Multiple Concerns in Aspect-Oriented Language Design:

A Language Engineering Approach to Balancing Benefits, with Examples

Gary T. Leavens
Department of Computer Science,
Iowa State University,
229 Atanasoff Hall, Ames, IA 50011 USA
leavens@cs.iastate.edu

Curtis Clifton
Computer Science and Software Engineering,
Rose-Hulman Institute of Technology,
5500 Wabash Ave, Terre Haute, IN 47803 USA
clifton@rose-hulman.edu

Abstract

Some in the aspect-oriented community view a programming language as aspect-oriented only if it allows programmers to perfectly eliminate scattering and tangling. That is, languages that do not allow programmers to have maximal quantification and perfect obliviousness are not viewed as aspect-oriented. On the other hand, some detractors of aspect-oriented software development view maximal quantification and perfect obliviousness as causing problems, such as difficulties in reasoning or maintenance.

Both views ignore good language design and engineering practice, which suggests trying to simultaneously optimize for several goals. That is, designers of aspect-oriented languages that are intended for wide use should be prepared to compromise quantification and obliviousness to some (small) extent, if doing so helps programmers solve other problems. Indeed, balancing competing requirements is an essential part of engineering. Simultaneously optimizing for several language design goals becomes possible when one views these goals, such as minimizing scattering and tangling, not as all-or-nothing predicates, but as measures on a continuous scale. Since most language design goals will only be partially met, it seems best to call them "concerns."

Categories and Subject Descriptors D.1.m [*Programming Techniques*]: Miscellaneous—aspect-oriented programming; D.2.10 [*Software Engineering*]: Design—methodologies; D.3.0 [*Programming Languages*]: General—aspect-oriented language design; D.3.3 [*Programming Languages*]: Language Constructs and Features—control structures, modules, packages, procedures, functions and subroutines, advice, spectators, assistants, aspects

General Terms Languages, Design

Keywords Aspect-oriented programming language design, concerns, goals, quantification, obliviousness, scattering, tangling, ease of reasoning, ease of maintenance, compromise, tradeoff, balance.

Workshop SPLAT '07 March 12, 2007, Vancouver, British Columbia, Canada.

1. Introduction

What makes a programming language aspect-oriented? Many agree with Filman and Friedman's definition [11, 12], that the language must have both quantification and obliviousness. They define "*quantified* program statements" as those that "have effect on many places in the underlying code" [12, Section 2.2]. Furthermore: "*Obliviousness* states that one can't tell that the aspect code will execute by examining the body of the base code" [12, Section 2.3] (italics added).

One of several authors who think well of Filman and Friedman's definition is Steimann [28]. In his *OOPSLA 2006* essay, "The Paradoxical Success of Aspect-Oriented Programming," he uses Filman and Friedman's definition as part of his argument that while aspect-oriented programming aims to "modularize crosscutting concerns," "Its very nature . . . breaks modularity" [28, Section 8].

1.1 Views of Aspect-Oriented Languages

Steimann's essay helped crystallize our thinking about the goals of aspect-oriented language design, in particular about the portion of such languages that provides quantification and obliviousness. In particular Leavens thought about that essay carefully in his role as the "discussant" for Steimann's presentation at *OOPSLA*. This position paper is, in large part, an expanded reflection of that discussion's key point.

This paper is also influenced by many conversations about aspect-oriented language design that we have had with others in the programming languages community. In these conversations, we have found that several members are openly skeptical about the benefits of aspect-oriented languages, saying that they think aspect-oriented programming is overhyped or that there are only 4 or 5 real uses for aspects. Others express worries about how software engineering concerns are affected by quantification and especially obliviousness. Steimann's *OOPSLA* essay discusses many of these worries, and is rare only in that it is published.

We are also influenced by many conversations with people in the aspect-oriented community, especially at the *FOAL* workshop that we help organize, and at the *AOSD* conference. In these conversations, people in the aspect-oriented community have emphasized the extreme importance of quantification and obliviousness for aspect-oriented languages. Indeed we have found that many are frankly dismissive of language designs that limit quantification or that do not have perfect obliviousness.

Our thesis is that both these viewpoints are short-sighted. First, the larger programming language community should recognize the value of quantification and obliviousness in promoting locality (i.e., reducing scattering) and separation of concerns (i.e., reducing tangling), which are the main contributions of the aspect-oriented paradigm. Second, the aspect-oriented community should recog-

nize the value of language designs that try to simultaneously optimize several goals (such as ease of static reasoning about correctness), even if doing so compromises quantification and obliviousness to some small extent.

1.2 Language Design Concerns

Ideally, a language design would perfectly fulfill all of its goals. Sometimes technical innovations make this happen; when they do, such innovations are justly celebrated. For example, the Hindley-Milner type system [25] simultaneously achieves the goals of type safety and brevity of notation, since its use of type inference avoids the need for type declarations.

However, it very often happens that a language design must make compromises among its goals. For example, C++ allows programmers to more efficiently control space allocation by deciding when objects should be stack or heap allocated, even though this makes the language more complex. The design of C++ trades some simplicity for fine-grained control of space allocation. Because of such compromises, the goals of a language design will not all be perfectly achieved but must be balanced and optimized as a group, with appropriate weightings. To avoid cheap criticism that a language design has "not perfectly achieved its goals" we suggest using the phrase "design concerns", or just "concerns" instead of "goals".

We believe that thinking about "concerns" of language design will also make it easier for language designers to keep the following two points in mind:

- there are multiple concerns in a language design, and
- all concerns should be addressed, jointly, as much as possible.

Of course, it is also scientifically interesting to experiment with languages that focus on a very narrow sets of features and concerns. Such efforts fit into our view of language design, by weighting some design concerns very heavily. For example, an aspect-oriented language design typically puts a very high weight on the concerns of code locality (reducing scattering) and separation of concerns (reducing tangling). This is fine, but we argue that the community should see value in aspect-oriented language designs that put somewhat less weight on these concerns and somewhat more weight on other design concerns. Such a weighting may be important for aspect-oriented languages that aspire to more widespread, general use.

1.3 Outline

In the rest of this paper we describe what some of these other concerns are for the aspect-oriented language designer, discuss how one can think about such concerns as measures, which allows them to be compromised to a measurable extent, and discuss some of the ways that quantification and obliviousness have been balanced with other concerns in the literature. These examples of compromise are intended as positive examples to provoke similar but better efforts in the future. We are not promoting any particular compromise, but rather the idea that some (small amount of) compromise is acceptable, and indeed even necessary.

2. What Other Concerns?

What should be the concerns of aspect-oriented language designers? For a start, language designers should think about concerns that interact strongly with the traditional aspect-oriented concerns of promoting locality and separating program concerns. The pointcut and advice mechanisms of languages like AspectJ [2, 18, 21] are ways to help promote these concerns, as are any mechanisms that promote quantification and obliviousness. So aspect-oriented language designers should consider any programming language or software engineering concerns that interact with quantification and obliviousness. From our own work and our discussions with people in the programming language community, including Steimann [28], it seems the most important such concerns are:

- ease of maintenance, and
- ease of reasoning (e.g., about correctness or efficiency).

It is precisely these important software engineering concerns that are most affected by the extra indirection that aspect-oriented techniques bring.

The concerns of easing maintenance and reasoning are broad, general concerns. It is thus helpful to break them into smaller concerns for purposes of evaluating a design. For example, the broad concern of easing maintenance is aided by the concern of allowing programmers to control coupling. (Coupling is the degree to which one part of the system depends on others.) The broad concern of easing reasoning (e.g., about correctness or efficiency) is aided by the concerns of allowing programmers to easily enforce information hiding and data encapsulation. Features that help these concerns should also help ease maintenance. The ease with which programmers can find code that may affect the values of a field or that may cause various transfers of control (such as throwing exceptions or looping) also affects both of these broad concerns. As others have noted, these concerns are not completely orthogonal to the traditional aspect-oriented concerns. For example, to the extent that a maintenance activity aligns with code modularization, support for locality of concerns strengthens support for the ease-of-maintenance concern [19].

3. Measuring Concerns

In thinking about how to simultaneously optimize concerns for a language design, it is important to keep two points in mind.

- Concerns should be measured on continuous scales.
- These scales should measure how easily one can write good programs.

Together, these points allow us to evaluate a language design in a multidimensional design space. We discuss each of these points in turn below.

3.1 Continuous Scales

Small amounts of compromise in a design concern can only be made if concerns are not thought of as all-or-nothing goals, but rather as continuous scales measuring what is to be achieved. By *continuous*, we mean that a scale or quality is not all-or-nothing (0 or 1), but that it permits some gradation and fractional values.

Most properties that seem inherently all-or-nothing can be made into continuous measures by simple changes in wording and measurement. For example, obliviousness as defined by Filman and Friedman seems like an all-or-nothing property: "one can't tell that the aspect code will execute by examining the body of the base code" [12, Section 2.3]. One can change this into a continuous scale by measuring the extent to which the base code is coupled to the aspect code. Similar remarks apply to the design concern that motivates obliviousness: separation of program concerns; thus one should measure how easy it is for programmers to separate concerns (e.g., by using advice). Similarly, quantification can be measured on a continuous scale by measuring what fraction of the types of places in the base code, or events in the execution of the base code, can be advised.

Concerns phrased as "ease of doing X", like "ease of reasoning," are already placed on a continuous scale.

Evaluating concerns on a continuous scale is necessarily more difficult than using simple predicates. However, while measurable

concerns may be difficult to evaluate with precision, we have found that language designers can get initial estimates for the measure of a given concern by doing case studies with their new language [9].

3.2 Measuring the Good

Many of the objections to aspect-oriented languages are worries that any use of aspect-oriented features will result in "bad" programs. What constitutes a "bad" program varies, but usually involves the program being incomprehensible, unmaintainable, overly long, overly complex, etc. A published example is Steimann's complaint about using aspects to address scattering. He says [28, p. 488], that use of aspects to avoid scattering makes programs bad in the sense that it hurts information hiding. Information hiding suffers because moving "scattered subfunctions" from a module (data abstraction) "into an aspect" introduces data dependencies between them. Such data dependencies are bad because, even if they are captured in an interface, "[i]ndependent evolability is therefore compromised" [28, p. 488]. Steimann also complains that use of advice causes "confusion" since it makes the control flow of a program less obvious [28, p. 490].

Such complaints recall the debates about the "goto" statement in the 1970s. Dijkstra famously [10] complained that any use of goto statements made programs hard to read and understand. However, Knuth [20] made the point that some uses of goto statements were necessary to address other program concerns, such as efficiency, reliability, and conciseness. Understanding of these other concerns led to the development of features that addressed them, such as exception handling [15, 23]. From our perspective, one can see Dijkstra's complaints as being focused on a single concern, namely understandability, and measuring the badness of programs by how much that feature was used.[1] On the other hand, the positive developments, such as exception handling mechanisms, resulted from language designers thinking about and measuring multiple concerns. That is, language designers thought not only about understandability, but also about the other program concerns that were being addressed by various uses of the goto statement.

How can language designers keep multiple concerns in mind, and not become sidetracked into endless debates about bad programs? Measuring what harm programmers might do, or be prevented from doing, does not help. Instead, language designers should measure the ease or extent to which programmers can write good programs.

Another reason for focusing on such positive measures was summarized by Flon [14], who said that no language can prevent programmers from writing bad programs. The implication is that since programmers are always able to abuse language mechanisms, write bad code, or avoid writing good code, it does no good to try to measure the extent to which a language "discourages" them from doing bad things. (The only exception is that a language could prohibit some bad coding practices completely when the cost was judged small enough; for example, by eliminating the goto statement entirely, or by prohibiting type errors.)

Therefore, it is vital to find positive measures that are aimed at addressing multiple program concerns. That is, language designers should use several measures that are aimed at helping programmers write good programs, such as ease of separating concerns or ease of reasoning.

3.3 A Multidimensional Design Space

Together, (1) positively stated concerns (2) measured on continuous scales give aspect-oriented language designers a multidimensional design space in which to work. Fig. 1 represents one such space. Several proposals for aspect-oriented languages have begun exploring such design spaces. We discuss this work in the next section.

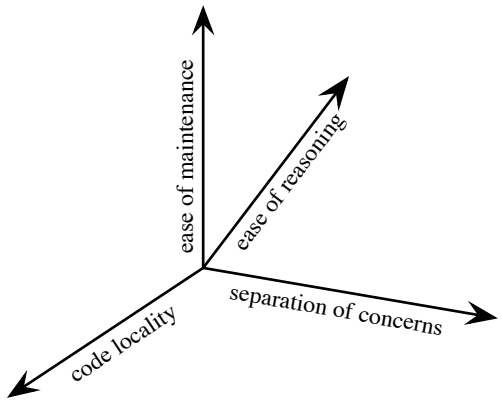

Figure 1. Multiple concerns in the language design space.

4. Compromises in the Literature

We illustrate our prescription for aspect-oriented language design by describing how some authors have balanced the aspect-oriented concerns of promoting locality (quantification) and separation of programmer concerns (obliviousness) in order to achieve better measures for other software engineering concerns.

4.1 Gudmundson and Kiczales

Gudmundson and Kiczales's paper "Addressing Practical Software Development Issues in AspectJ with a Pointcut Interface" [17] is a fascinating early presentation of such a compromise. The paper addresses the "problems of large-scale systems, team development, and maintenance in the context of AspectJ . . . programming" [17, Section 1]. The paper advocates having each class and package declare and export an interface consisting of named pointcuts. The idea is that each class and package agrees to support that interface, and that aspects only advise such exported pointcuts.[2] The authors note that their design balances obliviousness with other design concerns, saying: "By having the pointcut interface exported by the base code, we are clearly not making the aspects invisible" [17, Section 4]. Note that the compromise in obliviousness is small, since there is no direct coupling between the base code and aspects. Since it strongly supports our position on the value of compromise, it seems worth quoting in full the following paragraph defending their design [17, Section 4]:

> "Since obliviousness is such a desirable property, we feel that some additional justification of our position is required. In our experience and in several studies, special care must be taken to ensure that the right join points are exposed. While this might seem like a minor refactoring in the context of a small case study, it must be viewed objectively. Without having something to force the right join points to be exposed, there is no reason to expect this to happen naturally. Our pointcut interface serves as this

[1] Dijkstra's first sentence contains this measure "the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce" [10, p. 147].

[2] However, the authors do not propose enforcing this style by a language design change.

constraint. The obliviousness perspective does not allow for any influence to be exerted on the base code's structure, and thus cannot ensure that the join points will be exposed."

4.2 Griswold *et al.*'s XPIs

Closest to Gudmundson and Kiczales's idea of pointcut interfaces is the work on crosscut programming interfaces (XPIs) by Griswold *et al.* [16, 29]. Besides the maintenance difficulties described by Gudmundson and Kiczales, Griswold *et al.* also address the problem of ease of reasoning. They break ease of reasoning into two subconcerns: ease of reasoning about correctness, and minimizing the amount of code one needs to examine to "identify the relevant join points" [16, p. 51]. The XPIs declare a set of scoped, abstract pointcut definitions and a "partial implementation" of them [16, p. 52]. The partial implementation maps the abstract pointcuts (via patterns) to base code, and gives a set of "constraints" or "design rules" that describe how they may be used by aspects advising them. The base code does not itself contain the pointcut definitions (which are found in the XPIs), hence there is no compromise on obliviousness. However, advice must only advise the pointcuts from the XPIs in conformance with the stated constraints,[3] which limits quantification to some extent. The authors argue that this slight loss of locality (quantification) allows programmers to have better control over maintainability and ease of reasoning, and give two case studies to support these claims.

4.3 Aldrich's Open Modules

In contrast to the work described above, Aldrich's Open Modules [1] is a language design and not a coding convention. Aldrich's paper addresses both of the concerns raised in Section 2: ease of maintenance and ease of reasoning (especially about correctness). This paper is notable for making explicit compromises between quantification and obliviousness (called "openness"), and these other concerns (called "modularity"): "The goals of openness and modularity are in tension . . ., and so we try to achieve a compromise between them" [1, Section 2]. The language allows programmers to export pointcuts from modules, and advice that is external to a module can only be applied to the exported pointcuts. The language also restricts quantification, in that internal communication events (e.g., calls within a module) cannot be advised by external clients. These features together allow programmers to determine if a change to their aspects or base code might cause maintenance or reasoning problems. Note that, despite the small loss of quantification and a user-determined loss of obliviousness, modules are not strongly coupled to advice, as they do not know what, if any, advice is being applied.

4.4 Clifton and Leavens

In prior work, we introduced a distinction between spectators (originally called observers) and assistants [6, 7]. That work addresses the concerns of separate compilation and ease of reasoning. The language proposed makes users categorize AspectJ-style aspects as either spectators, which are not allowed to change the base program's computation, or assistants, which have no such limitations. Spectators are unlimited in their use of quantification and base code is oblivious to their existence. However, assistants must be acknowledged by the base code, which limits both quantification and obliviousness. Other features, such as aspect maps, make up for some of the loss of quantification and obliviousness.

Clifton's Ph.D. work focuses on a static analysis and annotations that allow the language to check that spectators really have no effect on the base program [5, 8]. A more recent design drops the requirement that base code explicitly accepts the advice of the assistants, and thus has no loss of obliviousness [9].

4.5 Other related work

We briefly review other relevant language designs below.

AspectJ [2, 18] limits quantification, since some events that are internal to a method, such as the get and set of local variables, are not considered to be join points. This limitation makes some kinds local reasoning within a method easier.

Bergmans and Aksit's Composition Filters [3, 4] is a language mechanism that allows programmers to declaratively describe how messages sent to objects are transformed. Composition Filters has limited quantification, compared to AspectJ, its "superimpositions" only apply at object interfaces, and cannot transform an object's internal messages. This loss of quantification is explicitly claimed to promote encapsulation [4, Section 5.5.3].

Similarly Tarr *et al.*'s multi-dimensional separation of concerns [30] has a similar limitation on quantification which enhances encapsulation, since it treats the "primitive units" of composition "as indivisible" [30, p. 111]. By choosing the granularity of different primitive units, the tradeoff between quantification and encapsulation can be adjusted [30, pp. 115–116].

Larochelle *et al.* describe a design for hiding join points [22]. It allows programmers to restrict quantification as an aid to easing modification and reasoning.

Ossher [26] also addresses the concerns of easing modification and reasoning. His mechanism allows a class to confirm or deny that a particular pointcut applies at a join point (such as a method or class). It thus allows programmers to trade a small amount of loss in obliviousness, in return for gains in ease of modification and reasoning.

Filman and Havelund [13] examine two aspect-oriented languages. They identify a large number of software engineering concerns that should be addressed in aspect-oriented language design, and consider the extent to which two languages deal with them.

Lopez-Herrejon and Batory [24] propose a model for composing aspects that "bounds the scope of quantification" to support ease of maintenance and "incremental development."

5. Conclusions

The idea of thinking about multiple concerns during programming language design is itself very aspect-oriented [30]. We have not proposed any meta-linguistic techniques for aspect-oriented language design in this paper. Such a way of composing separate concerns[4] has long been the holy grail of language design, and thus we leave it as future work. However, we have emphasized that good design principles apply equally to language designs, and thus our point is that a good language should strike an appropriate (weighted) compromise among all concerns.

As described in Section 4 on the previous page, many aspect-oriented language designs already involve some compromises in the aspect-oriented concerns of promoting locality (quantification) and separation of program concerns (obliviousness). Thus no one should complain that a language is not aspect-oriented, simply because it does not have maximal quantification and perfect obliviousness. By viewing these concerns as continuous measures, as opposed to all-or-nothing properties, one can discuss the extent to which aspect-oriented languages make such compromises, or allow their users to make such compromises for themselves.

Viewing these concerns as continuous measures also helps explain the "paradox" of aspect-oriented programming [28], since by

[3] These constraints are enforced using AspectJ's `declare error` mechanism. Thus enforcement does not require a change to AspectJ.

[4] These language design concerns are often thought of as monads by semantically-oriented language designers [27].

making slight compromises in quantification or obliviousness, language designers can promote other software engineering concerns, while only giving up a small amount of what makes them aspect-oriented. Put another way, aspect-oriented languages can promote increased locality (i.e., reduced scattering) and separation of concerns (i.e., reducing tangling), while still promoting ease of maintenance and ease of reasoning. By keeping in mind that all these concerns can be compromised in small amounts, language designers can find creative ways to simultaneously promote them all.

Acknowledgments

Thanks to Hridesh Rajan and the *SPLAT* program committee for comments on earlier drafts. Thanks to Friedrich Steimann for his OOPSLA 2006 essay [28], which greatly stimulated our thinking. Thanks to Richard Gabriel for asking Leavens to discuss Steimann's essay at *OOPSLA*. Thanks to many others in the aspect-oriented community, including Hridesh Rajan, James Noble, William Griswold, Kevin Sullivan, Gregor Kiczales, Mira Mezini, Klaus Ostermann, Jonathan Aldrich, Don Batory, Robert Filman, Ehud Lamm, Mitchell Wand, and other participants at the *FOAL* and *SPLAT* workshops, with whom we have had illuminating discussions. The work of both authors was supported in part by the US National Science Foundation under grant CCF-048078. Leavens's work was also supported in part by NSF grant CCF-0429567.

References

[1] J. Aldrich. Open modules: Modular reasoning about advice. In A. P. Black, editor, *ECOOP 2005 — Object-Oriented Programming 19th European Conference, Glasgow, UK*, volume 3586 of *Lecture Notes in Computer Science*, pages 144–168. Springer-Verlag, Berlin, July 2005.

[2] AspectJ Team. The AspectJ programming guide. Version 1.5.3. Available from `http://eclipse.org/aspectj`, 2006.

[3] L. Bergmans and M. Aksit. Composing crosscutting concerns using Composition Filters. *Commun. ACM*, 44(10):51–57, Oct. 2001.

[4] L. Bergmans and M. Aksit. Principles and design rationale of Composition Filters. In R. Filman, T. Elrad, S. Clarke, and M. Aksit, editors, *Aspect-Oriented Software Development*. Addison-Wesley, 2004.

[5] C. Clifton. A design discipline and language features for modular reasoning in aspect-oriented programs. Technical Report 05-15, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011, July 2005.

[6] C. Clifton and G. T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In G. T. Leavens and R. Cytron, editors, *FOAL 2002 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2002*, number 02-06 in Technical Reports, pages 33–44. Department of Computer Science, Iowa State University, Apr. 2002.

[7] C. Clifton and G. T. Leavens. Spectators and assistants: Enabling modular aspect-oriented reasoning. Technical Report 02-10, Iowa State University, Department of Computer Science, Oct. 2002.

[8] C. Clifton and G. T. Leavens. A design discipline and language features for formal modular reasoning in aspect-oriented programs. Technical Report 05-23, Dept. of Computer Science, Iowa State University, Ames, IA, 50011, Jan. 2005.

[9] C. Clifton, G. T. Leavens, and J. Noble. Ownership and effects for more effective reasoning about aspects. Technical Report 06-35, Dept. of Computer Science, Iowa State University, Ames, IA, 50011, Dec. 2006. To appear in *ECOOP 2007*.

[10] E. W. Dijkstra. Go to statement considered harmful. *Commun. ACM*, 11(3):147–148, Mar. 1968.

[11] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *OOPSLA 2000 Workshop on Advanced Separation of Concerns*, Minneapolis, MN, Oct. 2000.

[12] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In M. Akşit, S. Clarke, T. Elrad, and R. E. Filman, editors, *Aspect-Oriented Software Development*. Addison-Wesley, Reading, MA, 2004.

[13] R. E. Filman and K. Havelund. The effect of AOP on software engineering, with particular attention to OIF and event quantification. In *SPLAT '03*, Mar. 2003. `http://tinyurl.com/2euk95`.

[14] L. Flon. On research in structured programming. *ACM SIGPLAN Notices*, 10(10):16–17, Oct. 1975.

[15] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Commun. ACM*, 18(12):683–696, Dec. 1975.

[16] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, pages 51–60, Jan/Feb 2006.

[17] S. Gudmundson and G. Kiczales. Addressing practical software development issues in AspectJ with a pointcut interface. In *ECOOP 2001 Workshop on Advanced Separation of Concerns*, 2001.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference, Budapest Hungary*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, Berlin, June 2001.

[19] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *Proc. of the 27th International Conference on Software Engineering*, pages 49–58. ACM, 2005.

[20] D. E. Knuth. Structured programming with goto statements. *ACM Comput. Surv.*, 6(4):261–301, Dec. 1974.

[21] R. Laddad. *AspectJ in Action*. Manning Publications Co., Grennwich, Conn., 2003.

[22] D. Larochelle, K. Scheidt, K. Sullivan, Y. Wei, J. Winstead, and A. Wood. Join point encapsulation. In *SPLAT '03*, Mar. 2003. `http://tinyurl.com/26onl4`.

[23] B. H. Liskov and A. Snyder. Exception handling in CLU. *IEEE Transactions on Software Engineering*, SE-5(6):546–558, Nov. 1979.

[24] R. E. Lopez-Herrejon and D. Batory. Improving incremental development in AspectJ by bounding quantification. In *SPLAT '05*, Mar. 2005. `http://tinyurl.com/25shp3`.

[25] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, Dec. 1978.

[26] H. Ossher. Confirmed join points. In *SPLAT '05*, Mar. 2005. `http://tinyurl.com/2xzffu`.

[27] G. L. Steele, Jr. Building interpreters by composing monads. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon*, pages 472–492. ACM, Jan. 1994.

[28] F. Steimann. The paradoxical success of aspect-oriented programming. In *OOPSLA 2006: Proceedings of the 21st International Conference on Object-oriented Programming Systems, Languages, and Applications*, ACM SIGPLAN Notices, pages 481–497, New York, NY, Oct. 2006. ACM.

[29] K. Sullivan, W. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information hiding interfaces for aspect-oriented design. In *Proc. of the 13th ACM SIGSOFT symposium on the Foundations of software engineering (FSE-13)*, pages 166–175, Lisbon, Portugal, May 2005. ACM Press.

[30] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. Sutton Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 107–119, New York, NY, 1999. ACM.