Obliviousness, Modular Reasoning, and the Behavioral Subtyping Analogy

Curtis Clifton and Gary T. Leavens

TR #03-15
December 2003

Keywords: aspect-oriented programming, modular reasoning, behavioral subtyping, obliviousness, AspectJ language, superimposition, adaptation.

2002 CR Categories: D.1.5 [*Programming Techniques*] Object-oriented programming — aspect-oriented programming; D.2.1 [*Requirements/Specifications*] Languages — JML; D.2.4 [*Software/Program Verification*] Programming by Contract; D.3.2 [*Programming Languages*] Language Classifications — object-oriented languages, Java, AspectJ; D.3.3 [*Programming Languages*] Language Constructs and Features — control structures, modules, packages, procedures, functions and subroutines, advice, spectators, assistants, aspects.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

Obliviousness, Modular Reasoning, and the Behavioral Subtyping Analogy

Curtis Clifton and Gary T. Leavens[1]

Iowa State University, 226 Atanasoff Hall, Ames IA 50011, USA,
{cclifton,leavens}@cs.iastate.edu,
WWW home page: http://www.cs.iastate.edu/~{cclifton,leavens}

Abstract. The obliviousness property of AspectJ-like languages conflicts with the ability to reason about programs in a modular fashion. This can make debugging and maintenance difficult.

In object-oriented programming, the discipline of behavioral subtyping allows one to reason about programs modularly, despite the oblivious nature of dynamic binding; however, it is not clear what discipline would help programmers in AspectJ-like languages obtain modular reasoning. Behavioral subtyping was born out of the stories programmers were telling in their object-oriented programs and how they reasoned about them. Programmers use AspectJ-like languages to tell what we call "superimposition" and "adaptation" stories. Thus, a discipline of modular reasoning for an AspectJ-like language must account for both sorts of stories.

We describe the modular reasoning problem for AspectJ-like languages. We do not yet have a solution, but concisely articulate the issues involved.

1 Introduction

Much of the work on aspect-oriented programming languages refers to the work of Parnas [31]. Parnas argues that the modules into which a system is decomposed should provide benefits in three areas (p. 1054):

> The benefits expected of modular programming are:
> 1. managerial—development time should be shortened because separate groups would work on each module with little need for communication;
> 2. product flexibility—it should be possible to make drastic changes to one module without a need to change others;
> 3. comprehensibility—it should be possible to study the system one module at a time. The whole system can therefore be better designed because it is better understood.

It seems clear that aspect-oriented languages provide many benefits related to Parnas's second point [13, 18]. Among these benefits are a reduction in tangling

of code for unrelated concerns, a reduction in the scattering of code for a single concern, and the ability to modularize code on multiple axes, including the separation of code that implements functional and non-functional concerns.

Parnas's third point has received less attention in discussions of aspect-oriented languages and so is the primary concern of this paper. AspectJ-like languages [3, 17, 10], which have *dynamic join point models* that allow binding of advice based on dynamic events during a program's execution. AspectJ-like languages do not provide Parnas's third benefit in general, because they require that systems be studied in their entirety.

The whole-program analysis required by AspectJ-like languages is due to the form of obliviousness they allow. Filman and Friedman define *obliviousness* as the execution of additional aspect-oriented code, A, without effort by the programmer of the *client* code that A cross-cuts [14]. Filman and Friedman's paper argues that obliviousness is an essential property of aspect-oriented programming languages. But because of the obliviousness of AspectJ-like languages, it is difficult for programmers to find the code that may be executed at any given point in a program. This presents difficulties for debugging and for code understanding. This problem is currently being addressed via tool support (e.g., in Eclipse[1], the QJBrowser[2], and plug-ins for various IDEs) instead of via the language. Such tools perform the necessary whole-program analysis to direct the programmer to the applicable aspects that affect pieces of a module's source code. Furthermore, the need for such tools indicates that client programmers cannot really be completely oblivious to code introduced by aspects, except in a narrow syntactic sense.

It is interesting to examine the conflicting demands of comprehensibility and obliviousness in the context of object-oriented programming. Filman and Friedman point out that dynamic binding in object-oriented languages represents a form of obliviousness [14, §2]. Behavioral subtyping [12, 22, 24] restores Parnas's ideal of comprehensibility to object-oriented programs, in spite of the obliviousness offered by dynamic binding. We will use this as an analogy for considering the problem of comprehensibility and obliviousness in AspectJ-like languages. We focus on AspectJ-like languages because their dynamic join point models provide a closer analogy to dynamic binding in object-oriented languages than do the static join point models of languages like HyperJ [33, 30]. To keep the discussion and examples concrete, we will focus specifically on AspectJ.

2 The Goal: Modular Reasoning

We would like to have both obliviousness and comprehensibility in an AspectJ-like language. This comprehensibility is often termed "modular reasoning". Thus,

[1] Information on the AspectJ development tools for Eclipse is available from `http://www.eclipse.org/ajdt/`

[2] Information available from `http://www.cs.ubc.ca/labs/spl/projects/qjbrowser.html`

we begin by defining a notion of modular reasoning that corresponds to Parnas's comprehensibility benefit.

Modular reasoning is the process of understanding a system (or proving properties about it) one module at a time. A language *supports modular reasoning* if the actions of a module M written in that language can be understood based solely on the code contained in M and the code surrounding M, along with the signature and behavior of any modules referred to by that code. The notion of a *module* and the *surrounding code* for a module are determined by the programming language. In Java, we might consider a compilation unit to be a module, and in standard Java no code surrounds a module[3]. Code, C, *refers* to a module N if C explicitly names N, if C is lexically nested within N, or if N is a standard module in a fixed place (such as `Object` in Java).

We use contracts to specify the code's behavior. In the most concrete case, the contract is the method's code, but we prefer to think of more abstract contracts. These can be written in a formal specification language [21, 25], or informally by writing comments such as "This method returns true if the given file exists".

Our interest in modular reasoning in aspect-oriented programming languages is motivated in part by our initial work on combining MultiJava [6, 8] and JML [21].

3 The Problem: Undisciplined Obliviousness

In typical object-oriented languages, the dynamic type of the receiver object is used to select the appropriate method to invoke for a given message send. Such dynamic selection of the target method can prevent modular reasoning. For example, consider the declaration of `Point` in Figure 1 and its method, `move`. The `//@`-comments before `move`'s declaration give its behavioral specification in JML.

- The clause "`requires true`" says that clients are not obliged to establish any precondition.
- The clause "`assignable pos`" says that the `pos` field of the receiver object may be changed by the method, but not any other locations.
- The clause "`ensures` ..." says that, after `move` returns, the value returned by `getPos()` is equal to the sum of the `dist` argument and the value returned by `getPos()` before `move` was called.

Suppose an object of static type `Point` is passed to a method `client`, as in Figure 2. If modular reasoning is sound, then the programmer can reason about the invocation of `move` based on its specification in `Point`. That is, if the first assertion in Figure 2 holds, then the second assertion is valid based on the specification of `Point`'s `move` method. The definition of modular reasoning

[3] Another interpretation of "module" and "surrounding code" in Java might be that a type declaration is a module, and, for nested type declarations, the code for the enclosing type declaration is the surrounding code.

```
public class Point {
  private /*@ spec_public @*/ int pos;
  public final /*@ pure @*/ int getPos() {
    return pos;
  }
  /* ... */
  //@ requires true;
  //@ assignable pos;
  //@ ensures getPos() ==
  //@    dist + \old(getPos());
  public void move(int dist) {
    pos = pos + dist;
  }
}
```

Fig. 1. A `Point` class

```
public void client(Point p) {
  /* ... */
  assert p.getPos() == 0;
  p.move(-10);
  assert p.getPos() == -10;
}
```

Fig. 2. Sample client code

requires that the programmer should not have to consider possible subtypes of `Point` when doing this reasoning, since they are not mentioned in the client code. However, by subsumption, an instance of an unseen subtype of `Point`, say `RightMovingPoint`, may be passed to `client`. What if (as in Figure 3) `RightMovingPoint` overrides method `move`, but the override does not satisfy the specification of `move` in `Point`? Then modular reasoning such as that described for `client` is not valid. Using Figure 3, after the client's invocation of `p.move(-10)`, `p.getPos()` returns `10`, not `-10` as asserted.

The concept of behavioral subtyping restores sound modular reasoning by imposing the specification of `Point` on all its subtypes [11, 22, 24]. Thus, `RightMovingPoint` does not correctly implement a behavioral subtype of `Point`, because its implementation does not satisfy the specification of `move` in `Point`. Behavioral subtyping is often described by saying that the behavior of a subtype should not be *surprising*, with respect to the specified behavior of a supertype.

```
public class RightMovingPoint
                 extends Point {
  public void move(int dist) {
    if (dist < 0) super.move(-dist);
    else super.move(dist);
  }
}
```

Fig. 3. A `RightMovingPoint` class

As pointed out by Filman and Friedman [14], subtyping with subsumption, as above, is an example of obliviousness. Aspect-oriented programming languages allow programmers much greater latitude in defining oblivious behaviors.

3.1 Non-modular Reasoning in AspectJ

Next we show that, just as modular reasoning is not a general property of object-oriented programming languages in the absence of behavioral subtyping, modular reasoning is not a general property of AspectJ. We do this by considering an aspect-oriented extension to our `Point` example.

Figure 4 gives an aspect, `MoveLimiting`, that modifies the behavior of `Point` instances in the same way as `RightMovingPoint`. `MoveLimiting` declares a piece of around-advice. This advice intercepts calls to `Point`'s `move` method. If the argument passed to the client is negative, then, just as in `RightMovingPoint`, control proceeds to `Point`'s `move` method with the parameter set to the absolute value of the original parameter. As with `RightMovingPoint`, the `client` programmer's reasoning, as indicated by the `assert` statements in Figure 2, is not correct in the presence of the `MoveLimiting` aspect.

```
public aspect MoveLimiting {
  void around(int dist):
    call(void *.move(int))
      && args(dist)
  {
    if (dist < 0) proceed(-dist);
    else proceed(dist);
  }
}
```

Fig. 4. A `MoveLimiting` aspect

In AspectJ the advice is applied by the compiler without explicit reference to the aspect from either the `Point` module or a client module. (Instead the classes and aspect are simply passed as arguments to the same compiler invocation.) Thus, modular reasoning about the `Point` module or a client module has no way to detect that the behavior of the `move` method will be changed when the `Point` module and `MoveLimiting` are compiled together. In AspectJ the programmer must potentially consider all such aspects and the `Point` class together in order to reason about the `Point` module. Some potentially applicable aspects, such as `MoveLimiting`, may not even name `Point` directly, but instead may use wild card type patterns. So, in general, a programmer cannot "study the system one module at a time" [31].

Therefore, just as in object-oriented programming without behavioral subtypes, the obliviousness of AspectJ can prevent modular reasoning.

3.2 Programmers' Stories

The concept of behavioral subtyping was based on programmers' stories about how they used and reasoned about subtypes. The formalization of behavioral subtyping builds on a programming discipline that evolved "in the wild".

It is not yet clear how to modularly reason about AspectJ programs. What programming discipline, akin to behavioral subtyping, could be used to allow modular reasoning for AspectJ programs, while retaining obliviousness?

It seems that any such discipline must allow programmers to tell at least two sorts of stories[4]. We call these *superimposition stories* and *adaptation stories*.

Superimposition Stories Superimposition stories are about combining modules for separate concerns without introducing surprising behavior. The standard example of using aspects for logging is a superimposition story. Superimposition

[4] Thanks to Arno Schmidmeier, Juri Memmert, Karl Lieberherr, Frank Sauer, and others for discussions at AOSD '02 on the ways they are using aspect-oriented programming.

stories are analogous to the stories told with subtypes in object-oriented languages; both seek to enhance existing behavior without introducing surprising behavior.

In the behavioral subtyping literature, not introducing "surprising behavior" means that an overriding method cannot contradict the specified behavior of the method it overrides. Such methods can, however, introduce new behavior in areas that are unspecified in the supertype. The classic example is the specification of a `draw` method for shapes. The method is underspecified in the `Shape` supertype, which allows overriding methods in `Rectangle` and `Circle` to specialize its behavior in appropriate ways.

One way to think about behavioral subtyping is that an overriding method inherits the specification of the supertype method that it overrides [12]. This forces the overriding method to obey the supertype's specification. In an aspect designed for superimposition, the analogous requirement would be that the advice must obey the specifications of the code that it advises.

Adaptation Stories Adaptation stories are about modifying the behavior of existing programs without changing their code. These stories come in two styles:

- *Local adaptation* stories are about changing the behavior of a subset of a program as viewed by other client code within the program. An example of local adaptation is an aspect for persistence, where some code must explicitly establish links to a database while the remainder of the program benefits from persistence without any explicit changes to its code [32].
- *Global adaptation* stories are about changing the behavior of an entire program as viewed by users of the program. For example, using an aspect to add authentication to an existing web server would be a global adaptation story [19][20, Ch. 10]. Global adaptations are distinguished in that no part of the adapted program relies on the presence of the adaptation.

In contrast to aspects designed for superimposition, programmers expect aspects designed for adaptation to change the behavior of the modules they advise.

Adaptation stories can be told in object-oriented languages. Sometimes this is done using subclassing, but this violates the behavioral subtyping discipline and is generally discouraged [5, pp. 71–77][23, 26]. Instead, the recommended approach is to use a wrapper class whose instances contain a reference to the object to be adapted [15, pp. 139–150, 175–184]. The specifier of the wrapper class "starts from scratch"; she can provide whatever behavior is needed, independent of the original class's specification. This wrapper approach sacrifices the obliviousness of dynamic binding to achieve modular reasoning; a client of the wrapper class can reason using the wrapper class's specification.

It may be the case that a similar sacrifice of obliviousness is needed for modular reasoning in aspect-oriented programming languages. Like the wrapped class, the adapted code could remain oblivious to the adaptation aspect. But like the code that uses the wrapper class, perhaps the code that relies on the

adapted behavior must explicitly reference the adaptation aspect. (The filter specifications in Composition Filters [4] have this flavor, as does Hyper/J's meta-language for composing hyperslices into hypermodules [33, 30].)

Discussion It is tempting to relate the superimposition and adaptation stories to the distinction between production and development aspects made by others [17]. However, it seems that there are differences. For example, distinct production aspects for persistence and accounting rules might be part of a superimposition story if their specifications do not interact. We hope to better understand these distinctions by investigating possible solutions to the problem described above.

4 Verification vs. Conservative Restrictions

The behavioral subtyping analogy also suggests some ideas about the potential solution space.

Type systems for object-oriented programming languages enforce a subtyping property that is weaker than behavioral subtyping. For example, Java's type-checking is sufficient to ensure all invoked methods will exist at runtime and have the correct argument types. However, these rules still allow surprising behavior at runtime. To check behavioral subtyping, one would also have to add information about behavioral specifications. One would also have to deal proving that an overriding method implements the specification of the method it overrides—a task that cannot, in general, be completely automated. While it might be possible to greatly restrict specifications so that they could be automatically checked, to our knowledge this has never been done in a popular object-oriented language;[5] the price of formal specification and verification has seemed too large.

Similarly, solutions to the modular reasoning problems of AspectJ-like languages could fall anywhere along this continuum from conservative statically checked restrictions to obligations on behavioral specifications that are left to the programmer. A compromise that might be a fruitful area of investigation is to leave behavioral restrictions to external specification languages and tools, as is currently the case for most object-oriented languages (aside from Eiffel [26]).

4.1 Control Flow Restrictions

One intermediate approach is to restrict the points in a program where behavior changes might occur. An example of this in object-oriented programming is the `final` modifier in Java [16]. If a method is declared `final`, then a client of that class knows that the method cannot be overridden by an unseen subclass. An analogous approach for AspectJ-like languages might be a mechanism for a module to limit the join points that it exposes to advice binding.[6]

[5] Abadi and Leino's integrated type and specification language [1] perhaps comes closest to this idea.

[6] This idea is partly the result of conversations with Jonathan Aldrich at OOPSLA 2003.

4.2 Data Restrictions

A complementary intermediate approach is to restrict the data that may be modified by advice. This is exemplified in object-oriented programming languages by the work on various ownership type systems [2, 27–29]. These type systems are primarily concerned with restricting aliasing to promote data encapsulation, and hence modular reasoning. An analogous approach for AspectJ-like languages might be to limit the state that may be mutated by advice to just that state that it owns. Such a limitation is statically checkable, but requires additional annotations in the source code. However, these annotation offer the additional benefit of controlling some effects of aliasing.

5 Summary and Future Work

There is a parallel between the obliviousness of dynamic binding in object-oriented languages and that of advice binding in aspect-oriented languages [14]. We have shown that this parallel provokes a useful analogy between reasoning in the two styles. We have demonstrated that both sorts of obliviousness can prevent modular reasoning, as demonstrated by the `MoveLimiting` aspect and the (non-behavioral) subtype `RightMovingPoint`.

Behavioral subtyping adds modular reasoning to object-oriented programming languages. The problem is to find a similar programming discipline for AspectJ-like languages. We have described two sorts of stories that programmers tell about how they use AspectJ: superimposition and adaptation stories. These stories provide constraints on solutions to the modular reasoning problem, in that a completely viable solution should support both kinds.

The behavioral subtyping analogy seems to be a rich one for thinking about the relationship between object-oriented languages and AspectJ-like languages. Despite the absence of a solution to the modular reasoning problem, we hope that other researchers will benefit from the behavioral subtyping analogy. Superimposition and adaptation stories seem to characterize how aspect-oriented programming languages are actually used. We hope that this characterization will provide useful constraints in the design of new aspect-oriented langauges and reasoning mechanisms.

In addition to considering what behavioral subtyping may say about aspect-oriented programming, it might be interesting to consider whether the analogy works in reverse. Can aspect-oriented programming provide new insight on the earlier work on object-oriented languages?

Our future work is to formalize the modular reasoning problem and investigate the solution space by building a core calculus for an AspectJ-like language with support for modular reasoning. Our initial steps towards this calculus are available in a technical report [9], though that work does not include modularity features. Our next step is to extend the calculus with such features. After proving the appropriate modularity properties, we will then extend the calculus to a full-scale programming language.

6 Acknowledgments

We thank Yoonsik Cheon, Todd Millstein, Markus Lumpe, and Robyn Lutz, for their helpful comments on an early version of this work [7]. We also thank the workshop participants at Foundations of Aspect-oriented Languages 2002, in particular Gregor Kiczales, Doug Orleans, and Hidehiko Masushara, for discussions regarding this work. Finally, we thank the anonymous referees from the Aspect-Oriented Software Development 2003 Program Committee, for helping us to understand the scope of the problems we are addressing, and from Software-Engineering Properties of Languages for Aspect Technologies 2003, for helping us tighten the presentation in this paper.

The work of both authors was supported in part by a grant from the US National Science Foundation under grants CCR-0097907 and CCR-0113181.

References

1. M. Abadi and R. Leino. A logic of object-oriented programs. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer-Verlag, New York, NY, 1997. Expanded in DEC SRC report 161.
2. J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, volume 37(11) of *ACM SIGPLAN Notices*, pages 311–330. ACM, Nov. 2002.
3. AspectJ Team. The AspectJ programming guide. Available from `http://aspectj.org/doc/dist/progguide/index.html`, Feb. 2002.
4. L. Bergmans and M. Aksits. Composing crosscutting concerns using composition filters. *Commun. ACM*, 44(10):51–57, Oct. 2001.
5. J. Bloch. *Effective Java: Programming Language Guide*. Java series. Addison-Wesley, Boston, 2001.
6. C. Clifton. MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch. Technical Report 01-10, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, Nov. 2001. Available from `www.multijava.org`.
7. C. Clifton and G. T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. Technical Report 02-04a, Iowa State University, Department of Computer Science, Apr. 2002.
8. C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications*, volume 35(10) of *ACM SIGPLAN Notices*, pages 130–145, New York, Oct. 2000. ACM.
9. C. Clifton, G. T. Leavens, and M. Wand. Parameterized aspect calculus: A core calculus for the direct study of aspect-oriented languages. Technical Report 03-13, Iowa State University, Department of Computer Science, Oct. 2003. Submitted for publication.
10. Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. *ACM SIGSOFT Software Engineering Notes*, 26(5):88–98, Sept. 2001. Proceedings of the 8th

European software engineering conference held jointly with 9th ACM SIGSOFT symposium on Foundations of software engineering, Vienna, Austria.
11. K. K. Dhara and G. T. Leavens. Weak behavioral subtyping for types with mutable objects. In S. Brookes, M. Main, A. Melton, and M. Mislove, editors, *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995. Available from `http://www.elsevier.nl/locate/entcs/volume1.html`.
12. K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, Mar. 1996. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.
13. T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, Oct. 2001.
14. 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, to appear.
15. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.
16. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.
17. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Commun. ACM*, 44(10):59–65, Oct. 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. I. Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams Publishing, Indianapolis, 2003.
20. R. Laddad. *AspectJ in Action*. Manning Publications Co., Grennwich, Conn., 2003.
21. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06t, Iowa State University, Department of Computer Science, June 2002. See `www.jmlspecs.org`.
22. G. T. Leavens and W. E. Weihl. Reasoning about object-oriented programs that use subtypes (extended abstract). In N. Meyrowitz, editor, *OOPSLA ECOOP '90 Proceedings*, volume 25(10) of *ACM SIGPLAN Notices*, pages 212–223. ACM, Oct. 1990.
23. B. Liskov and J. Guttag. *Program Development in Java*. The MIT Press, Cambridge, Mass., 2001.
24. B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.
25. B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.
26. B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.
27. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. The author's Ph.D. Thesis. Available from `http://www.informatik.fernuni-hagen.de/import/pi5/publications.html`.

28. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. Technical Report 02-02a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, Oct. 2002. To appear in Concurrency, Computation Practice and Experience.
29. J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP '98 – Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer-Verlag, July 1998.
30. H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Commun. ACM*, 44(10):43–50, Oct. 2001.
31. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, Dec. 1972.
32. A. Rashid and R. Chitchyan. Persistence as an aspect. In M. Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 120–129. ACM Press, Mar. 2003.
33. P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. Sutton Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.