Usage and Testability of AOP: an empirical study of AspectJ☆

Freddy Munoz[1], Benoit Baudry[1,], Romain Delamare[1], Yves Le Traon[1]

[a] *INRIA / IRISA, Campus de Beaulieu, 35042 Rennes, Cedex, France*
[b] *University of Alabama, Department of Computer Science, Tuscaloosa, AL, USA*
[c] *University of Luxembourg, Campus Kirchberg, Luxembourg, Luxembourg*

Abstract

Context. Back in 2001, the MIT announced aspect-oriented programming as a key technology in the next 10 years. Nowadays, 10 years later, AOP is still not widely adopted.

Objective. The objective of this work is to understand the current status of AOP practice through the analysis of open-source project which use AspectJ.

Method. First we analyze different dimensions of AOP usage in 38 AspectJ projects. We investigate the degree of coupling between aspects and base programs, and the usage of the pointcut description language. A second part of our study focuses on testability as an indicator of maintainability. We also compare testability metrics on Java and AspectJ implementations of the HealthWatcher aspect-oriented benchmark.

Results. The first part of the analysis reveals that the number of aspects does not increase with the size of the base program, that most aspects are woven in every places in the base program and that only a small portion of the pointcut language is used. The second part about testability reveals that AspectJ reduces the size of modules, increases their cohesion but also increases global coupling, thus introducing a negative impact on testability.

Conclusion. These observations and measures reveal a major trend: AOP is currently used in a very cautious way. This cautious usage could come from a partial failure of AspectJ to deliver all promises of AOP, in particular an increased software maintainability.

Keywords: Aspect-oriented programming, metrics, empirical analysis

1. Introduction

Object-orientation (OO) pushes forward ideas such as *modularity*, *abstraction*, and *encapsulation* [?]. It promotes the separation of concerns as a corner-

☆This work was partially supported by the European project DiVA (EU FP7 STREP) and the NESSOS Network of Excellence.

*Corresponding author

stone to improve the maintainability, evolution, and comprehension of a software system. Concerns are features, behavior, data, etc., which are derived from the system requirements, domain, or even its internal details [**?**]. 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 [**?**]. 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 *pointcut descriptors*, which designate the points in the program where the advices are inserted. 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 the evolution of single units instead of scattered code fragments.

Since its introduction in 1997, many technical documents, research papers, books, and conference venues discussed and commented on AOP and its benefits. In 2001 the MIT announced AOP as a key technology for the future 10 years [**?**]. Later, in 2002 a growing scientific community launched the first International Conference on Aspect Oriented Software Development (AOSD), and about 300[2] documents cited AspectJ (the most popular incarnation of AOP) and AOP. The same year less than 10 open-source projects were actually using such technology in the source-forge repository.

Nowadays, 10 years after the MIT announcement, the number of documents about AOP and AspectJ has grown to more than 2500[2]. During the same period, the number of projects using AOP has increased only to about 60 projects (less than 0.5% of source-forge's projects developed using Java in the period from 2001 to 2008 integrate aspects). When facing this apparent paradox, we can wonder what prevents a more extensive use of AOP in what context it has been a good solution.

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

[2] According to an estimation using the google scholar search engine.

Previous work has identified two characteristics of aspect-oriented languages that hinder maintainability and evolvability: (1) the fragility of the pointcut descriptors that leads to the evolution paradox [? ?]; (2) the ability of aspects to break the object-oriented encapsulation [? ?]. Also, when looking at aspects for analysis or testing, another paradox seems to occur: aspects allow the extraction of scattered code in a single unit, thus improving the consistency of modules, but aspects can also increase coupling between modules when woven at multiple places. This increased coupling has a negative impact on testability, since it prevents an incremental approach for testing. In turn, this decreases maintainability because the testing effort will be impacted each time the program evolves.

In this paper we present a two-step empirical analysis of AOP, which is an extension of the experiment presented at ICSM'09 [?]. First, we analyze the current usage of aspect-related features in open source projects. We study 38 open source aspect-oriented projects developed with the Java and AspectJ languages in the first study. In particular, we analyze the number of aspects with respect to size of programs, the degree to which aspects break the object-oriented encapsulation and how much of the expressive power for pointcut descriptors is actually used. This analysis disregards the pointcuts leading to augmentation and crossing advices (*i.e.*, advices that do not disturb the proceed of base methods). This reveals that aspects are used in a very cautious way. This leads to the second part of our experiment in which we investigate a possible reason for this distrust.

The second step of the experiment aims at evaluating if AOP has kept its promises of better maintainability than OO. Our hypothesis here is that AOP does not keep its promises, it can be a reason why developers do not trust this techniques. We focus on testability as an indicator of maintainability. We compare the evolution of testability indicators over 3 versions of a system implemented with both Java and AspectJ technologies. This reveals that in the AspectJ versions, modules are more cohesive but are also more coupled. The increased coupling among modules suggests that AspectJ reduces testability by introducing modules that cannot be tested in isolation.

This empirical inquiry of aspects requires collecting and measuring data from aspect oriented programs. Thus, as an initial contribution for this work, we have developed tools for measuring different metrics on AspectJ programs. First, we extended Briand's OO metrics framework [?] with aspect-oriented specific features such as advices or invasive advices. The framework models all necessary information to compute metrics related to coupling, complexity, and modularity in aspect-oriented programs. Then, we developed a tool to measure these metrics on AspectJ programs. The tool also contains a module to measure AspectJ specific metrics.

We observe four 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 pointcut descriptors; (3) pointcut descriptors are defined with only half of the available expressions; (4) aspects modularize a series of concerns increasing the

```
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: Example of AspectJ pointcuts

software's modularity, however, this modularization introduces coupling that hinder testability.

This paper is structured as follows: Section **??** introduces the aspect-oriented programming concepts. Section **??** describes the theoretical framework and the tooling support backing our empirical study. Section **??** describes the experimental data and the research questions this study inquiries. Section **??** presents the analysis results for each research question. Section **??** discusses the related work. Section **??** concludes the paper by summarizing the main results and discussing their implications for maintenance and AOP adoption.

2. Aspect-oriented programming: the case of AspectJ

In aspect-oriented programming (AOP), aspects are defined in terms of two units: *advices*, and *pointcut descriptors* (PCD). *Advices* are units that realize the crosscutting behavior, and pointcuts 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 **??**) presents the PCD declaration for *logging* (lines 2-5) and *transaction* (lines 7-10) concern, whereas the second (Listing **??**) presents an advice (lines 3-14) realizing a *transaction* concern.

2.1. Pointcut descriptors

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 **??** (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`.

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:

```
1  public aspect BankAspect{
2     pointcut transaction(): ...
3  
4     boolean around(): transaction(){
5        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: Example of an AspectJ advice

wildcards, *logical operators*, and *primitive pointcut descriptors*. The combination of names and terms is referred to as *expression*.

Wildcards serve to enlarge the number of matches produced by a *name*. The AspectJ PCD language defines three wildcards: "*", "..", and "+". The PCD transaction (Listing **??**) presents an example of their usage. In line 8, the wildcard * enlarges the matchings of the name `boolean Account.*(int)` to any method in the class `Account`, which returns a type `boolean`, and receives a single argument of type `int`. The wildcard + is used at the end of a type pattern, and indicates that sub-types should also be matched.

logical 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 logical operators, "`&&`" (*conjunction*), "`||`" (*disjunction*), and "`!`" (*negation*).

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

Some primitive PCDs designate join points that can be computed only at runtime. The AspectJ PCD language defines 6 primitive PCDs for that purpose: *cflow*, *cflowbelow*, *if*, *args*, *this*, and *target*. The transaction PCD (lines 7-10) incorporates this kind of primitive PCDs. 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 dy-

namic 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*.

2.2. Advices

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 bound to a PCD designating the points where they will be executed. For instance, the advice in Listing **??** (lines 3-14) is bound to the PCD *transaction* (line 3). AspectJ provides three different kinds of advice *before*, *after*, and *around* indicating the moment when they are executed. Before and after advices are executed respectively just before, or after reaching the join points designated by the PCD. Around advices are special types of advices that are executed instead of the designated join points. For example, in Listing **??**, the advice is executed instead of all methods in *Account* which have one integer parameter and that are called by the *Bank.operation()* method.

By invoking the special operation "`proceed`" the advices can execute the captured join point. For instance, the advice in Listing **??** executes the captured join point only if the *balance* and the *credit* are strictly positive(lines 6-8); otherwise it replaces its execution (lines 10-13).

2.3. Invasive patterns in AspectJ

Advices such as the one presented in Listing **??** 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 [**?**] we identified 8 invasiveness patterns for advices, and 3 for aspects[3]. 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 follows:

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 **??**, 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.

[3]Different aspect classifications are discussed in Section **??**.

6. *Conditional replacement*: the advice replaces the behavior of the advised method under certain conditions (e.g advise in Listing **??**).
7. *Multiple*: the advice executes the advised method several times.
8. *Crossing*: the advice invokes one or more methods that it does not advise.

We can note that the '*Read*' pattern is not strictly invasive, in the sense that it does not modify the behavior nor the state of the program. Still, we include it in our list of patterns, for the sake of completeness with respect to the kind of interactions that can exist between advices and a base program.

For the purpose of analyzing the usage of the different invasiveness patterns, we disregard the patterns *augmentation* and *crossing*, in order to focus only on advices that can disturb the regular proceed of a method. Augmentation and crossing are the most frequently used advices (they are about 71% of the advices in the code base). They are often called observation advices because they only observe the flow (e.g., a logging aspect). Furthermore, in general for an advice to be useful it needs to interact with other classes, thus most advices are crossing.

3. Analysis Support

Before starting our inquiry of the different facets of aspect-oriented programs, we precisely define two sets of *aspect-oriented metrics* that we use in our experiments. The first set contains generic metrics, that are independent of any aspect-oriented programming language. Theses metrics are defined on an abstract model that captures the essential constructs of aspect-oriented programs. This model, that we call *theoretical framework*, is an extension of Briand's OO metrics framework [**?**] that supports aspect-oriented concepts such as advices or inter-type definitions. The second set contains metrics that are specific to the AspectJ language, particularly regarding the pointcut description language.

The abstract model for aspect-oriented programs and the two sets of *aspect-oriented metrics* have been implemented in a tool. This tool can process any AspectJ program and measure all metrics on this program.

3.1. Theoretical Framework

In the past, Briand *et al.* [**?**] proposed a formalism for analyzing object-oriented programs. The goal of Briand's work was to define a terminology and a formalism to capture the core concepts of object-oriented programming independently of any language. This formalism then allowed the definition of *generic* metrics for object-oriented programs.

In this section we extend the framework proposed in [**?**] to support aspect-oriented concepts. Similarly to Briand, our goal with this extension is to provide the support for aspect-oriented metrics independently of a particular aspect-oriented programming language. The benefit is that the we can reuse the metrics definition to measure programs implemented with AspectJ, CaesarJ or Spring AOP for example, as illustrated in Figure **??**. In order to measure metrics on a specific implementation it is necessary to extract a model from the program that

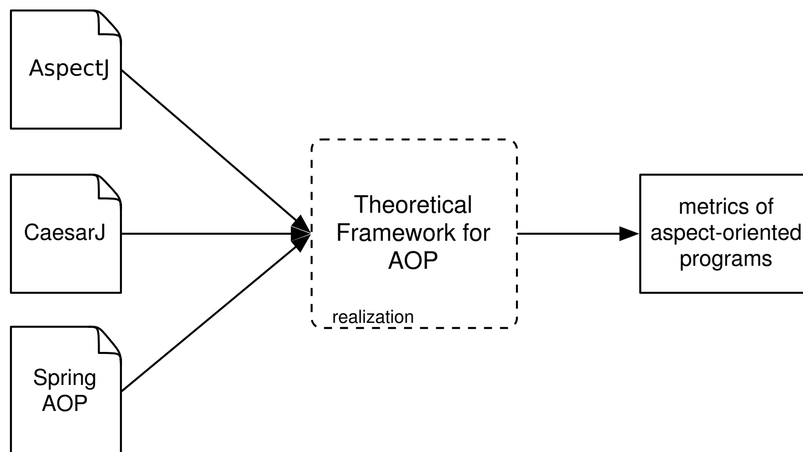

Figure 1: Overall work flow of the theoretical framework.

contains all the concepts defined in our theoretical framework. Then the metrics definitions can be reused. The theoretical framework is meant to be as language-independent as possible, thus language specific notions are not modeled.

We have kept everything that was in Briand's framework, since we target aspect-oriented languages that are extension of object-oriented languages. This also guarantees that, if an object-oriented program that has no aspects is analyzed, we can still gather the OO metrics using the same framework.

3.1.1. Choices when considering aspects in the framework

There exists several aspect-oriented languages that all have specificities and common concepts. Most of these languages can be abstracted using the proposed framework.

Specific classes. Some languages introduce a new kind of class that is used to encapsulate the new concepts of AOP. In AspectJ theses specific classes are called *aspects*, in CaesarJ [**?**] they are called *cclasses*. Other languages such as AspectWerkz [**?**] or JBoss AOP [**?**] have no specific classes.

In the proposed framework, classes and specific classes are not distinguished. As the differences are very specific and sometimes only syntactical, we have chosen to keep these differences out of the framework. In the theoretical , the term *'module'* will be used to designate either classes or specific classes (such as aspects in AspectJ).

Advices. In most AOP languages, an advice is a method and what is woven is just an invocation of this method. In AspectJ and CaesarJ, a specific syntax is used, whereas in JBoss AOP and AspectWerkz advices are regular java methods.

In the proposed formalism, advices and methods are not distinguished, for the same reason as for the specific classes. In the theoretical framework, the term *'operation'* will be used to designate either methods or advices.

Pointcut descriptors. Pointcut descriptors (PCDs) is the AOP concept that differs the most between the different languages. They can be defined using a special syntax (AspectJ, CaesarJ), annotations (AspectJ, AspectWerkz, JBoss AOP) or xml (AspectWerkz, JBoss AOP). In this formalism, we consider that the PCDs have been resolved and there is a relation between operations to determine the advised operations. Thus, the concept of PCD is not directly formalized.

3.1.2. Synthesis on the extension to Briand's framework

Inter-type definitions. Inter-type definitions are operations or attributes of a class that are defined by an aspect. The extension for inter-type definitions consist in the two sets $O_A(m$ and $A_A(o)$. $O_A(c)$ is the set of the operations of module m that are defined outside of m by other modules. $A_A(m)$ is the set of the attributes of module m that are defined outside of m by other modules.

Advices. Since advices and methods are unified as operations, the extension for the advices consists in $WA(o)$, $NWA(o, o')$, T and $IA(t)$. $WA(o)$ is the set of advices (operations) that are woven within the operation o. $NWA(o, o')$ is the number of time that the advice o' is woven within o. T is the set of the invasive pattern, and $IA(t)$ is the set of advices that realize the invasive pattern t. The predicate *uses* has also been modified to take the advices into account.

3.2. Definitions

In the following we present the framework. Extended definitions are marked with a '*', and new definitions are marked with '**'.

3.2.1. System

All entities that can contain operations and attributes are considered as modules. So for instance in AspectJ, interfaces and aspects are not distinguished from regular classes.

Definition 1. System, Modules, Inheritance Relationships. *An aspect-oriented system consists of a set of modules, M. There can exist inheritance relationships between modules such that for each module $m \in M$ let*

- $Parents(m) \subset M$ *be the set of parent modules of m.*
- $Children(m) \subset M$ *be the set of children modules of m.*
- $Ancestors(m) \subset M$ *be the set of ancestor modules of m.*
- $Descendents(m) \subset M$ *be the set of descendent modules of m.*

3.2.2. Operations

The operations of a module m are sorted in three partitions: operations that are declared in m, operations that are implemented in m, and operations introduced with inter-type definitions.

Definition 2. Operations of a Module. *For each module $m \in M$ let $O(m)$ be the set of operations of m.*

Definition 3. *** Declared, Implemented, and Inter-type Defined Operations. *For each module $m \in M$, let*

- *$O_D(m) \subseteq O(m)$ be the set of operations declared in m, i.e., operations that m inherits but does not override or virtual operations of m.*
- *$O_I(m) \subseteq O(m)$ be the set of operations implemented in m, i.e., operations that m inherits but overrides or non-virtual non-inherited operations of c.*
- *$O_A(m) \subseteq O(m)$ be the set of inter-type defined operations for m, i.e., operations of m declared and implemented by another module.*

where $\{O_D(m), O_I(m), O_A(m)\}$ is a partition of $O(m)$.

Definition 4. *$O(M)$ is the set of all operations in the system and is represented as*

$$O(M) = \bigcup_{m \in M} O(m)$$

3.2.3. Invocations

Two sets of invocations are defined for each operation: SIO (Statically Invoked Operations) is the set of the operations explicitly invoked and PIO (Polymorphically Invoked Operations) is the set of the operations that can be invoked because of polymorphism. Invocations counters are also defined.

Definition 5. *Let $SIO(o)$ be the set of operations statically invoked by o. Let $m \in M$, $o \in O_I(m)$, and $o' \in O(M)$. Then $o' \in SIO(o) \Leftrightarrow \exists m' \in M$ such that $o' \in O(m')$ and the body of o has an operation invocation where o' is invoked for an object of static type module m'.*

Definition 6. *Let $m \in M$, $o \in O_I(m)$, and $o' \in SIO(o)$. $NSI(o, o')$ is the number of operation invocation in o where o' is invoked for an object of static type module m' and $o' \in O(m')$.*

Definition 7. *Let $PIO(o)$ be the set of operations polymorphically invoked by o. Let $m \in M$, $o \in O_I(m)$, and $o' \in O(M)$. Then $o' \in PIO(o) \Leftrightarrow \exists m' \in M$ such that $o' \in O(m')$ and the body of o has an operation invocation where o' may, because of polymorphism and dynamic binding, be invoked for an object of dynamic module m'.*

Definition 8. *Let $m \in M$, $o \in O_I(m)$, and $o' \in PIO(o)$. $NPI(o, o')$ is the number of operation invocations in o where o' can be invoked for an object of dynamic type module m' and $o' \in M(o')$.*

3.2.4. Advices

Advices are operations that are woven in other operations. *NWA* approximates dynamic pointcuts by considering only the static part. The dynamic part, which can only be computed at runtime, does only constraint the static part, thus this is an over-approximation.

Definition 9. **** Woven Advices (WA). *Let $m \in M$, $o \in O_I(m)$, and $o' \in O(M)$. Then $o' \in WA(o) \Leftrightarrow \exists m' \in M, o' \in O_I(m')$ and o' is woven within the body of o.*

Definition 10. *** Let $m \in M$, $o \in O_I(m)$, and $o' \in WA(o)$. $NWA(o, o')$ is the number of times o' is woven within o.*

3.2.5. Invasive Advices

Definition 11. **** Invasive Advice Pattern Type. *Let T, be the set of all the types of invasiveness pattern. $T =$ {Write, Read, Argument passing, Augmentation, Replacement, Conditional replacement, Multiple, Crossing}.*

Definition 12. **** Invasive Advices (IA). *Let $t \in T$. Then, $IA(t, M) = \{o \in O(M) | \exists\ o' \in O(M), o \in WA(o')$ and o realizes the invasiveness pattern $t\}$.*

Definition 13. *** IA(M) is the set containing all the invasive advices in the system:*

$$IA(M) = \bigcup_{t \in T} IA(t, M)$$

3.2.6. Attributes

In a module m, there are three kind of attributes. Attributes that are inherited (declared attributes), attributes that are implemented by m, and attributes introduced with inter-type definitions.

Definition 14. Attributes of a Module. *For each module $m \in M$ let $A(m)$ be the set of attributes of module m.*

Definition 15. *** Declared and Implemented Attributes. *For each module $m \in M$, let*

- *$A_D(m) \subseteq A(m)$ be the set of attributes declared in module m (i.e., inherited attributes).*
- *$A_I(m) \subseteq A(m)$ be the set of attributes implemented in module m (i.e., non-inherited attributes).*
- *$A_A(m) \subseteq A(m)$ be the set of inter-type defined attributes for module m.*

where $\{A_D(m), A_I(m), A_A(m)\}$ is a partition of $A(m)$.

Definition 16. *$A(M)$ is the set of all attributes in the system and is represented as*

$$A(M) = \bigcup_{m \in M} A(m)$$

Definition 17. *For each $o \in O(M)$ let $AR(o)$ be the set of attributes referenced by operation o.*

3.2.7. Predicate

Definition 18. * Uses. *Let $m \in M$, $m' \in M$.*

$$\begin{aligned} uses(m, m') \quad \Leftrightarrow \quad & (\exists o \in O_I(o), \exists o' \in O_I(m'), o' \in PIO(o)) \vee & (1) \\ & (\exists o \in O_I(m), \exists a \in A_I(m'), a \in AR(o)) \vee & (2) \\ & (\exists o \in O_I(m), \exists o' \in O_I(m'), o' \in WA(o)) & (3) \end{aligned}$$

This predicate states that a module m *uses* a module m', (1) if it invokes an operation defined by m', (2) if it references an attribute of m', or (3) if m' has an operation that is woven in m.

3.3. Aspect-Oriented Metrics Definition

This section details the two sets of *aspect-oriented metrics* we use for our experimental inquiry of AOP.

3.3.1. Generic Aspect-Oriented metrics

We present a set of metrics defined only using the concepts defined in the theoretical framework for AOP. Since this framework captures language independent aspect-oriented concepts, these metrics are generic in the sense they can be measured on any aspect-oriented program. This is illustrated in figure **??**.

Definition 19. Coupling Between Objects (CBO and CBO').

$$\begin{aligned} CBO(c) &= |\{c' \in C - \{c\} | uses(c, c') \vee uses(c', c)\}| \\ CBO'(c) &= |\{c' \in C - (\{c\} \cup Ancestors(c)) | uses(c, c') \vee uses(c', c)\}| \end{aligned}$$

CBO has been defined by Chidamber and Kemerer [**?**]. Given a module m, CBO is the number of other modules it is directly coupled to (i.e. modules m uses or modules that use m). This definition is not different from the standard coupling definition because the aspects are already present in the *uses* predicate.

Definition 20. Response for module (RFM and RFM').

$$\begin{aligned} R_0(m) &= O(m) \\ R_{i+1}(m) &= \bigcup_{o \in R_i(m)} PIO(o) \cup WA(o) \\ RFM_\alpha(m) &= \left| \bigcup_{i=0}^{\alpha} R_i(m) \right| \\ RFM'(m) &= RFM_\infty(m) \\ RFM(m) &= RFM_1(m) \end{aligned}$$

The *RFM*, as defined by Chidamber and Kemerer [**?**], is the set of operations that can be called by operations in a module. *RFM'* is the transitive closure of RFM.

Inter-type definitions are taken into account as they are included in $O(m)$. Woven advices have been added and are considered as if the advice is invoked by the operations where it is woven.

Definition 21. Message passing coupling (MPC).

$$MPC(m) = \sum_{o \in O_I(m)} \sum_{o' \in SIO(o) - O_I(m)} NSI(o, o') + NWA(o, o')$$

MPC has been defined by Li and Henry [**?**] as the number of send statement in a module. Advice weaving have been added for the same reasons as for RFM.

Definition 22. Lack of Cohesion Of Operations (LCOO).

$$LCOO(m) = \frac{\left(\frac{\sum_{a \in A_I(m)} |\{o \in O_I(m) | a \in AR(o)\}|}{|A_I(m)|} \right) - |O_I(m)|}{1 - |O_I(m)|}$$

LCOO is based on the definition by Henderson-Sellers [**?**] and has been defined in the formalism of Briand *et al.* by Bruntink and van Deursen [**?**]. The LCOO value is zero, if all the attributes of m are referenced by all the operations of m. This indicates perfect cohesion. The LCOO value is one if there is a complete lack of cohesion. In that case each attribute is referenced by at most one operation. LCOO cannot be computed if there is no attribute or only one operation, because it would result in a division by zero, and it does not make sense in both cases.

Definition 23. Number of advices (NOAD):

$$NOAD(M) = \left| \bigcup_{o \in O(M)} WA(o) \right|$$

NOAD counts the number of advices that advise at least one join point in the system C. We consider that advices advising zero join points have no impact, and therefore are not significant for this study.

Definition 24. Number of advices realizing invasiveness patterns (NOIAD):

$$NOIAD(M) = \left| \bigcup_{t \in T} IA(t, M) \right|$$

NOIAD counts advices that realize one or more invasiveness patterns and advise at least one join point in the system M. It is worth mentioning that an advice realizing several invasiveness patterns counts only once.

Definition 25. Number of advices realizing each invasiveness pattern (NARI: Read, Write, Replace, Conditional, Multiple, Argument):

$$NARI(t, M) = |IA(t, M)|$$

NARI 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*.

Definition 26. Advices per method ratio (AMR):

$$AMR(M) = \frac{NOAD(M)}{|O(M)| - NOAD(M)}$$

AMR is the result of dividing the number of advices by the number of methods and corresponds to the number of advices per methods in the program.

Definition 27. Invasive advices per method ratio (IAMR):

$$IAMR(M) = \frac{NOIAD(M)}{|O(M)| - NOAD(M)}$$

IAMR is analogous to the previous but only considers the number of invasive advices.

Definition 28. Number of join points advised by an advice (NAJP):

$$NAJP(o) = \sum_{o' \in O(M)} NWA(o', o)$$

NAJP gives the number of advised join point for an advice m in a system C.

Definition 29. Cumulated number of join points matched by the advices (CAJP):

$$CAJP(M) = \sum_{o \in O(M)} \sum_{o' \in WA(o)} NWA(o, o')$$

CAJP is the cumulated number of advices advising join points in a system C.

Definition 30. Cumulated number of join points matched by the invasive advices (CIJP):

$$CIJP(M) = \sum_{o \in O(M)} \sum_{o' \in IA(M)} NWA(o, o')$$

CIJP is the cumulated number of join points advised by invasive advices in a system C.

3.3.2. AspectJ specific metrics

Some metrics are tightly coupled with a particular programming language. For instance, counting of specific language constructs and concepts such as the AspectJ *within* pointcut term. These metrics cannot be described on top of the proposed framework, and thereby we defined them separately. It is worth mentioning that these metrics are typically extracted directly from the program structure, for instance by counting the occurrence of a given construct such as a method call, or a pointcut declaration term.

Definition 31. Number of join point shadows (NOJP): counts the number of join point shadows in a system 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.*

Definition 32. Advised join points ratio (JMR)*:*

$$JMR(C) = \frac{CAJP(C)}{NOJP(C)}$$

The *JMR* gives a relation between the number of matchable join points in the system and the points that advices actually advice. This metric indicates the distribution of the crosscutting concerns realized by the aspects, independently of the size of the project.

Definition 33. Invasive advised join point ratio (IJMR):

$$IJMR(C) = \frac{CIJP(C)}{NOJP(C)}$$

The *IJMR* works analogously to the previous metric but considers the *cumulated number of join points matched by invasive advices* (*CIJP*).

Definition 34. 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* .

3.4. Tool Support

In order to extract and analyze the metrics we previously presented, we use a set of tools. To obtain OO metrics, we use the *Metrics plug-in*[4], a tool

[4]Available at http://metrics.sourceforge.net/

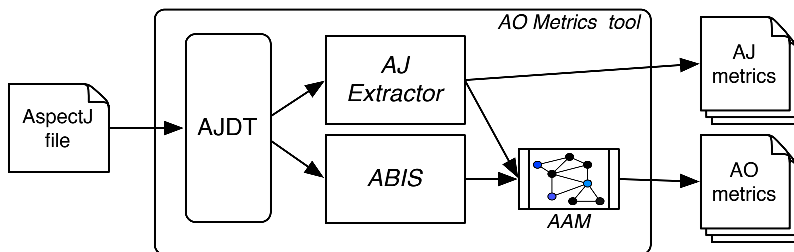

Figure 2: Architecture for extracting AO metrics from an AspectJ program.

for extracting elementary OO metrics from a Java program. We have used it to analyze the java sources on each project and extract the *number of classes (NOC)*, *number of methods (NOM)*, and *lines of code (LOC)*.

We have also developed a tool that can measure both generic AO metrics and AspectJ specific metrics on an AspectJ program. The major components of the tool are illustrated in Figure **??**. The tool is based on *AJDT* to statically extract an abstract syntax tree (AST) from the AspectJ program. Starting from the AST we build a model (AAM) of the program. This model contains all the elements defined by the theoretical framework. Then, once the model is built, it is possible to use the generic metrics definitions to compute all generic AO metrics.

This analysis is performed by two different modules: the ABIS tool [**?**], which extracts all information related to interaction patterns in AspectJ programs; the *AJExtractor* module, which extracts all information about methods, invocations, attributes and computes the Uses predicate. It is worth mentioning that this module can additionally compute the AspectJ specific metrics directly on the AST.

4. Experimental Design

In this section we present the experimental data and settings we used to empirically inquiry the different facets of aspect-oriented programs.

4.1. Experimental Data

Aspect-oriented open source projects:. One source of experimental data for this study consists of 38 aspect-oriented projects available under open source licenses[5]. We have collected these projects from public repositories in July 2008. We selected these projects according to the following criteria: (1) implemented in Java / AspectJ, (2) source code publicly available, (3) compiles using the AspectJ compiler version 1.5, (4) at least 10 classes and 1 aspect in the project, and (5) the advices in the project advise at least 1 join point. We started our

[5]Data summary available at `http://freddy.cellcore.org/research/study/aop`

Small (1000 to 5000 LOC)	Medium (5000 to 20000)	Large (more than 20000)
36%	36%	28%

Table 1: Distribution of projects according to their size in lines of code

1 - 10 advices	1 - 30 advices	more than 30 advices
62%	25%	13%

Table 2: Distribution of projects according to the number of advices

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). Only aspects defined with the *@Aspect* syntax, introduced with AspectJ 5, can be defined inside a regular Java file. This syntax is not widely used, thus only a few projects could have been left out because of it. Out of 2000 files, equivalent to 28 projects, only 10 fulfilled our selection criteria. Finally, we successfully gathered 38 open source aspect-oriented projects.

The 38 open source projects range from small to large size in lines of code (1116 - 80818 LOC). The proportions of small, medium and large projects are provided table **??**. Together, the 38 projects have 53083 methods scattered in 7343 classes, and $\sim 65 \times 10^4$ join point shadows.

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. The proportions of projects according to the number of advices are given in table **??**. Among the 38 projects, 57% of them comprise at least one advice realizing an invasive pattern, which corresponds to 30% of all the advices.

The studied projects were selected in July 2008, but we believe that the usage of aspects has changed in a way that the few relevant projects would contradict the conclusions of this study. As of June 2012, a request on Github (which is now more popular than Sourceforge) returns about 50 new projects using AspectJ. Most of these projects are simple example of the usage of AspectJ and thus are not very relevant to this study.

Health Watcher AOP systems. : Another source of experimental data for this study is the HealthWatcher system[6]. The HealthWatcher system is a middle-sized real life health complaint system developed to improve the quality of services provided by health care institutions. It has been evolved and re-factored several times in order to prove that aspects improve the modularity of concerns like *distribution* and *persistence* [**?**]. We experimented with three different

[6] http://www.comp.lancs.ac.uk/~greenwop/tao/

	Java	**AspectJ**
Version 1	76 classes and 12 interfaces	76 classes, 15 interfaces, and 11 aspects
Version 2	80 classes and 12 interfaces	81 classes, and 18 interfaces, and 13 aspects
Version 3	92 classes and 12 interfaces	111 classes, 18 interfaces, and 17 aspects

Table 3: Size (in number of classes, interfaces, and aspects) for each implementation of the HealthWatcher system.

versions of the HealthWatcher system. Each of these has two implementations, one developed using pure Java, and another using AspectJ. Table **??** shows the size of the different implementations.

AspectJ aspects.. It should be noted that, even if our metric framework can be used to analyze aspects expressed in diffent AOP languages (as mentioned in **??**), all our experimental data include aspects expressed in AspectJ only.

4.2. 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 testability. Based on this motivation, we study 4 research questions that inquiry on the different facets of the open source aspect-oriented projects we collected.

Q1. 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 [**? ?**], 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. The relationship of the aspects usage with the projects' size will support or contradict this intuition.

Q2. 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 difficult it is to manage and understand them. This question is meant to understand whether

	Mean	Median	2.6%	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 4: Descriptive statistics for *AMR* and *IAMR*

AO programmers tend to build aspects that are very specific and crosscut very few places in the program (as stated by Steimann in [**?** **?**]) 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.

Q3. 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.

Q4. How does AspectJ influence testability?* Testability captures attributes of a program that bear on the effort needed to validate the software product [**?**]. Testable programs are more maintainable because they allow programmers to better understand the impact of a change. Testability is correlated to modularity: small and cohesive modules loosely coupled increase testability because the functionality of a module is clearly identified and can be tested in isolation. The answer to this question will reveal qualitative dimensions of AspectJ and allows us to inquire whether AspectJ has kept one of AOP promises: *improving maintainability.*

5. Analysis Results

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

5.1. Aspect Usage (Q1)

We analyze three dimensions of advices usage to answer Q1: number of advices, evolution of this number with respect to project 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 **??** 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 aspects are scarcely used to modularize crosscutting concerns.

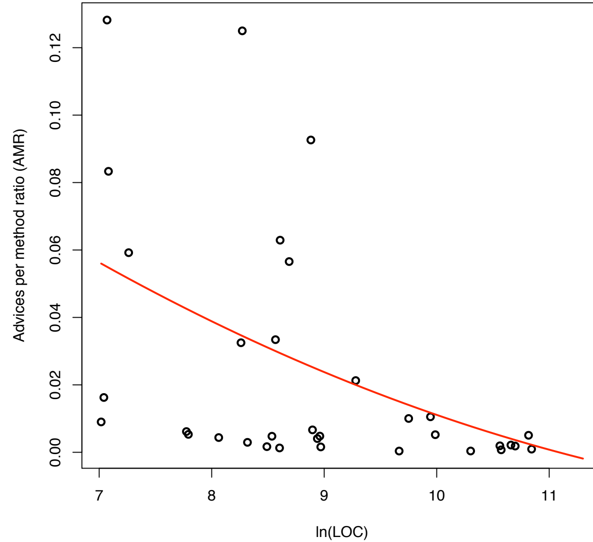

Figure 3: Scatter-plot illustrating the relationship between *AMR* and the projects' size.

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[7] for the *AMR* value and are not considered in Table **??**. These projects are small and represent punctual cases of the AOP usage. One of them (2391 LOC) uses 27 advices to implement concerns such as graphical user interface (GUI) management, and exception handling. The other project (4377 LOC) 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.

[7] According to Grubbs [**?**], "An outlying observation, or outlier, is one that appears to deviate markedly from other members of the sample in which it occurs."

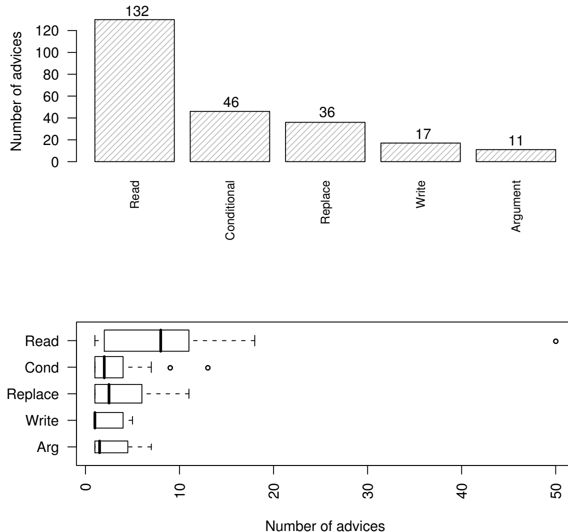

Figure 4: Bar-plot (top) of the cumulated *IAMR* value, and box-plot (bottom) of the *IAMR* value.

Figure **??**, 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 **??** 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.

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 [**?**], 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 [**?** **?**]. In next section we investigate if these few advices are widely spread through base programs.

5.2. 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 **??** presents the descriptive statistics for the advised join points ratio (*JMR*), and the Invasive advised join point ratio (*IJMR*), and Figure **??** presents a histogram comparing them (*JMR* in dark gray, *IJMR* in light gray).

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 PCD. The

	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 5: Descriptive statistics for *JMR* and *IJMR*.

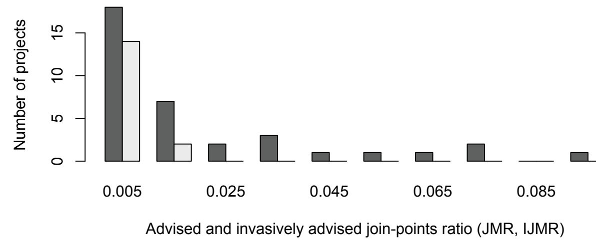

Figure 5: Histogram comparing the frequency of JMR and IJMR values.

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 **??** 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 *JMR* values and are not considered in Table **??**: 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 **??** 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 one or two points.

The relationship of *JMR* and the projects' size is illustrated by Figure **??**. In

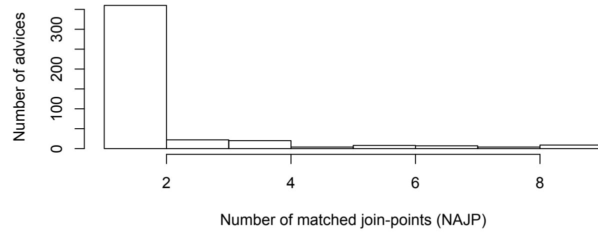

Figure 6: Histogram of NAJP frequency.

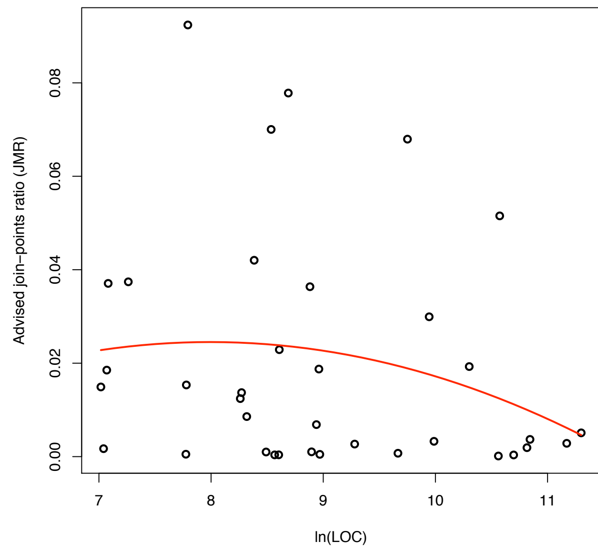

Figure 7: Scatter-plot illustrating the relationship between *JMR* and the projects' size.

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 advices very crosscutting, part of the 5% of advices advising between 80 and 2000 join points. These advices realize typical concerns [**?**] such as logging, debugging, and profiling among others.

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 [**? ?**].

- *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.

5.3. PCD usage (Q3)

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

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[8] 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*

[8]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.

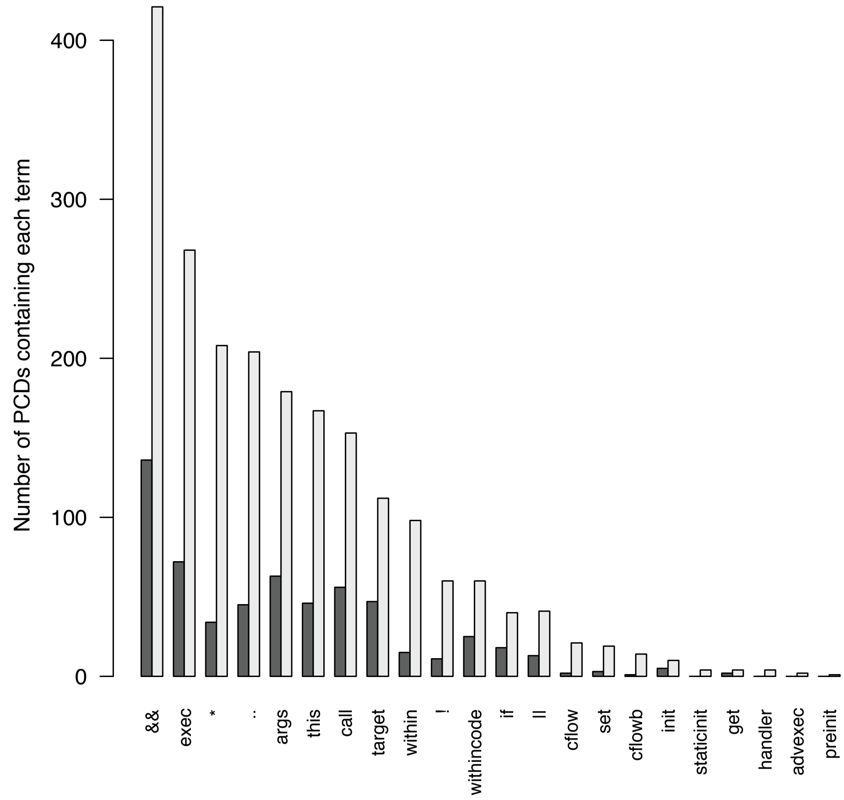

Figure 8: Bar-plot of the cumulated *NPCD* for all the projects, and the projects containing only invasive advices.

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 primitive PCDs *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* primitive PCDs 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 [**?**]. We explain this by the fact that in general developers want their advices to execute regardless the calling facility.

The usage of dynamic primitive PCDs is forked in two trends. The primitive PCDs *args*, *this*, and *target* are present in 20 to 35% of the PCDs, whereas *if*, *cflow*, and *cflowbelow* only in 4 to 8%. We explain this by the fact that the first group serves to capture data and specify types of the matched point, whereas the second is used 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 primitive PCDs 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 primitive PCDs 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 **??**.
- *Developers 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 join points*. This confirms the observations from previous section where we noticed that invasive advices crosscut a very small portion of the base program.

5.4. Discussing the usage of aspects

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*. When focusing only on invasive advices, we observe that *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*. 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*. In particular, `&&`, which restricts the number of matched join points, is the most commonly used construct; dynamic constructs such as `cflow`, which are difficult to precisely understand, are seldomly used.

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 [**?**].
- 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 [**?**].
- The AspectJ PCD language contains a large number of terms, but makes testing complex [**?**] and is paradoxically not very expressive [**?**].

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.

	Mean	Median	2.5%	97.5%	Min	Max	Total
Methods	-0.2	0	0	0	-4	0	-6
Attributes	0.042	0	0	0	-1	2	1
LoC	-14.792	-9,5	-12	-1.75	-101	0	-355
LCOM	0.021	0	0	0	-0.5	0.025	0.492

Table 6: Evolution of the number of Methods, number of attributes, LoC and LCOM, between the Java and AspectJ version of HealthWatcher (v1).

5.5. AspectJ influence on testability (Q4)

Previous work [**?**] has demonstrated a relationship between cohesion (LCOM metric), coupling (CBO, RFM) and testability. In this section we compare the evolution of coupling and cohesion metrics between the Java and AspectJ implementations of the HealthWatcher. Due to space limitations, we provide only the results for the first version of HealthWatcher, but we computed the same metrics for all three versions and observed similar trends. Data on the three versions is available online[9]

As a global observation, the number of modules increases in the AspectJ version, from 88 to 102. Table **??** synthesises the differences for the number of methods, attributes, lines of code and cohesion between the Java and AspectJ implementations. We compare only the modules that appear in both implementation and where at least one aspect is woven (which includes all modified modules). Consequently, we consider only 24 classes where advices are woven for analysis. The numbers given in table **??** correspond to the difference between the value measured on the AspectJ version minus the value on the Java version. Thus, a negative value indicates a decrease in the AspectJ version. The values are compared for each class, so for instance on average the number of lines of code has decreased by 14.792 per class in the AspectJ version.

The number of methods slightly decreases. One class has two methods less, and another class has four methods less. All other classes have the same number of methods. The number of attributes is almost the same. One class has two attributes more, another has one method less, and all the other classes have the same number of attributes.

The number of lines of code (LoC) clearly decreases. Almost 80% of the classes have a decreasing LoC, the others have the same LoC. Even the total number of LoC for the whole system decreases from 5 107 LoC to 4 744 LoC (more than 7% decrease), although 15 aspects have been added. In addition, the cohesion of modules is slightly improved in the AspectJ version. LCOM never increases and decreases in two cases. For one class LCOM decreases from 0.5 (medium cohesion) to 0 (perfect cohesion).

These initial results indicate that in the HealthWatcher example, AspectJ increases the number of modules while reducing the global size of the program.

[9] `http://www.romain-delamare.net/experiments/healthwatcher.xlsx`

	Mean	Median	2.5%	97.5%	Min	Max	Total
CBO	1.75	1	1	3	0	6	42
RFM	-1.33	-1	-2	0	-7	2	-32
RFM'	20.958	26	16	27.250	-1	31	503
MPC	-3.625	-7.5	-9	1	-32	34	-87

Table 7: Evolution of CBO, RFM, RFM', and MPC between the Java and AspectJ version of HealthWatcher (v1).

```
1  public class Library {
2     Collection<Book> books;
3
4     public void addBook(Book b) {
5        if (System.inMaintenance() || !AccessPolicy.
             isAuthorized())
6           throw ExceptionFactory.
                createUnauthorizedAccessException(this,b);
7        books.add(b);
8     }
9
10    public void removeBook(Book b) {
11       if (System.inMaintenance() || !AccessPolicy.
             isAuthorized())
12          throw ExceptionFactory.
                createUnauthorizedAccessException(this,b);
13       books.remove(b)
14    }
15 }
```

Listing 3: Example of a library implementation without aspects

As a consequence, all modules are smaller than in the Java version and tend to be more cohesive. This is consistent with AOP's promise to help isolate cross cutting concerns in separate cohesive modules. This is a necessary feature in order to improve the global modularity and testability of programs. Another important dimension for modularity is the coupling between the modules.

Table **??** shows statistics on the evolution of CBO, RFM, RFM', and MPC between the Java and the AspectJ implementations of the first version of HealthWatcher. Here we do not observe a homogeneous behaviour for all metrics. On one hand, CBO and RFM' increase in the AspectJ version. CBO is the same for one class, and increases for all the others. RFM' slightly decreases in one class and increases in all others. On the other hand, RFM and MPC globally decrease. In 20.8% of the classes, RFM increases, in 8.3% of the classes it is unchanged, and in 70.8% of the classes it decreases. In 37.5% of the classes MPC increases, and in 62.5% of the classes MPC decreases.

```
1  public class Library {
2     Collection<Book> books;
3
4     public void addBook(Book b) {
5        books.add(b);
6     }
7
8     public void removeBook(Book b) {
9        books.remove(b);
10    }
11 }
12
13 public aspect AccessPolicy {
14    before(): execution(void Library.*(Book)) {
15       Library l = getJoinpoint().getTarget();
16       if (System.inMaintenance() || !AccessPolicy.
            isAuthorized())
17          throw ExceptionFactory.
                createUnauthorizedAccessException(l,b);
18    }
19 }
```

Listing 4: Example of a library implementation with aspects

	Java	AspectJ
RFM	5	3
RFM'	x	$x+1$
MPC'	8	3

Table 8: Metrics of the two implementations of the library, in Listings **??** and **??**.

In the following we discuss an example to explain this apparent contradiction. Listing **??** shows an excerpt of a Java implementation of a Library. An access control checks if the user is authorized before adding or removing a book from the library, and throws an exception otherwise. The access policy concern is a cross-cutting concern. Listing **??** shows an implementation of the same functionality where an aspect encapsulates the access policy concern.

Table **??** shows the RFM, RFM', and MPC for the two implementations. RFM and MPC decrease in the AspectJ implementation because in each method, we have replaced three calls by the insertion of one advice. This decrease in the number of methods called directly by each module seems to indicate that modularity is improved in the AspectJ implementation. But eventually, the methods that were called are still executed. Thus, RFM' does not decrease, on the contrary it increases because one indirection level has been added. This explains why although RFM and MPC decrease, there is more coupling in the global system, and thus testability decreases.

5.6. Discussing testability of AspectJ aspects

When comparing Java and AspectJ implementations of three versions of the HealthWatcher, it appears that the number of modules slightly increases in the AspectJ version but that this is in favor of a smaller program as a whole, and more cohesive modules. On the other hand, these more cohesive modules also tend to be globally more coupled to each other. In addition to that, since AspectJ aspects need to be woven in a base program to be executed, they can not be tested in isolation. All these observations lead us to estimate that using AspectJ for AOP can hinder testability. As a consequence this can be an explanation why developers have been slowly adopting AOP through AspectJ. Indeed, if one technology improves the consistency of modules but also introduces new modules that are difficult to test and which interactions with the rest of the program are difficult to understand, it might be difficult to adopt this technology.

6. 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.

In the first part of the empirical study, a construction threat lies in the way we define our metrics and build the measurement tools. The relevance and accuracy of measurements depends on the capacity of the AspectJ compiler to statically detect join points and on the ability of ABIS and the AJAnalyzer to extract information correctly from the AST. It is also possible that our metrics result are too coarse grained to draw pertinent conclusions about the usage of AOP. For the second part of the study a construction threat lies in our choice of metrics to evaluate testability. Testability is related to the testing effort, but here we do not measure this effort directly. Instead, we measure the size

Primitive PCDs	# of occurrences
*	19
&&	13
execution	11
..	9
args	9
!	3
this	3
call	2
within	1

Table 9: Primitive PCDs and wildcards used in the HealthWatcher system

of modules, cohesion, and coupling under the assumption that if AOP has a negative impact on one of these factors, it has a negative impact on testability [? ?].

External threats lie on the statistical significance of our study. We acknowledge that we have only observed 38 open source projects. In addition, all aspects in these programs are expressed in AspectJ language only. We do not know to what extent this can be generalized to: (1) other AspectJ-like AOP languages such as CaesarJ [?]; (2) industrial projects under closed development; (3) systems developed in frameworks such as Spring AOP that tightly controls the use of AspectJ.

Another external threat is the representativeness of the HealthWatcher system and how the results of the the comparison of the different implementations can be generalized. Table **??** shows the number of occurrences for the primitive PCDs and wildcards used in the first version of the HealthWatcher system. This shows the same trend as the projects studied in this paper (see Figure **??**). As shown on Table **??**, the HealthWatcher is a medium sized project. Bigger or smaller systems, or systems using different kind of primitive PCDs, could lead to different results.

A confounding variable lies 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 and thus have a different usage of aspects.

7. Related Work

7.1. Usage of AOP

Apel *et al.*[?] study the usage of aspects in 11 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.* [**?**] 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 Lopez-Herrejon *et al.*'s 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.

Rashid *et al.* [**?**] have conducted a survey of eight industrial and academic AspectJ projects. They conclude that production projects have a basic usage of AOP for well-known pattern, which is consistent with our observation that developers are cautious with aspects. Rashid *et al.*'s survey also shows that AOP improves design stability and reduces the model size, and acknowledge the fragile pointcut problem.

7.2. Invasiveness-based aspect classifications

In this study we use a classification of aspects based on their invasiveness, introduced in our previous work [**?**]. Other classifications have been proposed in the past. Kienzle *et al.* [**?**] define four types of dependencies for aspects, orthogonal, uni-directional preserving, uni-directional modifying, and circular. In the invasive pattern we present, the *read* pattern has a uni-directional preserving depency while the others have a uni-directional modifying depency. Rinard *et al.* [**?**] propose categories of direct and indirect interactions between aspects and methods. Direct interaction is whether an advice interferes with the execution of a method, whereas indirect is whether advices and methods may read-/write the same fields. This classification is similar to ours, however, it addresses a different dimension. We identify invasiveness patterns instead of direct/indirect interactions. Moreover, in our work the identification of invasive patterns is only a portion of a whole specification framework. Katz [**?**] characterizes aspects among Spectative, Regulatory and Invasive aspects according to their invasiveness. Djoko *et al.* [**?**] have proposed a classification of aspects with three main categories: observers, which do not modify the base program, aborters, which are observers that can abort the execution, and confiners, which ensure constraints on the base program. These classifications are similar to ours, however, our characterization is more fine grained.

7.3. Metrics definition

Zhao [**?**] has previously proposed a framework for measuring metrics on aspect-oriented programs. Though this framework is also presented as an extension of Briand *et al.*'s framework [**?**], it is different from the extension presented in this paper. Zhao uses AspectJ as a target for the framework, thus his framework models features of AspectJ that may differ in other languages.

For instance, classes and aspects (in the sense of AspectJ) are distinguished and pointcuts are explicitly modeled as members of aspects. As discussed in Section **??**, our framework is more general and can be applied to other languages than AspectJ.

Zhao also defines coupling metrics that are based on relationships between entities. Each metric is the number of a certain type of relation in the system (e.g., relations between attributes and classes or relations between advice and classes). In this paper, the coupling metrics defined are based on OO metrics previously introduced [**? ? ?**], which measure more complex properties, such as indirect coupling.

Ceccato and Tonella [**?**] have adapted the metrics of Chidamber and Kemerer [**?**] for aspect-oriented programs. The metrics presented are close to metrics we define in this paper, but only a textual description is given by the authors. In this paper we present a formal framework in which we define metrics that are language independent.

Bartolomei *et al.* [**?**] have presented a metrics framework for aspect-oriented programs that targets different languages, as AspectJ or CaesarJ. This work discuss criterion for defining metrics (e.g., granularity, direction, etc.). The presentation of the framework is short, and some choice are not discussed, such as modeling code or join points. Only one metric, RFM, is defined in the proposed framework. The framework we propose is more detailed and closer to the one presented by Briand *et al.*, and we define more metrics, which are all used in the study.

Burrows *et al.* [**?**] have introduced new aspect-oriented metrics. These metrics, defined using the criterion of Briand *et al.*[**?**], are all new and specific to aspect-oriented programs. The authors conducted a study that compared the different metrics on aspect-oriented programs. This study showed that fine-grained metrics are a better indicator of fault-proneness than coarse-grained metrics. In our study, we compared object-oriented implementations with aspect-oriented implementations, thus we could only used metrics that can be applied on both paradigms. That is why we used metrics defined for object-oriented systems, extended for aspect-oriented systems.

8. Conclusion

In the first part of this study, we analyzed the usage of AspectJ 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. In the second part of the experiment, we analyzed the influence of aspects over testability by observing three versions of a system developed using both Java and AspectJ separately.

Our observations indicate that developers use few advices, and that these advices are scarcely crosscutting. The usage tendencies of the AspectJ PCD

language supports this: *Developers use only half of the PCD language expressivity*. The evidence shows that *developers write few advices that break the object-oriented encapsulation*, and these *advices advise a small number of very specific join points*. Sullivan et al. [**?**] studied the effect of using interface to explicit join points in AspectJ. Their findings support our evidence that the over expressiveness of AspectJ's PCD language in addition with obliviousness ..

Furthermore, our observations regarding the effect of aspect over testability indicate that, aspects have a negative effect on testability. The experiment reveals that *the usage of aspects improves the decomposition of code in modules*, concerns are better grouped into coherent units. However, *the coupling between the aspects and the base program is also increased*. As a consequence of this, modules cannot be tested as a single units.

The facts we collected during this study show that AspectJ is difficult to understand and control, in particular it is difficult to test programs implemented with AspectJ. It is difficult for developers to get confidence whether their code works the way they want it to. As a consequence developers use aspect-orientation in a very limited fashion when developing with AspectJ. On the other hand we have also observed that modules are smaller and more cohesive in aspect-oriented implementations than in object-oriented ones. All these observations lead us to conclude that aspect-orientation is an interesting technology to separate concerns, particularly crosscutting concerns that are likely to introduce defects [**?**]. However, *AspectJ seems to be an inappropriate language to implement AOP*. Some alternatives to alleviate the AspectJ limitation are providing more tools to assist the development with AspectJ and better Aspect-Oriented analysis and testing tools.

Another interesting phenomenon we observed through our analysis is that only a few crosscutting concerns are systematically realized through aspects. Developers usually implement concerns such as *logging*, *authorization*, *authentication*, *persistence*, and *distributions* [**? ? ? ? ? ?**]. The implementation of these concerns can actually help improving properties such as coupling, complexity, and the number of lines of code [**?**]. This supports Steimann's predictions about the AOP usage [**? ?**] – aspects are few and used in a very precise way. This might also indicate a general trend: technical cross-cutting concerns can be correctly dealt with at the code level and other cross cutting concerns should be dealt with at higher levels of abstraction (*e.g.*, with aspect-oriented modeling techniques).

Eight years after the AOP was announced as a key technology, this study offers a current view of AOP through its usage with AspectJ. We hope this will help researchers and practitioners to think about the future development of aspect-oriented environments and languages, tools for testing and analyzing AO programs AOP, and supporting aspect-oriented development through the software development.

In future work we will study the usage of aspects in other AOP solutions (such as Spring AOP). This will verify whether or not the overall usage of aspects is similar to the one we presented here. We think that this could be the case since these languages restrain expressivity and increase comprehension.