Information and Software Technology 55 (2013) 252–266

Contents lists available at SciVerse ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Usage and testability of AOP: An empirical study of AspectJ

Freddy Munoz [a], Benoit Baudry [a,*], Romain Delamare [b], Yves Le Traon [c]

[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

ARTICLE INFO

Article history:
Received 4 May 2011
Received in revised form 31 July 2012
Accepted 8 August 2012
Available online 4 September 2012

Keywords:
Aspect-oriented programming
Metrics
Empirical analysis

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.

1. Introduction

Object-orientation (OO) pushes forward ideas such as *modularity*, *abstraction*, and *encapsulation* [34]. It promotes the separation of concerns as a cornerstone 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 [44]. 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 [25]. 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 com-

* Corresponding author.
E-mail addresses: freddy.munoz@inria.fr (F. Munoz), benoit.baudry@inria.fr (B. Baudry), rdelamare@cs.ua.edu (R. Delamare), yves.letraon@uni.lu (Y. Le Traon).

[1] A recent study [17] demonstrates that crosscutting concerns increase the proneness to errors in OO system.

0950-5849/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2012.08.003

mented on AOP and its benefits. In 2001 the MIT announced AOP as a key technology for the future 10 years [46]. 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.

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 [45,27]; (2) the ability of aspects to break the object-oriented encapsulation [2,35]. 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 [36]. 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 three 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 [9] 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.

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

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

This paper is structured as follows: Section 2 introduces the aspect-oriented programming concepts. Section 3 describes the theoretical framework and the tooling support backing our empirical study. Section 4 describes the experimental data and the research questions this study inquiries. Section 5 presents the analysis results for each research question. Section 7 discusses the related work. Section 8 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 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.

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 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`.

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*, *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 1) 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.

```
 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.