A Design Discipline and Language Features for Formal Modular Reasoning in Aspect-Oriented Programs

Curtis Clifton and Gary T. Leavens

TR #05-23
December 2005

Keywords: Spectators, assistants, aspect-oriented programming, modular reasoning, AspectJ language.
2002 CR Categories: D.3.1 [*Programming Techniques*] Object-oriented programming — aspect-oriented programming; 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.

Submitted for publication.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

A Design Discipline and Language Features for Formal Modular Reasoning in Aspect-Oriented Programs*

Curtis Clifton[1] and Gary T. Leavens[2]

[1] Computer Science and Software Engineering
CM 4006, Rose-Hulman Institute of Technology
5500 Wabash Ave., Terre Haute, IN 47803 USA
clifton@rose-hulman.edu
[2] Department of Computer Science, Iowa State University
226 Atanasoff Hall, Ames, IA 50011-1040, USA
leavens@cs.iastate.edu

Abstract. Advice in aspect-oriented programming helps programmers modularize crosscutting concerns by allowing additions and changes to a program's execution. However, formal reasoning about the functional behavior of aspect-oriented programs requires a non-modular, whole-program search to find applicable advice.
To allow modular reasoning, we describe a discipline that categorizes aspects into two sorts: spectators and assistants. "Spectators" are statically checked to not modify the behavior of the code they advise; this restriction lets them remain unseen. Unlike spectators, "assistants" are not restricted in their behavior. However, for modular reasoning one must be able to identify all applicable assistants, hence assistants must be explicitly accepted by the code they advise. Besides allowing modular reasoning, this discipline permits the use of existing idioms, and appears to be statically verifiable and practical for software development. Indeed, expert aspect-oriented programmers seem to use such a discipline.

1 Introduction

Aspect-oriented programming [20] deals with the problem of modularizing cross-cutting concerns. *Cross-cutting concerns* are features of a program that are orthogonal to its main decomposition [35]. Such concerns inherently result in *scattering* of code across various modules in the main decomposition of the program. From the perspective of the main decomposition, such cross-cutting code is said to be *tangled*. Aspect-oriented programming moves code for cross-cutting concerns into separate modules, called aspects, and declares how code, *advice*, is associated with events in the main decomposition at run time.

This declarative association of advice with events is a powerful technique for eliminating the scattering of code. It can result in modules that are more

* This paper is adapted from Clifton's Ph.D. thesis [7] and builds on ideas from our earlier work [8]. The work of both authors was supported by NSF grant CCF-04-8078.

focused and concise. It promotes modularity and hence helps modular reasoning. As Laddad [23, §1.3.3] points out, well-written aspects consolidate code for a concern that would otherwise be scattered. Thus, one can check that such a concern is properly implemented (e.g., that a policy is enforced as desired) by just looking at a single aspect; without aspect-oriented techniques, such checks would need a whole-program search to find the scattered code and to check that it was written in all the required places.

However, aspects also present challenges to code understanding and maintenance. These problems occur because the semantics of weaving advice into a program's execution scatters the effects of advice throughout a program. Thus, to formally reason about functional behavior, programmers must, in general, know what advice applies to each piece of code.

In the early years of object-oriented programming, similar concerns were raised about polymorphic method invocations. Such method invocations can invoke code of a subtype, making it hard to formally reason about the functional behavior of method calls without a whole program search for all possible overriding methods in subtypes. The discipline of behavioral subtyping [3, 11, 25, 28, 29] addresses these concerns. It places a constraint on subtype programmers: overriding methods in a subtype must satisfy the specification of the overridden supertype methods. In exchange for programming with this constraint, clients of the supertype may reason about invocations of supertype methods without worrying about the effects of overriding methods in unseen subtypes. Behavioral subtyping thus provides both practical guidance to programmers and has been proven to allow sound modular reasoning. Our goal is to present an analogous discipline for aspect-oriented programming.

1.1 The Behavioral Subtyping Analogy

In this section we describe the desired analogy in more detail.

1.1.1 Modular Reasoning There are various definitions of modularity in the literature. The weakest definition might be that an analysis is modular if the portion of the program that must be considered to perform the analysis is a well-defined, proper subset of the whole program. At the other end of the spectrum, the definition might require that an analysis must consider only the code for a single compilation unit. Work that uses this definition, particularly in the area of component-based programming [16], typically requires a compilation unit to declare its expectations of external modules [34]. These expectations can then be verified during composition of the unit with external modules.

We will use a definition between these two extremes: a language allows *modular reasoning* if it is possible to reason about a compilation unit in that language based on the code of that compilation unit and the specifications of any *modules* (e.g., classes, interfaces, and packages) referred to by that compilation unit. A compilation unit *refers* to a module M if it explicitly names M, is lexically nested within M, or if M is a standard module in a fixed location (such as `Object` in Java). Java [17] with JML [26] and Eiffel [29] both satisfy this definition.

In languages that allow modular reasoning the cognitive burden on the programmer is reduced—the specifications of referenced modules serve as behavioral abstractions of all the code implementing those modules.

1.1.2 Object-Oriented Non-modularity In single-dispatch object-oriented languages, the dynamic type of the receiver object is used to select the appropriate method to execute for a given invocation. 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 "`assignable pos`" says that the `pos` field of the object, but no other locations, may be changed by the method.
- 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.

```
public class Point {
    private /*@ spec_public @*/ int pos;
    //@ ensures \result == pos;
    public final /*@ pure @*/ int getPos() {
        return pos;
    }
    /* ... */

    //@ assignable pos;
    //@ ensures getPos() == dist + \old(getPos());
    public Point move(int dist) {
        pos = pos + dist;
        return this;
    }
}
```

Fig. 1. `Point` Class

Suppose an object of static type `Point` is passed to a method `client`, as in Figure 2 on the next page. 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 the figure holds, then the second assertion is valid based on the specification of `Point`'s `move` method. The definition of modular reasoning requires that the programmer should not have to consider possible unseen subtypes of `Point` when reasoning, since they are not mentioned in the client code. But, by subsumption, an instance of just such an unseen subtype may

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

Fig. 2. Sample Client Code

be passed to `client`. What if (as in Figure 3) the subtype `RightMovingPoint` overrides method `move`, but does not satisfy the specification of `move` in `Point`? Then modular reasoning such as that described for `client` is not valid. If an instance of `RightMovingPoint` is passed to `client`, then after the invocation of `p.move(-10)`, the assertion fails: `p.getPos()` returns `10`, not `-10`.

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

Fig. 3. `RightMovingPoint` Class

1.1.3 Behavioral Subtyping Modular reasoning is not an inherent property of object-oriented languages. However, the discipline of behavioral subtyping restores sound modular reasoning by imposing the specification of `Point` on all its subtypes [11, 25, 28]. `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. Behavior is *surprising* if the (possibly unseen) code executed in response to a method invocation fails to satisfy the visible method's specification.

1.1.4 Non-modularity in Aspect-Oriented Languages Filman and Friedman [13] assert that aspect-oriented languages are characterized by two features: quantification and non-invasiveness (originally "obliviousness"). *Quantification* is the declarative specification of a set of points, in either the static code or the

dynamic control flow graph of a program, where aspect-oriented code is to be added. Non-invasiveness is the execution of additional aspect-oriented code at a program point, P, without effort by the programmer of the code containing P. Filman and Friedman point out that object-oriented subtyping with subsumption is a form of non-invasiveness. Aspect-oriented programming languages allow programmers much greater latitude in defining behaviors with unseen code.

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 aspect-oriented languages. To show this, we present an aspect-oriented extension to the `Point` example.

Figure 4 gives an aspect, `OneWayMoving`, that modifies the behavior of `Point` instances in the same way as `RightMovingPoint` [8]. `OneWayMoving` 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 in Figure 2 is not correct in the presence of `OneWayMoving`.

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

Fig. 4. `OneWayMoving` Aspect

In AspectJ the advice is applied without explicit reference to the aspect from either the `Point` module or a client module. 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 `OneWayMoving` are composed. 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 `OneWayMoving`, may not even name `Point` directly, but instead may use wild card type patterns. Therefore, just as in object-oriented programming without behavioral subtypes, the non-invasiveness of aspect-oriented languages can prevent modular reasoning.

1.1.5 Summary and Solution Characteristics The discipline of behavioral subtyping restores modular reasoning to object-oriented programming languages. It does this by requiring that an overriding method satisfy the specification that it inherits from the superclass method [11, 25, 28]. This discipline is enabled by the fact that *the superclass method is visible from the declaration*

of the overriding method. This is crucial. A class declares its superclass. The declaration allows the class to inherit or override methods from the superclass. So the class declaration containing an overriding method provides a reference to the overridden method.

Thus the non-invasiveness in object-oriented programming only cuts one way. From a method call site, the actual code to be executed may be in an unseen, overriding method. However, from the declaration site of the overriding method, the superclass method is visible, allowing the overriding method to satisfy the superclass method's specification.

In contrast, the non-invasiveness in aspect-oriented programming cuts both ways. From a method call site, the actual code to be executed may be in an unseen aspect. And from the declaration site of an aspect, because of quantification, the code to be advised may also be unseen. For example, the aspect might only advise code that implements some interface, and the code implementing that interface might not be known or even exist when the aspect is written. Thus, with aspect-oriented languages one cannot adopt the solution of behavioral subtyping: it is not enough to simply require that advice satisfy the specification of the code it augments.

This, then, is the core challenge in developing a programming discipline that allows modular reasoning about aspect-oriented programs. If reasoning is to be modular, then how can one reason about potentially advised code when (1) unseen aspects may apply to the code, and (2) aspects may be developed without (complete) knowledge of the code that will be advised?

1.1.6 Requirements for a Solution The `OneWayMoving` aspect represents poor aspect-oriented programming, just as the `RightMovingPoint` class in Figure 3 represents poor object-oriented programming. This is because both examples change the behavior of `move` with regard to a `Point`'s position; behavior that is, by virtue of `move`'s strong specification, restricted to `Point` itself.

How do aspect-oriented programmers avoid such problems? Experienced programmers separate concerns into orthogonal aspects. Orthogonality of these aspects helps the reader of a program to understand it, provided she can find the applicable aspects. Specifically, if she wants to reason just about the functional behavior of a code fragment, she must just consider the base program code. If she is concerned with the persistence behavior of a code fragment in the base program, she must just consider the single aspect for persistence. Based on these observations, a detailed design discipline must have two key features:

1. easy identification of applicable aspects, and
2. orthogonality of the concerns expressed by those aspects.

The first feature is provided in some aspect-oriented languages, for example, in Hyper/J's module interconnect language and in Weave.NET's XML-encoded aspect bindings [24]. In AspectJ, the first feature is generally provided by tool support, though this limits the possible analyses to those supported by the tool. We demonstrate that the easy identification of applicable aspects can be more generally accommodated at the language level.

The second feature is possible in existing aspect-oriented languages (indeed, this is largely what makes them aspect-oriented). However, we are not aware of any other languages that explicitly help software engineers to statically verify such orthogonality.

1.2 The MAO Discipline

In what follows we describe the *MAO discipline* for **m**odular, **a**spect-**o**riented programming that: (1) allows modular reasoning, (2) permits the use of existing aspect-oriented idioms for separation of concerns, (3) can be verified by a combination of static typechecking and simple verification conditions, and (4) can be incorporated into a practical, aspect-oriented language. We support these claims this paper by:

- describing the MAO discipline;
- presenting a small set of language features, as an extension to AspectJ (version 1.2), designed to facilitate the discipline;
- developing extensions to JML for specifying features of aspect-oriented programs; and
- sketching an algorithm for calculating the *effective specification* of an expression in our AspectJ extension, given the specifications for any potentially applicable advice.

The language features are formally described in detail in Clifton's dissertation [7]. Clifton has described, and proved sound, a static type system for a core calculus containing these features. He has also proved key meta-theoretic properties that demonstrate the effectiveness of the MAO discipline and our proposed language features for modular aspect-oriented reasoning.

While Clifton's dissertation is primarily formal, in this paper, we concentrate on the informal description of the MAO discipline and our proposed language extensions, to both AspectJ and JML. We refer the reader to Clifton's dissertation for the formal details that support the claims we make about the discipline.

2 Shared Responsibility

In the discipline of behavioral subtyping all the burden of ensuring modular reasoning is placed on the author of an overriding method. A "client" programmer—that is, a programmer writing code that calls the method—may reason about a call without seeing the overriding method. The specification of the superclass method is normally sufficient for reasoning.

However, in the MAO discipline, the burden is shared. The division of responsibility is mediated by the MAO discipline's division of aspects into two sorts based on whether the advice might introduce "surprising" behavior [8]. *Surprising* behavior causes the specification of the advised code to be violated.

For advice that is not surprising, the advice author must satisfy certain restrictions (detailed below). Having done so, the client programmer can remain

safely oblivious to the benign advice when reasoning about advised code. On the other hand, for advice that might introduce surprising behavior, the language must allow the client programmer to modularly identify what advice may apply. Having identified this advice, the client programmer must compose the specifications of the advice and the advised code, thus finding the *effective specification* of the code in the presence of the advice.

The MAO discipline satisfies the requirements for solving the modular reasoning problems that we identified in Section 1.1.6 as follows. (1) Aspects that would be unseen in regular AspectJ are divided into two kinds. Those with unsurprising advice remain unseen, but do not affect the behavior of the code within the bounds of its specification. Those with surprising advice must be made visible so that their effects can be considered. (2) For aspect development, the programmer of unsurprising advice must satisfy a set of restrictions, but having done so, she does not need to consider the specific behavior of the code that will be advised. For surprising advice, she must simply satisfy the specification of the advice; the responsibility for reasoning about the interaction of such surprising advice and the advised code falls to the client programmer.

3 Proposed Language Features

In this section we describe some language features that are sufficient to support the MAO discipline. For concreteness we describe these features as extensions to AspectJ. Our running example expands on the `Point` example of Figure 1.

The key feature to support modular reasoning in our proposal is to divide aspects into two sorts: spectators and assistants. "Spectators" are limited in that they may not introduce surprising behavior. "Assistants" are not limited in this way. Since spectators do not change the specified behavior of the modules to which they are applied, they preserve modular reasoning even when applied without explicit reference by said modules. Hence spectators preserve most of the flexibility of the current version of AspectJ. Because assistants can change the specified behavior, to maintain modular reasoning they can only be applied to modules that reference them.

3.1 Assistants

We call aspects that can change the specified behavior of a module *assistants*. The `MoveLimiting` aspect of Figure 5 on the facing page is an assistant; it changes the behavior of `Point`'s `move` method to limit the maximum change in position from any single call. The term "assistant" is intended to connote a participatory role for these aspects.

What information is needed to modularly reason about behavior when assistants are present? Quite simply, a module must explicitly name those assistants that may change its behavior or the behavior of modules that it uses. We say that a module *accepts assistance* when it names the assistants that are allowed to change its behavior or the behavior of modules that it uses. Assistance may be accepted by either:

```
1  public aspect MoveLimiting {
2      private static int MAX_DISTANCE = 10;
3      /* Constrains distance of any single movement */
4      Point around(int arg) :
5          execution(* mao.Point.move(int)) && args(arg)
6      {
7          if ( arg > MAX_DISTANCE ) {
8              return proceed( MAX_DISTANCE );
9          } else {
10             return proceed( arg );
11         }
12     }
13 }
```

Fig. 5. `MoveLimiting` Example (in AspectJ)

- the module to which the assistance applies (called the *implementation module*), or
- a client of that module.

3.1.1 Explicit Acceptance of Assistance AspectJ does not currently include syntax for explicitly accepting assistance. Explicit acceptance of assistance can, however, be roughly simulated by the "hyper-cutting" pattern in AspectJ. In this pattern, one creates a marker interface, and the pointcuts of assistants would only apply to types that implement that interface [22, pp. 214–216]. An implementation module can then implement the marker interface, and thus indirectly accept the advice of the assistant. However, if a single client declares that the implementation module is a subtype of the marker interface (using the `declare parent` syntax of AspectJ), then the change affects all clients of the implementation module, but no trace appears in the implementation module; hence such changes are not modular. Annotations can also be used for hyper-cutting, with similar modularity issues.

To automate the hyper-cutting pattern, and to avoid non-modular uses of it, we propose a simple syntax extension for accepting assistance:

`accept` *TypeName*`;`

where *TypeName* must be the name of an assistant respecting Java's usual namespace rules for packages and imports [17, §6.5]. Multiple accept clauses may appear in a single module, following any import clauses. For example, the `Point` module could accept the `MoveLimiting` assistant by declaring:

```
accept MoveLimiting;
```

We will generalize this idea with concern maps below.

When an implementation module accepts assistance, that assistance is applied to every applicable join point within the implementation module, regardless of the client making the call.

On the other hand, if the assistance is accepted by a client module, then that assistance is only applied to applicable join points in that client. Other clients that did not accept the assistance would not have it applied to their join points.

AspectJ includes two pointcut descriptors that roughly simulate this behavior. Advice on join points described via `call` pointcuts is woven into all client code. Advice on join points described via `execution` pointcuts is woven into the implementation code. Unfortunately, clients of such an implementation module have no (modular) way to know that such advice will be applied to their calls to the implementation module. In our proposal clients of such an implementation module would know about the advice; this is an example of how explicitly accepted assistance allows modular reasoning.

3.1.2 Concern Maps Modular reasoning in aspect-oriented programming languages can be achieved if modules explicitly accept assistance. But some assistants are applicable to code throughout an entire package or program, for example, a common exception handler. It would be inconvenient and error prone to include accept clauses for these assistants in every module. In fact, such accept clauses would represent code tangling.

We propose concern maps to avoid these problems. A *concern map* is a source code construct that specifies a mapping from modules in a package, or set of packages, to the assistance that is accepted by those modules. In our initial design, each package may contain at most one concern map. In file-system-based implementations, the concern map for a package would be given in a file named `package.map` stored in the directory containing the package source code. The syntax for concern maps is given in Figure 6 on the next page. Figure 7 gives an example concern map for the package named `mao`.

The type pattern `*`, in line 3 of Figure 7, says that all types in the `mao` package accept the `MoveLimiting` assistant. (We do not allow concern maps to specify fully qualified names in type patterns; instead we implicitly concatenate the name of the map's package with the given pattern. Thus the pattern "`*`" in the example, signifies the pattern `mao.*` in the global namespace. We do this because the map should only be able to specify acceptance of assistance for local types and types in "subpackages".) The `Rectangle` pattern in line 7 of the example says that, in addition to the `MoveLimiting` assistant, the `mao.Rectangle` class also accepts the `AreaStretching` assistant. As with accept clauses in modules, the identifier in an accept clause of a concern map is subject to Java's usual namespace rules for packages and imports.

One can think of concern maps as like an AspectJ "introduction"; they add accepts clauses to modules in the local package and subpackages. It would defeat the purpose of accepts clauses to allow their global introduction. So unlike AspectJ introductions, concern maps are lexically scoped.

$$
\begin{aligned}
\mathit{AspectMap} &::= \mathit{PackageDecl}\ \mathit{ImportDecls}_{opt}\ \mathit{MappingList}_{opt} \\
\mathit{PackageDecl} &::= \texttt{package}\ \mathit{Identifier}\texttt{;} \\
\mathit{MappingList} &::= \mathit{Mapping}\ \mathit{MappingList}_{opt} \\
\mathit{Mapping} &::= \mathit{TypePat}\ \texttt{\{}\ \mathit{AcceptList}_{opt}\ \texttt{\}} \\
\mathit{AcceptList} &::= \mathit{AcceptClause}\ \mathit{AcceptList}_{opt} \\
\mathit{AcceptClause} &::= \texttt{accept}\ \mathit{Identifier}\texttt{;} \\
\mathit{DomainClause} &::= \texttt{domain}\ \mathit{Identifier}\texttt{;}
\end{aligned}
$$

where *TypePat* refers to type patterns in the AspectJ Programming Guide [4, Appendix A], and *ImportDecls* refers to regular Java import declarations [17, §7.5].

Fig. 6. Syntax of Concern Maps

```
1  package mao;
2
3  * {
4      accept MoveLimiting;
5  }
6
7  Rectangle {
8      accept AreaStretching;
9  }
```

Fig. 7. Example Concern Map

The assistance accepted via concern maps still allows modular reasoning. To wit, the package clause at the beginning of a module names all the possible locations where a concern map naming that module might appear. The programmer, or a tool, must only look in that package, or possibly any outer packages, to find the applicable concern map. More specifically, the assistance accepted by a given module consists of:

1. all assistants named in accept clauses in the module,
2. all assistants to which the module is mapped by the `package.map` file for the module's package, and
3. all assistants to which the module is mapped by any `package.map` files in *outer packages* (i.e., packages surrounding the module's package).

This recursive search for acceptance of assistance in the module's package and outer packages allows the programmer to specify widely-applied assistance in the root of a package hierarchy, package-specific assistance in the concern map

of the package it applies to, and module-specific assistance in the modules it applies to. [3]

3.2 Spectators

Explicitly accepted assistance supports modular reasoning. Concern maps give the programmer flexibility in accepting assistance. But what about "development aspects"[21, p. 61], like tracing or debugging code, that are only sometimes included in an executing program? In a language that just supported explicitly accepted assistance, a programmer would need to edit concern maps or source code modules to control the application of development aspects.

To resolve this we propose that an aspect-oriented programming language should also support a category of aspects that we call *spectators*. A spectator is an aspect that does not change the specified behavior of any other module. Because it does not change the behavior, we will say that a spectator *views* (rather than "advises") methods. The term "spectator" is intended to connote the hands-off role of these aspects.

A spectator must not change the control flow to or from a viewed method and may only mutate the state that it owns. We define ownership using an alias controlling type system [1, 2, 6, 12, 31]. Each location in the heap is declared to belong to a "concern domain", as described below. We do not allow mutation of global state, as that is not handled in the supporting formalism, described in Clifton's dissertation.

3.2.1 Verifying Spectatorhood The primary challenge of implementing this part of our proposal lies in determining whether a given aspect is really a spectator. Clifton's dissertation formalizes a static analysis that conservatively verifies this. We outline this analysis here. It has two parts—verifying control flow and verifying that only appropriate locations are mutated.

In general the problem of verifying that a spectator does not disrupt control flow is undecidable (by reduction to the halting problem); however, one can restrict the sort of control flow allowed in spectators to achieve an approximate solution. In spectators:

- before advice must not throw a checked exception and must not explicitly throw an unchecked exception on any control flow path,
- around advice must proceed, exactly once, to the advised method on all control flow paths, and
- after advice must not throw a checked exception and must not explicitly throw an unchecked exception on any control flow path.

[3] Strictly speaking, packages in Java and AspectJ are not hierarchical. They merely provide a hierarchical namespace. For example, code in an inner-package in Java does not have access to package-privileged code from any outer packages. Our treatment of concern maps reflects the namespace hierarchy of packages, while still respecting their non-hierarchical encapsulation properties.

This solution is approximate because it still allows advice in spectators to include (possibly infinite) looping constructs and to call other (possibly non-terminating) methods, provided any checked exceptions declared by those methods are caught and handled. These conditions correspond to partial correctness (i.e., behavioral correctness ignoring termination) and ignore Java `Error`s, which we treat as outside the scope of specification.[4]

In addition to these control flow checks, the checks for "spectatorhood" must also verify that the proceed expression in around advice passes all arguments to the advised method in their original positions and without mutation. Any value returned from the advised method (or exception thrown) must be passed on by the advice without mutation.

The requirement that around advice in spectators proceeds exactly once, and with the same arguments, can be solved syntactically. Clifton's dissertation [7] separates spectator's around advice into before and after parts with a mandatory `proceed` that implicitly uses the original arguments. The problem of returning the value of the advised code can also be solved through language design, by changing the semantics of spectator's around advice to do that. (Clifton's dissertation uses a `surround` keyword for spectator's around advice and prohibits the use of `around` in spectators.) In spectator's around advice, the special variable `reply` can be used in the after part of surround advice to refer to the result of mandatory `proceed`. A similar, but more draconian, restriction would be to only allow spectators to use `before` and `after` advice.

The mutation analysis for spectators is more challenging. It is closely related to the problem of verifying frame axioms [5]. In fact one can think of spectators as having an implicit frame axiom that prevents modification of locations not in the home domain of the spectator. Clifton's dissertation proves the soundness of this approach.

3.2.2 Concern Domains Informally, concern domains represent a partitioning of the heap into sets representing orthogonal, or cross-cutting, concerns. Concern domains allow these cross-cutting concerns to be represented in the type system. This enables efficient static detection of tangled code. We sketch the design of this language feature here.

An additional declaration form in concern maps declares the concern domains that may be used to partition the heap. Thus, the programmer controls which actual concerns are expressed in the type system. Public concern domain declarations have the simple form `domain` g`;`, where the g is a concern domain name, drawn from the set of Java identifiers. A program may have zero or more concern domain declarations. Like accept clauses, concern domain declarations may also appear with class and interface declarations.

The signatures of declarations, along with object instantiation expressions, determine the actual partitioning. Figure 8 gives an updated version of the `Point` class using concern domains. Also in the figure is a sample spectator declaration. In class and aspect declarations a list of concern domain variables are given

[4] This also corresponds to JML's treatment of `Error`s.

following the class or aspect name. These concern domain variables are *in scope* for the remainder of the declaration (see lines 1 and 18). The first concern domain variable listed, `home` in both examples, represents the home domain for instances of the class or aspect. Any remaining concern domain variables (`other` in line 18 of the example) are used to endow instances of the class or aspect with permission to reference objects in other domains. An `extends` clause of a class declaration specifies the mapping of the concern domain variables to those of the superclass.

```
 1  public class Point in <home> {
 2      private /*@ spec_public @*/ int pos;
 3      //@ ensures \result == pos;
 4      public final /*@ pure @*/ int getPos() writes() {
 5          return pos;
 6      }
 7      /* ... */
 8
 9      //@ assignable pos;
10      //@ ensures getPos() == dist + \old(getPos());
11      public Point in <home> move(int dist) writes(home) {
12          pos = pos + dist;
13          return this;
14      }
15  }

16  package mao;
17
18  spectator DistanceTracking in <home, other> {
19
20      /** Records all the distances moved by all points. */
21      private ArrayList in <home,home> distances
22                            = new ArrayList() in <home,home>;
23      private readonly Point in <other> lastPoint;
24
25      before(int delta, readonly Point in <other> targ)
26        writes(home) :
27          execution(* Point in <other>.move(int))
28          && args(delta) && target(targ)
29      {
30          lastPoint = targ;
31          distances.add(delta);
32      }
33  }
```

Fig. 8. `Point` Class and DistanceTracking Spectator Using Concern Domains

We extend the object instantiation instruction, `new`, to include concern domain arguments (see line 22). The static type system ensures that `new` expressions within static contexts, such as the `main` method of a program, only use the names of declared, public concern domains. Furthermore, `new` expressions within a method or advice declaration must only use concern domain variables that are in scope. When the body expression of the method or advice is evaluated, the concern domain variables will be replaced with the concern domain names used to instantiate the self object of the evaluation.

Furthermore, methods and advice have an effects clause of the form `writes` $(g_1, \ldots, g_n)$, where the g_i are names of concern domains, added to their declarations (see line 11 and line 26). The type system ensures that no other domains may be modified at evaluation time (modulo domain dependencies).

(Although not needed for verifying spectatorhood, the language also adds concern domain dependency declarations, of the form g_1 `varies with` g_2`;` to aspects. These declarations allow an assistant aspect to declare that one concern domain may be modified when code that is declared to modify another domain is executed. These dependency declarations allow assistant aspects to modify other domains besides those written by some advised code, but they also allow a static analysis of what other domains might be modified.)

The type system enforces a non-interference property so that a non-global, signature-level search can identify all the code that might mutate a particular concern domain. By non-global we mean that only the types declared in a package or subpackage declaring a concern domain must be considered. By "signature-level", we mean that only method and advice headers, and not their bodies, must be considered. This search is related to the global configuration informally argued for by Kiczales and Mezini [19], but concern maps and explicit acceptance of advice allow the scope to be narrower.

The type system statically detects code tangling, based on a separation of concerns defined by the programmer [7].

More importantly the type system prevents spectators from modifying the public concern domains of the base program. The proof in Clifton's dissertation [7, §5] can be summarized as follows. Theorem 5.13 (Tag Frame Soundness) allows the unseen, private concern domains of spectators to be modified during method or advice execution. However, because of Theorem 5.16 (Respect for Privacy), one can still reason about the effects of a method or piece of advice. To reason about the execution of a method or piece of advice one must know its signature including its effects clause, the concern domains of the target object, and the configuration of assistants in the program, including the concern domains used for assistants and their dependency declarations. This same theorem says that if the concern domains of the target object do not include any private concern domains, then no changes made by unseen spectators will be visible in the code being considered. The side effects of spectators are effectively sequestered, and thus spectators can be used non-invasively.

3.2.3 Read-only Fields In addition to concern domains, we also add read-only fields. The field declaration in line 23 of Figure 8 is declared read-only. This allows the spectator to capture (in line 30) a reference to the target object of the advised call. The type system prevents the spectator from mutating this reference (or any objects transitively reachable through the reference) [7].

3.2.4 Spectator Example The `DistanceTracking` spectator in Figure 8 meets the restrictions on spectatorhood; our type system can statically detect this. The declaration (in line 18) that this aspect is a `spectator` says that this aspect does not change the behavior of any other module. This spectator mutates its own state by adding the (immutable) `int delta` to its own `ArrayList distances` (in line 31). It also captures a reference to the target object in line 30, though this is stored in a read-only field. The spectator does not change the behavior of `Point`'s `move` method. `DistanceTracking` merely views the arguments to the `move` method. The arguments are passed on to the method unchanged and the method's result is unchanged. In addition to cross-cutting concerns like this tracking example, spectators would also be useful for logging, tracing, and as the observer in the Observer design pattern [15, pp. 293–303]. For example, one can imagine a traffic simulation program that uses spectators for visualization, thus separating the visualization and simulation concerns.

Because spectators do not change the behavior of the methods they view, code outside an existing program can apply a spectator to any join point in the original program without loss of modular reasoning. In reasoning about the client and implementation code for a method, a maintainer of the original program does not need any information from the spectator.

3.3 Impact of the MAO Discipline

To better understand how our proposed restrictions might limit the practical expressiveness of AspectJ, Clifton [7, §2.3] surveyed examples from three separate sources—the *AspectJ Programming Guide* [4] and the books by Kiselev [22] and Laddad [23]. Clifton's study looked only at the categorization of aspects with respect to the MAO discipline. (Hence we leave a study of the constraints imposed by concern domains to future work.)

3.3.1 Spectators Many of the example aspects clearly meet our definition of spectator. To satisfy our restrictions they would only require the `spectator` syntax. In the AspectJ Programming Guide there were 9 spectators, including all of the examples in the `tracing` package, as well as `telecom.TimerLog` and `tjp.GetInfo`. In Kiselev's book there were 4 spectators, although `Profiler` and `NullChecker` required minor changes to be spectators. The changes to `Profiler` would require it to either swallow any I/O exceptions and record the problems or to convert I/O exceptions into unchecked `Error`s; alternatively, `Profiler` could be an assistant that was put in a root-level concern map. `NullChecker` throws an exception and so to be considered a spectator it would need to be modified to throw an `Error` instead.

3.3.2 Assistants Aspects in the examples that could be implemented as assistants can be divided into two kinds. *Client utilities* are used by client modules to change the effective behavior of objects whose types are declared in other modules. The changes in effective behavior do not affect the representation of those objects. To satisfy our restrictions, client utilities' assistance would have to be explicitly accepted by the clients. In fact, some of the client utility assistants are declared as nested aspects. These are similar in spirit to explicitly excepted assistance and would be implicitly accepted under our proposal.

In the AspectJ programming guide, there were 11 examples of client utilities. These include `Bean.BoundPoint`, the three Point aspects in the `introduction` package, the `SubjectObserverProtocol` and its implementation, three nested aspects the `spacewar` package, and the `telecom` package's `Billing` and `Timing` aspects. In Kiselev's book the `Authentication`, `Exceptions`, and `NewLogging` aspects were client utilities.

Other example aspects that could be implemented as assistants might be considered *implementation utilities*. These assistants encapsulate a unit of crosscutting concern related to a single module, e.g., enforcing a common precondition across the methods of a class. In our proposal each implementation utility would be accepted by the module that it advises, creating a mutual dependency. However, in all the examples this mutual dependency could be fixed by nesting the implementation utility inside the advised module. We would also require that the `call` join points in these aspects be changed to `execution` join points.

In the AspectJ programming guide, there were 4 examples of implementation utilities, all the `spacewar` package. All 4 of Kiselev's "runtime aspects" were also implementation utilities: `Pooling`, `ReadCache`, `ConnectionChecking`, and `OutputStreamBuffering`.

3.3.3 Combined To satisfy our restrictions, one example aspect, the `Debug` aspect of the AspectJ `spacewar` example, would require a combination of spectators and assistants. This aspect would be a spectator, except that it provides after advice to a GUI frame's constructor, to add debugging options to the frame's menu bar. To support this pattern with our restrictions, the GUI frame would have to accept assistance from an assistant, say `AdditionalMenuConcern`. This assistant would provide methods allowing other code to add to the GUI frame's menu bar. The debugging aspect would become a spectator viewing the program and using the methods provided by `AdditionalMenuConcern` to display the debugging menus.

3.3.4 Summary of Evaluation Our proposed language features add restrictions to AspectJ. But our evaluation shows that these restrictions do not restrict the expressiveness of the language. In fact, most of the examples studied fall neatly into three categories: spectators, implementation utilities, and client utilities. The latter two are assistants with natural locations for explicit acceptance. This supports our contention (in Section 1.2) that experienced aspect-oriented

programmers are already following disciplines, like the MAO discipline, that enable modular reasoning.

4 Discussion

The current work does not address AspectJ's introduction mechanisms and `declare parents` construct. An aspect that used introduction to replace an inherited method of a class with an overriding method would clearly change the behavior of that class and would therefore be an assistant. On the other hand, suppose an aspect introduced a new, non-overriding method to a class. Since no other code could have called that new method, this introduction should not change the behavior of existing code. So such an introduction could be allowed in a spectator. (This case is similar to the introduction of external generic functions via MultiJava's "open class" mechanism [9].) However, we leave this decision for future work, because introduction involves subtle modularity issues, particularly for avoiding runtime ambiguities. These issues are made more complex by the possibility that the newly introduced methods might be advised by existing aspects, or that a change in the base program might make a previously "fresh" introduced method into an overriding one.

This paper also does not address AspectJ's `declare error` and `declare warning` constructs. But these constructs do not change the behavior of a program in any way. Instead they provide advice to the compiler itself, telling the compiler that if certain join points are detected, then an error or warning should be issued. Thus, these constructs can be allowed in spectator aspects.

An aspect that used the `declare soft` construct, which converts checked exceptions to unchecked ones, would clearly change the control flow of a program to which it was applied. Such an aspect is thus an assistant.

An alternative technique in regular AspectJ for implementing the hyper-cutting pattern (discussed in Section 3.1.1) is to use `within` and `withincode` pointcut descriptors to statically limit the code to which a particular piece of advice applies. Unlike accepts clauses and concern maps, this approach buries the applicability of advice within pointcut descriptors. The `within` approach to hyper-cutting makes it even harder to find applicable advice than with the (already non-modular) marker interface approach. The `within` approach is a handy implementation technique, however. In our prototype implementation of accepts clauses and concern maps, we use the `within` technique in the intermediate code.

5 Related Work

Kiczales and Mezini [19] argue that the modularity properties of aspect-oriented programs should be understood in terms of "aspect-aware interfaces". These interfaces are based on the global configuration of a system and essentially provide a bi-directional mapping from methods to associated advice, and advice to advised methods. Assuming the existence of such an aspect-aware interface, the authors then argue that reasoning about cross-cutting concerns is simpler in an

aspect-oriented implementation than in a purely object-oriented implementation of the same program. However, the paper presents no formal analysis; instead a single example is used to support the claims. That example examines the process necessary to refactor a program written in an aspect-oriented style, versus the same program written in a pure object-oriented style. In either program an implementation-level search is needed to understand the design. In the aspect-oriented version, the change necessary to support the refactoring can be localized in a single module. But it seems that the localization of this change is enabled by AspectJ's predicates over the current call stack: `cflow` and `cflowbelow`, and not by the the cross-cutting expressiveness of aspect-oriented programming *per se*. That is, the example's refactoring could be done in a pure object-oriented language that had such predicates.[5] So essentially, their argument is that reasoning is no less modular in aspect-oriented programs than it is in object-oriented programs where cross-cutting concerns are scattered and tangled. The long-range objective of our work is to demonstrate that, given the appropriate design discipline, reasoning in aspect-oriented programs can actually be *more* modular.

The "aspectual collaborations" of Lieberherr et al. [27] are somewhat related to our concern maps. With aspectual collaborations, advice is declared within modules using abstract representations of the pointcuts to be matched. Modules must be explicitly composed. This composition reifies the pointcuts, making explicit the ways in which one module's advice might attach to another module's methods. While aspectual collaborations offer some nice modularity properties, they require all composition to be done at the top-level, instead of at any level of the module hierarchy as for concern maps. Aspectual collaborations do not address the problem of reasoning about AspectJ programs, since aspectual collaborations do not use AspectJ. Finally, it is unclear how anything with the flexibility of spectators could be expressed using aspectual collaborations.

Katz and Gil [18] suggest that the body of work on "superimposition", for reasoning about distributed algorithms, might provide a fertile ground for ideas in developing aspect-oriented programming. They briefly sketch three categories of aspects. Their "spectative" category matches our notion of spectators. The other two categories of aspects they mention map to our notion of assistants. However, they do not consider a language design that might help enforce and exploit these distinctions. Because of this they do not address anything like our concern maps and they do not talk about how one might enforce that declared spectators have no observable side effects.

Sullivan et al. [33] compare using non-invasive AspectJ versus a new approach, which they call "design rules". Design rules place control flow restrictions on programs, requiring an explicit specification of the join points to which advice may attach and placing pre- and post-conditions on any advice that attaches to those join points. They analyze the implementation of an application using the standard AspectJ approach and using their design rules approach. They conclude that the non-invasive AspectJ approach may decouple base pro-

[5] This idea is based on a post to the `aosd-discuss` mailing list by Christina Lopes that mentioned including call-stack predicates in a non-aspect-oriented language.

gram development from aspects, but tightly couples aspects to the development of the base program. Their system is really just a design methodology, it does not include static checking and has not been formalized.

Dantas and Walker [10] present a calculus for "harmless advice", based on an extension of the typed lambda calculus, with references and objects. Dantas and Walker use a type system with "protection domains" to keep aspects from altering the data of the base program. In keeping with this non-interference property, they do not allow advice to change values when proceeding to the base program. They use the lattice-ordered protection domains to prevent lower integrity data, such as that generated by advice, from interfering with higher integrity data, such as that from the base program. These protection domains in their core calculus are more expressive than our concern domains. This expressiveness is relinquished in their user-level calculus. Their user-level calculus generates a single protection domain for the base program and a separate protection domain for each declared aspect. Thus the protection system is tied to the program structure and, unlike concern domains, cannot represent designs where the protection domains cross-cut the modularity structure of the program. For example, they cannot represent potentially "surprising" assistants, which are important in many uses of AspectJ.

Aldrich and Chambers [1] present an ownership type system that is decoupled from the encapsulation relation in a program. Their system allows very fine-grained specification, and static typechecking, of the aliasing relationships in a program. The system replaces the traditional owners-as-dominators property of ownership type systems with a link soundness property. The link soundness property says that the only interdomain aliases are those between "ownership domains" that are explicitly given permission to hold such aliases. These permissions are closely related to our concern domains. The authors' ownership domains are significantly more fine-grained than our concern domains, with each *object* having its own member domains. Their system includes a single global domain, called "shared", to which objects belong by default. Their system is not designed to control mutation and does not distinguish between read-only and write-enabled pointers.

6 Conclusions And Future Work

In this paper we have shown that the MAO discipline: (1) allows modular reasoning, (2) permits the use of existing aspect-oriented idioms for separation of concerns, (3) can be verified by a combination of static typechecking and simple verification conditions, and (4) can be incorporated into a practical, aspect-oriented language.

The MAO discipline addresses the twin problems of modular reasoning in aspect-oriented languages: (1) unseen aspects may apply to the code, and (2) aspects may be developed without complete knowledge of the code that will be advised. The discipline addresses these problems by separating aspects into two sorts: benign spectators and surprising assistants. The discipline also requires

that the aspect author and the programmer of advised code share the burden of ensuring modular reasoning.

We have argued that a few additional language features are sufficient to support the MAO discipline in a language like AspectJ. Our proposed features statically separate aspects into assistants and spectators. Assistants have the full power of AspectJ's aspects, but to maintain modular reasoning we require that assistants be explicitly accepted. Spectators are constrained to not modify the behavior of the modules that they view. This allows modular reasoning about the advised code, even if spectators remain unseen.

Our proposal introduces concern maps to allow acceptance of assistance, while avoiding the scattering of duplicate accept clauses throughout a program. It also adds concern domain declarations to verify spectatorhood and to detect unwanted tangling.

We have described an evaluation of the practical effect of our proposed language features (with the exception of concern domains). Our evaluation looked at three sets of AspectJ examples: the *AspectJ Programming Guide*, and Kiselev's and Laddad's books. For the AspectJ constructs considered in the current work, our language features can handle their examples with no changes in most cases, and minor changes otherwise. The ready identification of places to accept assistance from client or implementation utilities in these examples supports our contention that experienced aspect-oriented programmers are already using disciplines, like the MAO discipline, that enable modular reasoning.

The major technical challenge for our proposal is checking that aspects declared as spectators meet our definition, as discussed in Section 3.2. We have specified constraints on spectators that allow modular reasoning about their (lack of) effect on control flow. A type system that restricts aliasing and mutation allows modular reasoning about spectators (lack of) effect on the relevant state of the modules they view. Clifton's dissertation [7] presents:

- an aspect-oriented calculus for investigating these ideas in a formal setting, and
- a sound type-system for the calculus that statically enforces our proposed restrictions on spectators.

Clifton's evaluation of existing AspectJ examples, discussed in Section 3.3, provides some evidence as to the practicality of our proposal. Clifton's formal study provides strong, theoretical support for our claims. However, some open problems remain.

- To fully support claim 3, we must formalize the verification conditions entailed by our proposed specification language constructs, and prove that reasoning using effective specifications is sound.
- To fully support claim 4, we must demonstrate the incorporation of our proposed language features into a practical programming language. We must use that language to implement realistic-scale programs.

Each of these open problems entails a significant research program.

6.1 Future Work

In additional to the open problems discussed above, the current work also suggests several other interesting lines of investigation.

Our study focused on sequential aspect-oriented programs. This focus excludes some interesting techniques. For example, Laddad's worker object creation pattern uses proceed closures [23, §8.1]. In this pattern, advice captures a `proceed` expression inside an instance of an anonymous `Runnable` class. This allows the advised code to be postponed, or executed immediately but in a new thread. Such use of `proceed` is fascinating, but to study it we would need a formalism that models concurrent processes. It may be that some variant of the π-calculus would be appropriate for this study [30].

It would also be interesting to compare the reasoning problem in aspect-oriented programming to reasoning in component-based programming [34].

Component-based programming requires components to specify their expectations of external modules. This allows separate verification of the components, provided that expectations are checked at composition time. In the MAO discipline, the steps are performed in a different order. Each piece of advice is separately verified. Then, for a given composition, the specifications are composed to determine the effective specification of an advised join point. This effective specification is used to verify the code that triggers the join point. A problem that would surface as an unsatisfied expectation in component-based programming, appears as an effective specification like `requires false; ensures true;`. It is essentially useless for verifying the client code.

It seems that the essential trade-off is that component-based programming provides simpler compositional reasoning, but less expressive composition mechanisms versus aspect-oriented programming. Aspectual collaborations [27] can be considered a mid-point between these extremes, though that system does not include formal specifications.

Bibliography

[1] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In Odersky [32], pages 1–25.

[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] P. America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/-Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, New York, NY, 1991.

[4] AspectJ Team. The AspectJ programming guide. Available from `http://eclipse.org/aspectj`, Oct. 2003.

[5] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10): 785–798, Oct. 1995.

[6] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, volume 37(11) of *ACM SIGPLAN Notices*, pages 292–310. ACM, Nov. 2002. URL `http://doi.acm.org/10.1145/582419.582447`.

[7] 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. URL `ftp://ftp.cs.iastate.edu/pub/techreports/TR05-15/TR.pdf`.

[8] 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 Report, pages 33–44. Department of Computer Science, Iowa State University, Apr. 2002. URL `ftp://ftp.cs.iastate.edu/pub/techreports/TR02-06/TR.pdf`.

[9] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 35(10) of *ACM SIGPLAN Notices*, pages 130–145, New York, NY, Oct. 2000. ACM. URL `ftp://ftp.cs.iastate.edu/pub/techreports/TR00-06/TR.ps.gz`.

[10] D. S. Dantas and D. Walker. Harmless advice. In *The 12th international workshop on Foundations of object-oriented languages*. ACM, 2005.

[11] 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.

[12] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 2005. To appear.

[13] 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. to appear.

[14] FSE. *Proc. of the 13th ACM SIGSOFT symposium on the Foundations of software engineering (FSE-13)*, Lisbon, Portugal, 2005. ACM Press.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[16] D. S. Gibson, B. W. Weide, S. M. Pike, and S. H. Edwards. Toward a normative theory for component-based system design. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 10, pages 211–230. Cambridge University Press, New York, NY, 2000. ISBN 0-521-77164-1.

[17] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2005.

[18] S. Katz and Y. Gil. Aspects and superimpositions. In *Int'l Workshop on Aspect-Oriented Programming (ECOOP 1999)*, June 1999. URL `http://trese.cs.utwente.nl/aop-ecoop99/papers/katz.pdf`.

[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] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *ECOOP '97—Object-Oriented Programming 11th European Conference, Jyväskylä, Finland, Proceedings*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, NY, June 1997.

[21] 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. URL `http://doi.acm.org/10.1145/383845.383858`.

[22] I. Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams Publishing, Indianapolis, 2003.

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

[24] D. Lafferty and V. Cahill. Language-independent aspect-oriented programming. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 1–12, Anaheim, California, USA, 2003. ACM Press.

[25] G. T. Leavens and W. E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8): 705–778, Nov. 1995.

[26] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3):185–208, Mar. 2005.

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

[28] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.

[29] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.

[30] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, LFCS, Oct. 1991. *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991.

[31] 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.

[32] M. Odersky, editor. *ECOOP '04 — Object-Oriented Programming European Conference*, volume 3086 of *Lecture Notes in Computer Science*, Oslo, Norway, 2004. Springer-Verlag.

[33] K. Sullivan, W. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information hiding interfaces for aspect-oriented design. In FSE FSE [14].

[34] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, second edition edition, 2002.

[35] 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. URL `citeseer.nj.nec.com/tarr99degrees.html`.