The Paradoxical Success of Aspect-Oriented Programming

Friedrich Steimann
Lehrgebiet Programmiersysteme
Fernuniversität in Hagen
D-58084 Hagen
steimann@acm.org

Abstract

Aspect-oriented programming is considered a promising new technology. As object-oriented programming did before, it is beginning to pervade all areas of software engineering. With its growing popularity, practitioners and academics alike are wondering whether they should start looking into it, or otherwise risk having missed an important development. The author of this essay finds that much of aspect-oriented programming's success seems to be based on the conception that it improves both modularity and the structure of code, while in fact, it works against the primary purposes of the two, namely independent development and understandability of programs. Not seeing any way of fixing this situation, he thinks the success of aspect-oriented programming to be paradoxical.

Categories and Subject Descriptors D.2.2 [**Software Engineering**]: Design Tools and Techniques – *Modules and interfaces; Structured programming.* D.3.2 [**Programming Languages**]: Language Classifications – *Multiparadigm languages.* D.3.3 [**Programming Languages**]: Language Constructs and Features – *Modules, packages; Control structures; Procedures, functions, and subroutines.*

General Terms Languages.

Keywords aspect-oriented programming; modularization; program structure; globalization of variables; independent development; readability; software engineering.

1. Introduction

I first encountered aspect-oriented programming (AOP) while writing my habilitation thesis, via the "detour" of subject-oriented programming (SOP) [30]. At that time, I was mostly interested in roles as first class modeling and programming concepts, and although I could see the practical problems SOP and AOP were addressing, I decided that their relationship to roles—at least the way I viewed them—was weak.

After finishing my habilitation, I was asked to take over the Software Engineering lectures. For Software Engineering II, I decided to include a short excursion into AOP, partly because I wanted to find out for myself what it was good for (if not for representing roles), partly because I wanted to communicate to my students that object-orientation and Java were not the last words in programming. AspectJ was particularly attractive for my purposes because it came with a compiler and a plugin for the Java IDE I was using. After fiddling with the versions I managed to get it installed and my first sample program running. What proved more difficult, though, was to find a conceptual motivation of AOP that convinced me (one comparable to how classes, associations, and roles can be motivated in OOP); unsuccessful as I was, I decided to stick with the material used for the AspectJ demo at OOPSLA 2002, which was available on the web. My students immediately bought it.

What impressed me most at that time was the fact that the developers of AspectJ had undergone the suffering of developing an IDE plugin that not only allowed compilation without preprocessing, but also provided tool support allowing me to deal with the features of the language rather than the technical obstacles to using it. In fact, all other language extensions proposed by academics I had looked into until that time either remained at the theoretical level entirely (with very impressive, page long soundness proofs convincing me that there are smarter guys out there than me, but not giving me any feeling of the practical impact of the formalism), or came with command level precompilers requiring me to undertake installation procedures so intimidating that I was sure I would never get it running. The AspectJ people on the other hand knew what it takes for a new language to be adopted by programmers: a set of simple concepts attacking a real problem, and good tool support. In this light, AspectJ was surely one piece of commendable work.

A little later, I attended a biannual national (German) meeting on teaching software engineering at universities. Someone had raised the question, what minimum half life does a new approach need to deserve to be taught? One highly respected participant said that we as lecturers should be able to judge the impact of, say, AOP right away so that there would be no need to wait for first signs of decay in order to be able to estimate the half life. I wondered why he used AOP as an example, and asked him for his judgment in this particular case. He responded by saying he was confident that AOP would have sufficient impact to grant teaching it. This made me wonder what made him so sure.

What disturbed me most about AOP at that time was the monotony of examples. In particular, to me the ever-recurrent logging, tracing, debugging, etc. aspects are all more or less "programming problems" in that they address concerns that a programmer has to deal with because he is programming, not because some particular problem domain or application demands it. This is in contrast to application classes such as *Person*, *Document*, etc., and also methods such as *attend*, *format*, and so on, which all represent problem domain level concepts. I conjectured that most, if not all of the programming problems addressed by AOP could either be tackled by adding a corresponding feature to an IDE (for example, tracing as done in Eclipse), or by extending the language with suitable constructs (for example, exception handling as in Java, or transaction management as in database languages). I expressed this in my provocative claim that "the number of useful aspects is not only finite, but also fairly small." Although obviously impossible to prove, I thought I could make it plausible by showing that aspects are not domain level abstractions and thus lack a significant source of diversity.

Because my claim was both provocative and unproven, I decided to test it against a pro-aspect audience and submitted it as a position paper to a small European workshop on aspect systems. Not surprisingly, most of the AOP proponents at the workshop would not follow my thoughts, and the discussion led, if I remember correctly, nowhere. However, it was there that I learned that to some in the community, AOP is all about modularity. To me, this came as a surprise, mainly because I missed interfaces in the AOP toolset, a construct which I had always thought (and taught) to be inseparable from that of modules. The good thing I took home from this workshop was that I decided to get myself a copy of a workshop paper by Filman and Friedman, titled "Aspect-oriented programming is quantification and obliviousness" [23].

When I first read this paper, I thought: "Wow, this is it!" It was the first paper I had come across that—without putting forward a particular language—seemed to be more concerned about what AOP *is* rather than what it *is good for*. Hence, it seemed one big step toward the conceptual justification of the approach and its constructs that I had been looking for. At the same time, it presented AOP as a next step in the history of the development of programming languages, and gave criteria for classifying whether a language is aspect-oriented. The third thing that struck me, however, was that the paper's characterization of aspect orientation almost completely covered one I had—believe it or not—independently devised for the purpose of making plausible "Why most domain models are aspect free" [59], my (vain) attempt to put an end to the (vain, in my eyes) attempts of aspectizing domain modeling and its graphical modeling languages. However, although I had based my argumentation mainly on the quantification property and the resulting second-orderedness of aspect-orientation, I quickly learned that not all in the community liked the article by Filman and Friedman as much as I did, apparently mostly because the notion of obliviousness was too much an invitation to question the modularity property thought to be crucial to AOP.

My most recent noteworthy encounter with AOP has been at a conference where someone explained to the audience how he had used AspectJ to factor out the various passes of a compiler from the node classes of the abstract syntax tree. As far as I could see, this inevitably meant that as aspects, the passes needing access to the data stored in the nodes had to break the nodes' modularity. When I remarked that he had just given us a perfect example of why AOP is fundamentally at odds with modularization, that in fact I believed that AOP should not be spoken of in connection with modularity in any other than a negative sense, someone from the audience responded that I was right, but only if I talked about "Parnas style modularization." Given that modularization is a fairly broadly accepted notion, I thought that this was a pretty odd thing to say.

To me programming is the process of creating a software artifact that, by repeated extension, adaptation, and correction, approximates an ideal solution to a given problem. The more complex the problem and its solution are, the more programming depends on recursive (de)composition, that is, on the possibility to repeatedly divide a problem into smaller parts whose solutions can either be taken off the shelf or be programmed independently. However, such an approach poses stern requirements on compositionality; in particular, it demands that the functionality of the whole is predictable from the functionality of its parts and how they are composed; because otherwise one has to understand the complete system in order to know what it does, making assembly from parts no easier than creating the software in one piece. To reach this level of compositionality, each part must come with a sufficiently accurate specification of what it does, and what it requires for doing it. While there may in fact be different styles of decomposition, the resulting specifications always involve modules and interfaces. Surely, keeping to interfaces means restricting the programmer's freedom, and programming without bounds (interfaces) is certainly more fun; yet I believe that for big systems, restriction is the key to success.

With hindsight, I personally have undergone a development that may not be so untypical for many like me. I started out trying to ignore AOP, but it came back to me with sufficient thrust to make me curious. By looking into it, I found its applicability to be fairly limited, but when I looked up again, I realized that it had begun to penetrate all areas of software engineering, at least in academic circles. Wondering what the key to its apparent success was, I tried to learn more about it, but the more I knew, the less I could see how AOP was going to live up to its claims. While this may not be the end of my personal development, I decided that I had collected enough material to share my thoughts with others. After all, AOP is not some shrinking violet that I could wipe out with a few well-chosen words. Quite the contrary.

2. Pinning Down a Moving Target

When I originally set out to write this tract, I tried to present a characterization of AOP that was

1. general enough to cover sufficiently many approaches huddling under the aspect-oriented umbrella, while at the same time was
2. specific enough to be able to base some conclusive argumentation on it.

I thought this was wise because although each aspect-oriented programming language (AOPL) comes with its own, formal and unambiguous definition of what AOP is, there seems to be no one such definition, not even on an abstract level, that

a) is common to all AOPLs and
b) sufficiently distinguishes it from other, long established programming concepts.

One problem with having no single, accepted definition is that it makes AOP a moving target for its opponents: whenever some problem is identified, answers of the kind "Ah well, this is a problem of <insert some AOPL here>, but if you look at, for instance, <insert another AOPL here>, then you don't have this problem!" can easily be generated. I tried to shield the points I was going to make from frustrating discussions of this kind, by basing my argumentation on as few defining characteristics of AOP as possible. I was hoping that my readers would agree that there must be some such characteristics, or else the subject of the discussion would dissolve away, at least from an engineering standpoint. Unfortunately, I did not get further than generating comments of the kind "While this may be a valid characterization of <insert some AOPL here> and like languages, it is certainly not one of <insert some other AOPL here>." With hindsight, this is what I should have expected; others much more involved in the subject than I am have come to the conclusion that capturing AOP is a very difficult undertaking (cf., for example, Mehner & Rashid [45] and also the discussions at `http://aosd.net/`).

Despite the lack of a common agreement of what AOP *is*, there is a common understanding of what AOP *is good for*, namely for

modularizing crosscutting concerns.

However, this understanding reflects the purpose, not the nature of AOP. What disturbs me most about this is that it forbids me from deriving myself what AOP is good for, since this is anticipated in its definition. In particular, with "modularizing crosscutting concerns" as the definition of AOP, finding that whatever is considered to be AOP modularizes crosscutting concerns is just begging the question,[1] and finding that it does not implies (via *modus tollens*) that whatever I have looked at and found to not modularize cannot have been AOP—basta! Also, if modularizing crosscutting concerns is a widespread problem whose solution bestows success on the approach, AOP must be a success, simply because it is by definition solving the problem. If it is successful without solving the problem, it can still be so for other reasons, but these cannot be explained by its definition; in fact, it leaves what is successful undefined.

I don't find this very satisfactory. Instead, as formulated by Filman and Friedman:

> *Understanding something involves both understanding how it works (mechanism) and what it's good for (methodology). In computer science, we're rarely shy about grandiose methodological claims (see, for example, the literature of AI or the Internet). But mechanism is important—appreciating mechanisms leads to improved mechanisms, recognition of commonalities and isomorphisms, and plain old clarity about what's actually happening.* (from [22], Chapter 2, a revised version of [23])

Because "recognition of commonalities and isomorphisms, and plain old clarity about what's actually happening" are precisely what I am interested in, I will need an understanding of the mechanisms of AOP. For this purpose I will resort to a rather simplistic capture of how AOP works, willing to accept that it does not cover all of AOP—in particular, that it ignores approaches such as SOP that, previous to joining the AOP family, led an independent life.

3. The Aspect Formula

Perhaps the best known definition of what AOP *is* (its nature) is the simple equation

aspect orientation = quantification + obliviousness (1)

put forward in a workshop paper by Filman and Friedman [23] and only recently repeated in a book on aspect-oriented software development edited by Filman and others [22]. Obliviousness basically implies that a program has no knowledge of which aspects modify it where or when, and quantification expresses the fact that an aspect can affect arbitrarily many different points in a program. One might be tempted to add "precisely which being specified by the aspect" to the last sentence, but this would make obliviousness a consequence of quantification, which would not allow one to be discussed independently from the other. In fact, while obliviousness has been the subject of some criticism, and in response to this has been questioned as a defining characteristic of AOP by its community (see, for example, Murphy and Schwanninger [47]), quantification seems to have been challenged less. Thus, the sentence,

In programs P, whenever condition C arises, perform action A. (2)

which is also from Filman and Friedman [22, 23] and captures much of the essence of Definition (1) without mentioning obliviousness explicitly[2], seems much more generally agreed upon. In fact, Characterization (2) specifies in a concise way the effect an aspect—defined as a pair (C, A)—has on a program P.[3]

As mentioned in the introduction, I had proposed a slightly more explicit formulation of Definition (2) independently in an earlier paper of mine [59]; in particular, that formulation captured a notion of context in which condition C arises, and in which action A is performed. This context is of particular importance when speaking about modularity (and thus the purpose of AOP), since whenever action A is not completely independent of the context in which C arises, A will require access to this context, possibly breaking the modularity of the program P. I will therefore assume in the rest of this essay that both C and A are parameterized by a set of context variables that are bound to actual program elements whenever C is satisfied (the existence of the context elements may in fact be part of the condition).

Note that it is somewhat typical for AOP—although perhaps not a necessary condition—that the context provided to an action A is expressed by the aspect (C, A), but not by the program elements that provide it. By contrast, a subroutine call explicitly specifies the context (parameters) passed to the subroutine at the call site (unless the subroutine has automatic access to this context, for instance through global variables). Generally, this means that the program elements satisfying a condition C are oblivious to which elements of their context an aspect relies on (they could assume all, though).

[1] an instance of the logical fallacy of that name, also known as *petitio principii*

[2] In fact, Definition (2) is presented in Filman and Friedman [23] as capturing the quantification part of aspect orientation. From Definition (2) alone, it remains unclear whether P (or its elements) have knowledge of C. Filman later added that "the oblivious claim is that real aspect languages do not require P to mention A" [24].

[3] One might argue that this capture of AOP is incomplete in that it leaves out the structural (as opposed to behavioral) changes made possible by certain AOPLs (for instance the inter-type declarations of AspectJ). But the same is also true for other popular definitions of AOP that are of the kind "when X happens, do Y" (Coyler et al. [12]). Besides, and decisive for this paper, the possibility to introduce structural changes does not alleviate any of the problems I am concerned about: the same line of argumentation can be applied to a definition of AOP that includes structural introductions.

3.1 Interpretations of the Aspect Formula

Admittedly, the characterization of AOP as captured by Definitions (1) and (2) seems to be influenced by the language definition of AspectJ. In fact, the definition of aspects as a pair (C, A) and their effect on programs covered by Definition (2) translates to the terms of AspectJ as follows:

- P is the execution of a program, which includes the execution of advice (see below);
- C is a set of so-called *pointcuts* specifying the target elements of the aspect in the program and the context in which they occur (mostly variables, but also stack content);
- A is a *piece of advice* that depends on the context captured by C; and
- the quantification is implicit in AspectJ's compiler/weaver.

Despite the influence of AspectJ, the generality of the above simple capture of AOP should not be underestimated:

1. First and foremost, it allows full flexibility for different AOPLs concerning *when* C is to be evaluated and, consequently, *which elements* of the program it has access to. So-called static AOPLs evaluate C at compile (or class loading) time, whereas dynamic AOPLs evaluate it at runtime. In the static case, evaluation of C has only access to the elements of the program text. In the dynamic case, C can range over execution elements as well as over temporal patterns of these.
2. Second, there are no theoretical bounds to what the condition C has access to, in particular, what the context may include: declared and actual types of all kinds of variables (not only parameters), their values (objects), the receiver of a method call, the caller, the current state of the program, past states, the call stack, an event, a sequence of events—C could even be a predicate over traces [3, 37, 65]. In brief: in Definition (2), condition C can have access to whatever is allowed by the AOPL and can be made available by the underlying runtime system.
3. Depending on viewpoint, action A is expanded to a sequence of program elements (static view) or to a set of join points (dynamic view). In either case, the condition C of an(other) aspect can range over the elements of A. In other words: actions A can be elements of programs P, and aspects can be the targets of aspects. Therefore, Definition (2) is not as asymmetric as it may seem. Instead, one should be aware that "target"—in the literature usually referred to as "base program"—and "advice" are relative terms, or roles of advising: what is the target in one aspect application can be the advice in another. This is somewhat analogous to the (not unrelated) distinction between object language and metalanguage, which comes with corresponding roles [7, 25].
4. In practice, the point nature of join points limits the variability regarding how the execution of an action (advice) A can be combined with that of the program elements (join points) triggering it: *before*, *after*, "*around*" (that is, one part before, one part after), and *instead*[4] seem to be the only options. This is of course different if the model also allows multi-point patterns [3, 37, 65] as join points. Since Definition (2) does not place any constraints on the nature of the elements of P other than that they can be selected by suitable conditions C, this model of AOP covers multi-point patterns as well.

In fact, with a little flexibility Definition (2) of how AOP works can even be stretched to be in accord with more recent characterizations of AOP, such as those proposed by Masuhara & Kiczales [44] or Kojarski & Lorenz [38]. In particular, conditions C can be interpreted as composition rules governing the composition of different action sets A (representing different concerns) into a composed program P. This would allow Definition (2) to cover other, so-called symmetric approaches to AOP such as Hyper/J [50], as well as the inter-type declarations of AspectJ (cf. Footnote 3). In the context of this essay, however, it is important to note that these newer characterizations of AOP do not assign it properties voiding the discussions that follow. In particular, in all but the most trivial cases the composition rules C (labeled R in [38]) will likely need some intimacy with the structure (including context) of the actions A (labeled C in [38]) to be composed.

3.2 But OOP…!

Some proponents of AOP say that trying to nail down AOP to its mechanisms is premature, or even unfair, simply because an analogous capture of OOP has long been—or is still—missing, and that this fact hasn't compromised OOP or its community. However, I don't agree with this argumentation.

Concerning the lack of definedness, I never thought that this was a problem for OOP. In fact, the first definition of OOP that I heard is still the one I use today:

object-orientation = abstract data types + inheritance (3)

Surely, this characterization depends on a common understanding of what abstract data types and inheritance are, but other than that, I think it's perfectly OK. In particular, it serves as a broadly usable criterion for deciding whether some language is object-oriented or not: all that needs to be done is to check whether it supports abstract data types and comes with some kind of inheritance. At the same time, this definition is sufficient—respective definitions of abstract data types and inheritance provided—to infer some properties of OOP. For instance, its modularity property can be inferred from that of abstract data types, and it can be shown that OOP has a problem with modularity if inheritance breaks the encapsulation provided by abstract data types. This susceptibility to rigorous reasoning is in sharp contrast to the "grandiose methodological claims" that OOP isn't at all devoid of: that object-orientation better captures the real world, that it allows seamless integration of analysis, design, and implementation, that it leads to productivity gains, etc. None of these claims serves the "plain old clarity about what's actually happening". But Definition (3) does.

4. Playing with the Options

That Characterization (2) is indeed a very general capture of AOP can be seen by running through a number of possible formulations of condition C. At one extreme, C could stand for the condition that a certain aspect with associated action (advice) A is referenced in the program text. Definition (2) then expresses no more than the semantics of a standard procedure call:

In programs P, whenever an aspect is referenced, perform its associated action A. (4)

[4] Note that allowing an aspect to block execution of its advised join point is somewhat contradictory to the interpretation of AOP as event-driven programming [18], in that an event that led to the execution of an aspect actually does not take place.

This is an interesting construction, since it shows that quantification can indeed be completely independent from obliviousness: all places where condition C can possibly arise are explicitly marked in the program text (cf. Footnote 2). Of course the programmer of P needs to know which aspects there are, how they are named, and where or when they should apply; and almost certainly, no one would accept this style as AOP, since it can be replaced by ordinary procedural programming; nevertheless, it goes to show that Definition (2) of how AOP works is quite stretchable.

At the other extreme, C can express some condition that does not allow a programmer of P to associate elements of P with aspects: for instance, C could express a random selection, invoking A by chance (including always or never). This would be expressed by

In programs P, whenever *Random* indicates it, perform action A. (5)

Then, aspect awareness of a program is reduced to the level that all places in a program may be regarded as implicitly marked, but performance of A remains uncertain (the "non-certainty of application" noted in a different context in "AOP considered harmful" [13]). This is largely the situation in which the programmer of P has no knowledge of the presence of aspects or which they are, but knows that AOP exists and that P may be subject to it; and also to a certain extent the situation in which the programmers of aspects (C, A) have no detailed knowledge of the programs the aspects are to apply to.

Surely, these formulations of C are theoretical extremes that no practical AOPL will adopt. The question that I find interesting, though, is whether the conditions C can be cast in such a form that AOP serves its methodological claims, in particular the modularization of crosscutting concerns, while at the same time makes its mechanisms sufficiently innovative to justify its reception as a new form of programming. I will therefore further explore the possibilities of tweaking Definition (2).

4.1 Taming Obliviousness

Returning to the first extreme, it is obvious that directly calling aspects from a program is not AOP, simply because it is indistinguishable from procedural programming. However, the program P need not make explicit reference to the aspects themselves—instead, it could also reference some third elements B exterior (that is, not directly contributing) to P that are not parts of the aspects, but nevertheless indicate that some aspects may evaluate these elements in their conditions C, possibly invoking the attached actions A. This would be expressed by the formula

In programs P, whenever condition C arises where element B is referenced, perform action A. (6)

In order to hook an aspect (C, A) to B, B will usually be integrated into the aspect's condition C so that Definition (6) collapses to Definition (2) with the additional constraint that C must check for the presence of B. As regards the additional program elements B, annotations can be employed[5] (which are in a way outside the program they annotate); however, if these annotations do not allow the inclusion of runtime values, they cannot capture the context of B which A may need access to[6]. Note that B need not necessarily occur *exactly* where A is to be invoked—it can also be attached to a program scope in which C should be checked. For instance, B could be an interface of a module to whose internals (execution of program elements inside the module) aspect (C, A) is to be applied.

Using additional program elements B to tag the places where aspects may apply gives the programmers of P the possibility to deny aspects access where it is not wanted, simply by not referencing B (or any other annotation that could be evaluated by aspects) in these places. To express where they could apply, however, the programmer must have some sense of what possible aspects might want to do, so as to be able to tag the corresponding points in a program and also to be able to expose their context through B, if that is linguistically possible. In fact, referencing B in a program is more or less equivalent to inserting (or announcing, in case B does not mark the exact position) a dynamically bound procedure call, the main difference being that there is an additional condition C guarding this call that is not expressed at the call site. Also, if B does not capture the context that A may have access to, A must take access to the context it depends on for granted.

Referencing some other element B in the program, while keeping the information about which aspects are hooked to B the secret of the aspects, adjusts obliviousness to a level at which the programmer knows that aspects may interact with the points in P tagged with B (and will not interact with all other points); yet he does not know which aspects. As with the dynamic binding of methods in OOP, the surprise induced by this ignorance can be reduced by specifying contracts that have to be fulfilled by each aspect advising points matched by B, as has been suggested by, for example, Clifton, Griswold, Sullivan, and their co-workers [10, 27, 63] (note that [27] and [63] also impose contracts on the targets; see the appendix).

But no matter how attractive annotating the targets of aspects as suggested by Definition (6) may appear, AOP has a problem with it: for massively crosscutting concerns, annotating every program element that can be the target of corresponding aspects leads to widely scattered annotations that are just as annoying as the scattering of code the aspect is to modularize. For instance, with tracing as a crosscutting concern, annotating every program element whose execution is to be traced is just as annoying as adding the tracing code (usually no more than the calling of a subroutine) on site. To avoid this, it has been suggested to use so-called annotator aspects that annotate program elements so that they can be advised by other aspects [39], as expressed by

In programs P, wherever condition C arises, add annotation B. (7)

Obviously, the annotator aspects could be used to add the advice directly, but this would mean returning to Definition (2), that is, AOP without annotations. Considering that the proposed use of abstract annotations is to "[translate] some of the best practices from the object-oriented world to AOP" [39], and that this translation requires other aspects to restore the characteristic of AOP, the

[5] called "abstract annotations" by Laddad [39], because they indicate the nature of the annotated rather than possible aspects that depend on it; also referred to as "annotation-properties" by Kiczales and Mezini [35]

[6] This is the case, for instance, in Java.

suggested tango of aspects and annotations[7] looks more like an egg dance to me, the eggs being obliviousness, quantification, and explicit procedure calls.[8]

4.2 Taming Quantification

Returning to the other extreme, thinking of the conditions C as random may seem absurd. However, for the programmer of P who is unaware of the active aspects, which of his statements get advised may indeed appear random;[9] for the programmers of the aspects, if the aspects use generic, intensional expressions for C, exactly which program elements of an (evolving) program an aspect advises is also subject to some coincidence. A first step toward reducing this apparent randomness is to resort to an extensional specification of C, that is, to explicitly listing all program elements to be advised. This is expressed by

In programs P, whenever execution reaches one of the points in $\{p_1, \ldots, p_n\}$, perform action A. (8)

While this maintains P's obliviousness to aspect application (the implicit invocation of A), it requires aspects to repeat explicitly those parts of a program that they are to advise. As with annotating targets of aspect application, for massively crosscutting concerns this also quickly becomes a nuisance, the biggest difference being that the previously scattered references are now all collected in one location. While such a programming technique (which reminds me of manually maintaining the tables a linker uses for putting together separately compiled parts of a program) may be useful for updating (or patching) existing and already deployed programs, it is most certainly not what the AOP community envisions.

Generally, the quantification property makes AOP suffer from the problem that the conditions C in Definition (2) and its variations are extremely sensitive to changes in the program P, a condition that has become known under the label *fragile pointcut problem* [61] (also called *arranged pattern problem* in [29]). Some researchers expect that this problem can be addressed by devising better languages for expressing C ("semantic" pointcut or crosscut languages [29, 42, 48, 51]; cf. also [4] for the semantics of pointcuts). However, no matter how "semantic" a selector predicate C may be: in order to be able to formulate its action in terms of a programming language, an aspect must make reference to the context it needs access to. I find it difficult to imagine how "semantic" conditions and their context specifications can be *automatically* mapped to the surface structure of a program. Ultimately, I believe, semantic referencing as envisioned by Lopes et al. [42] will require automatic program understanding which, once available, will revolutionize the entirety of programming, conceivably making AOP, as well as many other techniques *en vogue* today, obsolete.

4.3 Quintessence

It seems to me that its nature makes AOP a rather delicate creature: when developed to the full, both obliviousness and quantification conflict with its goals, but cutting back on them seems to deprive AOP of its core contribution. Reducing the obliviousness in a program not only comes close to reducing implicit invocation of advice to some variation of dynamically bound procedure calling, it also reintroduces the very scattering AOP was to avoid. Reducing the quantification in aspects ultimately amounts to maintaining lists of places in a program the aspects are to advise: the obliviousness of a program to its aspects is therefore bought at the price of far-reaching intimacy of the aspects with the program, which thwarts modularization and thus the purpose of AOP. It may be my ignorance or lack of imagination, but I cannot see how to get out of this dilemma.

5. AOP from a Software Engineering Perspective

In the previous sections I tried to characterize AOP in abstract terms, which allowed me to point to some of its problems and to give reason to my belief that it will be very difficult to fix them without giving up the essence of AOP. For much of the rest of this essay I will try to carry over my findings to the real world of programming, that is, to software engineering.

5.1 AOP and Modularization

One of the first things that I noted when digging into the body of available literature is that the AOP community revels in quoting Parnas, a man whose name is inseparably tied to the concept of information hiding and whose contributions to program modularization are respected to this date. In particular, Parnas's seminal article titled "On the criteria to be used in decomposing systems into modules" [55] is almost universally referenced when speaking of the separation of concerns, even though the term is never mentioned in it. This however is not a real problem, since the modularization criteria Parnas suggested do serve separation of concerns. My problem with citing Parnas's work is that in my eyes it does not accommodate the AOP form of modularity; if anything, it forbids it.

5.1.1 Modularity and Information Hiding

This requires some explanation. At the time when Parnas's cited article was written, modularity was already a well-established notion, basically known as one that helps manage the development of large programs by dividing them into separate chunks that can be developed largely independently of each other:

> *A well-defined segmentation of the project effort ensures system modularity. Each task forms a separate, distinct program module. At implementation time each module and its inputs and outputs are well-defined, there is no confusion in the intended interface with other system modules.* [...] *Finally, the system is maintained in modular fashion;*

[7] phrased after a talk titled "Metadata and aspect-oriented programming: It takes two to tango" presented at Java One in 2004

[8] Egg dance: "Lightly, nimbly, quickly, and with hairbreadth accuracy, she carried on the dance. She skipped so sharply and surely along between the eggs, and trod so closely down beside them, that you would have thought every instant she must trample one of them in pieces, or kick the rest away in her rapid turns. By no means! She touched no one of them, though winding herself through their mazes with all kinds of steps, wide and narrow, nay even with leaps, and at last half-kneeling." J. W. von Goethe, Wilhelm Meister's Apprenticeship, Book II, Chapter VIII.

[9] Proponents of AOP usually point to possible tool support here: a tool (such as an aspect-aware IDE) can highlight the potential targets of aspects in the program text. However, this highlighting is not without randomness, either: it depends on the availability of the conditions C at editing time (which is a problem for aspects that are added later), and it is subject to change without notice, namely when an aspect programmer chooses to change C.

system errors and deficiencies can be traced to specific system modules, thus limiting the scope of detailed error searching. [55], quoting Gauthier and Pont "Designing Systems Programs" (Prentice-Hall, Englewood Cliffs 1970).

At that time, so Parnas reported, modules typically consisted of one or more procedures that mapped to a phase or step of processing (as derived, for example, from a flowchart); the data structures on the other hand, on which the procedures operated, were shared among the modules and thus were part of their mutual interfaces. Parnas's contribution to modularization was to deliver a novel criterion that guided the process of creating modules: each module should fully encapsulate one design decision so that later changes of this decision would be less likely to affect more than the hosting module. Parnas named this criterion "information hiding" ([55], p. 1056), making reference to an earlier paper of his, titled "Information distribution aspects of design methodology" [54].

It is instructive to read this paper, too. Much to my surprise, Parnas did not write about (the benefits of) information hiding, but about the detrimental effects of the opposite, namely *information distribution* among programmers. In fact, he never even used the term "information hiding," nor did he suggest what we know today as "data encapsulation" (which for me had been more or less synonymous until then[10]). Instead, he made clear that information hiding (or, rather, information non-distribution) is not only a design issue affecting the structure of the product, but also a process issue: information of how something was implemented should not be shared among programmers working on different modules, because this would create untoward dependencies and hamper independent development. In fact, in a recent defense of his earlier work he said that

My early work clearly treated modularisation as a design issue, not a language issue. A module was a work assignment, not a subroutine or other language element. Although some tools could make the job easier, no special tools were needed to use the principal, just discipline and skill.

When language designers caught on to the idea, they assumed that modules had to be subroutines, or collections of subroutines, and introduced unreasonable restrictions on the design. They also spread the false impression that the important thing was to learn the language; in truth, the important thing is to learn how to design and document.

We are still trying to undo the damage caused by the early treatment of modularity as a language issue and, sadly, we still try to do it by inventing languages and tools. [15]

That Parnas's work was actually perceived that way is expressed by a contemporary quote from Brooks's classic "The Mythical Man-Month":

D. L. Parnas of Carnegie-Mellon University has proposed a […] *radical solution[11]. His thesis is that the programmer is most effective if shielded from, rather than exposed to the details of construction of system parts other than his own. This presupposes that all interfaces are completely and precisely defined. While that is definitely sound design, dependence upon its perfect accomplishment is a recipe for disaster. A good information system both exposes interface errors and stimulates their correction.* ([8], Chapter 7)

At that time, Brooks seems to have favored that all programmers should see all the material, so as to increase the overall quality of code and to spot flaws and bugs early. However, 25 years later he recanted and stated, "Parnas was right, and I was wrong about information hiding" ([8], Chapter 19), basing his change of mind on the insight that "information hiding […] is the only way of raising the level of software design."

5.1.2 Modularity and Data Encapsulation

Although Parnas—with his suggestions to improve modularity of designs—did not call for new programming language constructs, he did suggest that data encapsulation (once more without resorting to this term) is a viable criterion for decompositions respecting information hiding, although not the only one:

1. A data structure*, its internal linkings,* accessing procedures and modifying procedures *are part of a single module. They are not shared by many modules as is conventionally done.* ([55], the first item in a list of "some specific examples of decompositions which seem advisable."; Parnas's emphasis)

To today's object-oriented programmers, this seems a matter of course, but at that time, it revolutionized thinking about modularity: it suggested that modules can consist of procedures together with the data structures on which they operate (ideally hiding the latter behind the former), and that such a module no longer necessarily corresponds to a certain phase or step of the processing. This was the first move in the direction of object-oriented programming, as Brooks later acknowledged:

Parnas's information-hiding definition of modules is the first published step in [a] *crucially important research program, and it is an intellectual ancestor of object-oriented programming. He defined a module as a software entity with its own data model and its own set of operations. Its data can only be accessed via one of its proper operations. The second step was a contribution of several thinkers: the upgrading of the Parnas module into an abstract data type, from which many objects could be derived. The abstract data type provides a uniform way of thinking about and specifying module interfaces, and an access discipline that is easy to enforce.* ([8], Chapter 19)

The third step was the adding of inheritance which, as we know, breaks modularity of abstract data types (classes). But even without inheritance, programming with abstract data types (that is, information hiding enforced by linguistic data encapsulation mechanisms[12]) is not without problems: taken to the extreme, it leads to the situation in which a single system level function (use case, concern, or whatever you want to call it) is distributed among all modules whose encapsulated data are involved in that function. This leads to the scattering of functionality that is so characteristic of object-oriented programming. In this light, I found it interesting to see that a similar problem was already recognized by Parnas, who warned us:

[10] an instance of the eternal confusion of data and information

[11] Brooks refers to a technical report here that was a precursor to reference [54].

[12] which is not what Parnas had in mind! (personal communication)

If each of the functions is actually implemented as a procedure with an elaborate calling sequence there will be a great deal of such calling due to the repeated switching between modules. The first [traditional] *decomposition will not suffer from this problem because there is relatively infrequent transfer of control between modules.* [55]

Today this reads like a valid criticism of object-oriented programming: when trying to understand, or debug, a function of an object-oriented program, the frequent transfer of control between modules (classes) is indeed a problem. But can AOP solve it?

Before I proceed, let me make clear that I do understand that there is a difference between the scattering of code resulting from the decomposition of a function into subfunctions that are associated with the data they operate on, and the scattering of code implementing "crosscutting" concerns such as tracing or logging, which is reflected in more or less identical pieces of code being found in several places. Also, the implementation of crosscutting concerns is usually tangled with that of other (crosscutting) concerns, which is typically not the case for subfunctions. On the other hand, both the subfunctions and the crosscutting functions access and operate on data elements held by the objects of the classes they are associated with; therefore, I believe that the following thought experiment of applying AOP to the modularization resulting from data encapsulation via classes (which AOP applied to OOP is invariably about) is legitimate.

In order to arrive at a better modularization of concerns, AOP allows that the scattered subfunctions are moved into an aspect. But if the original design that led to the scattering is guided by data encapsulation, the subfunctions are assigned to a module because they operate on the data contained in that module, and because changing the representation of the data would likely affect the implementation of the subfunctions. By moving the subfunctions to an aspect, this data dependency is not lifted, but is either

- moved to the interface between the module and the aspect (if such an interface at all exists; see below), or is
- left implicit, by granting the aspect general access to the data hidden in the module.

The former makes evident in the interface the resulting coupling between an aspect and the original module. A change in the data structure captured by the module (its formerly hidden design decision) on which the subfunction and thus also the aspect depend likely entails a change of the interface and therefore also the aspect. Independent evolvability is therefore compromised. The latter suffers from the same problem, but is worse in that the programmers responsible for the design decision thought to be encapsulated by the module are not aware of the dependency of the aspect on that decision (because there is no explicit interface stating this dependency). In fact, granting the aspect the access that it needs amounts to a globalization of the data contained in the module. But this is exactly the situation that Parnas found to be prevailing at the outset of his work: modules hosting phases or steps of processing, and complex interfaces between modules that capture the shared design decisions, or dependence of all modules on global data structures.

Now one might blame me for suggesting an improper use of AOP, one in which it is misused to implement a questionable design. Surely, such abuse of concepts is possible in all programming paradigms. But my main concern is not reverting to some design ideal thought to be long overcome (the division of a program into functions corresponding to steps or phases of processing, which may in fact be justified even in OOP); my main concern is the existence of a strong coupling between an aspect and its target, particularly if this coupling is left implicit, that is, not reflected in an explicit interface; since this impairs independent development. And this is the same, at least as far as I can see, for all but the most trivial crosscutting concerns factored out to aspects: *when code is moved out of its context to some other place, it must take (a reference to) the context that it depends on with it, thereby establishing a coupling between its old and its new location*. Only if the context that it depends on is already published in the interface of its old host, independent development will not be compromised by this move. Given that classes hide design decisions, my feeling is that this will not often be the case. Therefore, the effect of AOP is likely less (or worse) modularity and not more (or better).

Now I will not ignore that data encapsulation as realized in object-oriented programming languages à la Java is not without problems. In fact, it is often difficult, or even impossible, to assign procedures to a class so that they depend only on the data structure represented (or hidden) by that class, and no other. The availability of C++'s friend functions and so-called multi-methods (methods whose late binding depends on the dynamic types of the receiver and the parameters) in other languages provide sufficient evidence for this. But as far as I can see, modularity problems of this kind can only be solved by introducing units larger than single objects (or their classes) as modules. Splitting a class into a class and an aspect produces smaller, strongly coupled units; it leads to more and larger interfaces, which is counterproductive to improving modularity.

5.1.3 Modularity and Interfaces

For all I know, the concept of a module is meaningless without that of an interface.[13] Interfaces form the borders between modules across which control flow and data is passed; they specify the functions that can be called and the variables that can be accessed.[14] Interfaces represent the coupling between modules—only if the interface between two modules is empty are the modules completely decoupled. If the interface is not empty, modules are decoupled to the extent that changes on either side are admissible without notice as long as the interface is kept constant. Note that this is independent of what is *explicitly specified in the program text to be the interface*: interfaces between modules exist regardless of what can be (or is) declared by the available means of the programming language used. However, leaving interfaces implicit is a bad start for independent development; quite the contrary, to ensure independent development, as much must be made explicit as possible.

[13] Both Gauthier & Pont and Brooks above stress the significance of interfaces in connection with modules, which has been a consistent theme in Parnas's writings. Also, the ACM Computing Classification System lists modules and interfaces as one common entry (under D.2.2, "Design Tools and Techniques").

[14] Depending on definition, interfaces also specify protocol, that is, the sequence in which procedures can be called and variables can be accessed. Certainly, this information was part of the interface Parnas had in mind. However, most contemporary programming languages support only weaker notions of interfaces, namely sets of signatures. The remaining information must be communicated using means outside the programming language (that is, specification or documentation).

5.1.4 Provided and Required Interfaces

Work on component-based programming, which relies heavily on components as modules and on the explicit specification of the interfaces between them, has led to the notion that interfaces come in two complementary forms: a module can have *provided* and *required* interfaces, and one module's required interface is another module's provided interface. A provided interface is basically a collection of program elements a module offers to its clients. A required interface on the other hand is a set of program elements a module needs from some other module for performing its function. In a system composed of modules, there needs to be a match between each required interface of one module and a provided interface of another.

It is instructive to try and apply these terms to aspects (as modules) and their targets (advised or base modules). Clearly, the items passed between the target and an aspect (C, A) are captured by the context attached to (C, A), which qualifies for an interface specification. But is this interface a required or a provided interface? Because the aspect provides a particular service through which it extends a program, one might be led to think of it as a provided interface. However, the matching required interface of the target remains implicit—the target program does not specify *that* it needs something, let alone specify *what precisely* it needs. Therefore, there is no visible (as made explicit by a required interface) coupling between the target module and the aspect—the target does not appear to depend on the aspect.

From a different viewpoint, one might argue that it is actually the target module that *provides* a set of program elements, which are *required* by the aspect to perform its function. And indeed, the aspect specifies a required interface in the guise of its condition C: it specifies the program elements the aspect needs to query from its target in order to achieve its function (see Ostermann et al. [51] for a similar view). This reflects the "inversion of dependency" [49] so characteristic of AOP: technically, although the aspect complements the target program, the aspect depends on the target and not vice versa. However, despite this dependency the target module comes without an explicit counterpart interface specification: its provided interface is implicit at best.

Seen either way, the target specifies no interfaces that could be matched with those of its aspects. For the programmer of the target module this means that there is no visible (explicit) coupling and, more importantly, that there is nothing to keep constant across all possible changes of the secrets of the module. This however ignores the part of the aspects which do specify interfaces that must not change if the aspects are to remain unaffected by target module modifications.

Now one might argue that requiring aspects and their targets to explicitly specify provided and required interfaces is unfair; after all, the interface between a class and its subclasses is not only not divided into a provided and a required interface, it is also mostly implicit in most object-oriented programming languages in use today (except for the rather weak notion of declaring members as "protected"). While this is certainly correct, it is also widely accepted as substantial and valid criticism of OOP as a form of modular programming: in fact, as exposed by the so-called fragile base class problem [46], subclassing breaks the modularity of classes. Needless to say that subclassing impedes independent development unless (a) the implicit interface between a class and its subclasses is made explicit, or (b) a class and all its subclasses are assigned to the responsibility of one team, and can thus be regarded as one module. Surely, adding interfaces on the targets' side means giving up much of the attractiveness of the approach, and packaging a class and its aspects into one module is counter to the intent of AOP. But denying attacks against the claimed modularity of AOP by pointing at similar weaknesses in object-oriented programming is no way out of this dilemma.

5.1.5 Modularity and Dynamic Interfaces

It should be clear that resorting to a purely dynamic AOPL (whose conditions make no reference to static parts of a program) is no escape: even if dynamic interface specifications (behavioral interface specifications [67] for example or event sequence specifications [3, 37, 65]) are supported by an AOPL, in order to also support modularity (viz. independent development), they will have to be provided at both sides, the target's and the aspect's. In particular, mutual conformance of the interfaces, as well as adherence of the implementations to their interfaces, are promises made at development time. After all, this is what modularization is about.

5.1.6 The Modularity of Aspects

Now one could argue that while modularity of the (crosscut) target program is sacrificed, modularity of the crosscutting concerns is won, and that this may be better in certain cases. However, crosscutting concerns may crosscut each other, and whenever the actions introduced by an aspect are part of the program and thus candidates for aspect application (as is the case for instance in AspectJ), the modularity of aspects is broken in exactly the same way as that of target programs.

5.1.7 Summary

Introducing explicit interfaces on the target modules' side (including annotations that indicate where aspects can apply) can declare the coupling with possible aspects, but then, as argued in Section 4.1, aspect activation not only becomes almost indistinguishable from late bound, guarded subroutine calling, it also re-introduces the very scattering AOP was to avoid. On the other hand, more abstract interfaces on the targets' side would require equal relaxation of the required/provided interfaces on the aspects' side, but it is unclear how an aspect (or any system functionality for that matter) can be programmed without concretely specifying somewhere what it needs access to. Once again, it may be my lack of imagination, but I can see no way of fixing this situation—to me, it appears that the idea of AOP is at odds with interfaces and thus also with modularization. (For a detailed discussion of related work, see the appendix).

To conclude: There may be means other than data encapsulation to realize information hiding as a design discipline, but one invariant is that they must grant independent development. While I agree that independent development is an important problem of programming even today, I find it hard to accept that the notion of a module—as one of the most fundamental to software engineering—is reinterpreted to the extent that its original meaning is no longer recognizable. A module is (and unless we manage to delegate programming to machines entirely, will continue to be) a unit of independent development, for such a concept is (and most probably always will be) needed. If aspects don't support it, please don't call them modular. Call them something else.

5.2 AOP and the Organization of Source Code

Regardless of whether aspects modularize, one could still argue that they are a good way of organizing source code. Since indeed every nontrivial application comes with several more or less independent criteria according to which its source code could be structured, and since both scattering and tangling of code is in fact a nuisance, such is a legitimate goal. In fact, the late Dijkstra, whose works are also commonly cited in the AOP community (and seem to be the bibliographical sources of the term "separation of concerns" [16, 17]), applied considerable thought to the organization of source code.

But as it turns out, AOP is also at odds with the work of Dijkstra, in particular the idea of structured programming. In his famous letter to the Communications of the ACM titled "Go to statement considered harmful" Dijkstra argued that a programming language should set up a coordinate system according to which any trace of a program is describable as a simple set of coordinates telling one precisely where the program is, and how it got there (by knowing the previously executed statement). For a sequence of statements and for a branch such a coordinate would be the program pointer (telling one that the program got there from the statement preceding in program text, including how a possible prior condition evaluated), for a loop the program counter plus a loop counter, and for a subroutine the program counter plus another program counter pointing to the site where the subroutine was called (basically the call stack). Dijkstra stressed that the coordinate system was to be set up automatically by the programming language, not the concrete program (and hence not the programmer). In other words, program organization should be promoted by the programming language, and not left to the wisdom of the programmer.

Dijkstra's complaints led to the maxim that each control structure of so-called structured programming should have precisely one entry and one exit point. `Goto` statements break this condition, torpedoing all conceivable coordinate systems. Now it can be argued that the net effect of AOP on any of the mentioned control structures is equally destructive: since an aspect can plug into just about any point of execution of a program, one can never tell the previous (or following) statement of any statement. In fact, as has been pointed out in "AOP considered harmful" [13], AOP introduces a modern variant of the `comefrom` statement, which was once suggested as a humorous contribution to the `goto` discussion, the joke being that such an inverse form of calling—very much like the implicit invocation mechanisms of AOP [20]—renders even small programs completely unreadable [9].

Today, Dijkstra's demand for a firm coordinate system is no longer dogma. In fact, as regards the single entry and exit point criterion, we now know that having multiple exit points from control structures can improve readability of programs, even though the reader does not know (without additional pointers) the statement executed immediately before the exit. The reason that we accept this breaking with formal structuredness is that the alternative, introducing guards that result in skipping the rest of a control structure, is often worse. Allowing multiple entry points, on the other hand, is widely rejected, but not because they formally break with Dijkstra's suggested coordinate system, but because experience has shown that they are rarely needed, yet are almost always difficult to understand. So we should take Dijkstra's coordinate system as one attempt at an explanation for what it takes for a human to be able to map the dynamic control flow of a program to its static structure.

Now Dijkstra's argument can be seen as basically one about the property of locality in programs. As discussed by Filman and Friedman [22, 23], many advances in the history of programming have broken with locality; in fact, even subroutine calling (as one of the four basic "structured" control structures) does. While Dijkstra's suggested coordinate system takes the non-locality of subroutines into account (by adding to the coordinates a pointer to the call site), the next step in programming language history, dynamically bound procedure calls, require addition of yet another pointer, namely one pointing to the bound procedure, for without this, one would not know which the statement executed immediately before the one statically succeeding the call was. Experience with object-oriented programming has shown that this advancement leads to problems in program understanding, in particular in (mentally) tracing program execution. This is worsened by the fact that in languages with dynamic class loading (such as Java), the number of possible branches depends on the configuration of a system, that is, on the set of alternative implementations (subclasses) provided at execution time.

The implicit invocation of aspects can be viewed as the next logical step in this development. In order to know the predecessor (in execution) of a statement, one must only add a pointer to the aspect just called (if any). The problem is that the points in the program in which I need this pointer (the selected join points) are not marked in place, as is the case for a (statically or dynamically bound) explicit procedure call. Even with tool support annotating the so-called join point shadows [31] in the program text (that is, the places where advice may be called), these places depend on the final configuration of the system, that is, the number and kind of aspects added (cf. Footnote 9). This is in contrast to the problem induced by dynamically bound procedure calls, in which I know, independent of configuration, *where* I need a pointer to the called procedure (only to which procedure I may not know). Thus, AOP adds another dimension of not knowing what just happened, or where I have come from, to programming. The question is whether the possible gains are worth the confusion it causes.

Certainly, more time will have to be allowed before this latter question can be answered. However, I will allow myself a little speculation here. While trading understandability of a program for expressiveness of a language may be in the tradition of progress in computing, it seems to me that AOP is pushing expressiveness a little too far. Surely, it is still some way from unconstrained metaprogramming (which is thought to be too difficult to be mastered by the average programmer), but it may just be that a healthy trade-off between expressiveness and understandability has already been found, and that this trade-off does not include the implicit invocation mechanisms of AOP.

5.3 AOP and the Globalization of Local Variables

In the wake of Dijkstra's letter, many other programming concepts were questioned along similar lines of argumentation. Among them, and with direct relevance to AOP, is an article by Wulf and Shaw titled "Global variable considered harmful" [66]. In it, the authors argue that visibility of variables outside a program segment under consideration strain the intellectual abilities of programmers, because of the phenomena of "indiscriminant access" and "vulnerability", where *the former reflects the fact that the declaror* [sic] *has no control over who uses his variables; the latter reflects the fact that the program itself has no control over which variables it operates on. Both problems force upon the programmer the need for a detailed global knowledge of the program which is not consistent with his human limitations.* [66]

Transferred to AOP, the "declaror" of a variable is the target program, which indeed has no control over which aspects use its variables, and the "program" is the aspect which, if programmed without knowledge of the target, "has no control over which variables it operates on". Because of its very nature, AOP not only makes the control flow unobvious from the program text, it also effectively "globalizes" all variables aspects can get access to. The conclusion drawn by Wulf and Shaw—that in the presence of global variables programmers need "a detailed global knowledge of the program"—is therefore also true for the presence of aspects (cf. also, for example, the work of Aldrich, Clifton, Kiczales, and their co-workers [2, 10, 34]).

While the globalization of local variables is a worrying problem, there is another one related to context that somewhat alleviates it, but at the same time severely restricts the feasibility of AOP: the problem of how to get hold of the context needed by an aspect. While it is difficult enough for a simple aspect to specify in itself (that is, locally) the context it needs access to in such a way that the specification applies to all points of a program which the aspect is to address (Sullivan et al. [63] provide a list of such problems found using AspectJ), more complex crosscutting behavior is much more intertwined with a single location of the target (it may in fact involve multi-point patterns in both time and space), and also much more diverse in its appearance among different locations. Also, in all but trivial cases combining separated concerns ("weaving") will be much harder than inserting one concern before, after or around another (see Ernst [21] for a concrete example of this). To phrase it in mathematical terms: AOP is based on the assumption that crosscutting concerns are scalars that can be factored out of a vector (a program) without leaving a trace, and that this factoring out (separation of concerns) can be reversed without any loss in meaning; however, weaving an aspect into a program is not always as simple as multiplying a scalar with a vector.

6. Some Observations on the Use and Usefulness of AOP

As expressed in some detail in the previous section, my opposition to aspects and AOP as a programming discipline is based mainly on my impression that it dismisses basic software engineering principles, and that in order to restore these principles, it must be stripped of its key characteristics. On the other hand, adhering to these principles is not always compulsory, so that there are application domains in which AOP should be unproblematic. However, is seems to me that domains of this kind are not core to the motivation of AOP; rather, what I find amply are application examples that, besides suffering from the modularity and structuring problems discussed above, are questionable with regard to AOP's net effect on systems. To make my point clearer, I will contrast examples for which I believe AOP may be useful with examples of how it seems to be actually used.

6.1 Usefulness of Aspects in Generated Code

First and foremost, AOP's unpunished use should be granted where modularity and structuredness are unimportant. This is for example the case in code generation, where the generated code—not the source to the generation!—may be aspect oriented without causing any problems of the above mentioned kind. Thus, it would seem that natural application areas of AOP are ad hoc language extensions (including domain-specific languages) and model-driven development (MDD). The former could for instance enhance an existing OOPL with language constructs specific for security, transaction management, or design by contract. The latter seems particularly interesting since the primary assets of MDD, models, usually come with many different views of a system which, like aspects, need to be woven together. And yet, model integration—that is, the integration of the information contained in diagrams of various kinds—is still mostly an open problem in modeling, and it will be interesting to see whether AOP can actually contribute to its solution [60].

6.2 Usefulness of AOP for Component-Based Programming

Also, I believe that AOP is useful where proper modules are a hindrance rather than an advantage. Ironically, this is to a certain extent the case in component-based programming: namely in the special (but not infrequent!) situation in which a composite cannot be formed out of available components without breaking into them—that is, without disrespecting their designed interfaces. Indeed, in the non-ideal world of programming practice, available components sometimes happen to be one bit off what is actually needed (see Kiczales et al. on open implementations [33] for an account of such cases), and the alternatives to breaking modularity seem just as unattractive. In these cases, AOP-related techniques may "digest" components (that is, dismantle and reassemble them) to form a new whole, giving this whole a new hull providing the interface to the rest of the world. Seen this way, AOP grants the writers of so-called glue code entirely new possibilities. The price is, obviously, that because the inner components are no longer modules, they cannot be evolved independently; instead, the newly formed component must be seen as an atomic whole that can only be read, understood, and changed *in toto*. Because this procedure of digesting components can be applied recursively, it can be abused to destroy all modularity in a system, turning it into one big monolith comprising what used to be modules.[15] Thus, it must be used with measure; in particular, because of the above-mentioned lack of independent evolvability associated with it, I don't think that it is justified to found a new discipline of *modular* software development on it.[16]

6.3 Observed Uses of AOP

While AOP seems to be useful for the above-mentioned coding problems, it appears that it is mostly used to solve quite different ones. When looked at more closely, some of these examples show that AOP can be used to fix problems to which it itself (although not alone) contributes. I admit that my observations presented in

[15] An alternative way of looking at it is that there exist no modules prior to system composition, and that modularization takes place only after this composition has been done [34]; see the appendix for a discussion.

[16] Aspectual collaborations [41, 52, 53] as promoted by Ovlinger et al. are an alternative approach to combining aspects and modularity in the form of components; again, see the appendix.

the following are somewhat nit-picking, nevertheless I think that the motivation of a new programming model should not resort to examples that can be attacked so easily.

6.3.1 Aspects for Logging, Tracing, and Debugging

Logging, tracing, and debugging are perhaps *the* canonical applications of AOP—they are returned to almost universally in papers on the subject. Although we already have excellent tools for logging, tracing, and debugging at hand that work without AOP (take for instance the Eclipse IDE), I will accept that AOP can offer an alternative approach. However, since generally aspects can plug their advice into just about any point in a program's execution, tracing, logging, and debugging become important concerns even in programs that without aspects would not need them (because they have been written in such a way that program flow is obvious from its static structure, or that the program is obviously correct). In the extreme case, one will find oneself introducing a tracing, logging, or debugging aspect only to trace, log, or debug other aspects executed. So in a way, while helping to solve a particular category of programming problems, AOP also adds to them.

6.3.2 Aspects for Security Issues

An aspect can intrude into a program (its components, or modules) in order to implement security, but what if it fails to do so? What if it never intended to? Can security aspects be installed that check the validity of aspects, that authorize and/or authenticate them? Perhaps they can, but not only is this a bootstrapping problem (or are there aspects that can ensure their own security?), but this also poses the question of what the net effect of AOP on system security is. As with tracing/logging/debugging above, not all applications will explicitly need to address security issues, but if they are executed in an AOP environment, they had better do so.

6.3.3 Aspects for Program Verification

Aliasing is a well-known problem for the verification of object-oriented programs, since an alteration of one object's value (as addressed through a reference) can change the value of what appears to be another object, but is really the same object addressed by a different reference (its alias) [32]. In short, with aliasing the simple verification problem

$$\{x.a = \text{true}\}\ y.a := \text{false}\ \{x.a = \text{true}\}$$

can become quite hard to prove in a modular fashion, because x and y might refer to the same object.

With AOP, the problem becomes even worse, because aspects may access and change the values of variables in their context even between the executions of two consecutive statements. In fact, the above aliasing problem can be rephrased as an *aspect problem of program verification*: in an aspect-oriented program, it is unclear how

$$\{x = \text{true}\}\ y := \text{false}\ \{x = \text{true}\}$$

could be proven correct without performing a whole-program analysis. Once more, aspects can come to the rescue, by introducing runtime verification of programs (for example, Bodden & Stolz [6, 62] and Lorenz & Skotiniotis [43]); however, introducing aspects that verify programs with aspects (including themselves) [36] sounds more like an academic exercise than a practical thing to do. To paraphrase Hoare, one should strive to write programs in such a way that they obviously contain no bugs; with AOP, however, the best a programmer not aspect aware can achieve is write programs that contain no obvious bugs.

6.4 Conclusion

The telephone was first used for broadcasting concerts, and radio for peer-to-peer communication. The gramophone was thought to be a replacement for newspapers, and Gutenberg's moveable type was designed to reproduce exactly handwritten letters. In each case, it took many years until a truly successful use of the invention was found. I wouldn't be surprised if AOP ended up being used for something quite different from what it is thought to be good for today.

7. New Programming?

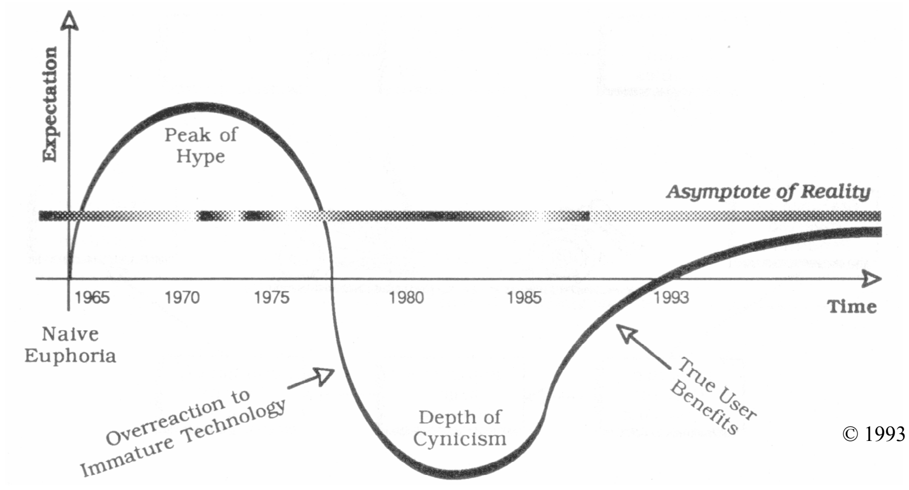
If I am right and if the problems I discussed above are all real problems of AOP, why, then, is it such a success? Is it a success?

Measured in terms of the number of successful commercial projects, it is perhaps still too early to judge. Measured in terms of the attention it receives, in academic circles in particular, it must be called a tremendous success. After only a few years, acceptance rates of the AOSD conference—*the* venue of the AOP community—seem to settle at approx. 20% (a score comparable to that of this conference (OOPSLA), which is now in its 21st year), and other major conferences in the field of programming have installed their own AOP tracks. In fact, within an extremely short period of time after its inception the number of papers and theses on the subject has risen beyond what can reasonably be overseen by a single researcher, a growth that is comparable only to the greatest revolutions in the history of programming.

There are several possible explanations to this phenomenon. One is given in Gabriel & Steele's report "The evolution of Lisp" [58], in which the authors describe a general evolution pattern of programming languages. According to this pattern, a successful language requires an acceptance group that is itself successful. Acceptance in turn requires, among other factors, solving a pressing problem and having the right *cachet*. Surely, AOP addresses an important problem, namely the modularization of crosscutting concerns, but the jury is still out on whether AOP can actually solve it (I have certainly written enough about my doubts in this essay). Regarding cachet, AOP seems to have plenty: it has the aura of a leading edge technology, it is supported by a number of OOP luminaries, and it comes with its own fancy lingo. In fact (and referring to [58] for an example of "right cachet"), seen from the outside watching someone do AOP is a little like watching someone own a Mac: one is not really sure of its advantages, but is willing to accept that it is superior technology.

But there is also a simple economics-based explanation for the success of AOP in academia. The last revolution in programming—object-oriented programming—is already more than a generation old, and although many new things have been tried since, none has had comparable impact. As a result, tremendous pressure (in the form of program committees expecting new ideas to be presented and funding agencies waiting for new, promising strains of research to be financed) has built up, and it is quite clear to everyone that the next "big thing" will attract enormous attention and resources. We have all been asking ourselves what this thing could be.

AOP blesses us with a whole concert of innovations: a *new* way of structuring code [64], resulting in *new* kinds of modules with



© 1993 IEEE

new interfaces [34], allowing *new* ways of composition [45], etc. In fact, there are so many new concepts (or, rather, new variants of old concepts) attached to AOP that one cannot help but view it a new programming paradigm. Therefore, from a purely phenomenological standpoint it certainly qualifies as the next big thing in programming, as *the* "post-object programming" mechanism [20].

Economically, the last century ended with the insight that traditional laws are not easily put to rest: the much praised "new economy" turned out to have some really old problems. "Old" modularity, interfaces, and independent development are so fundamental to disciplined programming that it is difficult to imagine how they could be replaced with new variants. Instead, it may be that the belief in AOP is just a belief in "new programming."

8. Conclusion

Given that AOP has set out to modularize crosscutting concerns (its methodological claim), but by its very nature (its mechanics) breaks modularity, I think the current success AOP enjoys is paradoxical. For all I can see, this paradox cannot be resolved by adjusting the mechanics of AOP so as to respect modularity, since then whatever remains of it appears to be only mildly different from other programming techniques currently not thought of as being aspect-oriented. As a way of organizing source code, AOP has its merits, namely the "localization" or "compartmentalizing" [40] of code belonging to one concern in one place, but almost ironically, this requires sacrificing locality ("local in that [a statement] was almost always proximate to the statements executing around it" [22], p. 22) and thus structuredness in Dijkstra's sense. The net effect on program understandability is not indisputable.

I would feel much better about AOP if it gave up its "modularizing the un-modularizable" [40] promise and instead focused on blending its key concepts with those of other programming models[17], reserving its unbridled use for coding problems for which modularity and structuredness are no issues. Alternatively, it could provide us with a definition of what it *is* that is consistent with what it *aims to be good for*.

[17] the coherence requested in the call for this conference!

Epilogue

If you think that my claims are polemic, or those of a cynic, or of an envier, I will agree, yet only to the extent of admitting that they are somewhat overstated. But why am I doing this?

During my works on my doctoral thesis in Medical Informatics back in the early nineties, I looked into Lotfi Zadeh's fuzzy set theory. At that time, the theory was already a generation old, and Charles Elkan's "The paradoxical success of fuzzy logic" [19] had just appeared. In the same year, James Bezdek wrote the following in the editorial of the inaugural issue of the IEEE Transactions on Fuzzy Systems:

> *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. Most technologies are overpromised, more often than not simply to generate funds to continue the work, for funding is an integral part of scientific development; without it, only the most imaginative and revolutionary ideas make it beyond the embryonic stage. Hype is a natural handmaiden to overpromise, and most technologies build rapidly to a peak of hype. Following this, there is almost always an overreaction to ideas that are not fully developed, and this inevitably leads to a crash of sorts, followed by a period of wallowing in the depths of cynicism. Many new technologies evolve to this point, and then fade away. The ones that survive do so because someone finds a good use (= true user benefit) for the basic ideas.* [5]

The timescale he assigned to his observation (specialized to fuzzy models) is depicted in the figure above (reproduced from the editorial [5] with kind permission by the IEEE).

Now reading Bezdek's observation in the context of this essay not only uncovers its author as a cynic, but also as one who hasn't realized that he is as much part of the game as the ones he criticizes for playing it. Yet the enthusiasts should forgive him, for his role is not an unimportant one: the earlier the depth of cynicism is reached, the sooner the true user benefits are discovered, and the sooner AOP can converge to the asymptote of reality. Sometimes, to be truly good, good cops need bad cops, so here I am, ready to take the bashing.

Acknowledgments

I am indebted to Alexander Pretschner, Colin Atkinson, Eric Bodden, and especially Stefan Hanenberg for their suggestions on improving the original manuscript. Also, fourteen anonymous reviewers spent their valuable time helping shape the content of this essay with their detailed comments. Needless to say, views differed greatly, and the sometimes conflicting proposals were difficult to integrate; yet, all expressed opinions and remaining misconceptions are exclusively my own.

Finally, I would like to thank my shepherd Richard P. Gabriel for conveying his sense of essayness to me, and for helping me form my many points into a cohesive argument.

Appendix: Known Attempts to Restore Modularity in Aspect-Oriented Programs

Pointcut Interfaces

A workshop paper by Gudmundson and Kiczales first proposed to reduce the adverse effect AspectJ style AOP has on modularization (information hiding) by moving the pointcut definition closer to the target modules, that is, in proximity of the places where they match [28]. For this purpose, it introduced what its authors called *pointcut interfaces*: basically collections of pointcut signatures (pointcut name plus argument types). According to their suggestion, the definition (implementation) of the pointcut interface, that is, the provision of a concrete pointcut expression, is the responsibility of the module that exports the interface, which can be a class, a package, or a whole program. This means that the declaration and the definition of the pointcut are contained in the same syntactical unit, but outside the aspects that depend on it. Particularly if this unit is a class, it should be comparatively easy to maintain the contract of a pointcut interface (keep the interface constant) when the definition of the class is changed. In order to allow independent development of target modules and aspects, pointcut interfaces should be defined together with all other module interfaces—that is, at the project's outset (with modifications possible as the design evolves).

It is not obvious to me why the idea of pointcut interfaces, which was picked up by other authors (for example, [1, 2, 27, 63], all discussed in the following subsections), appears to have not been pursued further by Kiczales, who now seems to favor other kinds of aspect interfaces ([34]; also discussed below). One possible reason for this may be that in order to have the *linguistically enforceable effect of pointcut awareness*, it is not sufficient that pointcuts reside in the proximity of the target modules they apply to: rather, the aware have to state explicitly what they are aware of. This however would amount to a kind of (target) tagging (comparable to that of classes declaring to implement certain interfaces) that would reduce obliviousness and increase intimacy [20, 63] as well as scattering to levels thought to be incompatible with the original idea of AOP. On the other hand, just keeping pointcut definitions separate from the aspects depending on them, in some third place but without any reference from the code they quantify over, is not a big improvement over keeping them within the aspect.

Open Modules

Aldrich notes that in order to retain some contribution of AOP while at the same time respecting the "intended information hiding boundaries" (aka interfaces) a compromise needs to be found [1, 2]. His so-called Open Modules enables aspects to advise all external uses of program elements exported in the module's provided interface, as well as internal joinpoints that are declared public ("open") by that interface. All other intrusions from aspects, including advice on internal use of published elements, are prohibited. Because in Open Modules all interfaces toward an aspect are explicit, a module can hide the information considered as its secret behind these interfaces, allowing it to evolve independently from aspects.

As Aldrich himself notes, the pointcut interfaces of Open Modules can be thought of as definitions of extension points and the execution of advice at these points as a kind of callback to client-provided functions [1] (which lets the pointcut interfaces appear as required interfaces; cf. discussion in Section 5.1.3). In fact, as pointed out by Aldrich [2], "explicitly exposing internal events in an interface pointcut means a loss of some obliviousness in the distributed development case, since the author of the module must anticipate that clients might be interested in the event." But modularity is all about distributed, independent development (see Parnas [54, 55] and also Section 5.1), and the price for modularity is, once more, the introduction of some "pluggable" procedure call through the back door.[18]

Crosscutting Interfaces

Griswold et al. suggest the introduction of crosscutting interfaces (XPIs) [27] as interfaces "that base code designers 'implement' and that aspects may depend upon" [63]. For this, they assign design rules to XPIs as a kind of contract which the programmers of the base code must observe. At the aspects' side, each XPI comes with a "syntactic part" that exposes the signature of named pointcuts, but not its "hidden implementation" ([27], p. 54), that is, the part that specifies the concrete pointcut expressions. Note that storing the implementation in the interface is somewhat unusual, but must be seen as technical tribute to AspectJ as the language in which XPIs are currently implemented. In fact, the authors of XPIs deliberately wanted AspectJ to remain as is, in order not to subject their work to the lack of adoption that is usual for language modifications [63]. However, this technicality impairs independent module evolution to a certain extent, since the implementation of the crosscutting interface is not part of the implementation of the module (cf. the discussion of Gudmundson and Kiczales's pointcut interfaces above, which suffer from the same problem if the pointcut crosscuts more than a single class).

Griswold et al. note that the decoupling of aspects from their bases through XPIs is comparable to that of a caller from the called module through the provided interface (the API) of that module. In fact, just as a module can remain (and usually is) oblivious of its specific callers (a property called *feature obliviousness* in Sullivan et al. [63]), and although the module needs to prepare for aspects (by providing an XPI), it may remain oblivious of which aspects exactly utilize this interface. However, while the API of a module is usually designed to serve a specific goal

[18] Another criticism of Open Modules expressed by Sullivan et al. is that since the pointcut interface is tied to a single (hierarchical) module, the interface is not crosscutting [63]. One could add that therefore, Open Modules are not aspect oriented. Note that the same argumentation can be applied to the pointcut interfaces suggested by Gudmundson and Kiczales, if they are assigned to single classes.

(the purpose of the module), specification of the XPI requires an a priori decision what the crosscutting behavior of a system is. To address this, Sullivan et al. [63] state they designed their XPIs by "ask[ing] the question, what constraints on the code would shape it to make it relatively easy to write the aspects at hand, as well as support future aspects?" This is exactly the loss of obliviousness noted by Aldrich.

Restoration of Modular Reasoning

Clifton and Leavens note that AOP, although heavily citing Parnas's article [55], is at conflict with it, basically because the obliviousness property contradicts the independent comprehensibility required of a module (a notion called *modular reasoning*) [10, 11].[19] Following the behavioral subtyping analogy they suggest that the effect of aspect application should be checkable so as to not alter the behavior of a module in unexpected ways. For this, they suggest to divide aspects into so-called *spectators* (formerly called *observers* [10]) and *assistants*, the former not changing ("in some well-defined sense") the behavior of the modules they advise, the latter only doing so to an extent made explicit in a suitable "module interconnection specification" to be found "in a well-defined place relative to the client module" [11]. This would retain some of the flexibility associated with the obliviousness property of AOP, and at the same time allow modular reasoning. Support for automated classification of aspects as spectators comes from a whole-program analysis described by Zhao & Rinard [57], which is also capable of pointing to specific problems of assistants. However, it seems that the module interconnection specification suffers from the same problems as the definition of pointcut interfaces spanning several modules (discussed above). Also—although only a marginal note—even spectators can be harmful, if only by spying on local (private) data and passing it on to some other, malevolent party. For a more detailed discussion, see Dantas & Walker's recent work on "Harmless advice" [14].

Aspect-Implied Interfaces

In another attempt to restore modular reasoning, Kiczales and Mezini argue that "aspects cut new interfaces through the primary module structure" [34]. *De facto*, this means that a module is no longer sovereign over its own interfaces; rather, they are forced upon it by system composition. One immediate consequence of this is that modules cannot be changed independently of their assembly, simply because it is unclear which interfaces to keep constant. This of course leads independent development and with it also the module concept *ad absurdum*.

A closer look at Kiczales and Mezini's proposal reveals that they suggest that a tool computes the aspect aware interfaces given a complete system configuration (cf. Footnote 9). While this may allow modular reasoning in the presence of aspects, it does so only *after* the system has been composed, a stage at which modules and their interfaces have done their service and might as well disappear. In fact, Parnas stressed explicitly that *after assembly*, two differently modularized programs *might conceivably be identical* ([55], p.1055)—modularization is a design-time issue! Also, using the same argumentation one could demand that programmers declare all members of all classes public, and only after system composition derive which ones may be declared private.

Last but not least, making interfaces aspect aware by adding the computed information which aspects apply to which members of the provided interface of a class does not really add to the interface, since no client of the class except the aspect from which it has been computed will ever use this information (at least not in a way that is enforced by the compiler). Instead, these aspect-aware interfaces publish implementation details of the aspects, namely which aspect is called where. This in turn means that with the suggested aspect awareness of interfaces, non-locality[20] is much the same as in conventional, procedure-call based implementations, which also need to import the modules containing the called procedures. So in a way, Kiczales and Mezini's proposal seems to support my observation that the problems of AOP cannot be fixed without giving up its distinguishing characteristics.

Aspectual Collaborations

The Aspectual Collaborations of Ovlinger et al. [41, 52, 53] continue previous work on aspectual components, enabling recursive composition of collaboration patterns as modules. So-called aspectual methods extend the usual binding between expected (or required) and provided interfaces by allowing a form of method call interception across modules in which the intercepting and the intercepted method can remain oblivious of each other. Aspectual Collaborations can be used to implement crosscutting concerns such as caching; yet this requires an explicit composition (binding) of the (collaboration representing the) aspect and the (collaboration representing the) base. Therefore, Aspectual Collaborations are more an exploitation of an aspect-oriented mechanism (method call interception, which is also a standard mechanism of metaprogramming) for the purpose of component-based programming, than a general reconciliation of AOP with modularity.

Information Transparency

But there are other ways of addressing crosscutting. One such way is explored by Griswold in his work on *information transparency* [26], a complement to information hiding that allows the ad hoc creation of localized descriptions of a design concern based on similarity of the scattered code implementing it. Griswold describes his approach as relying on naming conventions and other characteristics of code (including the use of particular variables, data structures, etc.) that can be evaluated by a tool, and sometimes even architectural information. One could add that today, source code annotations would lend themselves to explicitly associating code with concerns [56].

Perhaps the greatest advantage of information transparency over aspect-oriented approaches from a technical point of view is that it does not depend on weaving, that is, on the automatic tangling of code designed as separate units, but on its opposite, namely on the automatic disentangling of code designed to go together. In other words: rather than creating a system from different views of it, it creates different views of a system. Whether and how these views can be used to change and extend the system, however, remains an open challenge.

[19] The authors also touch on the verification problem mentioned in Section 6.3.3, namely that aspects can break the postconditions of a method in a way that is outside the control of its programmer [10].

[20] Here, locality refers to the property that all program elements relating to one concern are located in a single place, which is, according to Kiczales & Mezini [34], a necessary condition for modularity.

References

[1] Aldrich, J.: Open Modules: Reconciling extensibility and information hiding. In: *Software Engineering Properties of Languages for Aspect Technologies (SPLAT)*. Workshop at AOSD (2004).

[2] Aldrich, J.: Open Modules: Modular reasoning about advice. In: *ECOOP* (2005) 144–168.

[3] Allan, C. et al.: Adding trace matching with free variables to AspectJ. In: *OOPSLA* (2005) 345–364.

[4] Avgustinov, P. et al.: *Semantics of Static Pointcuts in AspectJ*. Technical Report abc-2006-3 (Oxford University Computing Laboratory, 2006).

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

[6] Bodden, E.: Efficient and expressive runtime verification for Java. In: *Proceedings of the Grand finals of the ACM Student Research Competition 2005*, San Francisco (2005).

[7] Bodden, E., Forster, F., Steimann, F.: Avoiding infinite recursion with stratified aspects. In: *NODe 2006 – Objects, Aspects, Services, the Web*. GI Lecture Notes in Informatics (2006) in press.

[8] Brooks, Jr., F.P.: *The Mythical Man-Month: Essays on Software Engineering*, 20th Anniversary Edition (Addison-Wesley 1995).

[9] Clark, L.R.: A linguistic contribution to goto-less programming. *Commun. ACM* 27:4 (1984) 349–350.

[10] Clifton, C., Leavens, G.T.: Obliviousness, modular reasoning, and the behavioral subtyping analogy. In: *SPLAT* (2003).

[11] Clifton, C., Leavens, G.T.: Observers and assistants: A proposal for modular aspect-oriented reasoning. In: *Workshop on Foundations of Aspect-Oriented Languages (FOAL)* (2002).

[12] Colyer, A., Harrop, R., Johnson, R., Vasseur, A.: AOP will see widespread adoption. *IEEE Software* 23:1 (2006) 72–74.

[13] Constantinides, C., Scotinides, T., Störzer, M.: AOP considered harmful. In: *1st European Interactive Workshop on Aspect Systems (EIWAS)* (2004).

[14] Dantas, D. S., Walker, D.: Harmless advice. In: *POPL, SIGPLAN Not.* 41:1 (2006) 383–396.

[15] Devanbu, P.T., Balzer, B., Batory, D.S., Kiczales, G., Launchbury, J., Parnas, D.L., Tarr, P.L.: Modularity in the new millenium: A panel summary. In: *ICSE* (2003) 723–724.

[16] Dijkstra, E.W.: *A Discipline of Programming*. (Prentice Hall, Englewood Cliffs, New Jersey 1976).

[17] Dijkstra, E.W.: On the role of scientific thought. In: Edsger W. Dijkstra: *Selected Writings on Computing: A Personal Perspective*. (Springer-Verlag 1982).

[18] Douence, R., Motelet, O., Südholt, M.: A formal definition of crosscuts. In: *Proc. of the 3rd Int. Conf. on Metalevel Architectures and Separation of Crosscutting Concerns*. Springer LNCS 2192 (2001) 170–186.

[19] Elkan, C: The paradoxical success of fuzzy logic. *IEEE Expert* 9:4 (1994) 3–8. First appeared at the *1993 National Conference on Artificial Intelligence (AAAI'93)*.

[20] Elrad, T., Filman, R.E., Bader, A.: Aspect-oriented programming: Introduction. *Commun. ACM* 44:10 (2001) 29–32.

[21] Ernst, E.: Separation of concerns and then what? In: *Position papers from the workshop on Aspects and Dimensions of Concern at ECOOP'00* (2000).

[22] Filman, R.E., Elrad, T., Clarke, S., Aksit, M.: *Aspect-Oriented Software Development*. (Addison-Wesley Professional, 2004).

[23] Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. In: *Workshop on Advanced Separation of Concerns at OOPSLA* (2000). Revised reprint appeared in [22].

[24] Filman, R.E.: What is AOP, revisited. In: *Workshop on Multi-Dimensional Separation of Concerns at ECOOP* (2001).

[25] Forster, F., Steimann, F.: AOP and the antinomy of the liar. In: *Workshop on the Foundations of Aspect-Oriented Languages (FOAL) at AOSD* (2006) 47–56.

[26] Griswold, W.G.: Coping with crosscutting software changes using information transparency. In: *Proceedings of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*. Springer LNCS 2192 (2001) 250–265.

[27] Griswold, W.G., Shonle, M., Sullivan, K., Song, Tewari, N., Cai, Y., Rajan, H.: Modular software design with crosscutting interfaces. *IEEE Software* 23:1 (2006) 51–60.

[28] Gudmundson, S., Kiczales, G.: Addressing practical software development issues in AspectJ with a pointcut interface. In: *Advanced Separation of Concerns, Workshop at ECOOP* (2001).

[29] Gybels, K., Brichau, J.: Arranging language features for more robust pattern-based crosscuts. In: *AOSD* (2003) 60–69.

[30] Harrison, W.H., Ossher, H.: Subject-Oriented Programming (A critique of pure objects). In: *OOPSLA* (1993) 411–428.

[31] Hilsdale, E., Hugunin, J.: Advice weaving in AspectJ. In: *AOSD* (2004) 26–35.

[32] Hogg, J., Lea, D., Wills, A., de Champeaux, D., Holt, R. C.: The Geneva convention on the treatment of object aliasing. *OOPS Messenger* 3:2 (1992) 11–16.

[33] Kiczales, G., Lamping, J., Lopes, C.V., Maeda, C., Mendhekar, A., Murphy, G.C.: Open implementation design guidelines. In: *ICSE* (1997) 481–490.

[34] Kiczales, G., Mezini, M.: Aspect-oriented programming and modular reasoning. In: *ICSE* (2005) 49–58.

[35] Kiczales, G., Mezini, M.: Separation of concerns with procedures, annotations, advice and pointcuts. In: *ECOOP* (2005) 195–213.

[36] Klaeren, H., Pulvermueller, E., Rashid, A., Speck, A.: Aspect composition applying the design by contract principle. In: *Proceedings of the GCSE 2000, Second International Symposium on Generative and Component-Based Software Engineering* (2000) 57–69.

[37] Klose, K., Ostermann, K.: Back to the future: pointcuts as predicates over traces. In: *Workshop on Foundations of Aspect-Oriented Languages (FOAL) at AOSD* (2005).

[38] Kojarski, S., Lorenz, D.H.: Modeling aspect mechanisms: a top-down approach. In: *ICSE* (2006) 212–221.

[39] Laddad, R.: AOP and metadata: A perfect match. In: *AOP@work* (http://www-128.ibm.com/developerworks/java, 2005).

[40] Lesiecki, N.: *Improve modularity with aspect-oriented programming* (http://www-128.ibm.com/developerworks/java/library/j-aspectj/, 2002).

[41] Lieberherr, K.J., Lorenz, D.H., Ovlinger, J.: Aspectual collaborations: combining modules and aspects. *The Computer Journal* 46:5 (2003) 542–565.

[42] Lopes, C.V., Dourish, P., Lorenz, D.H., Lieberherr, K.: Beyond AOP: toward naturalistic programming. In: *OOPSLA'03 Special Track on Onward! Seeking New Paradigms & New Thinking*. ACM (2003) 198–207.

[43] Lorenz, D.H., Skotiniotis, T.: *Extending design by contract for aspect-oriented programming*. http://arxiv.org/abs/cs.SE/0501070.

[44] Masuhara, H., Kiczales, G.: Modeling crosscutting in aspect-oriented mechanisms. In: *ECOOP* (2003) 2–28.

[45] Mehner, K., Rashid, A.: *Towards a generic model for AOP (GEMA)*. Technical Report CSEG/1/03, Computing Department, Lancaster University, UK (2003).

[46] Mikhajlov, L., Sekerinski, E.: A Study of the fragile base class problem. In: *ECOOP* (1998) 355–382.

[47] Murphy, G., Schwanninger, C.: Aspect-oriented programming. *IEEE Software* 23:1 (2006) 20–23.

[48] Nagy, I., Bergmans, L.: Towards semantic composition in aspect-oriented programming. In: *1st European Interactive Workshop on Aspects in Software (EIWAS)*. (Berlin, Germany 2004).

[49] Nordberg, III., M. E.: Aspect-oriented dependency inversion. In: *Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA* (2001).

[50] Ossher, H., Tarr, P.: Hyper/J: Multi-dimensional separation of concerns for Java. In: *ICSE* (2001) 729–730.

[51] Ostermann, K., Mezini, M., Bockisch, C.: Expressive pointcuts for increased modularity. In: *ECOOP* (2005) 214–240.

[52] Ovlinger, J.: Modular programming with aspectual collaborations. In:*OOPSLA 2002 Doctoral Symposium* (2002) 16–17.

[53] Ovlinger, J.: *Combining Aspects and Modules*. PhD Thesis (College of Computer and Information Science, Northeastern University, Boston, USA 2004).

[54] Parnas, D.L.: Information distribution aspects of design methodology. In: *Information Processing 71, Proceedings of the IFIP Congress* 1 (North-Holland, 1972) 339–344.

[55] Parnas, D.L.: On the criteria to be used in decomposing systems into modules. *Commun. ACM* 15:12 (1972) 1053–1058.

[56] Revelle, M., Broadbent, T., Coppit, D.: Understanding concerns in software: insights gained from two case studies. In: *IWPC* (2005) 23–32.

[57] Rinard, M., Salcianu, A., Bugrara, S.: A classification system and analysis for aspect-oriented programs. In: *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2004) 147–158.

[58] Steele, Jr., G.L., Gabriel, R.P.: *The Evolution of Lisp*. http://dreamsongs.com/NewFiles/HOPL2-Uncut.pdf

[59] Steimann, F.: Why most domain models are aspect free. In: *5th Aspect-Oriented Modeling Workshop AOM at UML* (2004); revised version appeared as Ref. [60].

[60] Steimann, F.: Domain models are aspect free. In: *MoDELS 2005, 8th International Conference on Model Driven Engineering Languages and Systems* (2005) 171–185.

[61] Störzer, M., Graf, J.: Using pointcut delta analysis to support evolution of aspect-oriented software. In: *21st IEEE International Conference on Software Maintenance* (2005) 653–656.

[62] Stolz, V., Bodden, E.: Temporal assertions using AspectJ. In: *RV'05 — 5th Workshop on Runtime Verification* (Edinburgh, Scotland, UK, 2005).

[63] Sullivan, K.J., et al.: Information hiding interfaces for aspect-oriented design. In: *Proc. 10th European Software Eng. Conf. Held Jointly with 13th ACM SIGSOFT Int'l Symp. Foundations of Software Eng. (ESEC/FSE 2005)* (ACM Press, 2005) 166–175.

[64] Tourwé, T., Brichau, J., Gybels, K.: On the existence of the AOSD-evolution paradox. In: *Workshop on Software-Engineering Properties of Languages for Aspect Technologies (SPLAT).* Workshop at AOSD (2003).

[65] Walker, R.J., Viggers, K.: Implementing protocols via declarative event patterns. In: *SIGSOFT FSE* (2004) 159–169.

[66] Wulf, W., Shaw, M.: Global variable considered harmful. *SIGPLAN Notices* 8:2 (1973) 28–34.

[67] Zhao, J., Rinard, M.C.: Pipa: A behavioral interface specification language for AspectJ. In: *Fundamental Approaches to Software Engineering, 6th International Conference* (2003) 150–165.