# Impact of aspect-oriented programming on software modularity

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

This paper is presented with assistance from the John Jenkins Memorial Fund of the Reengineering Forum (REF).

*Abstract*— **Over a decade ago, aspect-oriented programming (AOP) was proposed in the literature to "modularize the un-modularizable". Nowadays, the aspect-oriented paradigm pervades all areas of software engineering. With its growing popularity, practitioners are beginning to wonder whether they should start looking into it. However, every new paradigm makes claims that managers want to hear. The aim of this PhD dissertation is to find out how much of what has been written about AOP is true and how much is hype.**

***AOP, separation of concerns, software modularity***

## I. Introduction

The evolution of software development techniques has been driven by the need to achieve a better separation of concerns (SoC). A concern is a specific requirement or an interest which pertains to the system's development. The term SoC was coined by Dijkstra [1974] and it refers to the ability to decompose and organize system into manageable modules, which have as little knowledge about the other modules of the system as possible. In practice, this principle actually corresponds with finding the right decomposition of a problem [De Win et al., 2002].

Concerns can be mapped easily to different modules, if they are functional in nature [Beltagui, 2003]. Such concerns are called core concerns. Kiczales et. al. [1997] found that many systems contain also other kind of concerns that are hard to factor out in traditional units of decomposition, i.e. functions or classes. Such concerns are called crosscutting concerns. When they are implemented using a traditional language, their code spreads throughout the system. The reason is that traditional languages provide only one dimension along which systems can be decomposed. This limitation is known as the "tyranny of the dominant decomposition" [Tarr et al., 1999] and it states that concerns which do not match to the dominant decomposition must be implemented together with core concerns.

Symptoms of implementing crosscutting concerns in procedural or OO languages are "code scattering" and "code tangling". Code tangling occurs when implementations of different concerns coexist within the same module. Code scattering occurs when similar pieces of the implementation of one concern appear in many modules.

Code tangling and scattering run into problems significant enough for practitioners to begin questioning clasical programming paradigms. To achieve more advanced separation of concerns these questioners proposed a number of approaches such as composition filters, subject-oriented programming, feature-oriented programming and aspect-oriented programming (AOP). The most prominent and recognizable of these is AOP.

AOP introduces a new unit of modularity to implement crosscutting concerns. Although AOP allows programmers to avoid the phenomena of code tangling and scattering it comes with its own set of problems. The distinguishing characteristic of AO languages is that they provide quantification and obliviousness [Filman & Friedman, 2000]. Quantification is the idea that one can write unitary and separate statements that have effect in many, non-local places in a program [Filman, 2001]. Obliviousness states that the places these quantifications apply do not have to be specifically prepared to receive these enhancements [Filman, 2001]. Quantification and obliviousness cause problems such as difficulties in modular reasoning and maintenance [Leavens & Clifton, 2007; Figueiredo et al., 2008]. Hence, AOP by preventing code tangling and scattering improves code quality in one area, and at the same time by introducing quantification and obliviousness decreases it in the other area. The question is whether the possible gains are worth the confusion it causes.

## II. Problem statement

Every new technology begins with naive euphoria – its inventor(s) are usually submersed in the ideas themselves; it is their immediate colleagues that experience most of the wild enthusiasm [Bezdek, 1993]. As a paradigm grows in strength and moves beyond the embryonic stage, a battle over its acceptance starts. In 2005, Steimann stated the question: "Does aspect orientation really have the substance necessary to found a new software development paradigm, or is it just another term to feed the old buzzword-permutation based research proposal and PhD thesis generator?" Paradigms gain their status because they are more successful than their competitors in solving a few problems that the group of practitioners has come to recognize as acute [Kuhn, 1962]. AOP emerged in 1997 [Kiczales et. al., 1997] as a paradigm to implement the concerns that cannot be modularized either in procedural programming or in OOP because of the limited abstractions of the underlying programming languages. Nowadays, with its growing popularity, practitioners are beginning to wonder whether they should start looking into it. Several studies [Figueiredo et. al., 2008; Filho et. al., 2006; Garcia et. al., 2005; Greenwood et. al., 2007; Sant'Anna et. al., 2003; Soares et. al., 2002] suggest that AOP is successful in modularizing crosscutting concerns. Unfortunately, these studies either are based on intuition and gut feelings, rather than scientific evidence; or wrongly identify modularization with the lexical SoC offered by AOP; or wrongly measure coupling in AO systems. This dissertation argues for the following thesis: **Paradoxically, the programming abstractions proposed by aspect-oriented programming actually harm software modularity, instead of improving it**.

## III. Research methods

The main research method we employ is Multiple Embedded Case Study [Yin, 2003]. A case study is applied as a comparative research strategy, comparing the results of using OOP to the results of using AOP. The units of analysis are all systems (according to our knowledge) to which there is a Java implementation and an AspectJ implementation available. Because every individual case involves the examination of two subunits of analysis (OO and AO implementation), our study is called embedded.

In our research, we also do experimentation using the quasi-controlled experiment method. A controlled experiment is a scientific investigation that takes place in a setting especially created by the researcher [Boudreau et al., 2001]. With this research method, the researcher manipulates one or more independent variables to measure their effect on one or more dependent variables [Basili et al., 1999]. In our case, the dependent variables are are reuse level and atomic changes. Each combination of values of the independent variables is a treatment. Our experiment has two treatments representing two paradigms that we compare, i.e. OOP and AOP.

The measurement systems used to perform our empirical studies were defined with the Goal Question Metric (GQM) approach [Basili et al., 1994]. GQM is a top-down approach to establish a goal-driven measurement system on three levels: conceptual level (goal), operational level (question), quantitative level (metric).

To conduct the supporting research (developing visual notation for AOP), we applied the action research method. In this method, the researcher attempts to solve a real-world problem while simultaneously studying the experience of solving the problem [Davison et al., 2004]. He becomes a part of the research - to be affected by and to affect the research. A precondition for action research is to have a problem owner willing to collaborate to both identify a problem, and engage in an effort to solve it.

## IV. Contributions

### A. *Evaluating the impact of AOP on software modularity*

In the scientific literature, AOP is often claimed to improve software modularity compared to OOP. However, this dissertation denies it on theoretical as well as empirical grounds. We have surveyed the criteria for software modularity and found that aspects violate the basic principles on which software engineering has depended for the last 50 years. In our theoretical discussion we argue that AOP: (1) promotes unstructured programming; (2) breaks information hiding; (3) leaves interfaces implicit; (4) makes modular reasoning difficult; (5) breaks the contract between a base module and its clients; (6) escalates coupling.

We have also examined the research in AOP that propose to reduce obliviousness as a trade-off for an increase in modularity. In these approaches, AOP loses its ability to add new features to the code without having to intrusively modify the code.

Moreover, we have conducted two empirical studies that confirm our theoretical findings. We have compared OO and AO implementations of the 23 GoF design patterns in the first study, and 10 real-life applications in the second. In no case the AO implementation evidenced lower coupling than its OO counterpart. We are likely the first who experimentally demonstrated this effect. The impact of AOP on cohesion remains unclear.

### B. *Developing a new metric to measure aspect coupling*

Since AOP provides new programming constructs compared to OOP, we have adapted the CBO metric from the Chidamber & Kemerer suite to be applied in a paradigm-independent way, supporting the generation of comparable results. We have also given a rationale for our metric. At the same time, we have argued that since the coupling metrics introduced by Ceccato & Tonella, Sant'Anna et al., and Zhao do not take into account the semantic dependencies between aspects and other modules, they are invalid for evaluating AO systems.

### C. *Exploring the posibilities of AOP in the context of software reuse and evolvability*

We have performed some initial experiments on software reuse and evolution. The lessons learned from this study are as follows. In a AO system, one cannot tell whether an extension to the base code is safe simply by examining the base program in isolation. All pointcuts referring to the base program need to be examined as well. In addition, when writting a pointcut definition a programer needs a global knowledge about the structure of the application. This is due to the fact that pointcuts try to define intended conceptual properties about the base program, based on structural properties of the program. In most cases, aspects cannot be made generic, because pointcuts as well as advices encompass information specific to a particular use, such as the classes involved, in the concrete aspect. As a result, aspects are highly dependent on other modules and their reusability is decreased. Futhermore, we have confirmed that the reusability of aspects is also hampered in cases where "join points seem to dynamically jump around", depending on the context certain code is called from.

### D. *Elaborating an extension to the UML metamodel for Aspect Oriented Modeling*

We present an approach to integrate aspect orientation with the current state-of-the-art in modelling languages. The elaborated metamodel (AoUML) enriches UML with constructs for visualizing aspects. Thus we have improved the traceability from design to implementation by reducing the semantic gap between these development phases. We use AoUML in other parts of our thesis.

### *E. Improving AO implementations of three Gang of Four design patterns*

Although design patterns are an established strategy, the emergence of AO languages has given new insight into their implementation. Using AOP in combination with generics and reflective programming we developed reusable implementation of the Decorator, Proxy, and Prototype design pattern that can be introduced to existing code in a non-intrusive way. An adventage of having reusable implementation is, that the programmer does not have to do the implementation in each application he is building. Moreover, the pattern can be plugged or unplugged depending on the presence or absence of aspects.

### *F. Developing the guidelines outlining what is acceptable usage of AOP*

AOP should be considered as a noninvasive technique to introduce new functionality without disturbing the existing code. Although aspects can easily (un)plug the additional functionality, we suggest to use them only as a last resort, because they violate fundamental software engineering principles and make the source code hard to understand. AOP is convenient when changes have to be made only for a moment. Therefore AOP is appropriate for implementing development concerns such as tracing, profiling and testing. It is also an ideal prototyping tool for exploring new requirements.

## V. Dissertation Outline

The thesis is composed of scientific papers that arose during the research process. Table 1 shows how the contributions of this dissertation are linked to the publications of this dissertation.

TABLE I. The relations of contributions to papers

| **contribution** | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| **paper** | P6, P7, P8 | P7, P8 | P9 | P3 | P4, P5 | P1, P2 |

### *A. Post object-oriented paradigms in software development: a comparative analysis (P1), IMCSIT'07*

This paper gives an introduction to the problem of implementing crosscutting concerns in OO languages. The limitations of OO languages are explained and illustrated by 3 scenerio of adapting software to new requirements. Then, the paper presents the state-of-the-art in implementing cross-cutting concerns. The basic concepts of AOP and CF's are explored and applied to the scenerios to avoid code scattering and code tangling. Finally, the paper provides guidelines outlining which paradigm is appropriate for which software development issue.

### *B. Beyond object-oriented software development (P2), http://sciyo.com/books/show/title/advances-in-computer-science-and-it*

This paper is an extension of the previous one. It provides an overview of the background material necessary to understand the dissertation and evaluate its contributions. It surveys SoC and modularization techniques from the days of structured programming to post-OO paradigms of today's academic research. It also explains the „tyranny of the dominant decomposition" problem and the limitations of mainstream languages associated with it. Then, it outlines new programming constructs offered by AOP and CFs to avoid these limitations. Finally, the paper discusses the lessons learned from developing software in the post-OO paradigms. It suggests areas of future work related to post-OO paradigms.

### *C. Separation of crosscutting concerns at the design level: An extension to the UML metamodel (P3), IMCSIT'08*

This paper focuses on the design phase. It presents a new modelling language named AoUML that we have elaborated to incorporate aspects into class diagram. AoUML is an extension to the UML metamodel. Its specification is described by using a similar style to that of the UML metamodel. First, an overview class diagram is introduced to show the constructs that was included in the extension and how these constructs are built up in terms of the standard UML constructs. Then, the semantics and syntax are described in detail using natural language. The practical applicability of AoUML is demonstrated by modelling three GoF design patterns.

### *D. The Decorator pattern revisited: an aspect-oriented solution (P4), BIR'08*

This paper examines whether the existing implementations of the Decorator pattern can be improved using AspectJ and generics. In the first part of the paper we review the solutions to the Decorator pattern presented by Hannemann & Kiczales and Borella. It turns out that their solutions have some limitations and imperfections. In the second part of the paper we develop a reusable implementation of the Decorator pattern that can be easily (un)plugged into code.

### *E. Design Patterns with AspectJ, generics, and reflective programming (P5), ICSOFT'10*

This paper is a continuation of the previous one and further explores the possibilities for improving implementations of the GoF design patterns. It was found that not only Decorator, but also Proxy can take advantage by using generics. In addition, reflective programming was employed for Prototype to provide a default implementation for cloning. In each case, the obtained implementation is highly reusable.

### *F. What is wrong with AOP? (P6), ICSOFT'10*

The aim of this paper is to discuss whether AOP makes software more modular. First, we briefly review the literature of software engineering related to modularity and SoC techniques. Next, we show that AOP is in conflict with the well-established principles of modularity that Parnas, Dijkstra and other greats of the past laid out and on which software engineering has depended for the last 50 years.

### *G. An empirical assessment of the impact of AOP on software modularity (P7), ENASE'10*

This paper presents a quantitative study in which we compare OO and AO implementations of the 23 GoF design patterns with respect to modularity. We advocate for measuring modularity with the help of coupling and cohesion. The evaluation is performed applying the CBO and LCOM metrics from the CK suite, which we adapt to measure AO software. We argue that the existing metrics are invalid for evaluating coupling in AO systems, since they do not take into account semantic dependencies between the system modules. We aslo describe some bugs in the tool that is widely used to evaluate AO software. We have fixed and extended this tool to collect our metrics. Finally, we present the results of our experiment. We found that there is no pattern whose AO implementations exhibits lower coupling, while 22 patterns present lower coupling in the OO implementations. With the help of Dependency Structure Matrix we analyze in detail the coupling dependencies between modules of the Observer pattern. With regard to cohesion the OO implementations are superior in 9 cases, while the AO ones in 6 cases. 8 patterns exhibit the same cohesion in both implementations.

### *H. Where the truth lies: AOP and its impact on software modularity (P8), FASE'11*

In this paper we continue our earlier work on software modularity. We compare two versions (Java and AspectJ) of 10 real-life systems. The obtained results confirm our previous findings. We found that there is no evidence that AOP promotes better modularity of software than OOP. The OO implementation of every system exhibits lower coupling. With regard to cohesion the OO implementations are superior in 4 cases, while the AO ones in 6 cases. We also further explain semantic dependencies in AO software to give a rationale for our coupling metric.

### *I. Systems evolution and software reuse in OOP and AOP (P9), Springer-Verlag CCIS'11*

This paper describes a laboratory quasi-experiment which compares the evolution of two functionally equivalent programs, developed in two different paradigms. Using the GQM approach we define a measurement system for the comparison of AO and OO software with respect to software evolvability and reusability. The experiment encompasses five maintenance scenarios on a classical producer-consumer example. The empirical results show that the advantage of AOP over OOP is doubtful from the software evolution point of view on one hand, and on the other hand that AOP has a potential to increase software reusability. Nevertheless, more industrial data needs to be investigated before more definitive conclusions can be drawn about the impact of AOP on software evolvability and reusability.

## References

[1] Avison, D.E., Baskerville, R., Myers, M.: Controlling action research projects. Information Technology & People, 14 (1), pp. 28-45, 2001

[2] Basili, V.R., Caldiera, G., Rombach, H.D.: The Goal Question Metric Approach. In: Encyclopedia of Software Engineering, pp. 528-532, John Wiley & Sons, Inc., New York, 1994

[3] Basili, V.R., Shull, F., Lanubile, F.: Building Knowledge through Families of Experiments. In: IEEE Transactions on Software Engineering, vol. 25(4), pp. 456-473, July 1999

[4] Beltagui, F.: Features and Aspects: Exploring feature-oriented and aspect-oriented programming interactions. Technical Report No: COMP-003-2003; Lancaster University, Lancaster, 2003

[5] Bezdek, J.C.: Fuzzy models – what are they, and why. IEEE Transactions on Fuzzy Systems, vol. 1(1), pp. 1-6, 1993

[6] Boudreau, M.C., Gefen, D., Straub, D.: Validation in IS Research: A State-of-the-Art Assessment. MIS Quart., vol. 25(1), pp. 1-16, 2001

[7] Clifton, C., Leavens, G.T.: Spectators and Assistants: Enabling Modular Aspect-Oriented Reasoning. Technical Report 02-10, Iowa State University, 2002

[8] Davison, R.M., Martinsons, M.G., Kock, N.: Principles of Canonical Action Research. Information Systems Journal 14(1), pp. 65-86, 2004

[9] Dijkstra, E.W.: On the role of scientific thought. Netherlands, 1974

[10] Figueiredo, et al.: Evolving software product lines with aspects: An empirical study on design stability. In: 30th Inter. Conf. on Software Engineering (ICSE'08), Leipzig, Germany, 2008

[11] Filho, et al.: Exceptions and aspects: the devil is in the details. In: 14th ACM SIGSOFT international Symposium on Foundations of Software Engineering, Portland, Oregon, 2006

[12] Filman, R.E.: What is AOP, revisited. In: Workshop on Multi-Dimensional Separation of Concerns at ECOOP'01, Maribor, Slovenia, 2001

[13] Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. In: Workshop on Advanced Separation of Concerns at OOPSLA'00, Minneapolis, MN, 2000

[14] Garcia, A., et al.: Modularizing design patterns with aspects: a quantitative study. In: 4th Inter. Conf. on Aspect-Oriented Software Development (AOSD'05), Chicago, IL, 2005

[15] Greenwood, P., et al.: On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In: 21st European Conf. on Object-Oriented Programming (ECOOP'07), Berlin, Germany, 2007

[16] Kiczales, et al.: Aspect-Oriented Programming. LNCS, vol. 1241, pp. 220-242. Springer, New York, 1997

[17] Kuhn, T.S.: The Structure of Scientific Revolutions. University of Chicago Press, 1962

[18] Leavens, G.T., Clifton, C.: Multiple concerns in aspect-oriented language design: a language engineering approach to balancing benefits, with examples. In: 5th Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT'07), Vancouver, Canada, 2007

[19] Soares, S., Laureano, E., Borba, P.: Implementing Distribution and Persistence Aspects with AspectJ. In: 17th ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02), Seattle, Washington, 2002

[20] Steimann, F.: Domain Models Are Aspect Free. In: 8th Inter. Conf. on Model Driven Engineering Languages and Systems (MoDELS'05), Montego Bay, Jamaica, 2005

[21] Tarr, P., Ossher, H., Harrison, W., Sutton, S.M.: N degrees of separation: multi-dimensional separation of concerns. In: 21st Inter. Conf. on Software Engineering (ICSE'99), Los Angeles, CA, 1999

[22] De Win, B., Piessens, F., Joosen, W., Verhanneman, T.: On the importance of the separation-of-concerns principle in secure software engineering. Katholieke Universiteit Leuven, 2002

[23] Yin, R.K.: Case Study Research: Design and Methods. Sage Publications, Inc., California, 2003