WHAT IS WRONG WITH AOP?

Adam Przybylek
Department of Business Informatics, University of Gdansk, Piaskowa 9, 81-824 Sopot, Poland

Keywords: Aspect-oriented Programming, AOP, Separation of Concerns, Modular Programming.

Abstract: Modularity is a key concept that programmers wield in their struggle against the complexity of software systems. The implementation of crosscutting concerns in a traditional programming language (e.g. C, C#, Java) results in software that is difficult to maintain and reuse. Although modules have taken many forms over the years from functions and procedures to classes, no form has been capable of expressing a crosscutting concern in a modular way. The latest decomposition unit to overcome this problem is an aspect promoted by aspect-oriented programming (AOP). The aim of this paper is to review AOP within the context of software modularity.

1 INTRODUCTION

The history of programming languages may be seen as a perennial quest for better separation of concerns (SoC). The term SoC was coined by Dijkstra (1974) and it means "focusing one's attention upon some aspect" to study it in isolation for the sake of its own consistency; it does not mean completely ignoring the other ones, but temporarily forgetting them to the extent that they are irrelevant for the current topic. In the context of systems development, this term refers to the ability to decompose and organize systems into manageable modules, which have as little knowledge about the other modules of the system as possible.

Programming languages provide mechanisms that allow the programmer to define modules, and then compose those modules in different ways to produce the overall system. However, Kiczales et al. found that sometimes some issues of the problem cannot be represented as first-class entities in the adopted language. The reason why such issues are hard to capture is that they cut across the system's basic functionality, so their implementation will be spread throughout other modules (Kiczales et al. 1997). Such issues are called crosscutting concerns.

The symptoms of implementing crosscutting concerns in a procedural or object-oriented (OO) language are "code scattering" and "code tangling". Code tangling occurs when implementations of different concerns coexist within the same module. Code scattering occurs when the same implementation of a crosscutting concern spreads necessarily through many modules. Code tangling and scattering are damaging to the software architecture.

Efforts to deal with the phenomena of code tangling and scattering have resulted in aspect-oriented programming (AOP). Although AOP introduces a new unit of modularity to implement crosscutting concerns, it comes with its own set of problems. The distinguishing characteristic of AO languages is that they provide quantification and obliviousness (Filman & Friedman 2001). Quantification is the idea that one can write an aspect that can affect arbitrarily many non-local places in a program (Steimann 2006). Obliviousness states that one cannot know whether the aspect code will execute by examining the body of the base code (Filman 2001). Quantification and obliviousness cause problems such as difficulties in reasoning or maintenance (Leavens & Clifton 2007).

Hence, AOP, by preventing code tangling and scattering, improves software quality in one area, and at the same time, by introducing quantification and obliviousness, decreases it in the other area. In spite of the widely-held belief in the positive impact of AOP on software modularity, it has never been thoroughly investigated (according to our knowledge). The key problem this paper addresses is reasoning about whether the superior SoC offered by AOP makes software more modular.

Przybylek A. (2010).
WHAT IS WRONG WITH AOP?.
In *Proceedings of the 5th International Conference on Software and Data Technologies*, pages 125-130
DOI: 10.5220/0002921601250130

2 BACKGROUND

One of the well-known mechanisms to handle complexity proposed in the early days of software engineering is modularization. Modularization is the process of decomposing a system into loosely-coupled modules that are more easily understandable, manageable, and hide their implementation from each other (Parnas 1972). A module consists of two parts: an interface and a module body (implementation). An separates information needed by a client from implementation details. It represents a boundary across which control flow and data are passed. A module body is the code that actually realizes the module responsibility. It hides the design decisions and should not be accessible from outside the module. A programmer should be able to understand the responsibility of a module without understanding the module's internal design (Parnas, Clements, Weiss 1984). The interface and implementation parts are also called public and private, respectively. The users of a module need to know only its public part (Riel 1996). An interface serves as a contract between a module and its clients. Such contract allows the programmer to change the implementation without interfering with the rest of the program, so long as the public interface remains the same (Riel 1996).

An interface as presented above is often termed provided interface. A module can also stipulate a so-called required interface, which is another module's provided interface. A required interface specifies the services that an element needs from some other modules in order to perform its function and fulfill its own obligations.

In practice, modularization corresponds with finding the right decomposition of a problem (Win et al. 2002). Parnas (1972) argues that the primary criteria for system modularization should focus on hiding critical design decisions (i.e. difficult design decisions or design decisions which are likely to change). Yourdon & Constantine (1979) suggest to decompose a system so that (1) highly interrelated parts of the system should be in the same module; (2) unrelated parts of the system should reside in different modules. Although the different modules of one system cannot be entirely independent of each other, as they have to cooperate and communicate to solve the larger problem, the design process should support as much independence as possible (Jalote 2005). Dahl, Dijkstra & Hoare (1972) explain that "Good decomposition means that each component may be programmed independently and revised with no, or reasonably few, implications for the rest of the system." Parnas (1972) enumerates the benefits expected of modularization: (1) managerial – development time should be shortened because separate groups would work on each module with little need for communication; (2) product flexibility – it should be possible to make drastic changes to one module without a need to change others; (3) comprehensibility – it should be possible to study the system one module at a time. The whole system can therefore be better designed because it is better understood. This comprehensibility is often termed "modular reasoning". Clifton & Leavens (2003) clarify that a language supports modular reasoning if the actions of a module M written in that language can be understood based solely on the code contained in M along with the signature and behavior of any modules referred to by M. A module M refers to N if M explicitly names N, if M is lexically nested within N, or if N is a standard module in a fixed location (such as Object in Java).

3 MOTIVATIONS AND GOALS

Whenever new paradigms are proposed, they must be carefully assessed, so that their scopes of appropriate applicability can be identified (Lopes & Bajracharya 2006). Such is the case with AOP.

In the research community, some conclusions about AOP are the exact opposites of each other. On the one hand, "many traditional programming language researchers believe that aspect-oriented (AO) programs are ticking time bombs, which, if widely deployed, are bound to cause the software industry irreparable harm" (Dantas & Walker 2006). The best known skeptic – Steimann – in his OOPSLA'06 paper, argues that AOP "works against the primary purposes of the two, namely independent development and understandability of programs" and concludes that while AOP was set up to modularize crosscutting concerns, its very nature breaks modularity. Furthermore, Steimann claims that "the number of useful aspects is not only finite, but also fairly small" (Steimann 2006). Other opponents of aspect orientation, by rephrasing Dijkstra (1974), suggest that "the quality of programmers is indirectly proportional to the amount of advice they use in their programs" (Constantinides, Scotinides & Störzer 2004).

On the other hand, aspects are presented as "modular units of crosscutting implementation":

– AOP is attractive because of its ability to

modularize crosscutting concerns (Wampler 2007);
- AOP lets programmers modularize concerns that are orthogonal to the main decomposition of a program (Clifton 2005);
- AOP improves modularity by encapsulating crosscutting concerns into aspects (Munoz, Baudry & Barais 2008);
- Aspects allow for modularization of crosscutting concerns (Pawlak et al. 2005);
- AOP allows software to reach a higher modularity (Soares 2004).

However, the existing studies indicate only that AOP provides a better lexical SoC, but fail to show that AOP improves modularity. The advocates of AOP wrongly identify lexical SoC with the meaning attributed to SoC by Dijkstra. Next, they conclude that if AOP enriches SoC techniques, then it enables better modularity. However, something is wrong with the above reasoning. The results of some research (Tourwe, Brichau & Gybels 2003; Kästner, Apel & Batory 2007; Figueiredo et al. 2008) indicate that AOP leads to software that is as hard, or perhaps even harder, to evolve and to reuse than was the case before. Yet, it is documented that the improved modularity translates into easier maintenance and code reuse. The aim of this paper is to discuss whether the aspect-oriented SoC makes software more modular.

4 ASPECT-ORIENTED SOC: A THEORETICAL DISCUSSION

4.1 AOP Promotes Unstructured Programming

Constantinides, Scotinides & Störzer (2004) show that AOP has some of the problems associated with the GoTo statement. In particular, it does not allow for creating a coordinate system for the programmer. Since an advice can plug into just about any point of execution of a program, one can never know the previous (or following) statement of any statement (Steimann 2006). An advice is even worse than GoTo as the GoTo statement transfers control flow to a visible label, while an advice does not. As a result, just looking at the source code of the base module is not enough to deduce a variable value – an advice might have changed it invisibly for the programmer. Constantinides et al. compare Advice to the ComeFrom statement, which was proposed as a way to avoid GoTo – of course only as a joke (Constantinides, Scotinides & Störzer 2004).

4.2 AOP Breaks Information Hiding

A well designed module hides its implementation details from other modules. Prior to AOP, public interfaces together with private implementations guaranteed that changing a module's implementation would not break other modules as long as the interface would be kept the same. Since AOP this is no longer true. A pointcut expression allows a programmer to ignore the provided interface of a base module by capturing calls to the private methods. Then changes to the implementation of the base module might crash the pointcut expression. Hence, aspects can break down as classes evolve. Aldrich (2005) tightens this problem by restricting quantification, in that internal communication events (e.g., private calls within a module) cannot be advised by external clients.

In addition, an aspect can access the private members of any module by using the privileged modifier. In turn, it leads to a globalization of the data contained in modules. Hence, the conclusion drawn by Wulf & Shaw (1973) — that in the presence of global variables a programmer needs "a detailed global knowledge of the program" — is therefore also true for the presence of aspects (Steimann 2006).

4.3 AOP Leaves Interfaces Implicit

Steimann (2006) tries to apply the idea of provided/required interfaces to AOP. On the one hand, the aspect provides a particular service through which it extends the base module; therefore it should specify the provided interface. However, the matching required interface of the base module remains implicit — the base module does not specify that it needs something. On the other hand, the base module provides a set of program elements, which are required by the aspect to perform its function. Although the aspect depends on these elements, the base module comes without an explicit counterpart interface specification: its provided interface is implicit. Seen either way, the base module specifies no interfaces that could be matched with those of its aspects (Steimann 2006). For the programmer of the base module, this means that everything accessible for aspects should be kept constant.

The efforts of introducing an explicit interface between aspects and base modules were originated by Gudmundson & Kiczales (G&K) and then

continued by Aldrich. Gudmundson & Kiczales (2001) that the signature (a name and parameterization) of a pointcut can convey the abstract responsibility captured by the pointcut definition. Such as, pointcuts provide a basis for a new kind of interface, which G&K call the pointcut interface. A pointcut interface consists of a collection of named pointcuts and is exported by the base module, which can be a class or a package. The pointcut definition is kept within the module that exports the interface, so anyone looking at the definition would also be looking at the implementation of the base module. By having the exported pointcut, the programmer is aware that the base module may be influenced by aspects. Preserving the pointcut interface guarantees that upgrades to the base module will not disturb the dependent aspects.

Aldrich (2005) introduces a new modularization unit - Open Module - that "is intended to be open to extension with advice but modular in that the implementation details of a module are hidden". His language allows programmers to export pointcuts from an open module. Next, an external advice can be applied to these exported pointcuts. Because an advice needs to query exported pointcuts in order to achieve its function, the pointcuts can be thought of as a provided interface, while its counterpart in the advice header as a required interface. In addition, all calls to interface methods from outside the open module can also be advised. This property is important because many aspects rely only on calls to interface methods, so exporting pointcuts for all of these calls would be cumbersome.

The main drawbacks of open modules are: (1) Explicitly exposing an interface pointcut means a loss of some obliviousness; (2) The programmer of the base module must anticipate that clients might be interested in the internal event; (3) The programmer has to hide out some implementation details of the designed module to make the module open for advising; (4) When pointcuts are defined within base modules, many join points that have to be advised in the same way cannot be captured by quantified pointcuts, e.g., using wild-card notations. A separate pointcut is required for each base module.

Leavens & Clifton (2007) introduce a required interface in the base module by explicitly naming the aspects that may affect the module behaviour. Then, aspects can only be applied to the modules that reference them. Explicit acceptance of an aspect can be expressed by an annotation.

Hoffman & Eugster (2007) extend AspectJ with explicit join points (EJPs). EJPs introduce a new type of join point, which is explicitly declared by the programmer within aspect, has a unique name and signature. Base code then explicitly references these join points where crosscutting concerns should apply. The idea of EJPs is to represent cross-cutting concerns via explicit interfaces that act as mediators between aspects and base code.

As was pointed out by Steimann (2006), both the above solutions not only make advice activation almost indistinguishable from guarded subroutine calling but also they re-introduce the scattering that AOP was to avoid. For instance, with tracing as a crosscutting concern, annotating every method whose execution is to be traced is just as annoying as adding the tracing code on site (Steimann 2006). The use of annotations violates the "obliviousness" property of AOP pointcuts, and has potential scaling problems. In addition, this technique is invasive for base modules and unfeasible in case base modules are third party components.

4.4 AOP Makes Modular Reasoning Difficult

Aspects are most effective when the code they advise is oblivious to their presence (Filman & Friedman 2001). In other words, aspects are effective when a programmer is not required to annotate the base code in any particular way (Dantas & Walker 2006). However, the obliviousness property of AO languages implies that a base module has no knowledge of which aspects modify it where or when (Steimann 2006). It conflicts with the ability to study the system one module at a time. The whole-program analysis is required to find all the aspects that might advise a given piece of code (Clifton & Leavens 2003; Clifton 2005). This presents difficulties for code understanding and maintenance.

In addition, tight coupling between pointcuts and the semantics of methods and classes makes it impossible to understand aspects without first understanding methods and classes (Walker, Zdancewic & Ligatti 2003). Such as, it is not longer possible to reason about modules in isolation.

A proposal to maintain modular reasoning was put forward by Clifton & Leavens (C&L) and then expanded on by Dantas & Walker (2006) and Rinard et al. (57). Clifton & Leavens (2002) suggest to separate aspects into two categories, assistants and spectators, which provide complementary features. Assistants have the full power of AspectJ's aspects, but to maintain modular reasoning it is required that assistants are explicitly accepted (see Section 4.3).

Spectators are constrained to not modify the behavior of the modules that they advise. In concrete terms, a spectator may only mutate the state that it owns and it must not change the control flow to or from an advised method. In addition to mutating the owned state, it seems reasonable to allow spectators to change accessible global state as well, since a Java module cannot rely on that state not changing during an invocation (modulo synchronization mechanisms). This allows modular reasoning without requiring spectators to be explicitly accepted (Clifton & Leavens 2002). Nevertheless, when problems arise, a programmer must examine both the base and relevant aspect code to identify a bug.

Rinard et al. (57) proposed a more sophisticated classification system for AO programs. This system characterizes two kinds of interactions: direct interactions between an advice and methods that it crosscuts and indirect interactions between an advice and methods that may access the same object fields. They also implemented an application that may help programmers understand the interactions in AO programs and indicate the problematic ones.

4.5 AOP Breaks the Contract between a Base Module and its Clients

In the presence of aspects, clients of a base module can no longer trust that the provided service meets its specification. Each service can be affected by an advice.

Dantas & Walker (2006) introduce the notion of harmless advice, which is similar to the notation of spectators (see Section 4.4). Unlike an ordinary advice, a harmless advice is not allowed to influence the final result of the base code. Therefore, programmers may ignore harmless advices when reasoning about the partial correctness properties of their programs. Although harmless advices are useful for many common crosscutting concerns including: logging, tracing, profiling, invariant checking and debugging, they limit the powerful of AOP.

Gudmundson & Kiczales (2001) propose a pointcut interface (see Section 4.3) to restore the contract between a module and its clients. A pointcut interface allows a module to be evolved independently of its clients so long as the contract is preserved.

4.6 AOP doesn't Decrease Coupling

The fact that we must know something about another module is a priori evidence of some degree of interconnection even if the form of the interconnection is not known (Yourdon & Constantine 1979). An aspect establishes a strong dependency between itself and the base module, although this dependency is invisible from the base module's side. If a change occurs in any base module, all aspects need to be reviewed whether they are still working. Most AO languages in use today are based on structural information about join points, such as naming conventions and package structure, rather than the logical patterns of the software (Wampler 2007). In the result, a change in the method signature captured by the pointcut invalidates this pointcut definition, as well as the associated advice. This phenomenon is called the fragile pointcut problem.

5 SUMMARY

Since its inception over a decade ago, AOP is still a controversial idea. The advocates of AOP still repeat that AOP improves modularity. If a myth is repeated often enough, people believe it is true. Thus, even some opponents of AOP fall into the trap of saying that "AOSD leads to applications that are better modularized" (Tourwe, Brichau & Gybels 2003). However, this paper denies it on theoretical grounds. The main finding from the discussion is that aspects violate the basic software engineering principles and thus degrade modularity.

Some researchers propose to reduce obliviousness in return for increased modularity. In these approaches, AOP loses its ability to add new features to the code without having to intrusively modify the code, hence promises about non-invasive extensions are no longer true.

Nevertheless, the author appreciates the contribution of AOP in the development of SoC techniques and believes that AOP indicated directions in which further research should be conducted.

REFERENCES

Aldrich J., 2005. Open Modules: Modular Reasoning about Advice. In: *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*, Glasgow, U.K.

Clifton, C., 2005. A design discipline and language features for modular reasoning in aspect-oriented programs. *Phd thesis*, Department of Computer Science, Iowa State University, U.S.A.

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

Clifton, C., Leavens, G. T., 2003. Obliviousness, Modular Reasoning, and the Behavioral Subtyping Analogy. In: *Software-engineering Properties of Languages for Aspect Technologies (SPLAT'03)*, Boston.

Constantinides, C., Scotinides, T., Störzer, M., 2004. AOP considered harmful. In: *1st European Interactive Workshop on Aspect Systems*, Berlin, Germany

Dahl, O. J., Dijkstra, E.W., Hoare, C. A., 1972. Structured Programming. *Academic Press Ltd.*

Dantas, D.S., Walker, D., 2006. Harmless advice. In: *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM*, pp. 383–396, New York.

De Win, B., Piessens, F., Joosen, W., Verhanneman, T., 2002. 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.

Dijkstra, E. W., 1974. On the role of scientific thought. Netherlands, *http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD447.html*

Figueiredo et al., 2008. Evolving software product lines with aspects: An empirical study on design stability. In: *30th International Conference on Software Engineering (ICSE'08)*, Leipzig, Germany.

Filman, R.E., 2001. What is AOP, revisited. In: *Workshop on Multi-Dimensional Separation of Concerns at ECOOP'01*, Budapest, Hungary

Filman, R. E., Friedman, D. P., 2001. Aspect-oriented programming is quantification and obliviousness. In: *Workshop on Advanced Separation of Concerns at OOPSLA'00*, Minneapolis, Minnesota

Gudmundson S., Kiczales, G., 2001. Addressing practical software development issues in AspectJ with a pointcut interface. In: *Workshop on Advanced Separation of Concerns of ECOOP'01*, Budapest, Hungary.

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 (PPPJ'07)*, Lisboa, Portugal

Jalote, P., 2005. *An Integrated Approach to Software Engineering*. Springer, New York.

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.

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.

Leavens, G. T., Clifton, C., 2007. 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.

Lopes , C. V., Bajracharya, S., 2006. An Analysis of Modularity in Aspect-Oriented Design. *Springer LNCS 3880 Transactions on Aspect-Oriented Software Development I*, pp. 1–35.

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.

Parnas, D. L., 1972. On the criteria to be used in decomposing systems into modules. *Communications of the ACM, vol. 15(12)*. ACM Press, New York, pp. 1053–1058.

Parnas, D. L., Clements, P. C., Weiss, D. M., 1984. The modular structure of complex systems. In: *Proceedings of the 7th International Conference on Software Engineering*, Orlando, Florida.

Pawlak, R., Pawlak, R., Seinturier, L., Retaillé, J.P., Younessi, H., 2005. *Foundations of AOP for J2EE Development*. Apress.

Riel, A. J., 1996. *Object-oriented Design Heuristics*. Addison-Wesley, Boston.

Soares, S., 2004. An Aspect-Oriented Implementation Method. *PhD thesis*, Federal University of Pernambuco, Brazil.

Steimann, F., 2006. The paradoxical success of aspect-oriented programming. SIGPLAN Not. 41, 10 (Oct. 2006), pp. 481–497.

Tourwe, T., Brichau, J., Gybels, K., 2003. On the Existence of the AOSD-Evolution Paradox. In: *Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT'03) at AOSD'03*, Boston, Massachusetts.

Walker, D., Zdancewic, S., Ligatti, J., 2003. A Theory of Aspects. In: *8th ACM SIGPLAN International Conference on Functional Programming*, Uppsala, Sweden.

Wampler, D., 2007. Noninvasiveness and Aspect-Oriented Design: Lessons from Object-Oriented Design Principles. In: *6th International Conference on Aspect-Oriented Software Development (AOSD'07)*, Vancouver, Canada.

Wulf, W., Shaw, M., 1973. Global variable considered harmful. *SIGPLAN Notices 8:2*, pp. 28–34.

Yourdon, E., Constantine, L.L., 1979. *Structured Design: Fundamentals of a Discipline of Computer Program and System Design*. Prentice-Hall, New York.