Where the Truth Lies: AOP and Its Impact on Software Modularity

Adam Przybyłek

University of Gdańsk, Department of Business Informatics,
Piaskowa 9, 81-824 Sopot, Poland
adam@univ.gda.pl

Abstract. Modularity is the single attribute of software that allows a program to be intellectually manageable [29]. The recipe for modularizing is to define a narrow interface, hide an implementation detail, keep low coupling and high cohesion. Over a decade ago, aspect-oriented programming (AOP) was proposed in the literature to "modularize the un-modularizable" [24]. Since then, aspect-oriented languages have been providing new abstraction and composition mechanisms to deal with concerns that could not be modularized because of the limited abstractions of the underlying programming language. This paper is a continuation of our earlier work [32] and further investigates AO software with regard to coupling and cohesion. We compare two versions (Java and AspectJ) of ten applications to review AOP within the context of software modularity. It turns out that the claim that "the software built in AOP is more modular than the software built in OOP" is a myth.

Keywords: AOP, modularization, separation of concerns.

1 Introduction

Modularity is a key concept that programmers wield in their struggle against the complexity of software systems. Modularization is the process of decomposing a system into logically cohesive and loosely-coupled modules that hide their implementation from each other and present services to the outside world through a well-defined interface [3, 28, 30, 43]. Cohesion is the "intramodular functional relatedness" and describes how tightly bound the internal elements of a module are to one another, whereas coupling is "the degree of interdependence between modules" [43]. Modularization makes it possible to reason about every module in isolation, such that when a small change in requirements occurs, it will be possible to go to one place in code to make the necessary modifications [11].

In practice, modularization corresponds with finding the right decomposition of a problem [13]. However, in traditional programming languages, no matter how well a software system is decomposed into modules, there will always be concerns (typically non-functional ones) whose code cuts across the chosen decomposition [27]. The implementations of these crosscutting concerns will necessarily be spread over different modules, which has a negative impact on maintainability and reusability.

D. Giannakopoulou and F. Orejas (Eds.): FASE 2011, LNCS 6603, pp. 447–461, 2011.

Such concerns are called crosscutting concerns. The presented problem is known as the "tyranny of the dominant decomposition" [41] and several techniques have been invented to overcome it. The most prominent among them is AOP [23].

2 Motivations and Goals

Whenever a new technology is proposed, it has to prove its superiority over existing competitors. AOP emergenced as a new paradigm to modularize the concerns whose implementations would otherwise have been spread throughout the whole application, because of the limited abstractions of the underlying programming languages. Since then, several studies [15, 16, 17, 19, 35, 36] have suggested that AOP is successful in modularizing crosscutting concerns. Unfortunately, these studies either wrongly identify modularization with the lexical SoC offered by AOP, or wrongly measure coupling in AO systems. Indeed, in our previous study [32] we adapted the CBO metric to AOP and investigated implementations of the 23 GoF design patterns. We found no cases in which an AO implementation was more modular than its OO counterpart. Since these results cannot be generalized beyond simple academic examples, we continue our earlier work and evaluate real-life systems.

3 Modularity Metrics

3.1 Measurement System

In order to compare software modularity between the OO and AO paradigm, we used the G-Q-M (Goal-Question-Metric) approach [2]. G-Q-M defines a measurement system on three levels (Fig. 1) starting with a goal. The goal is refined in questions that break down the issue into quantifiable components. Each question is associated with metrics that, when measured, will provide information to answer the question.
Our goal is to compare AO and OO systems with respect to software modularity from the viewpoint of the developer.

In our previous paper [32], we argued for measuring modularity with the help of coupling and cohesion. This pair of attributes was firstly suggested to measure software modularity by Yourdon & Constantine [43] as part of their structured design methodology and then it was adapted to OO methodology by Coad & Yourdon [12], Booch [3], and Meyer [28]. Also, several empirical studies [6, 7, 20, 31] confirm that improvements in coupling and cohesion are linked to improved modularity.

Despite coupling and cohesion having been concepts in software design for almost 50 years, we still do not have widely-accepted metrics for them. In our previous paper [32], we supported CBO (Coupling Between Object classes) and LCOM (Lack of Cohesion in Methods), adapted from the Chidamber & Kemerer (CK) metrics suite [10]. CBO is a count of the number of other modules to which a module is coupled. Two modules are coupled when methods declared in one module use methods or instance variables of the other module [10]. LCOM is the degree to which methods within a module are related to one another. It is measured as the number of pairs of methods working on different attributes minus pairs of methods working on at least one shared attribute (zero if negative).

CBO and LCOM complement each other, and because of their dual nature, they are useful only when analyzed together. Attempting to optimize a design with respect to CBO alone would trivially yield to a single giant module with no coupling. However, such an extreme solution can be avoided by considering also the antagonistic attribute LCOM (which would yield inadmissibly high values in the single-module case) [20].

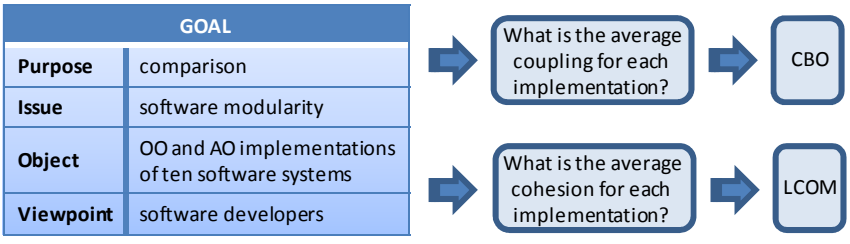

Fig. 1. Goal-Question-Metric

Since AOP provides new programming abstractions, existing OO measures cannot be directly applied to AO software. The efforts to make the CK metrics suite applicable to AO software were originated by Ceccato & Tonella [9] and Sant'Anna et al. [35]. Zhao [45] complemented their work by specifying the coupling dependencies in a formal way. Their general suggestion is to treat advices as methods and to consider introductions as members of the aspect that defines them. Although this suggestion is enough to adapt LCOM, the adjustment of CBO requires further explanation.

Coupling is a more complex attribute in AO systems, because new programming constructs introduce novel kinds of coupling relationships. We found that the existing coupling metrics [9, 35, 45] take into account only syntactic dependency. Syntactic dependency occurs when there is a direct reference between modules, such as inheritance or composition. However, in AO programs there is another kind of dependency that is not so easy to realize because it occurs without explicit references in the code. Ribeiro et al. [34] called this kind of coupling as semantic dependency. In our earlier work [32], we proposed a metric that considers this subtle kind of coupling. Our CBO metric considers a module M to be coupled to N if (in parentheses, we provide the abbreviations for the dependencies):

- M accesses attributes of N (A);
- M calls methods of N (M);
- M potentially captures messages to N (C);
- Messages to M are potentially captured by N (C_by);
- M declares an inter-type declaration for N (I);
- M is affected by an inter-type declaration declared in N (I_by);
- M uses pointcuts of N, excluding the case where N is an ancestor of M (P).

The C_by and I_by dependencies are semantic.

3.2 Rationale for Semantic Dependency

To construct our metric, we extrapolated the original CBO definition according to the question that underlies coupling: “How much of one module must be known in order to understand another module?” [43]. The syntactic dependencies (i.e. A, M, C, I, P) occurring in our metric do not raise any doubts even among proponents of AOP. Nevertheless, the debate on our previous paper [32], during and after the panel discussion at ENASE'10, demonstrated that further explanation of the semantic dependencies is required. It suffices to show that for understanding a given module M, we have to analyze N if the C_by or I_by dependency exists between M and N.

Suppose there are two modules as shown in Listing 1. Next, we send the inc(5) message to an instance of M. If we analyze M without considering the C_by dependency from M to N1, we will deduce (following program control flow) that the result is 6. However, the result is actually 11, and analyzing N1 is necessary to compute it correctly.

```
public class M {
  public int inc(int x) {
    return ++x;
  }
}
public aspect N1 {
  int around(int i):
   execution( int M.inc(int) ) && args(i) {
    return proceed(2*i);
  }
}
```

Listing 1. The C_by dependency

Now, suppose two new modules were added as shown in Listing 2. This time the same message is being sent to an instance of SubM. Once again, if we analyze M without considering the I_by dependency from SubM to N2, we will deduce an incorrect result. The correct is 20.

```
public aspect N2 {
  public int subM.inc(int x) {
    return x+10;
  }
}
public class SubM extends M {}
```

Listing 2. The I_by dependency

4 Empirical Study

4.1 Assessment Procedures

We compared two versions (Java and AspectJ) of 10 different systems: Telestrada, Pet Store, CVS Core, EImp, Health Watcher, JHotDraw, HyperCast, Prevayler,

Berkeley DB, and HyperSQL Database. To the best of our knowledge, these are all systems that have been implemented in both Java and AspectJ. They have also been widely used by other researchers to evaluate their work in the area of AOP.

The assessment of both versions bases on the application of metrics that quantify two fundamental modularity attributes, namely coupling and cohesion. In addition, we supplement our study by size metrics. Table 1 overviews the employed metrics and associates them with the attributes measured by each one of them. Detailed description of the coupling metric is provided in Sect. 3.1.

The gathering of data for the metrics was automated through the use of the extended version of AOPmetrics [37]. We extended AOPmetrics to support the CBO metric as defined in the previous section, except for capturing C and C_by (available at: http://przybylek.wzr.pl/AOP/). This is due to some inherent bugs in AOPmetrics [32]. Hence, we manually revised the CBO measures.

Table 1. Metric definitions

Attributes	Metrics	Definitions
Size	Vocabulary Size	Number of modules (classes, interfaces, and aspects) of the system
	Lines of Code	Number of lines in the text of the system's source code
Modularity	Coupling Between Object classes	Number of other modules to which a module is coupled
	Lack of Cohesion in Methods	Number of pairs of methods/advices working on different attributes minus pairs of methods working on at least one shared attribute (zero if negative)

4.2 Selected Systems

Our study uses ten real-world systems from different domains and of varying sizes. These are:

- Telestrada, which is a traveler information system being developed for a Brazilian national highway administrator. It allows its users to register and visualize information about Brazilian roads;
- Pet Store, which is a demo for the J2EE platform that is representative of existing e-commerce applications;
- CVS Core, which is an Eclipse Plugin that implements the basic functionalities of a CVS client, such as checkin and checkout of a system stored in a remote repository;
- EImp, which is an Eclipse Plugin that supports collaborative software development for distributed teams;
- Health Watcher, which is a web-based information system that was developed by Soares [36] for the healthcare bureau of the city of Recife, Brazil. The system aims to improve the quality of services provided by the healthcare institution, allowing

citizens to register complaints regarding health issues, and the healthcare institution to investigate and take the required actions. It involves a number of recurring concerns and technologies common in day-to-day software development, such as GUI, concurrency, RMI, Servlets and JDBC;

- JHotDraw (www.jhotdraw.org), which is a framework for technical and structured 2D graphics. Its design relies heavily on some well-known design patterns. JHotDraw's original authors were Gamma & Eggenschwiler;
- HyperCast (www.hypercast.org), which is software for developing protocols and application programs for application-layer overlay networks. It supports a variety of overlay protocols, delivery semantics and security schemes, and has a monitor and control capability. It was developed at the University of Virginia in cooperation with the Microsoft Corporation;
- Prevayler (www.prevayler.org), which is an object persistence library for Java. It is an implementation of the Prevalent System design pattern, in which business objects are kept live in memory and transactions are journaled for system recovery. Business object must be serializable, i.e., implement the java.io.Serializable interface, and deterministic, i.e., given an input, the object's methods must always return the same output;
- Berkeley DB Java Edition (oracle.com/technology/products/berkeley-db), which is a database system that can be embedded in other applications as a fast transactional storage engine. It stores arbitrary key/data pairs as byte arrays and supports multiple data items for a single key. Berkeley DB provides the underlying storage and retrieval system of several LDAP servers, database systems and many other applications;
- HyperSQL Database (hsqldb.org), which implements a relational database management system. It offers a small and fast database engine which supports both in-memory and disk-based tables. HSQLDB is currently being used as a database and persistence engine in many projects, such as Mathematica and OpenOffice.

All of these systems were originally implemented in Java and, afterwards, were refactored using AspectJ, so that the code responsible for some crosscutting concerns was moved to aspects. In each case, code refactoring was done by proponents of AOP to present the benefits of AOP over OOP.

In the first four systems, aspects were used to implement exception handling [8], [16]. Exception-handling is known to be a global design issue that affects almost all system modules, mostly in an application-specific manner.

The next work (Health Watcher) [19], [36] goes beyond refactoring of exception handling including concerns such as data persistence, concurrency and distribution (basic remote access to system services using Java RMI). Both the OO and AO designs of the Health Watcher system were developed with modularity and changeability principles as main driving design criteria.

AJHotDraw (ajhotdraw.sourceforge.net) is an aspect-oriented refactoring of JHotDraw with regard to persistence, design policies contract enforcement and undo command. It was started to experiment with the feasibility of adopting aspect-oriented solutions in existing software and demonstrate the strategies proposed by research of the Software Evolution Research Lab of Delft University of Technology in the

Netherlands. The aims, objectives and experience of the AJHotDraw project are summarized by Marin et al. [26].

Sullivan et al. [40] encountered two types of development problems when refactoring logging and event notification in HyperCast. First, the tight coupling between aspects and method names prevented the development of aspects in parallel with primary code refactoring, because the aspects could only be developed after inspecting the core concerns. Second, they found cases where joinpoints were not accessible, because AspectJ supports specifying joinpoints at the method call level and data member level, but not at the if or switch statement level. Next, they re-implemented the base version using AspectJ and crosscutting interfaces (XPI). What distinguishes that particular release, is the lack of introductions used. In our experiment, we evaluate the improved version.

Prevayler was refactored using AspectJ and horizontal decomposition by Godil & Jacobsen [18]. The horizontal decomposition principles were proposed by Zhang & Jacobsen [44] to guide the AO refactoring and implementation of complex software systems. The refactored code includes persistence, transaction, query, and replication management [18].

Störzer et al. [38] refactored version 1.8.0 of HSQLDB (sourceforge.net/projects/ajhsqldb/). They started with an accepted catalog of well-known crosscutting concerns and then tried to find classes, methods or fields related to the respective concerns. They used manual semantics-guided code inspection supported by Feature Exploration and analysis tool to find a relevant crosscutting code. They discovered and refactored many standard crosscutting concerns, including Logging, Tracing, Exception Handling, Caching, Pooling and Authentication/Authorization. When becoming familiar with the source code, they also found some application specific aspects, for example trigger firing or checking constraints before certain operations are performed [39].

By analyzing the domain, manual, configuration parameters, and source code, Kästner et al. [22] identified many parts of Berkeley DB that represented increments in program functionality that were candidates to be refactored into features. These features are implicit in the original code. They vary from small caches to entire transaction or persistence subsystems. All identified features represent program functionality, as a user would select or deselect them when customizing a database system. From these features, they chose 38 and manually refactored one feature after another (wwwiti.cs.uni-magdeburg.de/iti_db/berkeley/). They used various OOP-to-AOP refactoring techniques, including Extract Introduction, Extract Beginning and Extract End, Extract Before/After Call, Extract Method, and Extract Pointcut [22].

4.3 Experimental Results

Table 2 shows the obtained results for both size metrics, vocabulary size (n) and LOC, and both modularity metrics, CBO and LCOM. For all the employed metrics, a lower value implies a better result. The fourth and fifth column presents the mean values of the measures, over all modules per system. Rows labeled ‘Δ’ indicate the percentage difference between the OO and AO implementations relative to each metric. A positive value means that the original version performs better, whereas a negative value indicates that the refactored version exhibits better results.

Table 2. Results for size, coupling and cohesion metrics

		n	LOC	CBO	LCOM
Telestrada	OO	233	3424	0,81	1,86
	AO	242(18)	3350	0,95	2,17
	Δ	4%	-2%	18%	16%
PetStore	OO	345	17798	2,32	20,63
	AO	382(37)	17914	2,76	20,19
	Δ	11%	1%	19%	-2%
CVS	OO	257	18876	5,76	71,31
	AO	261(4)	19423	higher	73,90
	Δ	2%	3%	x	4%
Elmp	OO	123	8708	1,84	1,53
	AO	126(3)	9041	higher	1,68
	Δ	2%	4%	x	10%
Health Watcher	OO	88	6096	3,19	9,24
	AO	103(12)	5768	4,20	7,63
	Δ	17%	-5%	32%	-17%
JHotDraw	OO	398	22724	3,57	75,04
	AO	438(31)	23167	3,66	65,70
	Δ	10%	2%	3%	-12%
Hypercast	OO	370	50492	3,31	67,24
	AO	391(7)	51207	3,42	67,00
	Δ	6%	1%	4%	-0,4%
Prevayler	OO	167	5043	1,87	9,31
	AO	168(55)	4179	2,56	7,01
	Δ	1%	-17%	37%	-25%
Berkeley DB	OO	340	41651	4,38	126,31
	AO	452(107)	38770	4,73	78,21
	Δ	33%	-7%	8%	-38%
HSQLDB	OO	402	80736	4,11	226,91
	AO	413(25)	76210	4,12	247,30
	Δ	3%	-6%	0,3%	9%

In the case of both Eclipse Plugins, their refactored code is not publicly available, so we based our analysis on the measurements carried out by Castor et al. [8]. However, since they do not consider all types of coupling, we cannot present the exact CBO values. We can only say that coupling is greater for the refactored systems.

Contradicting the general intuition that AOP makes programs smaller, the refactored versions are larger with regard to the LOC metric in half the cases. Nevertheless, the increase ranges between 1% and 4%. In the remaining half, differences are greater than 5% (except for Telestrada) in favor of AOP.

The average coupling between modules is significantly higher in most of the refactored versions. For the refactored versions of Prevayler and Health Watcher, it is more than 30% higher than for the corresponding OO releases. Only for HSQLDB, JHotDraw and HyperCast is the increase rather slight. The higher coupling is the result of introducing new constructs intrinsic for AOP. In a typical scenario during AO refactoring, the coupling generated by explicit method call is replaced by the coupling generated by implicit advice triggering. Since an implicit advice triggering is associated with C and C_by in our CBO metric, the overall coupling grows. In addition, Filho et al. found [16] that new coupling was introduced when exception-handler aspects had to capture contextual information from classes.

Although the obtained results were as expected due to the above presented theoretical considerations, they contradict the outcomes achieved in several earlier studies. The advocates of AOP claim that the refactored versions of Telestrada [8, 16], Pet Store [8, 16] , CVS Core Plugin [8, 16], EImp Plugin [8], Health Watcher [19, 36], and Prevayler [18] exhibit lower coupling. However, they take into account only a subset of the dependencies that generate coupling in AO systems. Hence, the coupling measured with their metrics is underestimated.

The Lack of Cohesion in Methods is the metric for which the impact of AOP has remained unclear. For the refactored versions of Berkeley DB, Prevayler, Health Watcher and JHotDraw, the average LCOM is respectively 38%, 25%, 17%, and 12% lower than for the corresponding original versions. On the other hand, the average LCOM grew by 16% in the refactored version of Telestrada, 10% in the EImp Plugin, and 9% in HSQLDB. A partial explanation for this increase is the large number of methods that were created to expose join points (e.g. try-catch blocks in loops, etc.) that AspectJ can capture [21]. As discussed in [8], these new methods are not part of the implementation of the exception-handling concern but a direct consequence of using aspects to implement this concern. The average LCOM varied (positively or negatively) by less than 4% in the refactored versions of the remaining systems.

It is worth mentioning that most researchers compare aggregate coupling and cohesion between an OO and AO version of the same system. Aggregate coupling (cohesion) for a system is calculated as the sum of coupling (cohesion) taken over all modules. Hence, it can be derived from Table 2 as multiplication of the average value by vocabulary size. It should be also noted that the original versions perform better with regard to the aggregate coupling and cohesion, since the measures of vocabulary size grew in all cases, due to the introduction of aspects. Nevertheless, aggregate coupling does not satisfy the second axiom of Fenton & Melton [14] for coupling measures. That axiom states that system coupling should be independent from the number of modules in the system. If a module is added and shows the same level of pairwise coupling as the already existing modules, then the coupling of the system remains constant.

The obtained results confirm our previous findings. In [32], we compared Java and AspectJ implementations of 23 GoF design patterns. There was no pattern whose AO implementations would exhibit lower coupling, while 22 patterns presented lower coupling in the original implementations. With regard to cohesion, the OO implementations were superior in 9 cases, while the AO ones in 6 cases. 8 patterns exhibited the same cohesion in both implementations.

5 Threats to Validity

There are a number of limitations of this study that are worth stating. Firstly, we narrow software modularity to cohesion and coupling, despite of many other factors assigned to it. Nevertheless, cohesion and coupling are the concepts that lie at the heart of software modularity and are considered as main factors related to the goodness of modularization [3, 6, 7, 20, 28, 31]. The causal effect of AOP on software modularity was demonstrated in our earlier study [33].

Secondly, we could be criticised for applying metrics that are theoretically flawed. Briand et al. demonstrate [4] that LCOM is neither normalized nor monotonic. Normalization is intended to allow for comparison between modules of different size. To avoid this anomaly we weighted LCOM by the number of methods. Monotonicity states that adding a method which shares an attribute with any other method of the same module, must not increase LCOM. If we drop the very rare case where the methods of a module do not reference any of the attributes, the monotonicity anomaly disappears. The other problem with LCOM is that it does not differentiate modules well [1]. This is partly due to the fact that LCOM is set to zero whenever there are more pairs of methods which use an attribute in common than pairs of methods which do not [4]. In addition, the presence of access methods artificially decreases this metric. Access methods typically reference only one attribute, namely the one they provide access to, therefore they increase the number of pairs of methods in the class that do not use attributes in common [4]. The CBO metric also indicates inherent weakness. Briand et al. illustrate [5] that merging two unconnected modules may affect the overall coupling. Nevertheless, CBO as well as LCOM are widely applied and have been validated in many empirical studies [1, 5], and [6].

Thirdly, the applied metrics address only one possible dimension of cohesion and coupling. Moreover, CBO implicitly assumes that all basic couples are of equal strength [20]. In addition, it takes a binary approach to coupling between modules: two modules are either coupled or not. Multiple connections to the same module are counted as one [5]. In our defence we would point out that the OO community has yet to arrive at a consensus about the appropriate measurement of coupling and cohesion. The interested reader is referred to [4, 5], and [20] where an extensive surveys have been presented.

Finally, we could be criticised for generalizing findings from AspectJ to AOP. In our defence, most of the claims about the superiority of the AO modularization have been made in the context of AspectJ. It also should be noted that AspectJ is the only production-ready general purpose AO language.

To conclude, we are well aware that CBO and LCOM suffer from several disadvantages. We also known that the modularity evaluated in our setting may differ from the real modularity. The reason is that, it is not yet clear neither how to best measure attributes such as coupling and cohesion, nor how to compare modularity between systems that were developed in different paradigms. Nevertheless, the cases investigated provide enough evidence to challenge the claim that AOP improves software modularity.

6 Related Work

There are few studies focusing on the quantitative evaluation of the AO modularization. Sant'Anna et al. [35] conducted a semi-controlled experiment to compare the use of an OO approach (based on design patterns) and an AO approach to implement Portalware (about 60 modules and over 1 KLOC), a multi-agent system. Portalware is a web-based environment that supports the development and management of Internet portals. The collected metrics show that the AO version incorporates modules with higher coupling and lower cohesion. Their coupling metric is broader than the original CBO in the sense that it additionally counts modules declared in formal parameters, return types, throws declarations and local variables. However, it is not complete, since it does not take into account either the semantic dependencies, or the dependency that occurs when an advice refers to a pointcut defined in other, non-ancestor module.

The same suite of metrics was used by Garcia et al. [17] to compare the AO and OO implementations of the Gang-of-Four design patterns. They performed two studies, one on the original implementations from Hannemann & Kiczales and the other on the implementations with introduced changes. These changes were introduced because the H&K implementations encompassed few participant classes to play pattern roles [17]. Garcia and his team concluded that "the use of aspects helped to improve the coupling and cohesion of some pattern implementations." However, such conclusion may be misleading, according to the metrics they collected. The measures before the application of the changes exhibit that only Composite and Mediator present lower coupling for the AO solutions. The implementations of Adapter and State have the same coupling in both paradigms. In the case of the other patterns, the OO solutions indicate lower coupling. The superiority of OO solutions decreased a little after the changes were introduced. Although the AO implementations of Observer, Chain of responsibility, State and Visitor became better with respect to coupling than their OO counterparts, there are still 16 patterns for which the OO implementations provide superior results. With regard to cohesion, the OO implementations were also superior in most cases. They analyzed the absolute (aggregate) values.

Other studies can be classified into 2 groups. In the first group [8, 16, 19, 25], new kinds of coupling introduced by pointcuts are not considered at all. In the second group [21, 42], the coupling introduced by a pointcut is considered only if a module is explicitly named by the pointcut expression.

Greenwood et al. [19] chose the Health Watcher system as the base for their study. Their evaluation focused upon ten releases of the system, which underwent a number of typical maintenance tasks, including: refactorings, functionality increments, extensions of abstract modules and more complex system evolutions. Some of the crosscutting concerns were "aspectized" from the first release, while others were modularized as new HW versions were released. They found that modularity was improved with AOP. The average "coupling" as well as cohesion were enhanced by 17% in the initial version, and by 23% and 21% in the 10^{th} release.

Madeyski & Szała [25] examined the impact of AOP on software development efficiency and design quality in the context of a web-based manuscript submission and a review system (about 80 modules and 4 KLOC). Three students took part in their study. Two of them developed the system (labeled as OO1 and OO2) using Java, whilst one implemented the system using AspectJ. The observed results show that the AO version is 24% better than the others with regard to average "coupling" and it is 60% (3%) better than OO1 (OO2) with regard to average cohesion.

Filho et al. [8, 16] refactored to AOP four systems: Telestrada, Pet Store, CVS, and EImp. The average "coupling" was decreased by 6%, 9%, and 1% for the first three systems and increased by 2% for the last system. Nevertheless, Filho et al. [18] were aware that their study missed some coupling dependencies introduced by AOP: "a closer examination on the code (...) reveals a subtle kind of coupling that is not captured by the employed metrics."

The Telestrada and Pet Store systems were also used by Hoffman & Eugster. In their study [21], they calculated two coupling metrics, namely CBM and CIM. However, since CBM and CIM are not simply additive, the results are difficult to interpret.

Tsang et al. [42] compared AO vs. OO solutions in the context of real time traffic simulator. They found that aspects improved modularity by reducing "coupling" and cohesion. They considered aspects coupled to classes only if the aspects explicitly named the classes. "For instance, if we have the joinpoint call(* *(..)), then the aspect is not coupled to any classes. However, if we have the joinpoint call(void Test.methodName(..)), then the aspect is coupled to Test." In the conclusion of their work, they recommend the use of wildcards to maximize modularity improvements. Following this reasoning, one could recommend to replace the previous pointcut by call(void Test.methodNam*(..)), where '*' instead of 'e' eliminates "coupling".

7 Summary

This paper presents an empirical study in which we compare OO and AO implementations of ten software systems with respect to modularity. The evaluation is performed using the CBO and LCOM metrics from the CK suite, which were adapted to AOP. We hope that this paper regenerates some discussion about the role of AOP in software development.

The contribution of our work are twofold. Firstly, we gave a rationale for our coupling metric. At the same time, we argued why the existing coupling metrics are invalid for evaluating AO systems – they do not take into account all the composition mechanisms offered by the underlying paradigm. Secondly, we found that there is no evidence that AOP promotes better modularity of software than OOP. The OO implementation of every system exhibits lower average coupling. With regard to average cohesion the OO implementations are superior in 4 cases, while the AO ones in 6 cases. As far as we know, this is the first presentation of empirical evidence to this effect on real-life systems.

References

1. Basili, V.R., Briand, L.C., Melo, W.L.: A Validation of Object-Oriented Design Metrics as Quality Indicators. IEEE Transactions on Software Engineering 22(10), 751–761 (1996)
2. Basili, V.R., Caldiera, G., Rombach, H.D.: Goal Question Metric Approach. In: Encyclopedia of Software Engineering, pp. 528–532. John Wiley & Sons, Inc., Chichester (1994)
3. Booch, G.: Object-oriented Analysis and Design with Applications. Benjamin-Cummings, Redwood City (1994)
4. Briand, L.C., Daly, J.W., Wüst, J.: A Unified Framework for Cohesion Measurement in Object-Oriented Systems. Empirical Softw. Engg. 3(1), 65–117 (1998)
5. Briand, L.C., Daly, J.W., Wüst, J.K.: A Unified Framework for Coupling Measurement in Object-Oriented Systems. IEEE Transactions on Software Engineering 25(1), 91–121 (1999)
6. Briand, L.C., Morasca, S., Basili, V.R.: Defining and Validating Measures for Object-Based High-Level Design. IEEE Trans. Softw. Eng. 25(5), 722–743 (1999)
7. Briand, L.C., Wüst, J.K., Lounis, H.: Replicated Case Studies for Investigating Quality Factors in Object-Oriented Designs. Empirical Software Eng 6(1), 11–58 (2001)
8. Castor, F., Cacho, N., Figueiredo, E., Garcia, A., Rubira, C.M., de Amorim, J.S., da Silva, H.O.: On the modularization and reuse of exception handling with aspects. Softw. Pract. Exper. 39(17), 1377–1417 (2009)
9. Ceccato, M., Tonella, P.: Measuring the Effects of Software Aspectization. In: 1st Workshop on Aspect Reverse Engineering, Delft, Netherlands (2004)
10. Chidamber, S.R., Kemerer, C.F.: A Metrics Suite for Object Oriented Design. IEEE Trans. Softw. Eng. 20(6), 476–493 (1994)
11. Cline, M., Lomow, G., Girou, M.: C++ FAQs. Addison-Wesley, Reading (1998)
12. Coad, P., Yourdon, E.: Object-Oriented Analysis. Prentice-Hall, Englewood Cliffs (1991)
13. De Win, B., Piessens, F., Joosen, W., Verhanneman, T.: On the importance of the separation-of-concerns principle in secure software engineering. In: ACSA Workshop on the Application of Engineering Principles to System Security Design, Boston, Massachusetts (2002)
14. Fenton, N., Melton, A.: Deriving Structurally Based Software Measures. J. Syst. Software 12, 177–187 (1990)
15. Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Castor Filho, F., Dantas, F.: Evolving software product lines with aspects: An empirical study on design stability. In: 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany (2008)
16. Filho, F.C., Cacho, N., Figueiredo, E., Maranhão, R., Garcia, A., Rubira, C.M.: Exceptions and aspects: the devil is in the details. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Portland, Oregon (2006)
17. Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., von Staa, A.: Modularizing design patterns with aspects: a quantitative study. In: Proceedings of the 4th international Conference on Aspect-Oriented Software Development (AOSD 2005), Chicago, Illinois (2005)
18. Godil, I., Jacobsen, H.: Horizontal decomposition of Prevayler. In: The 2005 Conference of the Centre For Advanced Studies on Collaborative Research, Toronto, Canada (2005)

19. Greenwood, P., Bartolomei, T., Figueiredo, E., Dósea, M., Garcia, A.F., Cacho, N., Sant'Anna, C., Soares, S., Borba, P., Kulesza, U., Rashid, A.: On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In: Bateni, M. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 176–200. Springer, Heidelberg (2007)
20. Hitz, M., Montazeri, B.: Measuring Coupling and Cohesion in Object-Oriented Systems. In: Proceedings of the 3rd International Symposium on Applied Corporate Computing, Monterrey, Mexico (1995)
21. Hoffman, K., Eugster, P.: Bridging Java and AspectJ through explicit join points. In: 5th international Symposium on Principles and Practice of Programming in Java (PPPJ 2007), Lisboa, Portugal (2007)
22. Kästner, C., Apel, S., Batory, D.: A Case Study Implementing Features using AspectJ. In: 11th International Conference of Software Product Line Conference (SPLC 2007), Kyoto, Japan (2007)
23. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Cristina Lopes, C., Loingtier, J., Irwin, J.: Aspect-Oriented Programming. In: Liu, Y., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
24. Lesiecki, N.: Improve modularity with aspect-oriented programming (2002), http://www.ibm.com/developerworks/library/j-aspectj/
25. Madeyski, L., Szała, Ł.: Impact of aspect-oriented programming on software development efficiency and design quality: an empirical study. IET Software Journal 1(5), 180–187 (2007)
26. Marin, M., Moonen, L., van Deursen, A.: An Integrated Crosscutting Concern Migration Strategy and its Application to JHotDraw. In: IEEE International Conference on Source Code Analysis and Manipulation (SCAM 2007), Paris, France (2007)
27. Mens, T., Mens, K., Tourwé, T.: Software Evolution and Aspect-Oriented Software Development, a cross-fertilisation. ERCIM special issue on Automated Software Engineering. Vienna, Austria (2004)
28. Meyer, B.: Object-oriented Software Construction. Prentice-Hall, Englewood Cliffs (1989)
29. Myers, G.J.: Composite/Structured Design. Van Nostrand Reinhold (1978)
30. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12), 1053–1058 (1972)
31. Ponnambalam, K.: Characterization and Selection of Good Object-Oriented Design. In: Workshop on OO Design at OOPSLA 1997, Atlanta, Georgia (1997)
32. Przybyłek, A.: An empirical assessment of the impact of AOP on software modularity. In: 5th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2010), Athens, Greece (2010)
33. Przybyłek, A.: What is wrong with AOP? In: 5th International Conference on Software and Data Technologies (ICSOFT 2010), Athens, Greece (2010)
34. Ribeiro, M., Dósea, M., Bonifácio, R., Neto, A.C., Borba, P., Soares, S.: Analyzing Class and Crosscutting Modularity with Design Structure Matrixes. In: Proceedings of the 21th Brazilian Symposium on Software Engineering (SBES 2007), João Pessoa, Brazil (2007)
35. Sant'Anna, C., Garcia, A., Chavez, C., Lucena, C., von Staa, A.: On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In: 17th Brazilian Symposium on Software Engineering (SEES 2003), Manaus, Brazil (2003)
36. Soares, S., Laureano, E., Borba, P.: Implementing Distribution and Persistence Aspects with Aspect J. In: 17th ACM conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington (2002)
37. Stochmiałek, M.: AOPmetrics, http://aopmetrics.tigris.org

38. Störzer, M., Eibauer, U., Schöffmann, S.: Aspect Mining for Aspect Refactoring: An Experience Report. In: Workshop on Towards Evaluation of Aspect Mining at ECOOP 2006, Nantes, France (2006)
39. Störzer, M.: Impact Analysis for AspectJ – A Critical Analysis and Tool-based Approach to AOP. PhD thesis, School of Computer Science and Mathematics, University of Passau, Germany (2007)
40. Sullivan, K., Griswold, W.G., Song, Y., Cai, Y., Shonle, M., Tewari, N., Rajan, H.: Information hiding interfaces for aspect-oriented design. In: 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lisbon, Portugal (2005)
41. Tarr, P., Ossher, H., Harrison, W., Sutton, S.M.: N degrees of separation: multi-dimensional separation of concerns. In: 21st International Conference on Software Engineering (ICSE 2009), Los Angeles, California (1999)
42. Tsang, S.L., Clarke, S., Baniassad, E.L.A.: An evaluation of aspect-oriented programming for java-based real-time systems development. In: 7th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2004), Vienna, Austria (2004)
43. Yourdon, E., Constantine, L.L.: Structured Design: Fundamentals of a Discipline of Computer Program and System Design. Prentice-Hall, Englewood Cliffs (1979)
44. Zhang, C., Jacobsen, H.: Resolving Feature Convolution in Middleware Systems. In: 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, Canada, pp. 188–205 (2004)
45. Zhao, J.: Measuring Coupling in Aspect-Oriented Systems. In: 10th International Software Metrics Symposium, Chicago, IL (2004)