AN EMPIRICAL ASSESSMENT OF THE IMPACT OF ASPECT-ORIENTED PROGRAMMING ON SOFTWARE MODULARITY

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

Keywords: aspect-oriented programming, AOP, separation of concerns, modular programming.

Abstract: The term "crosscutting concern" describes a piece of system that cannot be cleanly modularized because of the limited abstractions offered by the traditional programming paradigms. Symptoms of implementing crosscutting concerns in the languages like C, C# or Java are "code scattering" and "code tangling" that both degrade software modularity. Aspect-oriented programming (AOP) was proposed as a new paradigm to overcome these problems. Although it is known that AOP allows programmers to lexically separate crosscutting concerns, the impact of AOP on software modularity is not yet well investigated. This paper reports a quantitative study comparing Java and AspectJ implementations of the Gang-of-Four design patterns with respect to modularity.

1 INTRODUCTION

A software system can be seen as a set of modules. Each module implements a concern, or a part of a concern. A concern is a specific requirement or an interest which pertains to the system's development.

Kiczales et al. [18] found that the abstractions offered by the traditional programming paradigms (e.g. structured programming, OOP, functional programming) are insufficient to express some kind of concerns in a modular way. Concerns like logging, persistence, concurrency control, or failure recovery tend to be scattered and tangled throughout the system modules, what adversely affect software modularity. These concerns are known as crosscutting concerns because they cross-cut the system's basic functionality [27].

Efforts to deal with crosscutting concerns have resulted in aspect-oriented programming (AOP). AOP brings new abstraction such as an aspect, a joinpoint, a pointcut, an advice, an introduction, and a parent declaration [19].

An aspect is a module that implements the behaviour and structure of a crosscutting concern. It can, like a class, realize interfaces, extend classes and declare attributes and operations. In addition, it can extend other aspects and declare advices, pointcuts, introductions and parent declarations.

A joinpoint is an identifiable location in the program flow where the implementation of a crosscutting concern can be plugged in. Typical examples of joinpoints include a throw of an exception, a call to a method and an object instantiation.

A pointcut is a language construct designed to specify a set of join-points and obtain the context (e.g. the target object and the operation arguments) surrounding the join-points as well.

An advice is a method-like construct used to define an additional behaviour that has to be inserted at all joinpoint picked out by the associated pointcut. The body of an advice is the implementation of a crosscutting functionality. The advice is able to access values in the execution context of the pointcut. Depending on the type of advice, whether "before", "after" or "around," the body of an advice is executed before, after or in place of the selected joinpoints. An around advice may cancel the

captured call, may wrap it or may execute it with the changed context [27].

An introduction is used to crosscut the static-type structure of a given class. It allows a programmer to add attributes and methods to the class without having to modify it explicitly. The power of introduction comes from the introduction being able to add methods to the interface.

A parent declaration may change the class's super-class or add implemented interfaces by defining an extends/implements relationship.

2 MOTIVATIONS AND GOALS

It is often taken as a given that AOP improves modularity [19], [20], [25], [29], [13]. However, there is no empirical evidence to support this assumption. The aim of this research is to perform a metrics-based comparison among AO and OO software with respect to modularity.

To date, there has been relatively little work in the area of developing methodologies for assessing the modularity of AO systems and for providing a means of comparison between AO systems and their OO equivalents [32]. To support such comparison we propose an approach based on coupling and cohesion.

Furthermore, the advent of a new paradigm requires defining new metrics to measure software quality. Ceccato & Tonella [7] and Sant'Anna et al. [28] refined the Chidamber & Kemerer metrics suite [8] regarding the effects of AOP, but they took into account only a subset of the dependencies that exist in AO systems. Hence, the coupling measured with their metrics is underestimated. We intend to revise the coupling metrics introduced by Ceccato & Tonella [7] to support a fair comparison between OO and AO implementations.

3 RATIONALE FOR THE STUDY

The effect of a new paradigm on software modularity can be evaluated through empirical studies. This section performs a quantitative assessment of Java and AspectJ implementations for the 23 GoF design patterns. As several design patterns involve crosscutting concerns [14], their implementations are good candidates for subjects of the study. Prior studies [14], [11] have shown that the AO implementations separate some concerns that are tangled and scattered in the OO decomposition counterpart. Hannemann & Kiczales report [14] that "AspectJ implementations of the GoF design patterns show modularity improvements in 17 of 23 cases". However, these modularity improvements "are manifested in terms of better code locality, reusability, composability, and (un)pluggability". Moreover, much of their assessment is based on intuition and gut feelings, rather than empirical support.

In this paper, the implementations are compared with regard to coupling and cohesion. This pair of attributes was firstly suggested to measure software modularity by Yourdon & Constantine [33] as part of their structured design methodology and then advocated by other software engineers [6], [23], [22]. Also, several empirical studies [4], [5], [15], [26] have found that improvements in coupling and cohesion are linked to improved modularity.

4 MODULARITY METRICS

Despite cohesion and coupling being concepts in software design for almost 50 years, we still don't have widely-accepted metrics for them. However the most well-known and widely used for OO assessment are CBO (Coupling Between Object classes) and LCOM (Lack of Cohesion in Methods), defined by Chidamber & Kemerer in their metrics suite [8]. CBO is a count of the number of other modules to which a module is coupled. Two classes are coupled when methods declared in one class use methods or instance variables of the other class [8]. 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). Lack of cohesion implies a module should probably be split into two or more sub-modules. CBO and LCOM complement each other, and because of their dual nature [15], they are useful only when analyzed together.

Since AOP provides new programming constructs, existing OO measures cannot be directly applied to AO software. Ceccato & Tonella [7] and Sant'Anna et al. [28] generalized the CK metrics suite to be applied in a paradigm-independent way, supporting the generation of comparable results between OO and AO solutions. The 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. Ceccato & Tonella defined the following metrics to measure different kinds of coupling [7]:

- CMC (Coupling on Method Call) is a number of modules declaring methods that are possibly called by a given module;
- CFA (Coupling on Field Access) is a number of modules declaring fields that are accessed by a given module;
- CAE (Coupling on Advice Execution) is a number of aspects containing advices possibly triggered by the execution of operations in a given module;
- CIM (Coupling on Intercepted Modules) is a number of modules explicitly named in the pointcuts belonging to a given aspect;
- CDA (Crosscutting Degree of an Aspect) is a number of modules affected by the pointcuts and by the introductions in a given aspect.

Nevertheless, to make the coupling comparable between the two paradigms, an AO counterpart to the CBO metric must be defined. For this purpose, we extrapolate the original CK definition according to the question that underlies coupling: „How much of one module must be known in order to understand another module?" [33]. Our CBO metric considers a module M to be coupled to N if:

- 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).

This metric gives a view of the overall effort needed to understand the module. It is worth noting that the dependencies C_by and I_by are semantic. A dependency is semantic when it can be implied from the source code though is not directly expressed. Such kind of dependency makes maintenance a nightmare without a tool that warns about extensions to a certain piece of code [17].

The coupling metric that seems to be similar to ours is the one of Sant'Anna et al. [28]. Their 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 take into account neither the implicit dependencies, nor the dependency that occurs when an advice refers to a pointcut defined in other, non-ancestor module.

5 STUDY SETTING

This study uses implementations of the GoF design patterns made freely available (http://hannemann.pbworks.com/Design-Patterns) by Hannemann & Kiczales. For each pattern they created a small example that makes use of the pattern, and implemented the example in both Java and AspectJ. The AspectJ implementations are considered as „one of the nearest things to examples of good AOP style and design" [24]. The Java implementations correspond to the sample C++ implementations in the GoF book.

In the measurement process, the data was gathered by the AOPmetrics tool [31]. To the best of our knowledge AOPmetrics is the sole tool to compute coupling metrics for AO systems. This tool implements the metric suite proposed by Ceccato & Tonella [7]. Although several researchers have used AOPmetrics [16], [21], they are not aware of its bugs. First of all, the implementation of CDA metric does not count modules coupled by pointcuts. A relationship between pointcut and the matched modules is typed by org.aspectj.weaver.AsmRelationshipProvider and marked by the MATCHES_DECLARE constant. The reverse relationship is marked by MATCHED_BY. However, the code-walker used by AOPmetrics does not count these relationships. Secondly, the values displayed as CDA should be displayed as CAE and vice versa. To fix this bug it is enough to swap the lines 107 and 109 in AspectualAffectedAndEffectedCalculator.java. We extended AOPmetrics to support the CBO metric as defined in the previous Section except for capturing the coupling introduced by pointcuts (the extended version is available at: http://przybylek.wzr.pl/AOP/). This is due to the inherent nature of the implementation as mentioned above. Hence, we manually revised the CBO values.

6 RESULTS

The CBO and LCOM (LCO in AOPmetrics) values were collected for each of the 128 modules across the OO implementations and 179 modules across the AO implementations. Table 1 presents the mean values of the metrics, over all modules per pattern. The lower numbers are better. The sixth and seventh column indicates the superior implementation with regard to the CBO and LCOM metric, respectively. The last column indicates the winning implementation. An implementation is

winning if it is better in at least one metric and not worse in the other. According to the last column, the patterns can be classified into two groups. Group 1 represents 16 patterns at which the OO implementations are better. All 7 remaining patterns belong to the second group. In this group, each implementation is superior with regard to one metric but inferior with regard to the other (except Adapter, which has the same metrics for both implementations). To our surprise, there is no pattern whose AO implementation exhibits better modularity.

The linear Pearson's correlation between CBO and LCOM on a class-by-class level is weak. It amounts to -0,04 for the OO implementations and 0,06 for the AO implementations. It means that the analyzed metrics do not capture redundant information.

For a further analysis of the effects of AOP, we break the results for this paradigm in two parts: (I) core concerns, and (II) crosscutting concerns (Table 2). Metrics in each part are calculated as arithmetic means taken over (I) all modules that implement the core concerns for a given pattern (it means all interfaces and classes except the Main class); (II) all aspects that comprise the pattern. Metrics in the first part reflect the modularization of core concerns, while metrics in the second part reflect the modularization of crosscutting concerns. The contribution of each part in the overall coupling and cohesion is shown as a percentage. In order to make a fair comparison between the two paradigms, Main classes were also excluded from the OO implementations. It is worth noting that in the AO versions most of the "badness" is generally accumulated within aspects. When comparing the CBO values for classes and interfaces only, the AO implementations are better in 4 cases and worse in 10 out of 23.

Table 1: Modularity metrics computed as arithmetic means.

	OOP		AOP		winner		
	CBO	LCOM	CBO	LCOM	CBO	LCOM	overall
Builder	0,75	2	1,8	2,2	OO	OO	OO
Command	0,71	0,14	1,58	2,67	OO	OO	OO
Iterator	0,75	0,25	1,4	1,8	OO	OO	OO
Mediator	0,86	0,14	1,13	0,5	OO	OO	OO
Proxy	1,2	0	1,38	0,13	OO	OO	OO
Chain	1,38	0,25	1,58	1,08	OO	OO	OO
Memento	0,67	0	0,75	0,5	OO	OO	OO
State	1,57	0,14	1,86	0,43	OO	OO	OO
Flyweight	0,8	0	0,86	0,14	OO	OO	OO
FactoryMethod	0,5	0	1,38	0	OO	-	OO
Facade	0,8	0	1,83	0	OO	-	OO
Strategy	0,8	0	1,67	0	OO	-	OO
Bridge	0,71	0	1,38	0	OO	-	OO
Composite	0,75	4	1,42	4	OO	-	OO
TemplateMethod	0,75	0	1	0	OO	-	OO
Decorator	1,17	0	1,25	0	OO	-	OO
Prototype	0,67	0,67	2,33	0	OO	AO	-
Singleton	0,67	0,33	1,33	0	OO	AO	-
Observer	1	5,60	1,67	2,11	OO	AO	-
Interpreter	1,56	0,11	2,4	0	OO	AO	-
AbstractFactory	0,9	0,1	1,18	0,09	OO	AO	-
Visitor	1,71	0,71	1,92	0,17	OO	AO	-
Adapter	1	0	1	0	-	-	-

Table 2: Modularity metrics – a detailed view.

	OOP		AOP							
			core			crosscutting			overall	
	CBO	LCOM	%	CBO	LCOM	%	CBO	LCOM	CBO	LCOM
Builder	0	2,67	75%	1,00	3,67	25%	3,00	0	1,50	2,75
Command	0,33	0,17	82%	0,89	0	18%	3,00	16,00	1,27	2,88
Iterator	0,67	0,33	75%	0,67	0,33	25%	3,00	8,00	1,25	2,25
Mediator	0,67	0,17	71%	0,60	0	29%	1,50	0,67	0,86	0,19
Proxy	0,50	0	43%	1,00	0	57%	1,75	0,25	1,43	0,14
Chain	1,14	0,29	82%	0,89	0	18%	3,5	6,5	1,36	1,17
Memento	0,50	0	71%	0,20	0	29%	1,50	2,00	0,58	0,58
Flyweight	0,50	0	67%	0,50	0	33%	1,00	0,50	0,67	0,17
FactoryMethod	0	0	71%	0,80	0	29%	2,00	0	1,15	0
Facade	0,75	0	80%	1,50	0	20%	3,00	0	1,80	0
Strategy	0,25	0	60%	1,00	0	40%	1,50	0	1,20	0
Bridge	0,17	0	86%	0,50	0	14%	4,00	0	0,99	0
Adapter	0,33	0	67%	0,5	0	33%	1,00	0	0,67	0
State	1,67	0,17	83%	1,40	0,6	17%	5,00	0	2,01	0,50
TemplateMethod	0	0	75%	0,33	0	25%	1,00	0	0,50	0
Decorator	0,60	0	33%	2,00	0	67%	1,00	0	1,33	0
Prototype	0	1,00	60%	1,67	0	40%	3,00	0	2,20	0
Singleton	0	0,50	60%	0,67	0	40%	2,00	0	1,20	0
Observer	0,75	7,00	50%	1,25	2,25	50%	1,25	2,50	1,25	2,38
Interpreter	0,88	0,13	89%	1,38	0	11%	6,00	0	1,89	0
AbstractFactory	0,67	0,11	90%	0,89	0,11	10%	2,00	0	1,00	0,10
Visitor	1,33	0,83	82%	1,33	0,22	18%	3,50	0	1,72	0,18
Composite	0	5,33	82%	0,78	0	18%	3,50	14,00	1,27	2,52

The problem with the arithmetic mean is that each of the modules contributes equally to the final result. Intuitively, modules which are more complex should contribute more. In addition, LCOM is not normalized, which means that the cohesion measures of different modules (as they all have different numbers of methods and attributes) should not be compared. Thus, weighted arithmetic means were also calculated. The individual CBO and LCOM values are weighted by the number of methods defined in the module, plus one. Table 3 presents the averages calculated in this way. As it turns out, no pattern changes its group.

7 DEEPER INSIGHT INTO MODULARITY

An established technique for analysing the dependencies among the modules of a system is Dependency Structure Matrix (DSM). A DSM is a square matrix in which the columns and rows are labelled with modules and a non-empty cell models that the module on the row depends on the module on the column. The type of dependency is represented by the value of the cell (the shortcuts are introduced in Section 4). The CBO metric for a module can be calculated from a DSM by counting non-empty cells in the row. To provide complex insight into modularity, LCOM for each module is also presented. The differences between modularity in the OO and AO implementations are shown on the Observer pattern. Figure 1 shows the dependency matrixes for this pattern.

The participants in the Observer pattern are subjects and observers. The subject is a data structure which changes over time (such as a point), and the observer (a screen) is an object whose own invariants depend on the state of the subject. The intention of the Observer pattern is to define a one-to-many dependency between a subject and multiple observers, so that when the subject changes its state, all its observers are notified [14].

Table 3: Modularity metrics computed as weighted arithmetic means.

	OOP		AOP		winner		
	CBO	LCOM	CBO	LCOM	CBO	LCOM	overall
Builder	0,53	2,35	1,48	3,67	OO	OO	OO
Command	0,89	0,28	2,06	9,7	OO	OO	OO
Iterator	0,74	0,26	1,48	2,44	OO	OO	OO
Mediator	1,00	0,26	1,15	1,00	OO	OO	OO
Proxy	0,9	0	1,52	0,14	OO	OO	OO
Chain	1,33	0,33	2,14	2,46	OO	OO	OO
Memento	0,64	0	1	0,91	OO	OO	OO
State	1,83	0,17	1,897	0,41	OO	OO	OO
Flyweight	0,67	0	0,94	0,18	OO	OO	OO
Composite	0,46	4,92	2,29	6,15	OO	OO	OO
FactoryMethod	0,33	0	1,43	0	OO	-	OO
Facade	1	0	1,8	0	OO	-	OO
Strategy	0,69	0,00	1,75	0	OO	-	OO
Bridge	0,5	0	1,22	0	OO	-	OO
TemplateMethod	0,4	0	0,75	0	OO	-	OO
Decorator	1,13	0	1,25	0	OO	-	OO
Prototype	0,33	0,83	2,53	0	OO	AO	-
Singleton	0,91	0,36	1,5	0	OO	AO	-
Observer	1,12	11,23	2,11	4,89	OO	AO	-
Interpreter	1,42	0,13	2,40	0	OO	AO	-
AbstractFactory	1,22	0,17	1,51	0,16	OO	AO	-
Visitor	1,64	0,92	2,24	0,24	OO	AO	-
Adapter	1	0	1	0	-	-	-

a) OO implementation

		1	2	3	4	5	CBO	LCOM
Main	1		M	M			2	0
Point	2				M		1	24
Screen	3				M	M	2	4
ChangeObserver	4						0	0
ChangeSubject	5						0	0

legend

class
interface
aspect

b) AO implementation

		1	2	3	4	5	6	7	8	9	CBO	LCOM
ColorObserver	1						C, I	I, M			2	0
CoordinateObserver	2						C, I	I, M			2	0
ScreenObserver	3							C, I, M			1	0
ObserverProtocol	4										0	10
Main	5	M	M	M			M	M			5	0
Point	6	C_by, I_by	C_by, I_by								2	9
Screen	7	I_by	I_by	C_by, I_by							3	0
Observer	8										0	0
Subject	9										0	0

Figure 1: DSMs for the Observer pattern.

In the OO implementation, the business logic and the pattern context are tangled within the participant classes. As a result, Point and Screen have a poor cohesion. Moreover, code for implementing the pattern is spread across all participants. In the AO implementation, all code pertaining to the relationship between observers and subjects is moved into aspects. Hence, the participant classes are entirely free of the pattern context, and as a consequence they are much more cohesive. In the OO version, a point directly informs its observers by sending a message to them. In the AO version, even though Point does not have any reference to its observers, the coupling has not disappeared. The coupling has changed its form from explicit method call to implicit join-points matching. Whenever a point changes its state, the relevant advice is triggered and the observers are notified. Since not all the dependencies between the modules are explicit, an AO programmer has to perform more efforts to get a mental model of the source code.

8 THREATS TO VALIDITY

There are a number of limitations of this study that are worth stating. Firstly, we could be criticised for narrowing the software modularity to cohesion and coupling. Although cohesion and coupling are considered as main factors related to the goodness of modularization [6], [23], [4], [5], [15], [26] other factors like obviousness, information hiding, and separation of concerns are also notable. However any reasonable model to estimate modularity has never been proposed and we do not make an effort to build ours. Instead, we assumed restrictive criteria to decide whether implementation in a given paradigm can be considered more modular: it must be better in at least one metric and not worse in the other.

Secondly, we could be criticised for applying metrics that are theoretically flawed. Briand et al. demonstrate [2] 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 [2]. 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 [2]. The CBO metric also indicates inherent weakness. Briand et al. illustrate [3] that merging two unconnected modules may affect the overall coupling. Nevertheless, CBO as well as LCOM are widely applied (even at NASA Goddard Space Flight Center) and have been validated in many empirical studies [1], [3], and [4].

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 [15]. 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 [3]. In our defence we would point out that even the OO community has yet to arrive at a consensus about the appropriate measurement of coupling and cohesion. The interested reader is referred to [15], [2], and [3] where an extensive surveys have been presented.

Fourthly, 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.

Finally, although „the GoF patterns effectively comprise a microcosm of many possible systems" [30], the conclusions obtained from our study are restricted to small-sized systems only. However, our experience indicates that in case of larger systems, when multiple advices apply to the same join point and when different aspects influence each other, modularity is even harder to achieve. The similar observation was reported by Kästner et al. [17]. We would also like to mention that the chosen sample favours AOP. This is due to the facts that: (1) a number of design patterns intensively involve crosscutting concerns [14]; and (2) recent studies have shown that OO constructs are not able to modularize these pattern-specific concerns and tend to lead to programs with poor modularity [11].

To conclude, we are mindful that the limited size of the examples restricts the extrapolation of our results. We are also 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 examples provide enough evidence to challenge the claim that AOP enables to achieve a better modularization.

9 RELATED WORK

There are a few studies focusing on the quantitative assessment of the AO modularization. However, the metrics of coupling they use are incomplete. It should be also noted that most researchers compare aggregate coupling and cohesion between an OO and AO version of the same systems. Aggregate coupling (cohesion) for a system is calculated as the sum of coupling (cohesion) taken over all modules. However, aggregate coupling (cohesion) says nothing about the goodness of modularization. In addition, aggregate coupling does not satisfy the second axiom of Fenton & Melton [9] for coupling measures. This 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 experiment closest to ours is the one conducted by Garcia et al. [11]. They also compared the AO and OO implementations of the Gang-of-Four patterns but in different settings. Firstly, they applied the metrics suite of Sant'Anna et al. [28]. Secondly, their results "represent the tally of metric values associated with all the classes and aspects for each pattern implementation", while our results represent the average of metric values. Thirdly, 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 encompass few participant classes to play pattern roles [11]. Garcia and his team conclude their study as follows: "We have found that most AO solutions improved the separation of pattern related concerns. In addition, we have found that: the use of aspects helped to improve the coupling and cohesion of some pattern implementations." However this 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 the both paradigms. In cases 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.

Sant'Anna et al. [28] 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.

The other studies either do not consider the coupling introduced by pointcuts at all [21], [10], [12] or consider it only if a module is explicitly named by the pointcut expression [32], [16]. No matter which of these two categories the study belongs to, the measured coupling is underestimated.

Greenwood et al. [12] chose the Health Watcher (HW) system (about 100 modules and over 5 KLOC) as the base for their study. HW is a web-based information system that was developed by Soares [29] for the healthcare bureau of the city of Recife, Brazil. It involves a number of recurring concerns and technologies common in day-to-day software development, such as GUI, persistence, concurrency, RMI, Servlets and JDBC. Both the OO and AO designs of the HW system were developed with modularity and changeability principles as main driving design criteria. Greenwood et al. found that "modularity" is improved with AOP; the average coupling as well as cohesion were enhanced by 17%.

Madeyski & Szała [21] 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 (labelled 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 then OO1 (OO2) with regard to average cohesion.

Filho et al. [10] investigated how metrics were affected in three real-world applications when exception handling was implemented using AspectJ instead of Java. The first application is a subset (224 modules) of Telestrada, a traveller information system being developed for a Brazilian national highway administrator. The second application is Pet Store (339 modules), a demo for the J2EE platform that is representative of existing e-commerce applications. The last application is CVS Core Plugin (257 modules), part of the basic distribution of the Eclipse platform. Filho and his team analyzed the aggregate values. After dividing these values by the number of modules, it turns out that the average "coupling" was decreased by 6%, 9%, and 1%. At the same time, the average cohesion was decreased by 3% for the second system and increased by 19% and 4% for the others. Filho et al. [10] are aware that their study does not consider the coupling introduced by pointcuts: "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 [16], 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. [32] 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 conclusion of their work, they recommend the use of wildcards to maximise modularity improvements. Following this reasoning, one could recommend to replace the previous pointcut by call(void Test.methodNam*(..)), where '*' instead of 'e' eliminates „coupling".

10 SUMMARY

This paper presents an empirical study in which we compare OO and AO implementations of the GoF patterns with respect to modularity. The evaluation is performed applying the CBO and LCOM metrics from the CK suite, which were adapted for use on AO systems.

The contribution of this research can be summarized as follows. Firstly, we defined a new metric for coupling. The existing metrics are invalid for evaluating coupling in AO systems, since they do not take into account semantic dependencies between the system modules. Our metric can be applied to OO as well as AO systems. Furthermore, we improved AOPmetrics whose objective is to collect metrics of Java and AspectJ source code.

Secondly, we demonstrated how to compare modularity between OO and AO implementations. We also gave several theoretical and intuitive arguments to support our approach.

Finally, we found that the claim that AOP promotes better modularity of software than OOP is a myth. There was no pattern whose AO implementation exhibited lower coupling, while 22 patterns presented lower coupling in the original version. The reason is that aspects are tightly connected with the affected classes. 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. As far as we know, this is the first presentation of empirical evidence to this effect. Although some empirical studies were undertaken in the context of AO modularity, none of them took into account all the significant dependencies. Hence, they favoured AOP. In our future work, we would like to perform further empirical evaluations on larger AO systems.

REFERENCES

1. Basili, V.R., Briand, L.C., Melo, W.L., 1996. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering vol. 22(10)*, pp. 751-761
2. Briand, L.C., Daly, J.W., Wüst, J., 1998. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Softw. Engg. vol. 3(1)*, pp. 65-117
3. Briand, L.C., Daly, J.W., Wüst, J.K., 1999. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering vol. 25(1)*, pp. 91-121
4. Briand, L.C., Morasca, S., Basili, V.R., 1999. Defining and Validating Measures for Object-Based High-Level Design. *IEEE Trans. Softw. Eng. vol. 25(5)*, pp. 722-743
5. Briand, L.C., Wüst, J.K., Lounis, H., 2001. Replicated Case Studies for Investigating Quality Factors in

Object-Oriented Designs. *Empirical Software Eng., vol. 6(1)*, pp. 11-58
6. Booch, G., 1994. *Object-oriented Analysis and Design with Applications*. Benjamin-Cummings, Redwood City, California
7. Ceccato, M., Tonella, P., 2004. Measuring the Effects of Software Aspectization. In: *1st Workshop on Aspect Reverse Engineering*, Delft, Netherlands
8. Chidamber, S.R., Kemerer, C.F., 1994. A Metrics Suite for Object Oriented Design. *IEEE Trans. Softw. Eng. 20, 6 (Jun. 1994)*, pp. 476-493
9. Fenton, N., Melton, A., 1990. Deriving Structurally Based Software Measures. *J. Syst. Software vol. 12*, pp. 177-187
10. Filho, F.C., Cacho, N., Figueiredo, E., Maranhão, R., Garcia, A., Rubira, C.M., 2006. Exceptions and aspects: the devil is in the details. In: *14th ACM SIGSOFT international Symposium on Foundations of Software Engineering*, Portland, Oregon
11. Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C., von Staa, A., 2005. Modularizing design patterns with aspects: a quantitative study. In: *4th international Conference on Aspect-Oriented Software Development (AOSD'05)*, Chicago, Illinois
12. Greenwood, P., Bartolomei, T.T., Figueiredo, E., Dósea, M., Garcia, A.F., Cacho, N., Sant'Anna, C., Soares, S., Borba, P., Kulesza, U., Rashid, A., 2007. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In: *21st European Conference on Object-Oriented Programming (ECOOP'07)*, Berlin, Germany
13. Guyomarc'h, J., Guéhéneuc, Y., 2005. On the Impact of Aspect-Oriented Programming on Object-Oriented Metrics. In: *Workshop on Quantitative Approaches in Object-Oriented Software Engineering at ECOOP'05*, Glasgow, UK
14. Hannemann, J., Kiczales, G., 2002. Design Pattern Implementation in Java and AspectJ. In: *17th Conference on Object-Oriented Programming Systems, Languages, and Applications*, Seattle
15. Hitz, M., Montazeri, B., 1995. Measuring Coupling and Cohesion in Object-Oriented Systems. In: *3rd International Symposium on Applied Corporate Computing*, Monterrey, Mexico
16. Hoffman, K., Eugster, P., 2007. Bridging Java and AspectJ through explicit join points. In: *5th international Symposium on Principles and Practice of Programming in Java*, Lisboa, Portugal
17. Kästner, C., Apel, S., Batory, D., 2007. A Case Study Implementing Features using AspectJ. In: *11th International Conference of Software Product Line Conference (SPLC'07)*, Kyoto, Japan
18. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, Ch., Lopes, C.V., Loingtier, J., Irwin, J., 1997. Aspect-Oriented Programming. *LNCS, vol. 1241*, pp. 220-242. Springer, New York
19. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G., 2001. An Overview of AspectJ. In: *15th European Conference on Object-Oriented Programming (ECOOP'01)*, Budapest, Hungary
20. Lesiecki, N., 2002. Improve modularity with aspect-oriented programming. *http://www.ibm.com/developerworks/library/j-aspectj/*
21. Madeyski, L., Szała, Ł., 2007. Impact of aspect-oriented programming on software development efficiency and design quality: an empirical study. *IET Software Journal, vol. 1(5)*, pp. 180–187
22. Mancoridis, S., Mitchell, B. S., Rorres, C., Chen, Y., Gansner, E.R., 1998. Using Automatic Clustering to Produce High-Level System Organizations of Source Code. In: *6th international Workshop on Program Comprehension (IWPC'98)*, Ischia, Italy
23. Meyer B., 1989. *Object-oriented Software Construction*, Prentice Hall
24. Monteiro, M.P., Fernandes, J.M., 2005. Towards a Catalog of Aspect-Oriented Refactorings. In: *4th international Conference on Aspect-Oriented Software Development (AOSD'05)*, Chicago, Illinois
25. Munoz, F., Baudry, B., Barais, O., 2008. A classification of invasive patterns in AOP. In: *24th IEEE International Conference on Software Maintenance (ICSM'08)*, Beijing, China
26. Ponnambalam, K., 1997. Characterization and Selection of Good Object-Oriented Design. In: *Workshop on OO Design at OOPSLA'97*, Atlanta, Georgia
27. Przybylek, A., 2007. Post Object-Oriented Paradigms in Software Development: a Comparative Analysis. In: *1st Workshop on Advances in Programming Languages at IMCSIT'07*, Wisła, Poland
28. Sant'Anna, C., Garcia, A., Chavez, C. Lucena, C., von Staa, A., 2003. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In: *17th Brazilian Symposium on Software Engineering (SEES'03)*, Manaus, Brazil
29. Soares, S., 2004. An Aspect-Oriented Implementation Method. *PhD thesis*, Federal University of Pernambuco, Brazil
30. Srivisut, K., Muenchaisri, P., 2006. Determining Threshold of Aspect-Oriented Software Metrics. In: *3rd International Joint Conference on Computer Science and Software Engineering*, Bangkok, Thailand
31. Stochmiałek, M., 2005. AOPmetrics. *http://aopmetrics.tigris.org*
32. Tsang, S.L., Clarke, S., Baniassad, E., 2004. Object metrics for aspect systems: Limiting empirical inference based on modularity. *Technical Report*, Trinity College Dublin
33. Yourdon, E., Constantine, L. L., 1979. *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*. Prentice-Hall, New York