finds that the method only creates a `Bullet` object if an `expendEnergy` check (of line number 3) is passed. To address this, he edits the `gunshot` pointcut as follows:

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

In this example, the developer diagnosed the bug in his program as an incorrectly formulated pointcut. To find the error in the pointcut, however, it was necessary for him to open and inspect the `Ship.fire` method code. This was because the information necessary to form a correct pointcut existed at the join point shadow location, rather than within the pointcut expression. The developer found by analyzing the `fire` method body code that an appropriate fix was to create a pointcut which matched the call to the `Bullet` constructor inside the `fire` method body. We propose that at this point, it would have been desirable if tool support had allowed the developer to perform this change by operating directly on join point representations provided by the tool. Instead the developer was forced to produce the desired effect through an extra level of indirection: navigating outside the location of the join point shadows to the aspect where he could edit the corresponding pointcut expression.

Following the terminology from [11], we say that AJDT produces a *non-effective* view of the join point representations in the code. A non-effective view allows you to see but not change what is contained in the view. The opposite of a "non-effective" is "effective". An effective view of the join points matched by the pointcut would allow the developer to both see and change what join points a pointcut matches.

This brings us back to the thesis of this dissertation, that it is possible to construct IDE-based tools that allow editing of pointcut expressions through direct manipulation of effective representations of join points in the code to which a pointcut applies.

1.3 Thesis Validation

The validation for our thesis is a constructive proof of existence. In other words we constructed a prototype tool that provides an effective view of join point representations in code. The bulk of this dissertation describes the design and implementation of the prototype.

In this section we explain the most important idea behind the tools design and provide an overview of the prototype.

1.3.1 Our Approach: Pointcut By Example

The main idea behind our approach towards building a tool that provides an effective representation of AspectJ pointcuts is to allow the developer to edit the pointcut expression by citing specific examples of shadows or non-shadows in the code.

We believe that since pointcut expressions are written in terms of patterns in the base code, it is most natural to allow the developer to work on a pointcut from within the base code. Our mechanism is called *pointcut by example* because join point shadows in the base code can be used by the developer as specific examples of the kinds of join points he wishes to add or remove from the set of join points the pointcut matches.

1.3.2 A Prototype Implementation

The **Pointcut Wizard** is a tool to support creating pointcuts by example. Using the tool, a developer can specify whether to **add** or **remove** a join point from the pointcut designator from the location of the join point site itself. By allowing the developer to effect changes in the pointcut designator during the exploration process without a constant switching back and forth from the aspect code to the join point shadow site, we believe a great deal of time in both creating and editing pointcut expressions can be saved.

The user interface to the Pointcut Wizard is similar to AJDT. It is an Eclipse plug-in and uses gutter annotations to highlight join point shadows. However the Pointcut Wizard uses *effective* representations of join point shadows, allowing the developer to edit the pointcut expression by specifying join point shadows in the code that it should or should-not match. The user interface the tool provides is in the form of a context menu at each possible join point site. If the shadow on a particular line is already matched by the pointcut expression, the developer is given the option of **removing** it from the set of shadows the pointcut matches. Alternatively, if a line of code contains shadows that are not currently matched, and the developer wishes for an advice to apply at that site, he may **add** the shadows to those matched by the pointcut through the same context menu.

1.4 Conclusion

In this chapter we introduced Aspect Oriented Programming, and in particular the AOP language AspectJ. AspectJ programs can be difficult to maintain without proper tool support for two reasons: the first reason is that the pointcut-advice mechanism can obscure what a given piece of code does. The second reason is that it can be difficult to write a pointcut expression that correctly matches the intended join points.

AJDT addresses these issues to some extent by alerting the developer to where the local join points exist in the code, and the advice that applies at that point. However, if the developer finds that the join points are incorrect he must open the aspect and edit the pointcut expression outside the context of the code where the join point should exist.

The thesis of this dissertation is that it is possible to construct IDE-based tools that allow the editing of AspectJ pointcut expressions through direct manipulation of effective representations of join points in the code to which a pointcut applies.

To validate this thesis, we present the notion of *Pointcuts by Example*. Using this technique, a developer is able to specify examples in the code of join points that a pointcut expression *should* or *should-not* match. As a constructive proof of existence, we developed prototype tool support for editing pointcuts by example called the *Pointcut*

Wizard. The Pointcut Wizard presents a GUI interface for editing pointcut expressions by selecting **add** or **remove** operations at join point shadow sites.

In the following chapter we will examine related work in the area of pointcut editing tools. Chapter 3 will present a more detailed explanation of how the Pointcut Wizard can be used to edit pointcut expressions. In Chapter 4 we will describe the architecture of the Pointcut Wizard. We will conclude with Chapter 5, where we will present our plans for future work in this area and concluding remarks.

Chapter 2

Related Work

This dissertation introduces new tool support for creating and editing AspectJ pointcut expressions.

In this chapter we present alternative methods for supporting the developer in these tasks. The chapter is divided into two sections. The first section reviews related work that motivates the need for better tool support for building and editing pointcut expressions. The second section discusses related work on developing tool support for writing and maintaining pointcut expressions.

2.1 The Difficulty of Writing Pointcuts

In this section we will motivate the need for better tool support for writing and maintaining AspectJ pointcut expressions. We will first examine work by the developers of AJDT which highlights the proper way to extract patterns from the base code in order to form pointcuts that express an intention as well as capture the correct set of join points. Then we will examine work that shows why the process of creating such a pointcut can often be difficult, due to the difficulty in both extracting patterns from the code and enforcing them once the pointcut has been written.

In [5] the authors guide the AspectJ developer in effectively using patterns to design robust pointcuts. A *robust* pointcut accurately portrays the intentions of the developer, capturing join points that exist in the current version of the code as well as raising the probability that pertinent shadows in future versions will be captured. According to the authors, the developer should include in the pointcut only that information which pertains to the join points he wishes to match. All other information should be considered extraneous and replaced with the appropriate wild-card symbol in the pointcut expression. There are three types of wild-card symbols available in AspectJ. The "*" will match any number of alpha-numeric characters in a type name or signature. The ". ." will match a dotted list containing alpha-numeric characters. The "+" must follow a type name and will match any sub-type of the type to which it is attached.

Although the wild-card symbols of the AspectJ language offer the user a great deal of freedom in expressing his intentions, they may not always be sufficient. We see an example of such a case in [6], where the authors identify a major obstacle in extracting a particular cross-cutting concern. The authors state that extracting a concern *often necessitated pointcuts referencing seemingly random, yet specific places within method boundaries of the original implementation.* To capture the correct set of join points, the developer created pointcut expressions which were essentially an enumeration of pointcuts - each matching one of the so called *random, yet specific places*. The cause of this situation, as stated by the authors, was that the code was (obviously) not designed with the Aspect in mind.

This situation is described in more direct terms by the authors of [20]. This work states that although the use of AspectJ may lead to greater modularity, it has not been proven to actually increase evolvability. They describe current AOP languages, and namely AspectJ, as *simplistic* because they are often unable to capture a crosscutting concern without resorting to the enumeration method described above. The authors argue that join points that share common properties that the developer would like to capture with a single pointcut often do not share similarities in naming patterns. Since AspectJ relies on these naming patterns to capture crosscutting concerns, the developer may often find himself searching for patterns in the code that do not exist. When a pattern can not be found, he is left to capture a crosscutting concern by enumerating the patterns found at each join point location. In doing so, he creates very specific references to program elements spread across many modules - greatly increasing the coupling between aspect and base code. The authors believe that this situation occurs frequently enough to make a good case against using languages like AspectJ. They describe the situation as the *AOP paradox*: aspects, which are meant to increase the evolvability of a system may be so closely coupled (and therefore dependent) on the systems current implementation that its evolvability is lessened.

Creating a pointcut that enumerates each join point is a time consuming and error-prone process, and currently available tools do not directly support this method for AspectJ since the resulting pointcut creates a high degree of coupling between aspect and base code. In the following section we will introduce work that uses a more expressive pointcut syntax than AspectJ allows - creating the ability to base pointcuts on properties of join points, rather than patterns in their signatures. Additionally, we will introduce current work on providing tool support to assist the developer in evolving a system that has been made fragile by the coupling between aspect and base code.

2.2 Current Language & Tool Support

In this section we discuss related work on developing language and tool support for writing and maintaining pointcut expressions. Each work is an attempt to address the problem motivated in the previous section: that when a developer resorts to enumerating the set of desired join points in a pointcut, a high degree of coupling between aspect and base code is introduced into the system. This decreases the robustness of the aspect, lowering the evolvability of the system as a whole.

The set of works we will look at can be loosely divided into two groups. The first group suggests an alternative, more expressive pointcut language to answer the coupling problem. The second presents a set of tools to alert the developer of the impact of any changes he is making to the base or aspect code.

2.2.1 Alternative Pointcut Languages

AspectJ's pointcut language is based on patterns, rather than properties in the code. The following two areas of research challenge this implementation. The first is based on using a more expressive logic language to capture properties of join points. The second allows the developer to use meta-tags to annotate the code.

Soul and Inductive Logic

In [4], the authors present the use of a specialized pointcut managing environment to assist the developer in creating pointcut-like expressions written in the logic language *Soul* [21].

As an initial step, the environment gathers background knowledge of the source code in the form of logic facts. Once a database has been formed, the developer begins to create a pointcut by iteratively selecting and de-selecting elements from the source code that he wishes to be captured by the pointcut. As this occurs, the tool actively maintains a set of logic rules that match the developers examples. As a final step, the tool uses an inductive logic technique called *relative least general generalization* [16] to reduce the amount of redundancy in the expression. When an incorrect set of shadows is captured by an expression, the developer can either continue with more iterations of providing positive and negative source code examples to the inductive logic engine, or attempt to manually fix the expression.

An important aspect of this environment is that it continually checks the correctness of pointcut expressions. If the examples used to create the pointcut are refactored or removed, the logic engine recomputes the pointcut to take these changes into effect. This dynamic quality greatly increases the evolvability of an aspect oriented system, because the developer has the option of refactoring the base code without needing to worry about removing a necessary join point.

This work is relevant to ours because the technique used to form and edit pointcut expressions is what this dissertation refers to as *pointcut by example*. The difference between the two is that Soul is a more expressive logic language than AspectJ. Using Soul, the developer is able to express properties of a join point shadow other than its signature and lexical context. The challenge of our work was to create a environment that supported editing AspectJ pointcuts by example.

Annotations

Annotations are meta-data tags that developers can use to classify elements of the code. They are typically processed by build-time tools or run-time libraries. The AOP community has embraced the use of annotations, with tools such as JBoss [12], AspectWerkz [3] and AspectJ [2] each providing tool support. Since we are focused on AspectJ in this dissertation, we will here illustrate the support AJDT provides for programming with annotations using AspectJ.

AJDT allows the developer to both `declare` annotations on elements of the code and to create pointcut expressions that match join points based on annotations. We will here provide an example based on the EJB 3.0 specifications [8], where methods have a transaction policy associated with them that can be annotated as follows:

```
@Tx(TxType.REQUIRED)
void credit(Money amount) {...}
```

Here, the @Tx denotes the type of the annotation, and the TxType.REQUIRED is the value of the annotation. If the developer wishes to use AspectJ to advise all method annotated with the REQUIRED type, he can write the following pointcut:

```
pointcut transactionalMethodExecution(Tx tx) :
   execution(* *(..)) && @annotation(tx) &&
   if(tx == TxType.REQUIRED);
```

The @annotation pointcut is new to AspectJ version 5. It allows the developer to retrieve the value of the annotation (in this case the TxType type) for each join point matched by the pointcut.

This work is pertinent to ours because by using annotations the developer can explore a system of code and annotate program elements as having a property he wishes to capture with a pointcut expression. With AspectJ, it is also possible to use a declare statement to automatically annotate program elements.

The Pointcut Wizard performs a similar task, but the tools gutter annotations are only used by the tool; no meta-data is added directly to the code. Another difference is that the Pointcut Wizard allows the developer to edit a pointcut expression based on examples from the code. Using annotations, the developer has no control over the pointcut - he may only annotate program elements in the code and then write a pointcut expression to capture those join points based on their annotations.

In the next section we will introduce other tools that support the developer in writing or editing pointcut expressions.

2.2.2 Pointcut Development Tools

Another important issue for AspectJ developers is that of making sure pointcuts continue to match intended join points as changes are made to the base code or the Aspect. This can be difficult without proper tool support because the developer may be unaware of where the join point shadows in the code are - and what properties they posses that make them a target join point for advice. In this section we will present some examples of how current tool support supports the developer in editing code while maintaining the operability of both aspect and base code.

AJDT

AJDT [5] is the AspectJ tool suite developed by the AspectJ team. The tools contained in the suite are geared towards helping the developer to understand what impact an Aspect will have on the base code when the program is run. The suite is made up of four tools:

- The Aspect Visualizer presents the developer with a macro-view of the impact an aspect will have on his system. By selecting an aspect, a view is presented in the manner of SeeSoft [7], with join point shadows in the code highlighted in a color determined by the developer.

- The Cross-References view, shown on page 5, provides a structured view of advice and join point shadows in the base code. This view allows a developer to view an outline of which advice applies to the join points in the code with which he is working. Alternatively, an Aspect developer can use this view to determine where an advice he is working on will apply in the base code. This view updates dynamically as changes in the code are made.
- The Crosscutting Comparison View presents a view of how changes the developer has made to either the base code or Aspect code have affected the number of join points. Whereas the other views reveal information about how the aspect is *currently* joining with the base code, this view shows join points that have either recently appeared or recently disappeared as a result of changes to the code.

Gutter annotations appear in both Aspects and Java classes for a developer using AJDT. If the developer is viewing an Aspect, the annotations provide a list of hyperlinks to shadows in the code. If the developer is viewing a join point shadow in the base code, hyperlinks are provided for the developer to open the advice which will be executed at the shadow. These annotations are an intuitive way of keeping the developer alert to the interaction of Aspect and base code in the system he is working with.

Each of these tools provides the developer with an important view of how the aspect code is interacting with the base code. However, none of the tools directly supports the editing of pointcut expressions. It is up to the developer to deduce from the current set of join points how and why a pointcut expression is failing to match shadows he wishes to capture. Additionally, none of these tools are *effective*; they present information dynamically, but provide no support in diagnosing or fixing problems. Therefore, if the developer needs to troubleshoot a pointcut that is capturing too many join points, he must first navigate to the advice which will execute at the join point. Then he must navigate to the pointcut on which the advice is based. He must then begin an iterative process of changing the pointcut and observing join points listed in the outline view until the correct set appears.

PCDiff

PCDiff [18] provides the same information as the Crosscutting Comparison View of AJDT, but with a better visualization. The tool uses three different types of views. Firstly, if a change the developer has made in the base code has added or removed a join point, a *plus* or *minus* sign appears in the gutter next to the affected position. Additionally, the creators of this tool make use of the Eclipse task view. Entries to the task view are made every time a shadow is added or removed. Should the developer find that a change he has made has negatively impacted the Aspect's functionality, he may find this view very useful to scan through, in order to diagnose the problem. Lastly, the tool provides a separate view which is very similar to the Crosscutting Comparison View. As the shadows are added or removed they appear in a structured view which lists the Java compilation unit, the Java class, the shadow type, and the advice whose join points have been affected. Although this tool provides the developer with a better visualization than AJDT for how a change has impacted the number of join points, similarly to AJDT it provides no assistance in diagnosing or fixing the associated pointcut.

2.3 Conclusions

Our dissertation states that it is possible to construct IDE-based tools that allow editing of AspectJ pointcut expressions through direct manipulation of effective representations of join points in the code to which a pointcut applies. In this chapter we presented two groups of work focused on supporting the developer in creating and editing pointcut expressions.

The first set of works presented language-based features geared toward allowing the developer to match join points based on their properties rather than on their patterns. The first group developed an environment with support for editing Soul pointcut-expressions *by example*. The second group provided support that allowed the developer to annotate and match join point shadows with Java annotations.

The second group of works presented tool support for managing pointcut expressions. The first work we looked at was the AJDT tool suite. The tools in this suite were designed to illustrate the relationship between AspectJ aspects and the base code. This involved gutter annotations at join point locations, and a cross-references view that provided an outline of information regarding local join points. The second tool, *PCDiff*, was focused on alerting the developer of any impact on the set of local join points that a current refactoring or editing of the code was having.

Chapter 3

The Tool

The Pointcut Wizard is unique because it allows the developer to edit a pointcut expression from the location of the join point shadows. This support is made possible through the use of *effective* gutter annotations. The operations supported by the Pointcut Wizard include adding or removing shadows in the code from the set of shadows matched by a pointcut expression. As the developer explores the base code, shadows matched by the pointcut are shown with annotations in the gutter of the editor next to the line where the shadow exists. As shadows are added or removed through either interaction with the gutter annotation or direct manipulation of the pointcut expression, the gutter annotations are refreshed to reflect each change.

This chapter will begin with a more formal description of the AspectJ pointcut language and an explanation of the features the Pointcut Wizard supports. Following that, we will give a detailed explanation of how to use the Pointcut Wizard to edit a pointcut while browsing Java code. Additionally, we will explain the transformation effected on the pointcut expression through operations performed by the Pointcut Wizard.

3.1 The AspectJ Pointcut Language

The Pointcut Wizard is able to parse and annotate shadows for *lexical-structure based pointcuts*, *binding pointcuts* and a subset of *kinded pointcuts.*[1] In this section we will explain each of these categories and which members of each category are supported by the tool.

Kinded pointcuts: are pointcuts that each match a corresponding category of join points. For example, the `execution` pointcut matches method executions and the `call` pointcut matches method calls. Table 3.1 is a listing of all kinded pointcuts in AspectJ version 1.1. The Pointcut Wizard supports a subset of kinded pointcuts. In the table, the right-most column states whether the pointcut for that row is supported or not.

We believe that the subset of kinded pointcuts we implemented for the prototype of the Pointcut Wizard is enough for a developer to use the tool effectively. We plan to add support for the rest of this set in production versions of the tool. We describe our plans to do so in the *Future Work* section of Chapter 5.

Lexical-structure based pointcuts: are pointcuts that capture all join points occurring inside the lexical scope of specified classes, aspects or methods. The Pointcut Wizard supports each of the two lexical-structure based pointcuts of the AspectJ pointcut language - shown in Table 3.2.

[1]This categorization of pointcut expressions was used in [14]

Table 3.1: Kinded Pointcuts

Join Point Category	Pointcut Syntax	Supported by PW
Method execution	execution(MethodSignature)	yes
Method call	call(MethodSignature)	yes
Constructor execution	execution(ConstructorSignature)	yes
Constructor call	call(ConstructorSignature)	yes
Class initialization	staticinitializatiion(TypeSignature)	no
Field read access	get(FieldSignature)	yes
Field write access	set(FieldSignature)	yes
Exception handler execution	handler(TypeSignature)	no
Object initialization	initialization(ConstructorSignature)	no
Object pre-initialization	preinitialization(ConstructorSignature)	no
Advice execution	adviceexecution()	no

Table 3.2: Lexical-Structure based pointcuts

Pointcut Syntax	Join Point Category
within(TypePattern)	Any join point occurring within a class or aspect
withincode(MethodSignature)	Any join point occurring within a method body
withincode(ConstructorSignature)	Any join point occurring within a constructor body.

Binding pointcuts: can be used in two ways. First, if the developer wishes to access an object's state at run-time, he can edit the pointcut expression by hand to take an object of that type as an argument. He can then use a binding pointcut to bind the argument to run-time objects during program execution.

Second, the developer can pass a type name to a binding expression. Any run-time object that matches a binding pointcut in both category and signature will be matched by the pointcut. For example, in Table 3.3 below, there is a binding pointcut of type `this`. If a developer were to use this pointcut and pass to it as an argument a type `Foo`, the pointcut would match every join point in the code where `this` is of type `Foo`.

Table 3.3 presents a list of all binding pointcut types:
To support binding pointcuts, the Pointcut Wizard performs a static analysis on the code to determine (and annotate) all join point shadows where the run-time object is possibly of the type passed to a binding pointcut as an argument. However, it does not allow the developer to create pointcut designators of this type while browsing the code. To make such a feature useful, the PW would need to also support the developer in selecting variables and fields in the code that he wishes to bind. We do believe each of these features is useful though - a topic we discuss in the Future Work section of Chapter 5.

Control-flow and conditional pointcuts, shown in Table 3.4, are dynamic pointcuts and do not have a direct representation in the source code. The conditional check pointcut *if* also depends on run-time information to determine a shadow and was therefore not included.
A developer using AJDT is alerted to the dynamic nature of a join point shadow by

Table 3.3: Binding pointcuts

Pointcut Syntax	Join Point Category
this(TypePattern or Object Identifier)	All join points where `this` is an object of the type passed as an argument to the pointcut.
target(TypePattern or Object Identifier)	All join points where the object on which a method call is performed is of the type passed as an argument to the pointcut.
args(List of TypePattern or Object Identifier)	All join points where the method or constructor signature matches the list of types passed in as an argument to the pointcut.

Table 3.4: Control-flow and conditional pointcuts

Pointcut Syntax	Join Point Category
cflow(pointcut)	All join points in the control flow of any join point matched by the pointcut passed as an argument.
cflowbelow(pointcut)	All join points within the control from of any join point matched by the pointcut, excluding the actual join points themselves.
if(boolean expression)	Does not match any join points, only evaluates to a boolean value.

tool-tips that state a run-time check is performed at that location.

The Pointcut Wizard prototype does not support these types of pointcuts. In order to support them, the tool would require a more expressive user interface. It would also need a *resolver* component capable of resolving `cflow` pointcuts. Our thoughts on design features to address these issues are discussed in the *Future Work* section of Chapter 5.

3.1.1 Wild-card symbols

The Pointcut Wizard's support for wild-card symbols can be best explained broken down into two categories: support for finding all shadows that match a pointcut expression, and support for building pointcut expressions with wild-card symbols based on examples from the code. We will discuss the extent of support for each category here.

The three wild-card symbols that AspectJ supports are "*", "+", and "`..`". The following list presents an explanation of the meaning of each type of wild-card, as well as the extent to which each is supported by the Pointcut Wizard.

The **star pattern**, "*", denotes any number of characters except the period. Using this pattern the AspectJ developer is able to form regular expressions to match patterns in the code. For example, he can match all method names that contain the word *log* with the following expression: `*log*`. Using AspectJ, the star can be used at the middle, beginning, end or both sides of a set of characters.

The Pointcut Wizard supports the star pattern only for the case where the star is not combined with any other characters. A developer can therefore either use a star to capture *all* names or he can use the actual name. In our *Future Work* section in Chapter 5 we present our plans for providing full support for the star pattern in a future version of the Pointcut Wizard.

The **plus pattern**, "+" denotes all *sub-types* of any type name it follows. For example, the following pointcut:

```
pointcut starPatEx(): call(* Foo+.*(..));
```

will match any method call where the receiver class is of type `Foo` or any of its sub-types. This operator can be used on any type name, in any type of pointcut designator AspectJ supports.

The Pointcut Wizard does not currently support this operator.

The **dot-dot operator** "`..`" denotes any number of characters including any number of periods. Using this pattern the developer is able to match fully qualified path names and lists of arguments. Consider the following example:

```
pointcut dotDotPatEx(): call(* java.util..*.*(.., Bar))
```

The receiver of this method call can be any type contained `java.util` or any of its sub-packages. The arguments list can match any arguments list as long as it ends in type `Bar` - including the case where `Bar` is the only argument.

The Pointcut Wizard prototype supports the use of this pattern in place of an arguments list to match signatures with any number of arguments.

3.2 A User Interface for Shadow Browsing

In this section we will present the user interface for exploring the set of shadows matched by a pointcut expression.

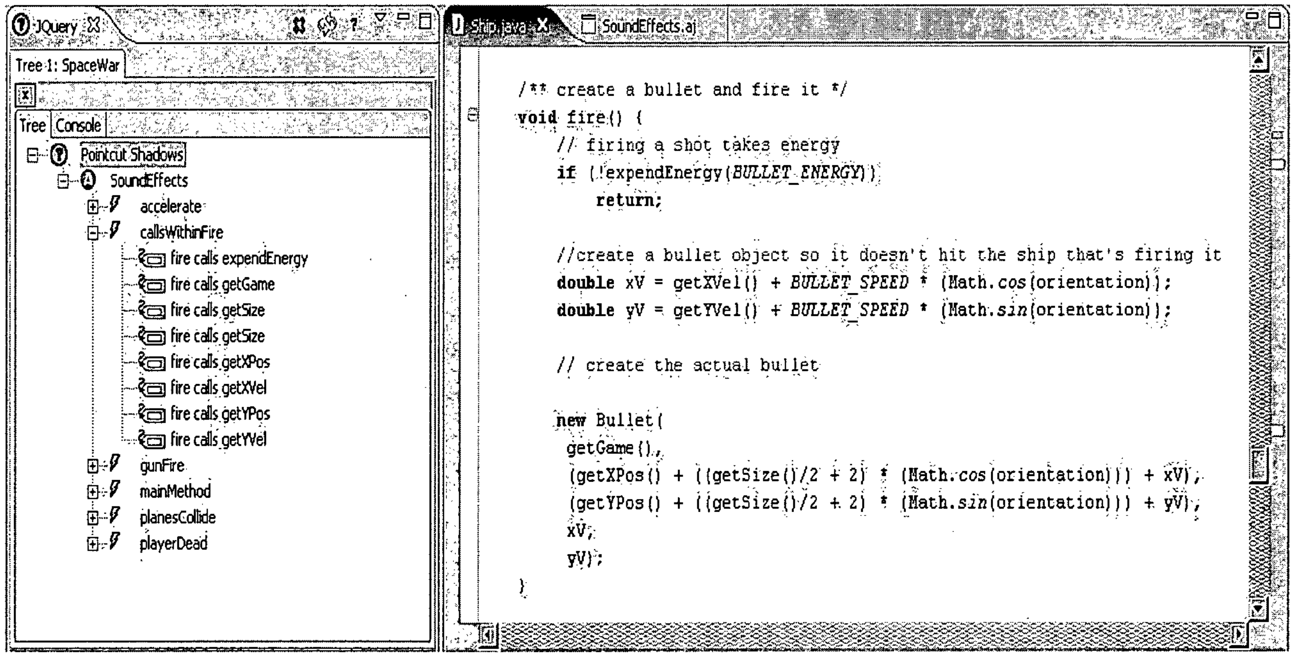
The developer can begin using the Pointcut Wizard view by opening an Aspect. The view organizes information about the Aspect as a tree hierarchy, with the Aspect as the root node, pointcut expressions as child nodes of each Aspect, and shadows as the children of each pointcut. The view on the left-hand side of Figure 3.1 shows an example of an Aspect open for editing in the Pointcut Wizard view. In the view the top level node, named *Pointcut Shadows*, notes the selection made by the developer to reveal all shadows matched by a pointcut expression in any of the Aspects with which the developer is working. Selecting any of the nodes in this view will open the associated program element in the code editor.

In this example, the developer has opened the *SoundEffects* Aspect and selected the node for a pointcut named *callsWithinFire*. The node has expanded to reveal a list of all join point shadows in the code that match *callsWithinFire*. The expression for the pointcut is shown bellow. It was written to capture all method calls made from within the method `fire` in class `Ship`.

```
pointcut callsWithinFire(): (call(* *.*(..)) &&
       withincode(void Ship.fire()));
```

To view shadows in the Java code editor, the developer must select an **Active Pointcut**. In order to reduce the complexity involved in navigating context menus at the shadow sites, only the shadows for an active pointcut are shown in the code editor. When a pointcut is *active*, context menus at join point shadow sites pertain only to that particular pointcut. To make a pointcut active, the developer must select that option from a context menu at the pointcut node in the Pointcut Wizard view.

Once the developer has chosen an active pointcut, lines in the code editor that contain a shadow matched by the pointcut are marked by a pointcut gutter annotation. In Figure 3.2, the developer has selected *callsWithinFire* as his active pointcut. The lightning bolt gutter annotations along the left-hand side of the code editor each signify that at least one matched shadow exists on that line.



```
Pointcut Shadows
  SoundEffects
    accelerate
    callsWithinFire
      fire calls expendEnergy
      fire calls getGame
      fire calls getSize
      fire calls getSize
      fire calls getXPos
      fire calls getXVel
      fire calls getYPos
      fire calls getYVel
    gunFire
    mainMethod
    planesCollide
    playerDead
```

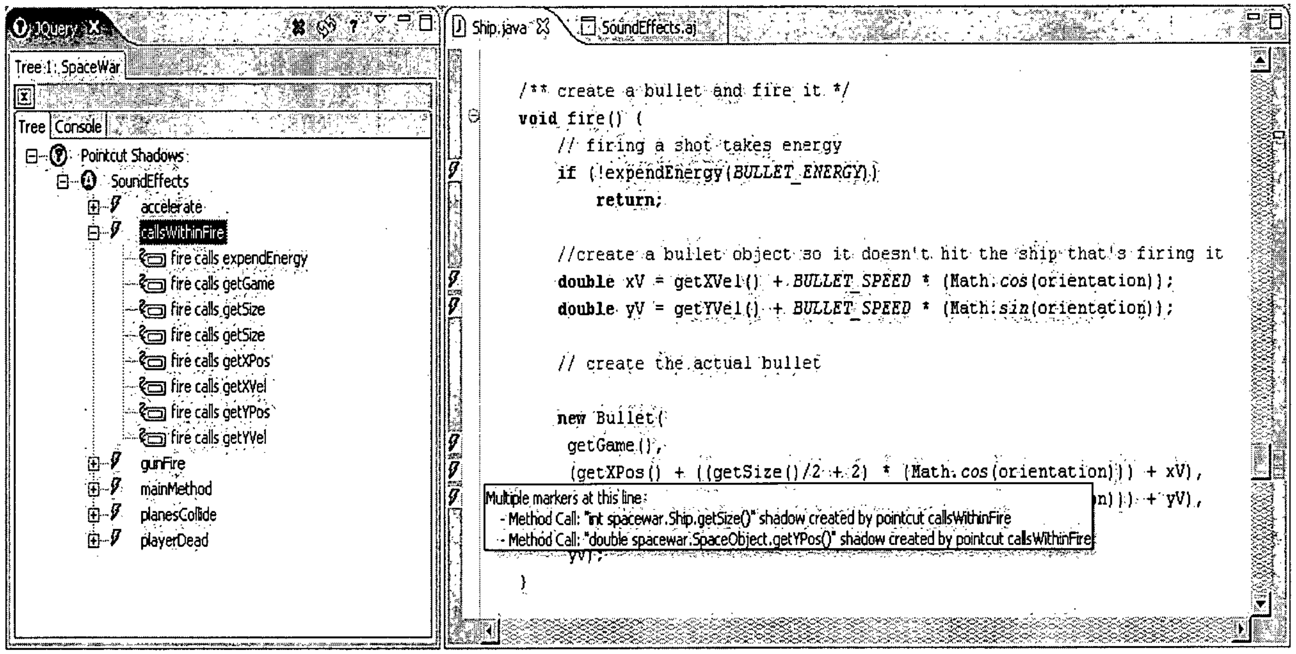
```
/** create a bullet and fire it */
void fire() {
    // firing a shot takes energy
    if (!expendEnergy(BULLET_ENERGY))
        return;

    //create a bullet object so it doesn't hit the ship that's firing it
    double xV = getXVel() + BULLET_SPEED * (Math.cos(orientation));
    double yV = getYVel() + BULLET_SPEED * (Math.sin(orientation));

    // create the actual bullet

    new Bullet(
     getGame(),
     (getXPos() + ((getSize()/2 + 2) * (Math.cos(orientation))) + xV),
     (getYPos() + ((getSize()/2 + 2) * (Math.sin(orientation))) + yV),
     xV,
     yV);
}
```

Figure 3.1: The Pointcut Wizard



```
Pointcut Shadows
  SoundEffects
    accelerate
    callsWithinFire
      fire calls expendEnergy
      fire calls getGame
      fire calls getSize
      fire calls getSize
      fire calls getXPos
      fire calls getXVel
      fire calls getYPos
      fire calls getYVel
    gunFire
    mainMethod
    planesCollide
    playerDead
```

```
/** create a bullet and fire it */
void fire() {
    // firing a shot takes energy
    if (!expendEnergy(BULLET_ENERGY))
        return;

    //create a bullet object so it doesn't hit the ship that's firing it
    double xV = getXVel() + BULLET_SPEED * (Math.cos(orientation));
    double yV = getYVel() + BULLET_SPEED * (Math.sin(orientation));

    // create the actual bullet

    new Bullet(
     getGame(),
     (getXPos() + ((getSize()/2 + 2) * (Math.cos(orientation))) + xV),
                                                                   n))) + yV),
     yV);
}
```

```
Multiple markers at this line:
 - Method Call: "int spacewar.Ship.getSize()" shadow created by pointcut callsWithinFire
 - Method Call: "double spacewar.SpaceObject.getYPos()" shadow created by pointcut callsWithinFire
```

Figure 3.2: Pointcut callsWithinFire is now an Active Pointcut

Since more than one join point shadow may exist on a particular line, the developer can hover over an annotation to view a tool tip listing the pointcut, the shadow, and the type of shadow for each shadow in that line of code. In Figure 3.2, he has prompted a tool tip which reveals the two shadows on a line of the `fire` method: a method call to `Ship.getSize()` and a method call to `SpaceObject.getYPos()`. We will now explain how the developer can interact with a menu structure at these shadow sites to effect changes in the active pointcut.

3.3 The UI for Editing Pointcuts

In this section we will illustrate the process a developer can use to edit a pointcut while browsing code with the Pointcut Wizard.

There are a range of possibilities for implementing a system where the developer can edit a pointcut by selecting examples from the code. The most popular of these is by applying inductive logic. In Chapter 2 we presented an example of such work [4] where a pointcut editing environment used inductive logic to reason about the examples using information based on the current structure of the pointcut expression combined with information from a database containing all structural relationships of the program. However, in order to build this tool, the authors used a logic language called Soul to express and edit pointcut expressions.

The Pointcut Wizard was designed to work with AspectJ pointcut expressions, which are written in a less expressive language than Soul. Therefore, the information used in performing an edit operation on a pointcut comes only from the join point shadow chosen as an *example* by the developer.

Because it is unclear what the developer is interested in when selecting a shadow as an example, the Pointcut Wizard uses what we call a *most-specific* description to identify it. This description identifies a shadows type, its signature, and the lexical context within which it exists. This set of properties regarding a shadow make up what we call the *Shadow Model* component of the Pointcut Wizard. A further explanation of the model will be presented in Chapter 4.

In the remainder of this section we will present the user interface for **adding** and **removing** shadows matched by a pointcut expression. Following the description of each operation, we will also present the impact for each type of operation on the pointcut expression.

3.3.1 Adding Shadows

Using the Pointcut Wizard to add a shadow is a very simple process. To illustrate the steps involved, we will continue with the same example from Figures 3.1 and 3.2.

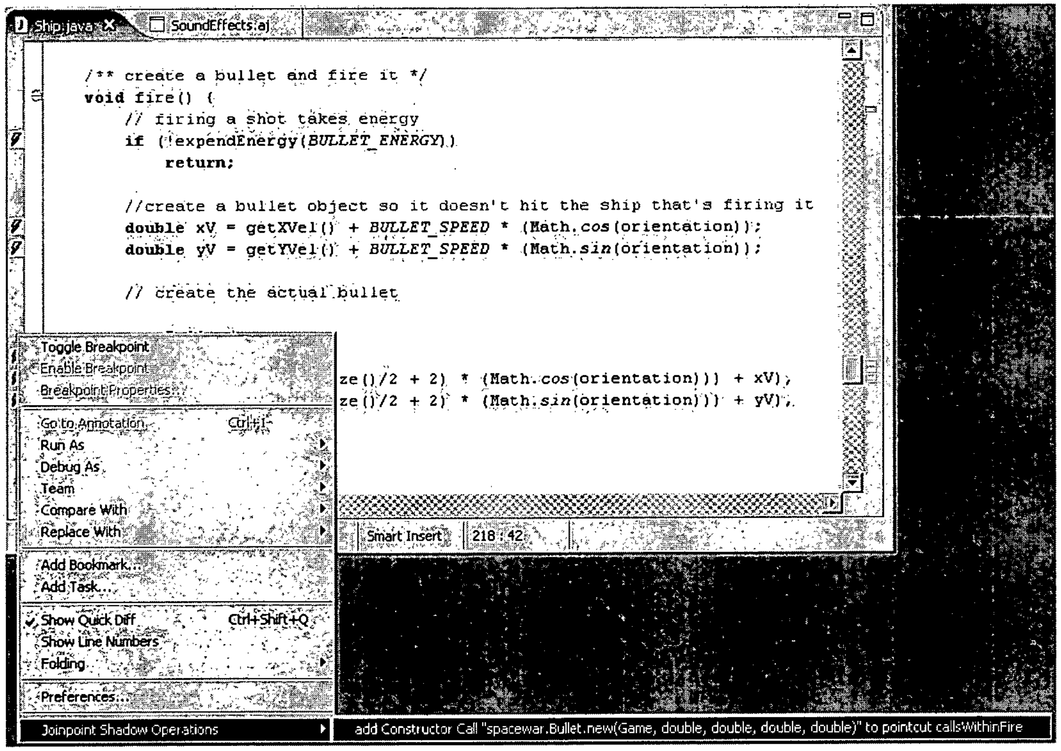
In the previous examples, the pointcut expression *callsWithinFire* was used to capture all method calls made from within `Ship.fire()`. Suppose that the developer had written a *tracing* advice with the intention that it trace *all* outgoing calls from `fire`. If this were the case, the developer would soon realize that the constructor call to `Bullet` was not being included in the trace. The reason is that a *call* pointcut can use a method signature or a constructor signature, but one signature type can not match both types of join points. In the remainder of this section we will highlight the steps involved in using the Pointcut Wizard to change *callsWithinFire* so that it captures the constructor call.

As a first step, the developer navigates to the `fire` method in the class `Ship`. He then opens the Pointcut Wizard view and selects *callsWithinFire* as his active pointcut. Once the shadows have been marked with a gutter annotation (as shown in Figure 3.2), the developer notices that there is no gutter annotation next to the constructor call to `Bullet`.

To add the constructor call to the set of shadows captured by *callsWithinFire*, the developer opens a context menu at the shadow site, as shown in Figure 3.3 on page 24. Under the *Joinpoint Shadow Operations* sub-menu, he selects the option:

```
add Constructor Call:
     ''spacewar.Bullet.new(Game, double, double,
                           double, double)''
to pointcut callsWithinFire
```

As shown in the figure, the Pointcut Wizard includes the *type* of the shadow (constructor call), the full *signature* of the shadow (`spacewar.Bullet.new(Game, double, double, double, double)` and the active pointcuts name (*callsWithinFire*). Once the section has been made, both the Pointcut Wizard view and the code editor are refreshed to show the new shadow being captured by *callsWithinFire*. The developer runs the trace program and finds that the fix was successful.



```
    /** create a bullet and fire it */
    void fire() {
        // firing a shot takes energy
        if (!expendEnergy(BULLET_ENERGY))
            return;

        //create a bullet object so it doesn't hit the ship that's firing it
        double xV = getXVel() + BULLET_SPEED * (Math.cos(orientation));
        double yV = getYVel() + BULLET_SPEED * (Math.sin(orientation));

        // create the actual bullet
```

```
ze()/2 + 2) * (Math.cos(orientation))) + xV),
ze()/2 + 2) * (Math.sin(orientation))) + yV),
```

add Constructor Call "spacewar.Bullet.new(Game, double, double, double, double)" to pointcut callsWithinFire

Figure 3.3: Adding a Constructor Call to the Active Pointcut

Impact on the Pointcut Expression

The Pointcut Wizard uses detailed information about a shadow when forming the pointcut to append to the current pointcut expression. In the above example, before the developer had used the tool to change the pointcut expression, it had appeared as follows:

```
pointcut callsWithinFire(): (call(* *.*(..)) &&
         withincode(void Ship.fire()));
```

When the selection was made, the Pointcut Wizard queried the site for the full signature of the **Bullet** constructor and the method from within which it was made. All of this information was used in creating the *most-specific* pcd, as follows:

```
call(spacewar.Bullet.new(Game, double, double, double, double))
&& withincode(void Ship.fire())
```

The current implementation of the Pointcut Expression uses detailed information to describe the join point shadow so that no false positives are formed when the active pointcut expression is edited to match the new join point. Following the **add** operation, the pointcut expression will look as follows:

```
pointcut callsWithinFire():
((call(* *.*(..)) && withincode(void Ship.fire())) ||
(call(spacewar.Bullet.new(Game, double, double, double, double))
 && withincode(void Ship.fire())))
```

A side effect of this method, however, is that the pointcut expression grows considerably longer with each newly added shadow. Our methods for dealing with this issue are presented in the following chapter, where we explain the tool's implementation.

3.3.2 Removing Shadows

To illustrate the process of removing shadows from the set of shadows matching a pointcut, we will continue with our *callsWithinFire* example.

Suppose that while the developer was inspecting the shadows within the method `Ship.fire()`, he realized that the call to `Ship.expendEnergy` should not be captured by the pointcut. To remove this shadow, the developer must open the context menu, as shown in 3.4, and select the following option:

```
   remove Method Call:
        ''boolean spacewar.Ship.expendEnergy(double)''
   from pointcut callsWithinFire
```

Once the selection has been made, the Pointcut Wizard gathers a set of detailed information about the shadow and generates the following pcd:

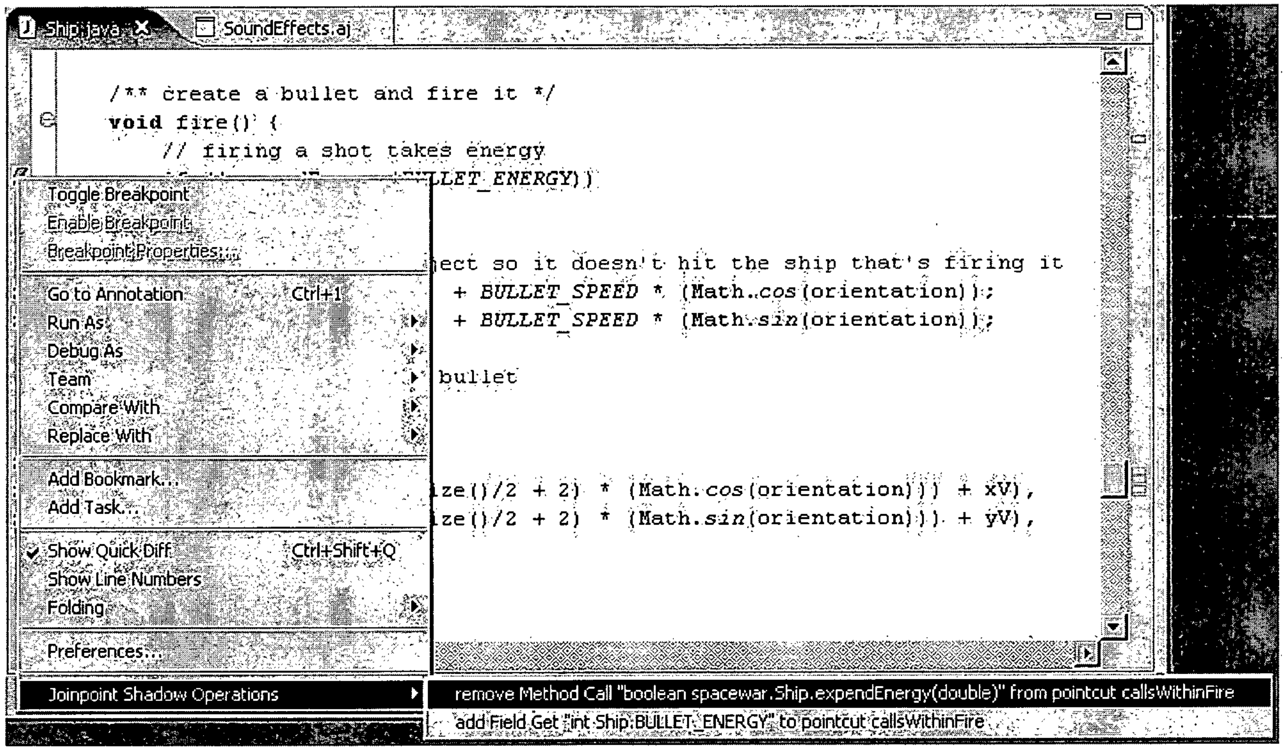
```
call(boolean spacewar.Ship.expendEnergy(double)) &&
withincode(void Ship.fire())
```

Because the selection was to remove the shadow from the set of shadows matching the pointcut expression, the pcd is **negated** and then appended to the pointcut with a logical **AND**.

Having used the Pointcut Wizard to add a shadow and remove a shadow, the developer has fixed the incorrect pointcut expression without ever leaving the code where the shadows exist. Additionally, the pointcut expression *callsWithinFire*, shown below, has been edited in a methodical manner, so that it can be read and edited directly if necessary:

```
(
((call(* *.*(..)) && withincode(void Ship.fire())) &&
  !(call(boolean spacewar.Ship.expendEnergy(double))))
||
(call(spacewar.Bullet.new(Game, double, double, double, double))
 && withincode(void Ship.fire()))
)
```

The resulting pointcut expresses the same semantic meaning, but has been *normalized*. The technique used is described in Chapter 4.



```
    /** create a bullet and fire it */
    void fire() {
        // firing a shot takes energy
                                  LLET_ENERGY))

                                  ect so it doesn't hit the ship that's firing it
                                    + BULLET_SPEED * (Math.cos(orientation));
                                    + BULLET_SPEED * (Math.sin(orientation));

                                  bullet



                                  ze()/2 + 2) * (Math.cos(orientation)) + xV),
                                  ze()/2 + 2) * (Math.sin(orientation)) + yV),
```

remove Method Call "boolean spacewar.Ship.expendEnergy(double)" from pointcut callsWithinFire

add Field Get "int Ship.BULLET_ENERGY" to pointcut callsWithinFire

Figure 3.4: Removing a Method Call from the Active Pointcut

3.3.3 Evaluating the Resulting Expression

The original intention of the `callsWithinFire` pointcut expression was to capture all calls made from within the method `Ship.fire()`. The original version of the pointcut appeared as follows:

```
pointcut callsWithinFire(): (call(* *.*(..)) &&
         withincode(void Ship.fire()));
```

The developer then edited the pointcut twice. The first edit was necessary because the pointcut did not capture constructor calls made from within `Ship.fire()`. The **add** operation he performed with the Pointcut Wizard resulted in the following expression:

```
((call(* *.*(..)) && withincode(void Ship.fire()))
||
(call(spacewar.Bullet.new(Game, double, double, double, double))
 && withincode(void Ship.fire())))
```

The developer then performed a **remove** operation using the Pointcut Wizard to express his new intention to capture all method and constructor calls *except* the one to `Ship.expendEnergy`. The edit resulted in a pointcut of the following form:

```
(
((call(* *.*(..)) && withincode(void Ship.fire())) &&
 !(call(boolean spacewar.Ship.expendEnergy(double))))
||
(call(spacewar.Bullet.new(Game, double, double, double, double))
 && withincode(void Ship.fire()))
)
```

We compare this to the following *best case* form of the pointcut expression.

```
((call(* *.*(..)) || call(*.new(..)) &&
withincode(void Ship.fire())
```

Each version of the pointcut expression `callsWithinFire` correctly matches all intended join points. However, the use of wild-card symbols in the second version creates a pointcut expression that matches not only the constructor call to `Bullet`, but also any other constructor calls that may exist in a future version of the method.

In future versions of the Pointcut Wizard we plan to increase support in the user interface for allowing the developer more freedom in creating a pointcut expression that best reflects his intentions. One possible option is presented in the *Future Work* section of Chapter 5. The Figure 5.1 on page 45 presents a mock-up of a possible user interface that provides the developer with check-box options for specifying which properties of a signature should be used to match join points.

3.4 Conclusion

This chapter covered two aspects of the Pointcut Wizard design. The first half of the chapter provided an explanation of the types of AspectJ pointcut expressions the tool currently supports. Following that explanation we provided an overview of the user interface presented to the developer for the adding and removing of shadows from the set matched by a pointcut expression.

We felt it was important to provide a brief explanation of the types of changes a developer using the PW will see in a pointcut expression after performing **add** or **remove** operations. This explanation was provided as an overview of the tool's functionality, rather than as an explanation of how the tool is implemented. In Chapter 4 we take a more detailed approach to describing the implementation of the Pointcut Wizard.

Chapter 4

Implementation

4.1 Introduction

In this chapter we will present the design of the Pointcut Wizard. First we will begin with an overview of the tool's architecture. The Pointcut Wizard tool consists of four components: the *shadow model*, the *disambiguator*, the *normalizer* and the *resolver*. The shadow model was built to model all join point shadows in the code. The other three components share information pertaining to a shadow based on this model. The disambiguator is responsible for deciding the meaning of an operation performed at a join point shadow and then mapping the information found at the join point site to a poincut designator that will match it. The normalizer is responsible for reducing the pointcut to normal form. Doing so allows the Pointcut wizard to remove redundancies that may appear in the pointcut expression following successive **add** and **remove** operations. The resolver is responsible for identifying the set of shadows in the code which match the pointcut expression. This information is sent to the user-interface so that each shadow can be identified with a gutter annotation.

The data flow between these components is illustrated in Figure 4.1. To understand the diagram, imagine that a developer has selected an *active pointcut* and opened the Pointcut Wizard menu for a particular line of code. Before the menu is opened, the disambiguator queries the shadow model for the set of shadows that exist on that line. If the shadows are currently matched by the pointcut expression, the tool presents the shadows information with a **remove** option, and if a shadow is not currently matched by the expression, the tool will present an **add** operation.

The tool then waits for a developer to make a selection from the menu. Once a selection has been made, the disambiguator creates a pointcut expression *PCD^* that matches the selected shadow. Then, depending on the operation selected, the tool acts one of two ways. If the **add** operation has been selected, the tool forms a new pointcut *PCD'* by ORing the active pointcut *PCD* with the pointcut that matches the selected shadow *PCD^*. If a **remove** operation is chosen, the tool creates *PCD'* by AND-NOTing *PCD* with *PCD^*.

The outcome *PCD'* of whichever operation has been chosen is then taken by the normalizer component which reduces it to normal form. The newly formed pointcut expression is then written to the aspect as a replacement for the active pointcut. The resolver component then annotates all join point shadows that match the new version of the pointcut expression.

The remainder of this chapter will provide a more detailed description of the implementation of each of these four components.

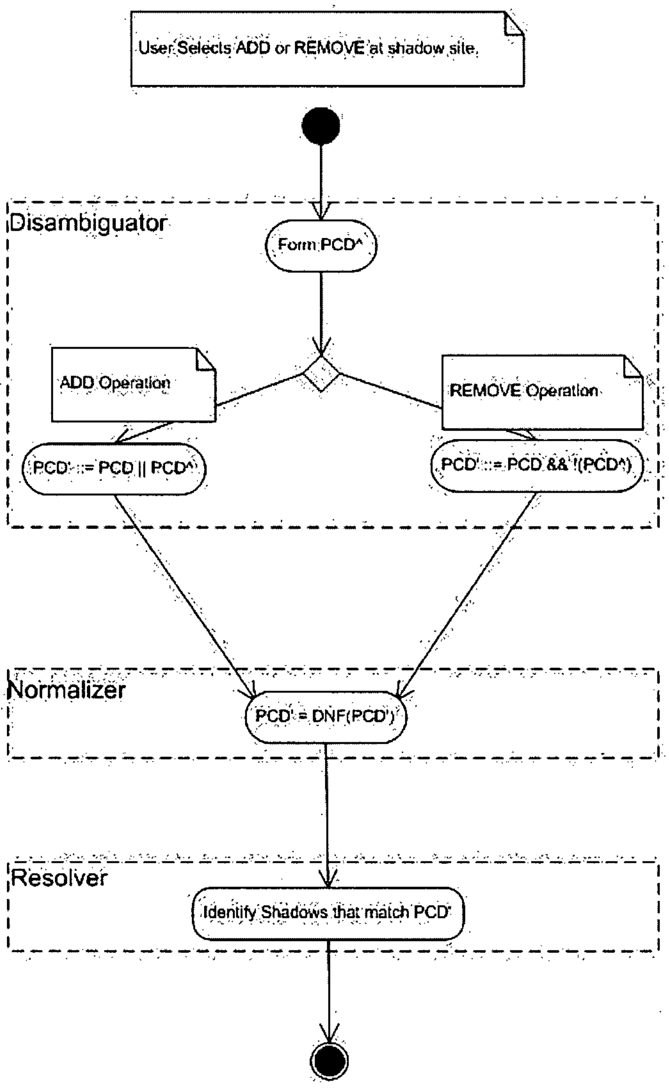


Figure 4.1: Data Flow Between Components

4.2 The Shadow Model

The shadow model is used by the Pointcut Wizard to model all shadows in the code. It is used by the other components of the tool as a mechanism for sharing information regarding shadows. In this section we will illustrate the design and implementation of the model.

The shadow model was built on top of a source code browser called JQuery [10]. JQuery uses a declarative database as a back-end to query on properties of elements of a code base. When the Pointcut Wizard is being used, the properties of each shadow in the code can be modeled using this same database.

The class diagram in Figure 4.3 on page 34 is a representation for all of the properties stored in the database for each shadow in the code. The diagram shows that all shadows contain a `SourceLocation` field which represents their location in the source code. Each *SourceLocation* object in the database is unique. Therefore, the shadow model can extract information about a shadow from the database based on a location in the source code.

As an example of how the shadow model is used, suppose the developer wishes to perform an **add** operation for a shadow on a particular line of code. The *disambiguator* component first creates a query that returns all shadows whose location in the source code lies within that line. Then, another query is performed to find the types of shadows that exist on that line. The possible shadow types are represented by the third layer of the hierarchy in Figure 4.3.

When the shadow types for the shadows that exist on a line of code are known, the signature properties for those shadows can be found through another set of queries. The shadow types must be known before these queries can be created because different shadow types are parameterized by different signature properties.

The second layer of the hierarchy in Figure 4.3 represents the three categories of shadows that the Pointcut Wizard supports. The difference between the categories of shadows are the signatures by which they are parameterized. The `CallShadow` and `FieldAccessShadow` shadow types each contain a *thisSignature* field which is of type `SignaturePattern`. A class diagram representation of signature types is shown in Figure 4.2. The diagram shows that an object of type `SignaturePattern` will either be a *constructor signature* or a *method signature*. This is logical, because a field access or call site will be made either from within a constructor or a method call.

Each of the shadow types expressed as the third layer of the shadow model hierarchy is parameterized by a *targetSignature*. This signature represents the signature properties of the shadow itself, so each type of shadow contains a *targetSignature* of the appropriate type.

In the next three sections we will describe the implementations of the components illustrated in Figure 4.1.

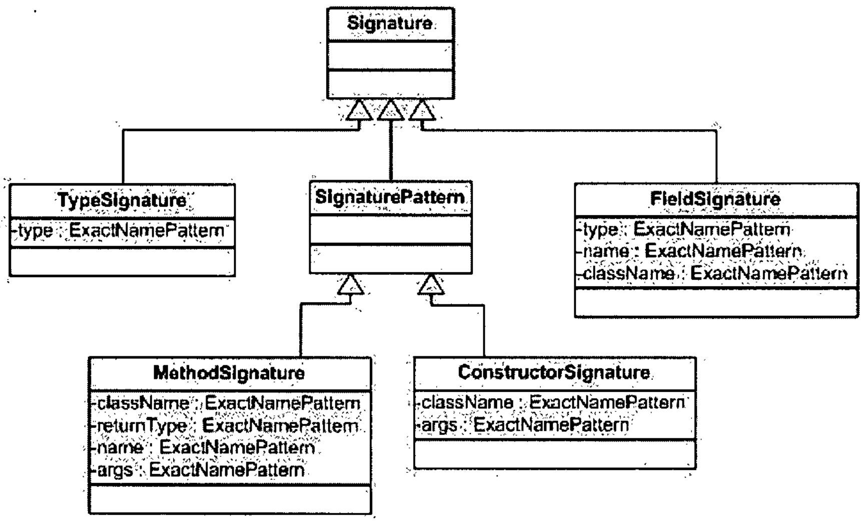


Figure 4.2: Shadow Signatures

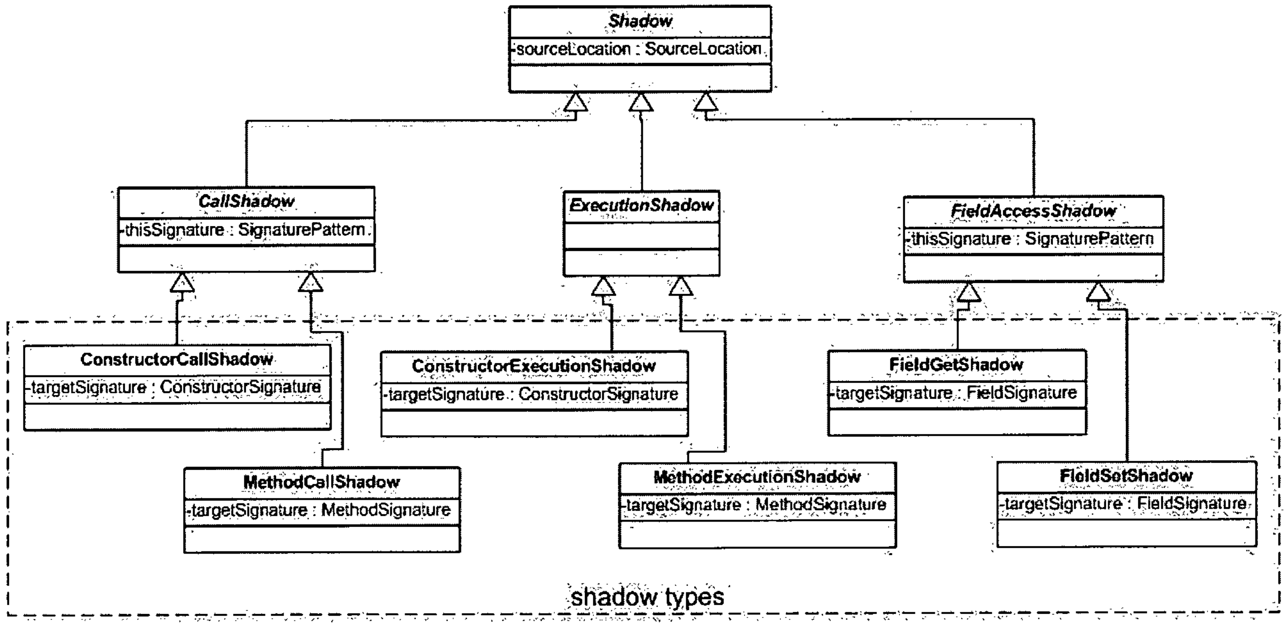

Shadow
-sourceLocation : SourceLocation
CallShadow
-thisSignature : SignaturePattern
ExecutionShadow
FieldAccessShadow
-thisSignature : SignaturePattern
ConstructorCallShadow
-targetSignature : ConstructorSignature
MethodCallShadow
-targetSignature : MethodSignature
ConstructorExecutionShadow
-targetSignature : ConstructorSignature
MethodExecutionShadow
-targetSignature : MethodSignature
FieldGetShadow
-targetSignature : FieldSignature
FieldSetShadow
-targetSignature : FieldSignature
shadow types

Figure 4.3: The Shadow Model

4.3 The Disambiguator

The disambiguator is responsible for deciding the meaning of an operation performed at a join point shadow and then mapping relevant information from the shadow site to a pointcut designator. In this section we will describe the implementation of the component.

When the developer selects an *active pointcut*, the resolver component of the Pointcut Wizard creates a representation for each shadow in the code that it matches. The UI of the tool then creates menu options at each site where a shadow exists. When the developer selects an option, the disambiguator builds a new pointcut designator based on the type of operations selected (**add** or **remove**) and information based on the shadow model's representation for that shadow.

In the current version of the disambigutor, **add** and **remove** operations translate to logical operations on pointcut expressions. Figure 4.1 on page 31 provides an overview of each operation. The diagram shows that when the developer selects an **add** operation, the disambiguator forms a new pointcut from the logical *OR* of the *active pointcut* and the *most specific* pointcut which matches the selected shadow. If a **remove** operation is selected, the disambiguator forms a new pointcut from the *active* pointcut *AND-NOT* the selected shadow's *most specific* pointcut.

The *most specific* pointcut is the pointcut that matches the smallest set of join points that includes the shadow. This may at times include more shadows than the developer has selected. For example, if two call sites with the same signature exist within the same method, there is no way for the developer to specify one and not the other using AspectJ - and therefore the Pointcut Wizard. If new versions of the AspectJ language grow to support such cases, the shadow model of the Pointcut Wizard will be adapted to fit the new semantics.

The properties of a join point that the Pointcut Wizard uses to specify a *most specific* pointcut are based on AspectJ's join point model. The AspectJ join point model can be divided into two areas: dynamic join points and statically determined join points. Statically determined join points are the join points which have a direct representation, or *shadow* in the code. Dynamically determined join points are unknown until the program is running. The prototype version of the disambiguator only supports statically determined join points, although the *Future Work* section of Chapter 5 presents our plans for extending the tool's support to cover dynamically determined join points as well.

A statically determined join point can be expressed in AspectJ using two types of pointcut designators: *kinded* pointcuts and *lexical-structure* pointcuts. As noted in Chapter 3, kinded pointcuts each match a type of action that may occur during the execution of the program, such as a method call or execution. Lexical-structure pointcuts match join points based on where they exist in the source code. To create a *most specific* pointcut to match a shadow, the Pointcut Wizard creates a pointcut by *AND*ing the kinded pointcut and the lexical-structure pointcut for that shadow.

There are other possibilities for disambiguating an operation performed at a join point shadow. For example, in Chapter 2 we presented inductive logic as a means of interpreting operations on join point shadows [4]. The pointcut editing environment presented in this work uses information from the current state of the pointcut as well as from the shadow site to make a best guess of the intentions behind the developers' **add** or **remove** operations.

We believe that there would be benefits to providing the developer with more control in specifying the properties of a join point shadow he is interested in matching with the pointcut. Also, we believe that support for dynamic pointcut designators such as `cflow` and `if` is possible. Each of these ideas are discussed in the *Future work* section of Chapter 5.

4.4 The Normalizer

The *normalizer* is responsible for maintaining the active pointcut in normalized form while the developer edits it through effective representations of the shadows it matches.

By maintaining the pointcut expression in a *normalized* form, the Pointcut Wizard is able to reduce the occurance of pointcut expressions which either conflict with each other or imply one another. Both of these situations make the pointcut unnecessarily difficult to understand and edit.

For example, the following pointcut expression specifies method execution join points with two sets of disjunct properties.

```
(execution(* Bar.foo(..)) && execution(void * *(Zed)))
```

A normalized pointcut with the same semantic meaning would appear as follows:

```
execution(void Bar.foo(Zed))
```

In this section we describe how the normalizer is constructed. First we illustrate how each AspectJ pointcut expression is broken down into a more primitive but equivalent logic expression for testing properties of the join points. Following that, we describe the normal form of the logic expression, which the normalizer uses as an internal representation of the pointcut expression. Last, we explain how this internal representation is converted into an equivalent AspectJ pointcut expression which matches the selected join point shadows.

4.4.1 Rules for Converting an AspectJ pointcut to an Expression on Join Points

In this section we present the set of rules the normalizer uses to convert AspectJ pointcuts to a more primitive expression over join point properties.

The language in which this internal representation of pointcuts is expressed can be thought of as of the AspectJ pointcut language with three additional expressions:

`TRUE` : *This expression trivially matches any joinpoint*

`FALSE` : *This expression doesn't match any joinpoints.*

`<property_name> == <property_Value>`

This expression matches if the join point property denoted by propertyName is of the value propertyValue.

The following is a list of transformations the normalizer uses to convert `kinded` and `lexical` pointcuts supported by the Pointcut Wizard. For each type of pointcut, a breakdown of the property tests performed implicitly by the pointcut is followed by the semantically equivalent expression used by the normalizer.

method call:

```
call( <return_type> <class_name>.<name> (<args>) )
```

converted to:

```
actionType==call &&
target.signatureType==method &&
target.returnType==<return_type> &&
target.className==<class_name> &&
target.methodName==<name> &&
target.args==<args>
```

constructor call:

```
call( <class_name>."new" (<args>) )
```

converted to:

```
actionType==call &&
target.signatureType==constructor &&
target.className==<class_name> &&
target.name==""
target.returnType=="" &&
target.args==<args>
```

method execution:

```
execution( <return_type> <class_name>.<name> (<args>) )
```

converted to:

```
actionType==execution &&
this.signatureType==method &&
this.returnType==<return_type> &&
this.className==<class_name> &&
this.methodName==<name> &&
this.args==<args>
```

constructor execution:

```
execution( <class_name>."new" (<args>) )
```

converted to:

```
actionType==execution &&
this.signatureType==constructor &&
this.className==<class_name> &&
this.name=="" &&
this.returnType=="" &&
this.args==<args>
```

field get:

```
get( <field_type> <container_class>.<field_name>)
```

converted to:

```
actionType==<fieldGet> &&
target.returnType==<field_type> &&
target.className==<container_class> &&
target.name==<field_name>
```

field set:

```
set( <field_type> <container_class>.<field_name> )
```

converted to:

```
actionType==<fieldSet> &&
target.returnType==<field_type> &&
target.className==<container_class> &&
target.name==<field_name>
```

within method code:

```
withincode( <return_type> <class_name>.<name>(<args>) )
```

converted to:

```
actionType==withinCode &&
this.signatureType==method &&
this.returnType==<return_type> &&
this.className==<class_name> &&
this.methodName==<name> &&
this.args==<args>
```

within constructor code:

```
withincode( <class_name>."new"(<args>) )
```

converted to:

```
actionType==withinCode &&
this.signatureType==constructor &&
this.className==<class_name> &&
this.name=="" &&
this.returnType=="" &&
this.args==<args>
```

within:

```
within(<className>)
```

converted to:

```
actionType==within && this.className==<className>
```

In Chapter 3 the Pointcut Wizard's support for matching join points by using wild-cards was discussed. To summarize here: the tool supports the wildcard symbols "`*`" and "`..`" when they are not used in conjunction with other characters to express a join point property. In the normalizer's property expression language, the "`*`" and "`..`" wild-cards are represented by the absence of a property expression of that type.

The empty string is used for the `name` and `returnType` properties of constructor signatures for the `constructor call`, `constructor execution` and `constructor withincode` pointcut types. At the implementation level, this allows the normalizer to treat the arguments for these types of pointcuts similarly to the corresponding *method* pointcuts.

4.4.2 PCDNF: A Pointcut Expression in Disjunctive Normal Form

When all AspectJ primitive pointcut expressions have been broken down into more primitive property tests as described above, it is relatively straight forward to use logic laws such as DeMorgan's law to reduce this expression into a disjunctive normal form. We call this normal form **PCDNF**.

A pointcut expression is in PCDNF if it matches the grammar shown in Figure 4.4.2 and there does not exist any conjunction in the expression that contains two `PropertyEqualTests` where one implies the other or the negation of the other (such conjunctions can be further reduced).

```
<Disjunction> := FALSE
               | <Conjunction>
               | <Conjunction> || <Disjunction>

<Conjunction> := TRUE
               | <PropertyExpression>
               | <PropertyExpression> && <Conjunction>

<PropertyExpression> := <PropertyEqualTest>

<PropertyEqualTest> := [<property_name> "==" <property_value>]
                     | [<property_name> "!=" <property_value>]
```

4.4.3 Converting a PCDNF expression to AspectJ Pointcut Syntax

PCDNF is not valid AspectJ syntax. Before the pointcut expression can be inserted back into the aspect it must be converted to AspectJ syntax. This means re-grouping

```
   target.className==Bullet && target.return== &&
   target.name== && target.sigType==constructor &&
   actionType==call && this.name!=bar
|
   target.className==Bullet && target.return== &&
   target.name== && target.sigType==constructor &&
   actionType==call && this.className!=Foo
|
   target.className==Bullet & target.return== &
   target.name== & target.sigType==constructor &
   actionType==call & this.sigType!=method
|
   this.className==Ship && this.name==fire &&
   this.sigType==method && actionType==execution
```

Figure 4.4: PCDNF representation

primitive property test expressions back into AspectJ pointcut expressions. The process is somewhat complicated because AspectJ's primitive pointcut expressions provide somewhat adhoc combinations of tests on different properties of a join point.

To perform this converstion we have implemented an adhoc solution that searches for property test expressions in the disjunctive normal form that can be grouped together into meaningful AspectJ expressions. These expressions are then used to build up an AspectJ pointcut - and are then removed from the disjunctive normal form. This process repeats until all property test expressions have been consumed. Rather than explain the details of this process we will show an example of the kinds of results that it produces.

The example is based on the following pointcut expression, which was sent as input to the normalizer component.

```
(call(Bullet.new(..)) || execution(Ship.fire(..))) &&
 !withincode(* Foo.bar(..))
```

The resulting PCDNF representation of this expression is shown in Figure 4.4.3.

From this PCDNF representation, the following AspectJ pointcut expression is formed:

```
   call(Bullet.new(..)) && !withincode(* Foo.bar(..))
|| execution(* Ship.fire(..))
```

Note that the `!withincode(* Foo.bar(..))` expression has been omitted from the `execution(* Ship.fire(..)))` conjunct of the resulting PCD. This is because during the conversion to PCDNF it was determined that it is implied by the conditions of `execution(* Ship.fire(..))`.

4.5 The Resolver

The resolver is responsible for identifying the set of shadows in the code that match the pointcut expression. This information is sent to the user interface so that a gutter annotation can be placed at each shadow matched by the active pointcut. In this section we will describe the implementation of this component.

To identify the location of each shadow in the code, the resolver relies on information from the shadow model. When a pointcut expression is received from the normalizer, the resolver creates a semantically equivalent query that can be executed by the shadow model. The results of this query are objects of type `SourceLocation`. As was stated in the *shadow model* section, each `SourceLocation` object maintained in the shadow model is unique - so the results of this query each represent a shadow matched by the pointcut expression.

For each `SourceLocation` object returned, the resolver queries the shadow model for the type of shadow that exists at that location. The results of this query must be one of the types specified in the diagram shown in Figure 4.3 on page 34. The figure also reflects the information pertinant to each shadow type. For example, a `ConstructorCallShadow` contains two sets of signature properties: one for the method or constructor from which it is called, and the signature of the constructor call itself. All of this information is gathered from the shadow model through a series of shadow type specific queries.

Once each shadow has been resolved, the resolver component sends information to the user interface of the Pointcut Wizard. The UI places a gutter annotation at each shadow location, along with the type specific properties of the shadow and the pointcut by which it was matched.

Once this process has finished, the developer is able to perform operations at any of the shadow sites. If a **remove** operation is selected, the disambiguator uses the shadow properties saved at the annotation site to begin the process of disambiguating the meaning of the operation, and the process shown in Figure 4.1 on page 31 begins again.

4.6 Conclusion

In this chapter we presented the design of the Pointcut Wizard. The tools architechure consists of the following four components:

- **Shadow Model:** The role of this component is to model all shadows in the code. The prototype implementation of the shadow model is as a database built on top of JQuery. [10]
- **Disambiguator:** The disambiguator is responsible for deciding the meaning of an operation performed at a join point shadow and then mapping the information found at the join point site to a pointcut designator that will match it.
- **Normalizer:** The normalizer is responsible for reducing the occurance of certain redundancies that may appear in the pointcut expression following successive **add** and **remove** operations.

- **Resolver:** The resolver is responsible for identifying the set of shadows in the code which match the pointcut expression. This information is sent to the user-interface so that each shadow can be identified with a gutter annotation.

The thesis of this dissertation is that it is possible to construct IDE-based tools that allow editing of AspectJ pointcut expressions through direct manipulation of effective representations of joinpoints in the code to which a pointcut applies. In this chapter we have presented the architecture of the Pointcut Wizard - a constructive proof of the thesis.

Chapter 5

Conclusion

The thesis of this dissertation is that it is possible to construct IDE-based tools that allow editing of AspectJ pointcut expressions through direct manipulation of effective representations of join points in the code to which a pointcut applies.

We presented *Pointcut by Example* as a method for accomplishing this goal. To create a pointcut by example the developer can specify join point shadows in the code as examples of the kinds of join points he wishes for a pointcut expression to match or not to match.

As a constructive proof that a pointcut can be created in this manner we presented the Pointcut Wizard. The Pointcut Wizard is tool support for editing a pointcut through effective annotations at locations where join point shadows exist. By selecting an **add** or **remove** operation at a join point shadow site, the developer can edit the pointcut expression so that it either captures or no longer captures the join point.

In this chapter we will reiterate the contributions made in this dissertation. Following that, we will present our plans for future work in this area. The dissertation will conclude with a brief discussion about where a tool for editing pointcuts by example fits into the world of Aspect Oriented Software Development.

5.1 Contributions

The central contribution made by this dissertation is the Pointcut Wizard: a constructive proof of pointcut by example for AspectJ. Our description of the tool was divided into two areas: the user interface (Chapter 3) and the underlying implementation (Chapter 4). We believe that each presents a layer of a novel architecture for modeling shadows and editing pointcuts. In this section we will summarize the important aspects of each area.

5.1.1 A UI for Effective Views of Join Point Shadows

The Pointcut Wizard provides effective representations for join point shadows in the code. To inform the developer of which join point shadows in the code are matched by a pointcut expression, we decided to use gutter annotations next to each line that containing a matched shadow. This technique is similar to that used by AJDT to mark locations where an advice will apply. By hovering over an annotation, the developer can access an Eclipse tool-tip containing information about the shadow itself and the pointcut by which it is matched. The tool provides an *effective* view of join point shadows because the developer is able to access context menus at any join point location and perform an **add** or **remove** operation.

If the shadow is already matched by the pointcut expression, the developer is provided with the option of removing it. Selecting the **remove shadow** option edits the pointcut expression so that it no longer captures the selected shadow. The tool provides immediate feedback as to the result of the operation.

The Pointcut Wizard also allows the developer to add shadows in the code that are not yet matched by the pointcut. If the developer wishes to capture a potential join point shadow, he can access the **add shadow** menu selection from the line the shadow exists on and the pointcut expression will be edited to capture that shadow.

Because the tool is implemented on top of a query engine, finding the results of a change operation on a pointcut is as simple as re-running a query. Therefore, all changes to the pointcut have an immediate impact on the annotations shown in the code editors.

5.1.2 An Architecture to Support Effective Views of Join Point Shadows

The Pointcut Wizard was implemented as four components: the *shadow model*, the *disambiguator*, the *normalizer* and the *resolver*. In this section we will summarize the role of each.

- The shadow model was built to model all shadows in the code. This was a necessary means for each of the other three components to share information about a shadow.
- The *disambiguator* is responsible for deciding the *meaning* of an operation performed at a join point shadow, and then mapping relevant information from the shadow site to a pointcut designator.
- The *normalizer* was built to maintain the pointcut expression in normalized form while the developer performs operations on it. This was necessary because of the additive nature of the operations performed on the pointcut following an **add** or **remove** operation.
- The *resolver* is responsible for identifying the set of shadows in the code which together represent the pointcut expression. This information is sent to the user-interface so that each shadow can be annotated.

5.2 Future Work

In this section we will present what we see as future work for this tool. We divide the work into two areas: improvements to current features and validation of the tool's usefulness and usability. In this section we will present our ideas for each area.

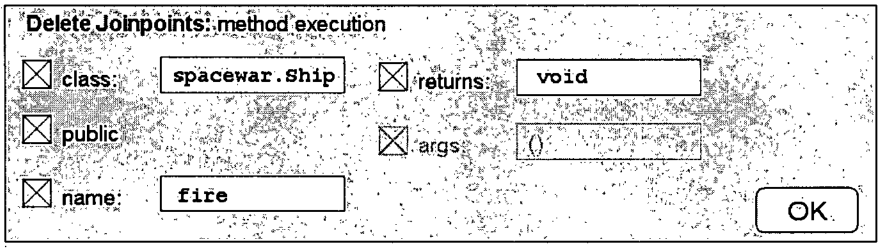


Figure 5.1: A UI for Editing Pointcut Properties

5.2.1 Usability and Usefulness

This dissertation has shown that it is possible to construct IDE-based tools that allow editing of pointcut expressions through direct manipulation of effective representations of join points in the code to which a pointcut applies. However, for future work we would like to validate the usefulness and usability of our technique to actual AspectJ developers.

We believe that the prototype of the Pointcut Wizard provides enough functionality to be able to determine its value to developers. Therefore, these experiments will occur prior to any of the following feature updates.

5.2.2 Improvements to Current Features

In this section we will present our plans for improving features that the Pointcut Wizard currently supports. Our first challenge will be to support the full AspectJ language. This will include full support for wild-card symbols, binding pointcuts, and cflow and conditional pointcuts. Additionally, we wish to provide support for editing join point shadows that occur within aspects. Each of these new language features brings with it new challenges for *effective* editing from the join point shadow location. In the following subsections we will describe our ideas for how to adapt the Pointcut Wizard to meet these challenges.

Support for Wildcards

We believe that it may be useful for a developer to specify a pointcut designator that matches only certain properties of a shadow he has selected. Through a menu selection, we plan to give the developer access to the dialog box with check-box options for specifying which properties of a signature by which to match join points. A mock-up of the dialog box is shown in Figure 5.1.
Upon opening this dialog box, each box would initially contain values specific to the shadow of interest. The developer would have the options of either un-checking a box to use a wild-card symbol in place of the value, or filling in a value. If he wishes to fill in the value by hand, we plan to support using regular expressions to match join points.

To implement this feature, we plan to update the *resolver* component of the Pointcut Wizard to include full support for wild-card symbols. Using the new interface, the developer should be able to create a pcd such as `call(* *.set*(..))` that matches all methods where the method name starts with *set*.

The developer would also have access to the "`..`" operator, allowing him to match argument lists by a certain type contained at the beginning, end, or both of an operator list. For instance, the following pointcut designator could be used to capture all calls to a method `set` where the first argument is `int` and the last is `bool` - and any number of arguments appear between them:
`call(* *.set(int, .., bool))`.

If the developer un-checks a box in the new interface, the value for that property will be either a star pattern or a dot-dot pattern, depending on the property.

Support for Binding PCDs

In the next version of the Pointcut Wizard we also plan to increase support for binding pointcuts. As explained in Chapter 3, the Pointcut Wizard does not fully support the binding pointcuts `this`, `target` or `args`. Using the prototype, a developer can specify a type name as an argument to these pcds and match any join point where an object takes on that role.

However, the binding pcds are most useful when the developer needs to access an object's state at run-time. We therefore plan to add a feature to the Pointcut Wizard that will allow the developer to specify variables or fields located at a join point shadow that should be bound in the active pointcut expression. Designing this feature should not be difficult, because the role the developer may wish to bind an object to (i.e. whether to use `this`, `target` or `args`) can be determined by the role it takes at the join point shadow.

As an example, suppose the developer selects an **add** operation on a method call `foo.fire();`. We plan to add UI support that would allow the developer to select a **bind** operation on the variable (or field) `foo`. Because `foo` is the *target* at this location, the Pointcut Wizard could automatically use that type of binding operation. The type of the variable `foo` could then be determined by JQuery, and added to the pointcut's signature as a parameter.

Support for Cflow and If PCDs

Supporting `cflow` and `if` pointcut designators will require a change to the *shadow model* and the *disambiguator* of the Pointcut Wizard.

Currently, the disambiguator attempts to form a one-to-one correspondence between the shadow of interest and the pcd that matches it. Therefore, the shadow model was designed to capture the shadow's signature and its lexical context. However, if the Pointcut Wizard were updated to support dynamic pcds such as `cflow` and `if` pcds, the model could no longer identify shadows based only upon signature and lexical context. The next version of the Pointcut Wizard will create a new shadow model that contains a shadow's signature, its lexical context, and information pertaining to any run-time checks associated with it.

The disambiguator would also need to be updated to handle the dynamic quality of such shadows. AJDT [5] addresses this issue with tool-tips that alert the developer to the possibility that the shadow may or may not be matched by the pointcut designator at run time. Our case is rather more complex however because the annotations must also be *effective*. To illustrate this point, take for example the following pointcut expression:

```
pointcut foo(): call(* *.foo()) && cflow(call((* *.bar)))
```

Suppose the developer is at a call site for a method `foo`. Along with the **add** and **remove** operations that the tool currently supports, the developer may also wish to have the option of adding, removing or changing the `cflow` pcd. The problem is that the context on which the cflow is based (i.e. the pointcut taken as an argument by the cflow pcd), may exist outside the context of the `foo` call site. Since a central design goal of the Pointcut Wizard was to allow the developer to edit a pointcut expression without facing the disorientation involved with leaving the context where the shadow exists, we believe this new feature could require fundamental change to the design of the Pointcut Wizard. Currently, this issue remains unresolved.

Editing Join Point Shadows in Aspects

According to AspectJ semantics, join point shadows can occur in aspects as well as classes. The prototype of the Pointcut Wizard supports the latter. In order to support the editing of join point shadows in *aspects* with effective views, the tool would need to support the *kinded* pcd type `adviceexecution`, shown in Table 3.1 on page 16.

In order to support this feature we would need to update the *shadow model* to account for a new shadow type that exists within an advice. Additionally, the tool only supports the parsing of java code in classes, and pointcut expressions in aspects. To fully support this new feature, the tool would also need to parse both *introductions* and *advice*.

5.3 Concluding Remarks

This dissertation applied the notion of creating *Pointcuts by Example* to AspectJ's join point model. We argued that AspectJ pointcut expressions are particularly difficult to edit because the pointcut language is based on patterns in the code, rather than the properties the patterns are meant to represent. Editing pointcuts by example allows the developer to gather more information about the property of a join point based on the context in which it exists. The goal of this dissertation was to prove that tool support which was capable of allowing the developer to edit pointcuts by example was possible. We have provided proof in the form of the Pointcut Wizard that such support is indeed possible.

Bibliography

[1] Aspectj documentation.

[2] AspectJ home page. http://eclipse.org/aspectj/.

[3] AspectWerkz home page. http://aspectwerkz.codehaus.org/.

[4] Tom Tourwe Centrum. Inductively generated pointcuts to support refactoring to aspects.

[5] Andy Clement, Adrian Colyer, and Mik Kersten. Aspect-oriented programming with ajdt. In *Analysis of Aspect-Oriented Software (AAOS)*, 2003.

[6] Bart De Win, Wouter Joosen, and Frank Piessens. Aosd & security: a practical assessment. In *Workshop on Software engineering Properties of Languages for Aspect Technologies (SPLAT, AOSD)*, Boston, Masachussets, March 2003.

[7] Stephen G. Eick, Joseph L. Steffen, and Jr. Eric E. Sumner. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng.*, 18(11):957–968, 1992.

[8] Enterprise Java Beans 3.0 spec. http://www.jcp.org/en/jsr/detail?id=220.

[9] Geoff Hulten, Karl Lieberherr, Josh Marshall, Doug Orleans, and Binoy Samuel. *DemeterJ User Manual*.
`http://www.ccs.neu.edu/research/demeter/`.

[10] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. pages 178–187.

[11] Doug Janzen and Kris De Volder. Programming with crosscutting effective views. In *ECOOP*, pages 195–218, Washington, DC, USA, 2004.

[12] Bugdel home page. http://jboss.com.

[13] 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.

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

[15] Karl Lieberherr, David H. Lorenz, and Johan Ovlinger. Aspectual collaborations for collaboration-oriented concerns. Technical Report NU-CCS-01-08, College of Computer Science, Northeastern University, Boston, MA 02115, November 2001.

[16] S. Muggleton and C. Feng. Efficient induction of logic programs, 1990.

[17] Klaus Ostermann and Mira Mezini. Conquering aspects with Caesar. pages 90–99.

[18] M Strzer and C Koppen. Pcdiff: Attacking the fragile pointcut problem. In *Interactive Workshop on Aspects in Software (EIWAS)*, September 2004.

[19] Klaas van den Berg and Jose Maria Conejero. A conceptual formalization of crosscutting in aosd. In *DSOA'2005 Iberian Workshop on Aspect Oriented Software Development*, Technical Report TR-24/05, pages 46–52. University of Extremadura, 2005. ISBN 84-7723-670-4.

[20] Tom Tourw Vrije. On the existence of the aosd-evolution paradox.

[21] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 112, Washington, DC, USA, 1998. IEEE Computer Society.