PointcutDoctor: IDE Support for Understanding and Diagnosing Pointcut Expressions

[AOSD 2007 demonstration proposal]

Lingdong Ye, Kris de Volder
Department of Computer Science
University of British Columbia
201-2336 Main Mall
Vancouver, B.C. V6T 1Z4
604-822-1290
{lintonye, kdvolder}@cs.ubc.ca

ABSTRACT

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

In this demonstration we present PointcutDoctor, an extension of AJDT tools which helps developers write correct pointcuts by providing immediate diagnostic feedback.

PointcutDoctor provides several kinds of information for a given pointcut. Firstly it shows which join points (shadows) the pointcut matches or doesn't match. This helps a developer to verify whether her pointcut is correct. Furthermore, PointcutDoctor also provides an explanation of *why* the pointcut matches or does not match a given join point(shadow). This information helps a developer diagnose the cause of problems—unintended matches or failures to match certain join points—in her pointcut.

1. PROBLEM STATEMENT, MOTIVATION

Pointcuts in general and AspectJ pointcuts in particular are hard to write. To write a correct pointcut, a developer needs to understand the subtleties of the pointcut language as well as how the pointcut matches against various program elements across her entire code base.

We present PointcutDoctor, an IDE based tool that helps developers write correct pointcuts by providing them with easy access to the right kind of information. This information helps developers by making it easier for them to:

1. ascertain whether a given pointcut is correct (or not)
2. diagnose and correct problems if and when they are discovered

Ascertaining that a pointcut is correct in a given code base means understanding how this pointcut matches join points across the code base [4]. This requires a global understanding of the code at a level of detail that is not easy to obtain or remember for developers. Fortunately, developers can be assisted by IDE-based tools that provide an explicit representation of the join points a pointcut matches in a given code base. AJDT [1] already provides some useful information in this regard. However, we claim that AJDT has a kind of "blind spot": the information provided by AJDT is insufficient for a developer to determine a pointcut's correctness. We illustrate this problem with a concrete example[1]:

```
pointcut threadCreation(Runnable worker)
: call(Thread.new(Runnable)) && args(worker);
```

Assume that a developer formulated this pointcut to capture the creation of *all* `Thread` instances from an instance of `Runnable`. How would the developer know this pointcut is correct? Since the intention is to capture *all* creations, verifying correctness means ascertaining that there are no accidentally missed creation sites. We argue that AJDT does not provide the right kind of information for the developer to make this determination. Indeed, AJDT's cross references view only shows which join point shadows produce join points matched by the pointcut, but it does not show any explicit information about join points that are *not* being matched. This is an important "blind spot" and as a result the view is rather unhelpful in discovering unintended misses. Indeed, it is hard for a developer to scan a list of matches and realize that something that should be there isn't. However, as our example illustrates, this kind of determination, is often critical in understanding whether a pointcut is correct. There are actually 5 `Thread` constructors that have a `Runnable` parameter:

[1]This example was taken from [5]

```
Thread(Runnable)
Thread(Runnable, String)
Thread(ThreadGroup, Runnable)
Thread(ThreadGroup, Runnable, String)
Thread(ThreadGroup, Runnable, String, long)
```

Only calls to the first of these 5 constructors will be matched by the pointcut. The pointcut is therefore incorrect, failing to match calls to the other 4. AJDT's cross references view provides no explicit information about non-matched join points and so does not help to discover this important fact. As a result this bug in the pointcut is likely to go unnoticed.

Besides the complexity of dealing with pointcut (mis)matches across an entire code base, another problem with regard to pointcut writing is the complexity and subtleties of the pointcut-language semantics itself. Once it is clear that a pointcut is incorrect, i.e. it unintentionally matches or misses certain join points, it is often unclear to developers *why* a particular join point is (not) being matched. The AspectJ pointcut language has some subtleties that appear confusing especially to novices. For example, `call` pointcut matches calls to methods overridden in subclasses, but does not match any constructors of subclasses, even if their parameter lists are the same as that of their super class (appearing as if they are "overridden constructors"). That is, the example pointcut above will not match calls to `MyThread(Runnable)`. There are similar issues for `execution` pointcuts. This is a typical confusion that novices are confronted with and ask questions about on the aspectj-user mailing list[2]. There are other caveats such as `call` pointcut only matches according to the static type of a call join point's target, constructor and static method calls don't have targets, `handler` doesn't match subclasses, `args` often restricts the parameter pattern in `call` or `execution` pointcut etc. Failing to understand these subtleties, a developer may find it challenging to diagnose problems in her pointcuts. A tool that offers an explanation of why a certain join point is matched or not matched would be of great assistance. To our knowledge, AJDT only provides explanations for a couple of cases so far by means of xlint warning (e.g. Xlint:unmatchedSuperTypeInCall) when there are suspicious misses.

2. TOOL DESCRIPTION

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

The main point where PointcutDoctor functionality has been added is in the cross references view. First, the original cross references view only displays information about matched join point shadows. PointcutDoctor extends this with additional information about "almost" matched join point shadows. This fixes AJDT's "blind spot" discussed earlier. The second extension is a "pointcut explainer" feature which provides diagnostic information that may be helpful to developers when they see incorrectly matched or missed join point shadows in the cross references view. We next explain each of these extensions in more detail.

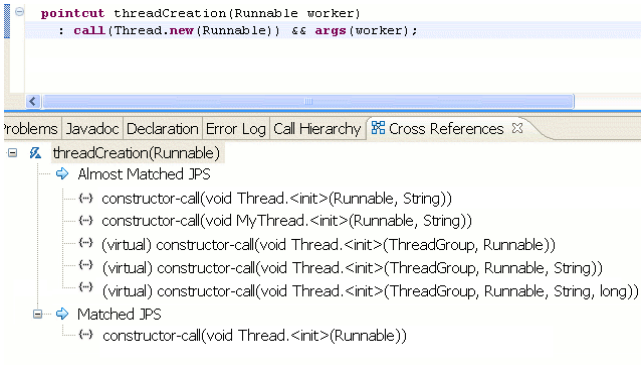

Figure 1: Almost Matched Virtual Shadows

2.1 Almost Matched Join Points

As explained earlier, to ascertain that a pointcut is correct it is often just as important to obtain information about the join points that it didn't match as those that it did match. However, it is impractical in most cases to display all non-matched join points because there are too many of them. Our answer to this dilemma is to display information only about shadows that are "almost" matched. These "almost matched" join point shadows are computed by a technique we call *pointcut relaxation*. The technique is described in more detail in Section 3.

Figure 1 shows the augmented cross references view and what information it displays for the pointcut in our motivating example. Notice that this view shows the missed calls to some of the `Thread` constructors as "almost matched" join point shadows. Notice also that some of the almost matched shadows are marked as "virtual". Our algorithm produces virtual shadows for calls to methods that have been declared but for which no actual calls exist within the code base. The rationale behind this is that even though such calls do not exist in the current code base it is likely they will be added in future versions and a developer should take such potential join points into account when trying to write "robust" pointcuts.

2.2 Pointcut Explainer

The purpose of the pointcut explainer is to help developers diagnose problems with pointcuts when they discover wrongly matched or missed join point shadows in the cross references view. When the developer clicks on a shadow in the cross references view the pointcut explainer provides an "explanation" by highlighting specific parts of the pointcut right in the editor. The highlighted parts are those that are responsible for matching or not matching this join point shadow. The highlighting uses a color-coding scheme based on the matching status of the pointcut fragment. Three highlighting colors are chosen for "Matching" (colored green), "Mismatching" (colored red) and "Maybe"[2](colored

[2]"Maybe" indicates that matching requires runtime determination.

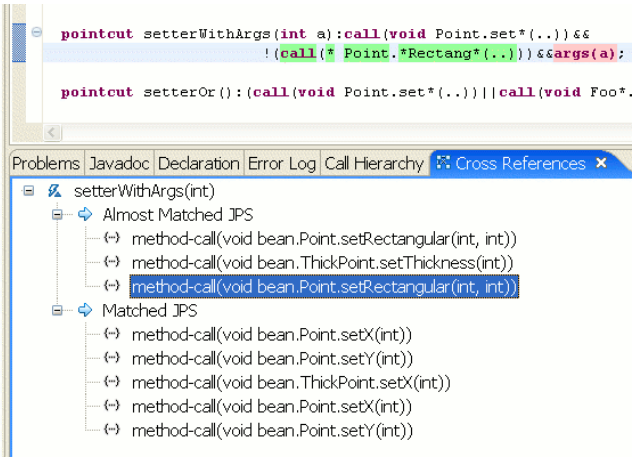

Figure 2: Highlighting Explanation

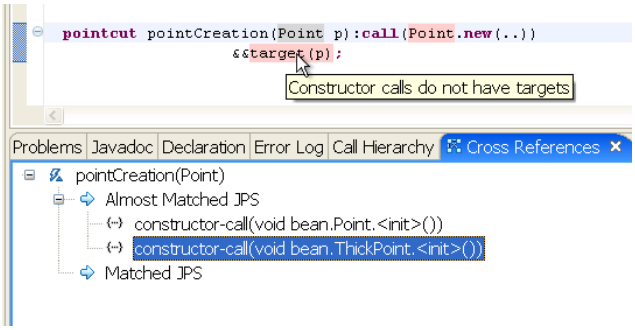

Figure 3: In-place Explanation

yellow). A screenshot is shown in Figure 2.

If the colored highlighting alone is not a sufficient explanation, the developer may elicit more information by hovering over the highlighted part of the pointcut as shown in Figure 3 to request an explanation why this particular pointcut fragment is colored the way it is. By hovering over the shadow itself reveals a broader explanation about the entire pointcut (shown in Figure 4). The textual explanation elaborates the causes and tries to educate the user on the subtleties of AspectJ. The explanations are tailored to the specific context of the user's code.

3. UNDERLYING IMPLEMENTATION TECHNOLOGIES

The current implementation of PointcutDoctor is based on various modifications to the weaver component of AspectJ compiler 1.5.2, and extensions to the cross reference view of AJDT.

Pointcut relaxation: by a process called relaxation, the original pointcut expression is turned into a series of increasingly more relaxed pointcut expressions which potentially match more join points than the original one. The relaxed pointcuts are produced by a number of heuristic rules to ensure a reasonable selection and ordering of different kinds of relaxation methods being applied. Without these heuristics, some very broad pointcuts will basically match all join

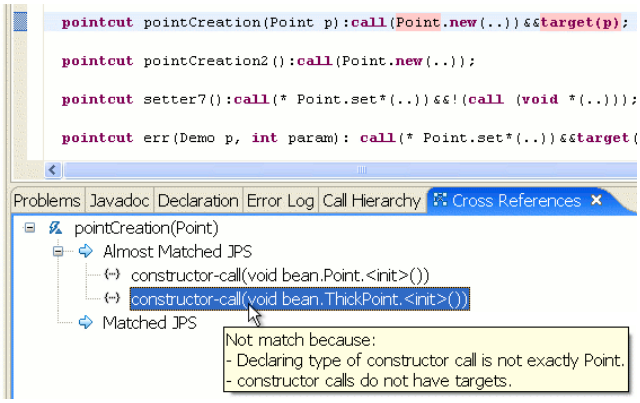

Figure 4: Full Textual Explanation

points in the code base, which makes the tool less useful, e.g. if the original pointcut is `call(void *.*(..))`, and if we simply relax it by replacing the return type with *, it will match all call join points. The original matching process of the weaver is augmented so that the relaxed pointcuts are matched in parallel to the original pointcut. The correspondence between pointcuts and shadows are then kept for later use in the explanation algorithm.

Virtual shadows: In some cases, even when a join point does not exist in the current code base, it is very likely these join points will occur in the future, e.g. if a method is declared but not used (this is often the case when using code libraries). PointcutDoctor is able to detect this case and identify these "virtual join points" as matched/almost matched join points. It is implemented by introducing a group of virtual shadow classes which extend the standard `Shadow` abstract class and will be instantiated right after the related code element is gone through, say, a method declaration. Then these virtual shadows will act almost the same as other shadows in the matching process, except that they are not matched against the standard shadow mungers, such as advices, declares etc. As a future work, this technique can be further explored and used for improving the robustness of a pointcut.

Recursive explanation: To explain the reason why a pointcut doesn't match (or does match) a join point, a recursive algorithm is devised. The algorithm is based on an and follows a similar structure to the AspectJ pointcut matcher, but instead of producing a result of "match/no-match" for a given shadow it produces an explanation of the reason for the match. Two kinds of explanations can be computed. The first is a highlighting (color-mapping) of the pointcut expression AST and the second is a textual elaboration. The "reason" for a match/no-match is defined as the set of minimal sufficient conditions that makes a pointcut expression True, False or Maybe. For example, the reason `call(* Foo.bar(..))&&!call(void *.bar(int))` matches `method-call(void Foo.bar(boolean))` is that `call(* Foo.bar(..))` matches and the parameter list of `call(void *.bar(int))` doesn't match. The challenge here is the definition of reason and the fact that the pointcut expression could consist of `&&`, `||` and `!`. To explain why an expression matches needs to explain why its sub-expressions

match, or even why they don't match. The textual explanation is computed and combined in the same manner but is more heuristics based.

4. WHAT THE AUDIENCE WILL SEE

The demonstration will start with a short presentation introducing the motivation and ideas, followed by a live demo of the tool being used in typical scenarios for AspectJ development. The demo consists of two parts and the features of PointcutDoctor will be explored incrementally.

The first part will show several short scenarios of a user making mistakes when writing pointcuts. We will show how these kind of mistakes can be easily identified using the augmented cross references view and demonstrate the usefulness of the diagnostic information presented by the explainer to understand the cause of the problem and make corrections. The audience will also get a sense that PointcutDoctor requires minimal learning effort assuming that the user is already familiar with AJDT.

The second part of the demo will show a more comprehensive development task: implementing a specific crosscutting concern. The display of matched and almost matched join point shadows, including normal shadows and virtual shadows, and difference kinds of explanations will be demonstrated, illustrating the usefulness of PointcutDoctor in a typical software development context.

5. RELATED WORK AND UNIQUENESS OF THE TOOL

In the current version of AJDT, only matched join points are shown for advices. Some xlint warnings(e.g. Xlint:unmatchedSuperTypeInCall) are provided for explaining several cases of suspicious misses. A series of explanation related tasks are planned in AJDT project (e.g. Pointcut Reader).

PCDiff [3][6] tracks and presents the difference a pointcut matches before and after a change made to the code base, while our work focuses on helping developers write correct and robust pointcuts rather than on tracking the impact of changes to the code over time. Tools like PCDiff are complementary to PointcutDoctor and it would be interesting to try to combine their functionalities in a single integrated tool.

As far as we know, there is no existing work with regard to presenting join points that are not matched by the pointcut, and explaining why a *specific* join point shadow is (not) matched by a pointcut.

6. HARDWARE AND PRESENTATION REQUIREMENTS

This demonstration does not require any special hardware other than a projector and a screen. The presenter will come with a laptop prepared for the talk.

7. REFERENCES

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

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

[3] M. S. C. Koppen. Pcdiff: Attacking the fragile pointcut problem. *European Interactive Workshop on Aspects in Software*, 2004.

[4] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, 2005.

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

[6] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 653–656, Washington, DC, USA, 2005. IEEE Computer Society.