Inquiring the Usage of Aspect-Oriented Programming: An Empirical Study*

Freddy Munoz[1], Benoit Baudry[1], Romain Delamare[1], Yves Le Traon[2]
[1]*INRIA, Centre Rennes - Bretagne Atlantique*
Campus de Beaulieu, F-35042 Rennes Cedex, France
{freddy.munoz,romain.delamare,benoit.baudry}@inria.fr
[2]*IT-Telecom Bretagne*
2, rue de la Châtaigneraie, 35576 Cesson Sévigné Cedex, France
Yves.letraon@telecom-bretagne.eu

Abstract

Back in 2001, the MIT announced aspect-oriented programming as a key technology in the next 10 years. Nowadays, 8 years later, AOP is not widely adopted. Several reasons can explain this distrust in front of AOP, and one of them is the lack of robust tools for analysis, testing and maintenance. In order to develop dedicated solutions for assisting the development with AOP, and increase its adoption, we need to understand how it is actually used. In this paper we analyze 38 aspect-oriented open source projects with respect to the impact of aspects on the projects, and to coverage of the language features. This reveals that AOP is currently used in a cautious way. This work is a first step to built support and development tools dedicated to actual practices for AOP, based on empirical usage profiles.

1. Introduction

Object-orientation (OO) pushes forward ideas such as *modularity*, *abstraction*, and *encapsulation* [1]. It promotes the separation of concerns as a cornerstone to improve the maintainability, evolution, and comprehension of a software system. Since a modular unit encapsulates the behavior of a single concern, its maintenance and evolution should require modifying a single module. This results in a major improvement in comparison to non-modular design, which requires modifying several pieces of code several times. Thus, maintaining a system conceived with object-orientation requires less effort than maintaining non-object oriented systems.

However, separation of concerns and modularity cannot always be achieved with OO. Some concerns cannot be neatly separated in objects, and hence, they are scattered across several modules in the software system. Such concerns are referred as *crosscutting concerns* because they are realized by fragments of code that bear identical behavior across several modules. Maintaining a crosscutting concern means modifying each fragment of the scattered code realizing that concern. Therefore, increasing the coding time, error proneness[1], and the maintenance cost.

Aspect oriented programming (AOP) appeared in 1997 as a mean to cope with this problem [3]. The idea underlying AOP is to encapsulate the crosscutting behavior into modular units called *aspects*. These units are composed of *advices* that realize the crosscutting behavior, and *point-cut descriptors*, which designate the points in the program where the advices are inserted.

In 2001 the MIT announced AOP as a key technology for the future 10 years [4]. The expressive features provided by aspect-oriented languages were meant to enable developers to encapsulate tangled code in a very versatile way; therefore improving maintainability of the system by allowing developers to modify single units instead of scattered code fragments. This should have led developers to rapidly adopt AOP. However, those features introduced difficulties for maintenance, validation, and evolution as shown by several studies [5, 6, 7, 8]. As a consequence, 8 years after the MIT announcement AOP is still not widely spread. For instance, in the *source-forge* open source repository, less than 0.5% of the projects developed using Java in the period from 2001 to 2008 integrate aspects.

Previous work has identified two characteristics of aspect-oriented languages that hinder maintainability and evolution: (1) the fragility of the point-cut

[1] A recent study [2], demonstrates that crosscutting concerns increase the proneness to errors in OO system.

* This work was partially supported by the European project DiVA (EU FP7 STREP).

descriptors [6]; (2) the ability of aspects to break the object-oriented encapsulation [5]. In order to develop dedicated solutions for assisting the development with AOP, we need to understand how developers use aspect-oriented features, and how they deal with these characteristics.

In this paper we present an empirical study that analyzes 38 open source aspect-oriented projects developed with the Java and AspectJ languages. This study provides a better understanding of how developers use the AOP features in open source projects. In particular, we analyze the degree to which aspects break the OO encapsulation, and how much of the expressive power for point-cut descriptors is actually used. This reveals that aspects are used in a cautious way.

We observe three major trends: (1) advices affect a small portion of points in the project, and this proportion decreases with the project size; (2) few advices break the encapsulation, and those who break it are used with very precise point-cut descriptors; (3) point-cut descriptors are defined with only half of the available expressions.

This paper is structured as follows: Section 2 introduces the aspect-oriented programming concepts. Section 3 describes the important aspects of our experiments, and present the precise research question this study inquiries. Section 4 presents the analysis results for each research question. Section 5 presents related work. Section 6 concludes the paper by summarizing the main results and discussing their implications for maintenance, and AOP adoption.

2. Aspect-Oriented Programming

In aspect-oriented programming (AOP), aspects are defined in terms of two units: *advices*, and *point-cut descriptors (PCD). Advices* are units that realize the crosscutting behavior, and point-cuts are pointing elements that designate well-defined points in the program execution or structure (*join-points*) where the crosscutting behavior is executed. We illustrate these elements through two code fragments belonging to a banking aspect-oriented application. The first (listing 1) presents the PCD declaration for *logging* (lines 2-5) and *transaction* (lines 7-10) concern, whereas the second (listing 2) presents an advice (lines 3-14) realizing a *transaction* concern.

In AspectJ, a PCD is defined as a combination of *names* and *terms*.

Names are used to match specific places in the base program and typically correspond to a method's qualified signature. For instance, the name boolean Account.withdraw(int) in listing 1 (line 3) matches a method named withdraw that returns a type boolean, receives a single argument of type int, and is declared in the class Account.

```
1: public aspect BankAspect {
2:  pointcut logTrans(int amount):
3:        ( call(boolean Account.withdraw(int)) ||
4:          call(boolean Account.deposit(int))
5:        ) && args(amount);
6:
7:  pointcut transaction():
8:        execution(boolean Account.*(int))
9:        && cflow(execution(void Bank.operation(..))
10: }
```

Listing 1.

Terms are used to complete names and define in which conditions the places matched by names should be intercepted. AspectJ defines three types of terms: *wildcards*, *logic operators*, and *keywords.* The combination of *names* and *terms* is referred as *expression*.

W*ildcards* serve to enlarge the number of matches produced by a *name*. The AspectJ PCD language defines two wildcards: "*" and "..".

Logic operators serve to compose two expressions into a single expression, or to change the logic value of an expression. The AspectJ PCD language provides three logic operators, "&&" (*conjunction*), "||" (*disjunction*), and "!" (*negation*).

Keywords define when and in which conditions the places matched by names should be intercepted. The AspectJ PCD language defines 17 keywords for that purpose. For instances, the keyword call in *logTrans* (lines 3, 4) indicates the interception of all the calls to the enclosed names, whereas the keyword args (line 5) indicates that the PCD argument amount should be the argument of those invocations.

Some keywords point to joint-points that can be computed only at runtime. The AspectJ PCD language defines 6 keywords for that purpose: *cflow, cflowbelow, if, arg, this,* and *target*. The transaction PCD (lines 7-10) incorporates this kind of keywords. It contains two expressions: (1) a static expression that intercepts the execution of any method returning a boolean in the class Account (line 8); (2) a dynamic expression that constrains the interception of the static expression to the execution occurring inside the control flow of the execution of the method operation in the class Bank. This is a dynamic expression since determining whether the execution of a method occurs during the execution of another can be done only at runtime. We refer to join-points occurring only at runtime as *dynamic join-points* and PCDs pointing these points as *dynamic-PCDs*.

AspectJ extends the Java syntax to allow developers to implement advices as natural as possible. Advices can be seen as routines that are executed at some point.

Typically AspectJ advices are bounded to a PCD designating the points where they will be executed. For instance, the advice in listing 2 (lines 3-14) is bounded to the PCD *transaction* (line 3). AspectJ provides three different kinds of advices *before*, *after*, and *around* indicating the moment when they are executed.

```
1:  public aspect BankAspect{
2:    pointcut transaction...
3:    boolean around(): transaction(){
4:         Account account=..
6:         if(account.balance>0 && account.credit>0){
7:             commit(account);
8:             return proceed();
9:         }
10:        else{
11:            rollback(account);
12:            return false;
13:        }
14:    }
15: }
```

Listing 2.

Advices such as the one presented in listing 2 are called *invasive advices* and the aspects containing the advices *invasive aspects*. These names refer to their ability to break the object oriented-encapsulation and disturb the control flow, or modify the data structures of a modular unit. Typically, invasive aspects and advices are characterized by an invasive pattern, which describes the interaction of the aspect/advice with the base program in which it is woven. In previous work [5] we identified 8 invasiveness patterns for advices, and 3 for aspects. Since advices are realization of crosscutting behavior, and hence promoters of the modularization enhancement proposed by AOP, in this work we focus on the advice invasiveness patterns. The 8 invasiveness patterns for advices are as follow. *(1) Write:* the advice assigns a value to an object attribute. *(2) Read:* the advice accesses the value of an object attribute (advice in listing 2, field access in line 6 account.balance). *(3) Argument passing*: the advice captures and modifies the argument passed to the advised method. *(4) Augmentation:* the advice augments the behavior of the advised method always executing it. *(5) Replacement:* the advice replaces the behavior of the advised method. *(6) Conditional replacement:* the advice replaces the behavior of the advised method under certain conditions (e.g advise in listing 2). *(7) Multiple:* the advice executes the advised method several times. *(8) Crossing:* the advice invokes one or more methods that it does not advise.

Our analysis of invasive aspects disregards the invasiveness patterns *augmentation*, and *crossing* in order to focus only on advices that can disturb the regular proceed of a method. This leaves only 6 patterns for invasive advices.

3. Experimental Set-Up

In this section we present the experimental data, and settings we used to empirically inquiry the usage of AspectJ.

3.1. Experimental data

The experimental data for this study consists of 38[2] aspect-oriented projects available under open source licenses. We have collected these projects from public repositories in July 2008. We selected these projects according to the following criteria: (1) Project implemented in Java / AspectJ, (2) project source code publicly available, (3) project compiles using the AspectJ compiler version 1.5, (4) project size of at least 10 classes and 1 aspect, and (5) the advices in the project advise at least 1 join-point. We started our search at sourceforge.net, at that date the most popular open source repository in Internet. Out of 74 aspect-oriented projects, only 28 fulfilled our criteria. Then, we continued gathering projects by inspecting other repositories by using the Google™ code search engine. It is worth mentioning that we queried files with the AspectJ file extension (.aj). This leaves out of our search aspects defined inside java class files (.java). Out of 2000 files, equivalent to 28 projects, only 10 fulfilled our selection criteria. Finally, we successfully gathered 38 open source aspect-oriented projects.

Out of 38 open source projects ranging from small to large size in lines of code (1116 - 80818 LOC), 36% of them have between 1000 and 5000 LOC (*small size*), 36% between 5000 and 20000 LOC (*mid size*), and 28% more than 20000 LOC (*large size*). Together, the 38 projects have 53083 methods scattered in 7343 classes, and $\sim 65 \times 10^4$ statically calculable join-points.

Regarding the number of crosscutting units, the 38 projects have a total of 479 aspects, and 522 advices advising a total of 21245 join-points. Out of them 62% of the projects comprise between 1 and 10 advices, 25% between 10 and 30, and 13% more than 30 advices. Among the 38 projects, 57% of them comprise at least one advice realizing an invasive pattern, which corresponds to 30% of all the advices.

3.2. Analysis tools

We have used a variety of tools to analyze each project and collect the data needed to answer our inquiry:

Metrics plug-in: The *Metrics plug-in*[3] is a tool for extracting well-known OO metrics from a java program. We have used the *Metrics plug-in* to analyze

[2] Data summary available at http://freddy.cellcore.org/research/study/aop
[3] Available at http://metrics.sourceforge.net/

the java sources on each project and extract the OO metrics of interest for this study.

ABIS framework: *ABIS* [5] is a framework built on top of *AJDT* that aims at checking the interactions between aspects and the base code of an aspect-oriented program. We have extended *ABIS* in order to use its analysis capabilities and extract data about the crosscutting units and the presence of invasiveness patterns.

PCD Analyzer: *PCD Analyzer* is a tool built on top of *AJDT* that aims at analyzing the particular point-cut expressions used by an aspect-oriented project. We have used *PCD Analyzer* to gather data related to the PCD usage.

3.3. Experimental design and metrics

We have seized the 38 aspect-oriented projects from their repositories, and then processed them using the previously described tools. This resulted in the computation of a set of metrics described below.

The *Metrics plug-in* provided 2 metrics of interest for our study: *lines of code (LOC)* and *number of methods (NOM)*.

ABIS and *PCD analyzer* provided the count of the different units and relations comprising an aspect-oriented project. It is worth mentioning that according to the classical measurement theory [10] these metrics are in a *ratio* scale. In the following we summarize these metrics.

Number of advices (*NOAD*): counts advices defined with the keywords *before, around*, or *after*. This captures only the advices advising at least one join-point. We consider that advices advising zero join-points have no impact, and therefore are not significant for this study.

Number of advices realizing invasiveness patterns (*NOIAD*): counts advices that realize one or more invasiveness patterns and advise at least one join-point. It is worth mentioning that an advice realizing several invasiveness patterns counts only once.

Number of advices realizing each invasiveness pattern (*NARI*: *Read, Write, Replace, Conditional, Multiple, Argument*): counts the occurrence of invasiveness patterns concerned with this study (*read, write, replacement, conditional replacement, multiple, argument*). An advice realizing multiple patterns will count once for each pattern it realizes. For instance, the advice in listing 2 increases the count of *Conditional* and *Read*.

Number of join-points advised by each advice (*NAJP*): The *NAJP* metric is calculated for each advice, and the sum of the *NAJP* for all the advice in a project results in the *cumulated number of join-points matched by the advices* (C*AJP*).

Number of join-points advised by each invasive advice (*NIJP*): The *NIJP* metric is calculated for each invasive advice. The sum of the *NIJP* for all the advice in a project results in the *cumulated number of join-points matched by the invasive advices* (*CIJP*).

Number of statically calculable join-points (*NOJP*): counts the number of points in a project that can be statically matched by an advice. Although AspectJ allows developers to match join-points in libraries (jar files) and other sources, we limit our count only to the project's java source files. The points we count are: *method call, constructor call, initializations, assignments, exception handling, method declarations, and constructor declarations*.

Advices per method ratio (*AMR = NOAD / NOM*): this metric is the result of dividing the number of advices by the number of methods *(NOAD / NOM)* and corresponds to the number of advices per methods in the program.

Invasive advices per method ratio (*IAMR = NOIAD / NOM*): this metric is analogous to the previous but only considers the number of invasive advices (*NOIAD*)

Advised join-points ratio (*JMR* = C*AJP / NOJP*): this metric is the result of dividing the cumulated number of matched join-points by the number of statically calculable join-points (C*AJP / NOJP)*. This metric is an indicator of how spread are the crosscutting concerns realized by the advices regardless the project size.

Invasive advised join-point ratio (*IJMR = CIJP / NOJP*): this metric is analogous to the previous but considers the cumulated number of join-points matched by invasive advices (*CIJP*).

Number of PCDs comprising each terms of the AspectJ PCD language (*NPCD*: *And, Or, Not, Exec, Arg-dots, Star, Args, Call, Target, Within, Set, Init, Get, Withincode, If, This, Cflow, Cflowb, Staticinit, Handler, Advexec, Preinit*): counts the occurrence of each PCD term bound to an advice. A PCD containing multiple terms will count once for each term. For instance the first PCD in listing 1 will increment the count of *And, Args,* and *Call*, whereas the second will increment the count of *Exec, Target, Cflow,* and *If.*

4. Analysis results

In this section, we present the results of our analysis, addressing each research question in turn.

4.1. Research questions

As we stated in the introduction, the motivation of this paper is to better understand the usage of aspects in open-source projects, and their potential impact on maintenance. Based on this motivation, we study 3 sets of research questions that inquiry on the different facets of the open source aspect-oriented projects we collected.

Q 1: *What is the extent of aspect and invasive aspect usage in AO projects? Does this usage vary with the size of the project?* AOP promised to modularize crosscutting concerns into advices, but as we can observe, just a few projects integrate aspects. This question is important because it inquiries the usage of aspects in those projects containing them. The answer to this question will reveal whether aspects are used in a very reduced and precise way as predicted by Steimann in [11, 12], or the few projects containing aspects use them intensively. Furthermore, since large projects have potentially more concerns that can crosscut, it is natural to think that they will contain more aspects than small projects.

Q 2: *To what extent do aspects and invasive aspects really crosscut AO systems? Does this depend on the size of the systems?* This is important since AOP modularizes crosscutting concerns to later weave them with other concerns. Knowing the crosscutting of aspects will reveal whether the number of points where advices are woven is significant, or not. That is, whether the concerns modularized through AOP are spread enough to consider such modularization important. It is fair to say that the more points an advice advises, the more difficulty it is to manage and understand them. This question is meant to understand whether AO programmers tend to build aspects that are very specific and crosscut very few places in the program (as stated by Steimann in [11, 12]) or if they tend to write aspects widely spread through the whole program. Moreover, since large projects contain a large number of join-points where crosscutting concerns can be woven; it is reasonable to expect that aspects should be more crosscutting in large projects that in small projects. The relationship between crosscutting and projects' size will provide evidence supporting or contradicting this intuition.

Q 3: *Do PCDs use the full expressivity provided by the AspectJ pointcut language? Are invasive advices woven with precise PCDs?* This is important since the AspectJ language provides a large set of terms for writing PCDs. The usage of these terms indicates the way in which developers exploit the PCD language to capture the desired join-points, and the trust that developers put on them.

4.2. Aspect usage (Q1)

We analyze three dimensions of advices usage to answer Q1: number of advices, evolution of this number with respect to projects size, and the partition of invasive patterns among invasive advices.

First, let us analyze the advice per method ratio (*AMR*), and the invasive advice per method ratio (*IAMR*).

Table 1. Descriptive statistics for *AMR* and *IAMR*.

	Mean	Median	2.5%	97.5%	Min	Max	Std Dev
AMR	0.027	0.005	0.002	0.036	0.0003	0.128	0.039
IAMR	0.011	0.004	0.001	0.006	0.0003	0.074	0.018

Table 1 presents the descriptive statistics for *AMR* and *IAMR*.

AMR values are computed based on 36 projects and indicate that the quantity of advices in the projects is very small. It clearly appears (median = 0.005) that the number of advices per methods is very small, which means that scarcely used to modularize crosscutting concerns.

The project with the largest quantity of advices has 1 advice for 8 methods (out of 189 methods), whereas the project with the smallest quantity of advices has only one advice. Concerning the density, 68% of the projects have at maximum 1 advice per 37 methods, and 40% have at maximum 1 advice per 200 methods.

Two of the 38 projects are outliers for the *AMR* value and are not considered in table 1. These projects are small (less than 5000 *LOC*) and represent punctual cases of the AOP usage. One of them uses 27 advices to implement concerns such as graphical user interface (GUI) management, and exception handling. The other project uses 84 advices to implement concerns such as censoring, multithreading, persistence, replication, exception handling, and logging.

We analyze *IAMR* for the 21 projects containing invasive aspects (57% of the 38 projects). One of the outliers for *AMR* is also an outlier for *IAMR*. Out of 84 advices in this project, 39 implement invasive patterns such as replacement, and conditional replacement patterns, among others. Therefore, the *IAMR* is calculated from a universe of 20 projects.

The *IAMR* values (median = 0.004) indicate that there are even less invasive advices than regular ones (*IAMR* inferior to *AMR*), with a maximum of 1 advice for 13 methods in a small project. Concerning the density, 76% of the projects with invasive aspects have at maximum 1 advice per 90 methods, and 47% have 1 advice per 250 methods.

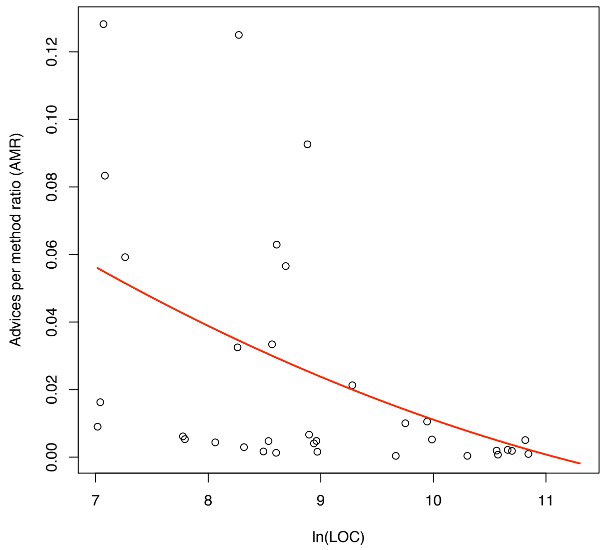

Figure 1. Scatter-plot illustrating the relationship between *AMR* and the projects' size.

Figure 1, illustrates the relationship between *AMR* and the projects' size. It appears that *AMR* decreases with the project size. The curve fit represented by the bold line in the plot endorses this thesis. Furthermore, the size of the projects does not imply a larger number of advices. We observe the same phenomenon for the evolution of *IAMR* with projects size. However, for both *AMR* and *IAMR*, some projects have a behavior that differs from the general tendency.

We highlight four projects (two small and two mid-size) having an *AMR* value over 0.08. *AMR* is high for the small projects because they comprise very few methods (20 and 78), and a few (2 and 10) very specific advices realizing concerns such as debugging mode or authorization that are woven at large number of locations in the base code. In one of the mid-size projects, a total of 38 advices realize 20 GUI functionalities such as drag and drop, redo-undo, etc. The other mid-size project has a logging concern that is realized at least in 10 different ways by 49 advices. As can be noticed, these are very specific cases of the aspect usage.

The single project having an *IAMR* value over 0.07 is a small project, which has 14 invasive advices realizing optional functionalities for a GUI and results to have also an *AMR* value over 0.08.

Regarding the different invasiveness patterns, we look at the *NARI* metric. Figure 3 shows a view of this metric. On top it shows a bar-plot of its cumulated value (sum of all the projects *NARI* metric), whereas on bottom it shows a box-plot of its value on the projects. Notice that the invasiveness pattern *multiple* has been removed from the plots because no advice out of the 21 projects realizes it.

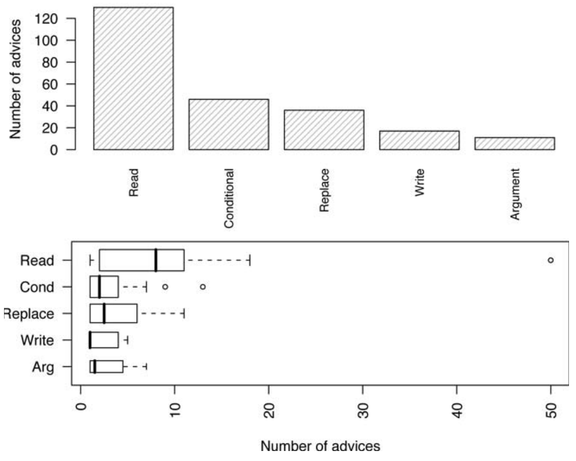

Figure 3. Bar-plot (top) of the cumulated *IAMR* value, and box-plot (bottom) of the *IAMR* value.

The bar-plot indicates that the number of advices realizing the *read* pattern outmatches all the others (80% of the projects). The next patterns in the list are *conditional replacement* (71 % of the projects) and *replacement* (61% of the projects), with less than half of the advices that realize the *read* pattern. The box-plot ratifies the dominance of the *read* pattern. It also shows that the value of the *conditional replacement* pattern is influenced by two extreme values and that instead, the *replacement* pattern follows the *read* pattern. We explain this situation by the fact that the *read* pattern is practically side effect free, and hence, developers trust it more than the other patterns.

Concerning the high values of the *conditional replacement* and *replacement* patterns, we observe the following: (1) the advices realizing the *conditional replacement* pattern are in most of the cases the implementation of transaction, authorization, and tracing concerns, 68% of them are in 3 projects (14% of the projects); (2) the advices realizing the *replacement* pattern are in most cases the implementation of alternative GUI functionalities, once again 63% of them are in 3 projects.

The *argument* pattern is mostly used (60%) to preprocess the request arguments of a web server, in a single project.

These results yield to several conclusions for Q1:

- *Developers use very few advices to implement crosscutting concerns*; this is ratified by a very small *AMR* maximum (0.128), with a median of 0.005.

– *Developers use few invasive advices*. Only 30% of all the advices realize an invasiveness pattern. This might be due to the fact that invasive advices can introduce side effects [5], and therefore, developers do not trust them. The observations of the *NARI* metric sustain this thesis, since the *read* pattern, that has no side effect, is dominant.

– *The projects' size does not imply an increment in the number of advices*. This contradicts the intuition that larger projects having more methods should have more advices to encapsulate the crosscutting concerns. This ratifies the postulate of Steimann that *aspects are few* [11, 12]. In next section we investigate if these few advices are widely spread through base programs.

4.3. Aspects crosscutting (Q2)

In this section we address Q2 by analyzing the proportion of join-points matched by all advices and by invasive advices.

Table 2. Descriptive statistics for *JMR* and *IJMR*.

	Mean	Median	2.5%	97.5%	Min	Max	Std Dev
JMR	0.020	0.001	0.0015	0.031	0.0001	0.092	0.024
IJMR	0.003	0.002	0.0007	0.003	0.0001	0.013	0.003

Table 2 presents the descriptive statistics for the advised join-points ratio (*JMR*), and the Invasive advised join-point ratio (*IJMR*), and figure 4 presents a histogram comparing them (*JMR* in dark gray, *IJMR* in light gray).

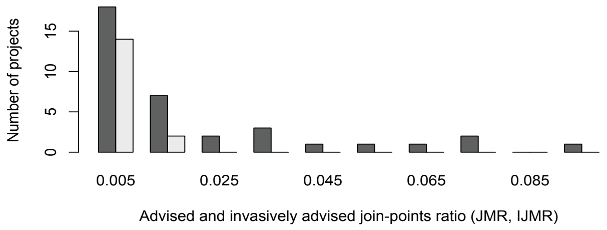

Figure 4. Histogram comparing the frequency of *JMR* and *IJMR* values.

A 0,092 value for the maximum *JMR* means that, at most, 9,2 % of the join-points that could be matched (*NOJP*) are actually matched by one join-point. The project with this maximum *JMR* has 11 advices that match less than 170 join-points in total. This means that, in average, there are 15 join-points per advice, which is a manageable amount of join-points that can all be checked and tested manually. Part from this maximum, the mean and the median indicate that, in general, advices advise from 1 to 2 percent of the *NOJP*. More important, from the histogram in figure 4 we notice that advices advise less than 0.5% of the *NOJP* in 41% (16) of the projects, whereas in 27% (10) of them between 0.5 and 2%.

Two projects are outliers for the *AMR* values and are not considered in table 2: a large project (more than 20000 *LOC*), that uses aspects to implement a performance measurement and profiling system and advise almost every method invocation in the project (a total of 13440 join-points); a small project (less than 5000 *LOC*), that uses 4 advices to handle GUI exit events. Since these projects contained very particular crosscutting advices, we considered them as outliers.

The *IJMR* values indicate that invasive advices are much less crosscutting than regular advices. All the *IJMR* values are less than the half the *JMR* values. In the project with the maximum *IJMR*, a small project, the advices advise 1.3% of the *NOJP*, equivalent to 16 join-points for 16 advices. If we look at the mean and median values, we notice that in general invasive advices advise less than 0.3% of the *NOJP*.

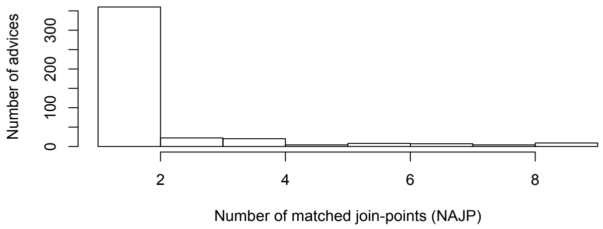

Figure 5. Histogram of *NAJP* frequency.

Figure 5 presents the individual crosscutting of the advices: it displays the number of advices that match a given number of join-points (*NAJP*). Since only 16% of advices advise more than 10 join-points, the histogram only shows *NAJP* below 10. What we see in this histogram is that 69% of the advices advise between 1 and 2 join-points, and only 15% of the advices advise between 2 and 5 join-points. These results confirm that most of the advices under study are very precise and the concerns they realize are usually woven in or 2 points.

The relationship of *JMR* and the projects' size is illustrated by figure 6. In the figure, a scatter-plot and a curve fit of the *JMR* values versus the projects' size.

From the curve fit and the shape of the plot, we observe that the general trend for *JMR* is to decrease with the projects' size. Furthermore, the size of the projects does not imply that advices are more crosscutting. We observe that this phenomenon is more accentuated for *IJMR*. However, locally, some projects have a behavior that differs from the general tendency. Notice that five projects have a high *JMR* value. These projects are well apportioned in the size spectra, 2 small, 2 mid, and 1 large project. More important, regardless their size, the commonality of these projects is that they comprise advises very crosscutting, part of the 5% of advices advising between 80 and 2000 join-points. These advices realize typical concerns [13] such as logging, debugging, and profiling among others.

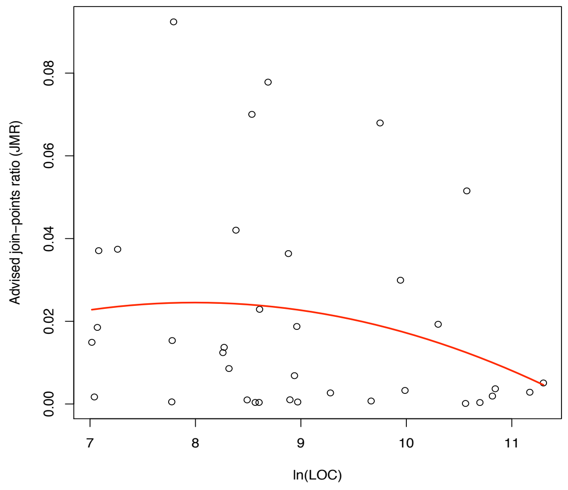

Figure 6. Scatter-plot illustrating the relationship between *JMR* and the projects' size.

These observations yield to several conclusions for Q2:

- *Developers write precise advices that advise few join-points*. The high number of advices advising less than 2 join-points (69%) and the high percentage of projects having a low *JMR* value (below 0.005) ratify this. This is congruent with the intuition that too many advised points imply less control on the effect of advices and on the maintainability of the project. This confirms the postulate of Steimann that *aspects are few and very precise* [11, 12].
- *Developers use the invasive facilities of AspectJ very carefully*. The *IJMR* values show that in general invasive advices advise few and precise join-points. The reason for this might be that invasive advices realize very precise concerns, and that since they can introduce side effects developers tend to keep and increased control over them.
- *The projects' size does not imply an increment in the advices crosscutting*. This contradicts the intuition that in large projects advices should be more crosscutting.

4.4. PCD usage (Q3)

This section investigates question Q3 through the analysis of the *NPCD* metric. Figure 7, shows a bar-plot of the cumulated sum of *NPCD* for all the projects (light gray), and projects containing only invasive advices (dark gray).

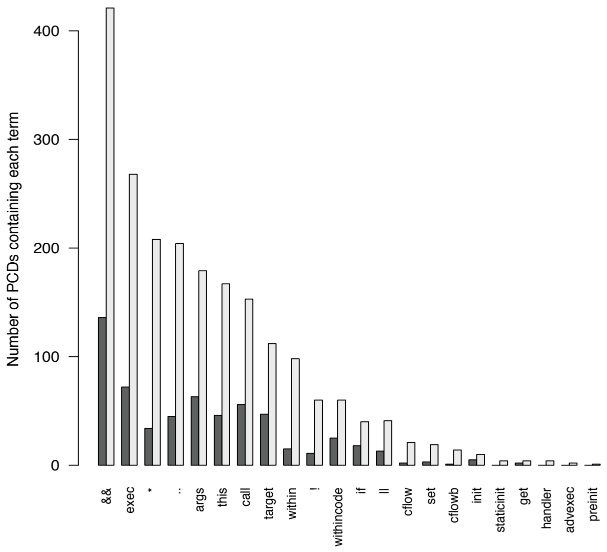

Figure 7. Bar-plot of the cumulated NPCD for all the projects, and the projects containing only invasive advices.

First, we can observe that a series of terms are present in very few PCDs (less than 1% of the PCDs). Terms such as *preinit, adviceExecution*, and *handler* are used at maximum by 4 out of 522[4] PCDs. Furthermore, 50% of the terms are present in less than 8% of the PCDs. This suggests that developers rarely use more than half of the AspectJ PCD language's expressivity.

We can also observe the large and low occurrence of the terms "&&" (80%) and "||" (8%) respectively. This indicates that *developers tend to narrow the number of matched join-points.* Since the "&&" forces the combination of two conditions (expressions) to be satisfied, it is used to narrow the scope of the base program that is advised by an aspect. Likewise, the presence of the keywords *within* and *withincode* supports this trend because they narrow the scope where join-points could be matched.

The large and small number of PCDs including *execution* (51%) and *call* (29%) respectively, indicate that *developers prefer to target the method execution instead of it calls. Execution* and *call* keywords indicate when the advice should be executed. The first forces the advice to be woven in the advised method code, whereas the second in the caller code [14]. We explain this by the fact that in general developers want their advices to execute regardless the calling facility.

The usage of dynamic keywords is forked in two trends. The keywords *args, this*, and *target* are present in 20 to 35% of the PCDs, whereas *if, cflow*, and

[4] Since PCD are always attached to an advice, we count each advice as having a single PCD. Therefore the number of PCDs is equivalent to the number of advices.

cflowbelow only in 4 to 8%. This suggests that developers may trust the first group of keywords and distrust the second. We explain this by the fact that the first group serves to capture data and specify types of the matched point, whereas second to specify a given moment or condition occurring during the program execution. Consequently, it is difficult to foresee the effect of this second group of keywords in complex PCDs, which can explain why, developers, prefer to avoid them.

Regarding the terms used in PCDs related to invasive advices, the low number of wildcards (less than 28%) indicates that developers tend to enumerate the points where invasive advices are woven. Besides, dynamics keywords such as *if*, *cflow*, and *cflowbelow* are almost never used to weave invasive advices.

These results yield to several conclusions for Q3:

- *In general, developers use only half of the expressiveness power provided by the AspectJ language.* This is ratified by the fact that half of the AspectJ PCD terms are present in less than 8% of the advices.
- *Developers write PCDs targeting precise join-points*. The large numbers of PCDs including terms that narrow the scope of matchable join-points sustain this thesis. Besides this endorses the conclusions drawn in section 4.3.
- *Developes use PCDs containing dynamic terms if, cflow, and cfowbelow in a very cautious way.* Evidence of this is the very small amount of PCDs comprising these terms.
- *PCDs for invasive advices tend to target a very specific list of joinpoints*. This confirms the observations from previous section where we noticed that invasive advices crosscut a very small portion of the base program.

4.5. Threats to validity

There exists no perfect data, or perfectly trustable analysis results, and this study is not an exception. For this reason we identify the construction, internal and external threats to validity for this study.

Internal threats lie on the source of the empirical data. We have selected our subject upon the available open source projects. Since we seized only open source projects, we have no pointer about the skills of the developers who have written the aspects in these projects. It is possible that well trained and skilled developers could write better advices, and modularize more crosscutting concerns.

Construction threats lie in the way we define our metrics and their measurement. The number of advices, join-points, and advised join-points, depends on the capacity of the current version of the AspectJ compiler to detect them and to weave the advices in the right places. However, due to unknown bug in the AspectJ compiler, this might not necessarily represent the developer wishes, or the real amount of advices in the source files. It is also possible that our metrics result are too coarse grained to draw pertinent conclusions, and that other metrics will be better fitted for this purpose.

External threats lie on the statistical significance of our study. We acknowledge that we have only observed 38 open source projects written in AspectJ language. We do not know to what extent this can be generalized to: (1) other AspectJ like AOP languages such as CaesarJ [15]; (2) industrial projects under closed development.

5. Related work

Apel et al [9], study the usage of aspects in eleven academic aspect-oriented programs. They divide the aspect usage in basic (inter-type declarations) and advanced (advices) and conclude that in general aspects are very few (14% of the code), and only a small portion corresponds to advanced usage. Lopez-Herrejon et al [18], define a set of metric for aspect-oriented programming that categorize crosscutting according to the number of classes crosscut and their language constructs. The authors observed these metrics on four aspect-oriented programs concluding that the number of classes crosscut by advices is very small and their crosscut reduced. The metrics defined by this study are very similar to ours; however, our metrics are oriented to the study of the particular usage of each language construction (including the PCD language) and their interaction with the base program. Furthermore, our inquiry reaffirms the results of these studies and extends them to a wider number of subject programs that goes beyond academic examples.

6. Conclusions

In this paper we analyzed the usage of AOP in 38 open source aspect-oriented projects, from small (less than 5000 LOC) to large size (more than 20000 KLOC) comprising a total of 479 aspects, and 522 advices. Our aim was to provide a better understanding of how and to which extent developers use AOP, its invasiveness facilities, and the PCD language. Through the analysis of different metrics we observed the trends regarding the amount of advices, their crosscutting, and the coverage of the AspectJ PCD language.

Our observations reveal that *developers use few advices to modularize crosscutting concerns,* and that *these advices are scarcely crosscutting.* The observations on the coverage of the PCD language confirm this: *developers write specific PCDs using only half of the AspectJ PCD language's* *expressiveness.* Furthermore, *developers write very few advices that break object-oriented encapsulation,* and *the small number of invasive advices, advise a small number of very specific join-points.*

These observations can suggest two types of interpretation. A pessimistic interpretation considers theses results as a proof of the distrust of developers for the aspect-oriented principles and as evidence that they intentionally ignore AOP even when their systems contain many crosscutting concerns. We discuss possible reasons for that below:

- Developers do not precisely know how to reason about crosscutting concerns and how to modularize them with aspects.
- Developers find it difficult to reason about units that seem modular but crosscut other units. Particularly when they think about AspectJ as an extension to OO, which can improve modularity but paradoxically reduces maintainability [6].
- The AspectJ language is not flexible enough to allow developers modularizing the total of crosscutting concerns.
- The invasive capabilities of AspectJ, which should help modularizing precise crosscutting concerns are not used because they can introduce side effects [5].
- The AspectJ PCD language contains a large number of terms, but makes testing complex [17] and is paradoxically not very expressive [16].

On the other hand, there can be an optimistic interpretation for these observations. This interpretation consists in viewing the presence of aspects in open source projects as a sign that developers have experimented AOP and that they have identified some interesting usages of aspect-oriented principles for specific purposes. According to such an interpretation, we can envision the trends identified in this empirical inquiry of AOP as usages that are useful and relevant for the development of software systems. It is then possible to increase the adoption of these specific usages of AOP by developing robust IDEs, analysis, testing and debugging tools based on simplified aspect-oriented features. For example, assuming there are no dynamic PCDs eases the development of efficient testing and analysis tools for AOP.

Eight years after the AOP was announced as a key technology, this study offers an actual view of AOP in practice. This should help researchers and practitioners think about the future development of aspect-oriented environments and languages, and also analysis and testing tools for AOP, supporting software development with AOP.

10. References

[1] B. Meyer, *Object-Oriented Software Construction 2nd edition*: Prentice Hall PTR, 2000.

[2] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho, "Do Crosscutting Concerns Cause Defects?," *IEEE Transactions on Software Engineering,* vol. 34, pp. 497-515, 2008.

[3] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proc. European Conf. on Object-Oriented Programming*. vol. 1241, M. A. a. S. Matsuoka, Ed. Jyväskylä, Finland: Springer-Verlag, 1997.

[4] T. J. v. d. Werff, "10 Emerging Technologies That Will Change the World," in *Technology Review*. vol. 1 Massachusetts: MIT, 2001.

[5] F. Munoz, B. Baudry, and O. Barais, "Improving maintenance in AOP through an interaction specification framework," in *IEEE Intl. Conf. on Software Maintenance*, Beijing, China 2008, pp. 77-86.

[6] T. Tourwé, J. Brichau, and K. Gybels, "On the existence of the aosd-evolution paradox," in *SPLAT: Software Engineering Properties of Languages for Aspect Technologies*, J. B. L. Bergmans , P. Tarr, and E. Ernst Editors, Ed. Boston, USA, 2003.

[7] M. Storzer and J. Graf., "Using pointcut delta analysis to support evolution of aspect-oriented software," in *Intl. Conf. on Software Maintenance*, Budapest, Hungary 2005.

[8] W. G. Griswold, M. Shonle, K. Sullivan, Y. Song, N. Tewari, Y. Cai, and H. Rajan, "Modular software design with crosscutting interfaces," *IEEE Software* vol. 23, pp. 51-60, 2006.

[9] S. Apel and D. Batory. "How AspectJ is Used: An Analysis of Eleven AspectJ Programs", *Technical report MIP-0801*, Universität Passau, Passau, Germany, 2008.

[10] D. Krantz, R. Luce, P. Suppes, and A. Tversky, *Foundations of Measurement*: Dover Publications, 2006.

[11] F. Steimann, "The paradoxical success of aspect-oriented programming," in *Proc. Conf. on Object-oriented programming systems, languages, and applications* Portland, Oregon, USA, 2006, pp. 481-497.

[12] F. Steimann, "Aspects are technical, and they are few," in *European Interactive Workshop on Aspects in Software EIWAS'04* Berlin, Germany, 2004.

[13] M. Marin, L. Moonen, and A. van Deursen, "Documenting Typical Crosscutting Concerns" in *14th Working Conference on Reverse Engineering,* Vancouver, Canada 2007.

[14] H. Erik and H. Jim, "Advice weaving in AspectJ," in *Proc. Conf. on Aspect-oriented software development,* Lancaster, UK, 2004.

[15] I. Aracic, V. Gasiunas, M. Mezini, and K.Ostermann, "Overview of CaesarJ," *Transactions on Aspect-Oriented Software Development,* vol. 3880, pp. 135 - 173, Feb 2006.

[16] K. Ostermann, M. Mezini, and C. Bockisch, "Expressive Pointcuts for Increased Modularity," in *Intl. Conf. Object-Oriented Programming*, Glassgow, UK 2005, pp. 214-240.

[17] R. Delamare, B. Baudry, Sudipto Ghosh and Y. LeTraon, "A test-driven approach to developing pointcut descriptors in aspectj" in *Intl Conf. on Software Testing, Verification, and Validation*, Denver, Colorardo, USA 2009.

[18] R. E. Lopez-herrejon and S. Apel, "Measuring and Characterizing Crosscutting in Aspect-Based Programs: Basic Metrics and Case Studies" in *Intl Conf. Fundamental Approaches to Software Engineering*, Braga, Portugal, 2007.