# Spectators and Assistants: Enabling Modular Aspect-Oriented Reasoning

Curtis Clifton and Gary T. Leavens
Department of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA 50011-1040 USA
+1 515 294 1580

{cclifton,leavens}@cs.iastate.edu

## ABSTRACT

In current aspect-oriented languages, separate compilation and modular reasoning are not possible. This detracts from comprehensibility and impedes maintenance efforts, contrary to a stated goal of aspect-oriented programming. We describe language features that would allow aspect-oriented languages to provide separate compilation and modular reasoning. We demonstrate that existing programs written in AspectJ can be easily rewritten using these features.

## Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*classes and objects, control structures, inheritance, modules, packages, procedures, functions and subroutines, advice, spectators, assistants, aspects*; D.3.2 [**Programming Languages**]: Language Classifications—*object-oriented languages, multi-paradigm languages, AspectJ*; D.3.2 [**Programming Techniques**]: Object-oriented programming—*aspect-oriented programming*

## Keywords

Spectators, assistants, aspect-oriented programming, modular reasoning, separate compilation, AspectJ language

## 1. INTRODUCTION

Much of the work on aspect-oriented programming languages makes reference to the work of Parnas [27]. That work argues that the modules into which a system is decomposed should be chosen to provide benefits in three areas. Parnas writes (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.

While much has been written about aspect-oriented programming as it relates to Parnas's second point, his third point is the primary concern of this paper. We contend that current aspect-oriented programming languages do not provide this third benefit in general, because they require systems to be studied in their entirety.

After describing and motivating the problem in this introduction, in Section 2 we propose a simple categorization of aspects that would bring these languages much closer to Parnas's ideal without any practical loss of expressiveness.

We begin by defining a notion of modular reasoning corresponding to Parnas's third benefit. Subsequent subsections in this introduction show how such modular reasoning is possible in the Java Programming Language [1, 11] but problematic in AspectJ [14].

Modular reasoning is a proxy for separate compilation—it is straightforward for a language that allows modular reasoning to also allow separate compilation.

We work with Java and AspectJ for concreteness. However, we believe our ideas are independent of these languages and are generally applicable to the class of aspect-oriented languages.

### 1.1 Modular Reasoning

Before delving into the details, it is useful to define our terms. *Modular reasoning* is the process of understanding a system 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 $M$ along with the signature and behavior of any modules referred to by $M$. For example, in Java a single compilation unit, typically a file declaring a single top-level type (class or interface), is a module. The behavior of a module can be thought of concretely as its code. Often programmers reason about modules using informal abstractions, e.g., "This method returns true if the given file exists". (In a more expressive language, such as Eiffel [21] or Java annotated with

JML [18], the abstract behavior can be given explicitly using pre- and postconditions, frame axioms, and invariants; such specifications serve as contracts that allow one to separately reason about the behavior and correctness of an implementation.)

Our interest in modular reasoning in aspect-oriented programming languages is motivated in part by our earlier work on MultiJava [6, 8]. In that work we were concerned with modular static type checking and compilation. These tasks are closely related to the issue of modular reasoning. A language that supports modular reasoning can also permit separate compilation, as well as modular implementations of other tools (e.g., optimizers, verifiers, and model checkers). Thus, mechanisms that permit modular reasoning have many benefits. In Section 2 we will discuss how mechanisms that allow module reasoning also allow separate compilation.

## 1.2 Modular Reasoning in Java

Java without aspect-oriented extensions supports modular reasoning. Consider the examples in Figure 1 and Figure 2, motivated by examples in Kiczales, *et al.* [14] and annotated with javadoc comments [9].

Suppose one wanted to write code that manipulates objects of type `FigureElement`. One could reason about such objects based solely on the information contained in Figure 1. For example, one would know the objects support a method named `move` that takes two arguments, `dx` and `dy`, of type `int` and that when invoked the method would leave the object in a state where the values returned by `getX` and `getY` were increased by `dx` and `dy`, respectively.

Similarly, suppose one wanted to write code that manipulated instances of `Rectangle`. One could reason about these instances based on Figure 2, along with the modules referred to in that code. To reason about `Rectangle`'s `getArea` method one would just need to consider the `Rectangle` module. However, to reason about a call to `move` on an instance of `Rectangle` one would have to consider the behavior of the `FigureElement` module, since the `Rectangle` module inherits `move` (and several other methods) from `FigureElement`. This consideration of `FigureElement` is modular because `FigureElement` is explicitly referred to by the clause

```
extends FigureElement
```

in the declaration of `Rectangle`.[1]

## 1.3 Non-modular Reasoning in AspectJ

Next we show that modular reasoning is not a general property of AspectJ by considering an aspect-oriented extension to our previous example. Figure 3 gives an aspect, `MoveLimiting`, that modifies the behavior of `FigureElement` instances. `MoveLimiting` declares two pieces of *before-advice*, or code to be executed before traversing a join point into a method body. A *join point* is an arc in the dynamic call graph of a program.[2] The first before-advice in `MoveLimiting` is applicable to all join points described by the *pointcut*, or join point description, `setPC`. This pointcut describes the join point where a method with the signature `void set*(int)` is called. The `args` keyword is used to give names to the arguments of the method call. The asterisk (`*`) in the signature is a wildcard; wildcards can be used to give very general pointcuts. The `setPC` pointcut matches both the `setX` and `setY` methods of `FigureElement`, and binds the argument of the method to a variable `z` that can be used within advice that references the pointcut. The second piece of before-advice in `MoveLimiting` uses the pointcut `move`. This pointcut uses the `target` keyword to give the before-advice access to the target object of the advised method call. (AspectJ also has *after-advice*, executed after traversing a join point out of a method body, and *around-advice*, which applies to the join points into and out of a method body.)

Both pieces of before-advice shown in Figure 3 throw an exception if allowing the invocation to continue would result in either the x- or y-coordinate of the target object becoming negative. If the exception is not thrown, then control flow is passed on to the advised method. In AspectJ the advice is applied by the compiler without explicit reference to the aspect from either the `FigureElement` 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 `FigureElement` module or a client module has no way to detect that the behavior of the `move` method will be changed when the `FigureElement` module and `MoveLimiting` are compiled together. In AspectJ the programmer must potentially consider every combination of such aspects and the `FigureElement` class in order to reason about the `FigureElement` module. Some potentially applicable aspects may not even name `FigureElement` directly, but instead may use wildcards type patterns. So, in general, a programmer cannot "study the system one module at a time" [27].

## 1.4 Problem Summary

In a paper from ECOOP 2001, arguing for aspect-oriented programming, Kiczales, *et al.* state [15, p. 327]:

> We would like the modularity of a system to reflect the way 'we want to think about it' rather than the way the language or other tools force us to think about it.

However, we have seen that the lack of support for modular reasoning can sometimes prevent us from thinking about a system "the way we want to think about it". In AspectJ, tool support is provided to compensate for this lack of modularity. 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. Other tools for processing AspectJ source code (e.g., type checkers, compilers, and optimizers) also require a whole-program analysis.

The problem is to design a small set of language features that obviate the need for this whole-program analysis either by the programmer or by supporting tools.

The remainder of this paper is organized as follows. Section 2 describes some language features that are sufficient for modular reasoning in an aspect-oriented programming language. Section 3 evaluates our proposal. Section 4 discusses some additional advantages and limitations of our proposal. Section 5 outlines related work. Section 6 concludes.

[1] In Java every class is implicitly a subclass of `java.lang.Object`. Thus reasoning in Java also requires that one consider `Object`'s behavior. However, because it is common to all classes we do not consider this implicit reference to be non-modular.

[2] Join points in AspectJ are actually more general than what we describe. For example, join points can refer to field references and exception handlers [2].

```
package aosd03;

public abstract class FigureElement {
    private int _x = 0, _y = 0;

    /** Initializes this figure element to be
     *  at the given coordinates. */
    public FigureElement( int x, int y ) {
        _x = x;  _y = y;
    }

    /** Returns the x-coordinate of this. */
    public int getX() { return _x; }

    /** Returns the y-coordinate of this. */
    public int getY() { return _y; }


    /** Returns the area of this. */
    public abstract int getArea();

    /** Makes the x-coordinate of this figure
     *  element be as given. */
    public void setX( int x ) {
        _x = x;
    }

    /** Makes the y-coordinate of this figure
     *  element be as given. */
    public void setY( int y ) {
        _y = y;
    }

    /** Changes the x-coordinate of this figure
     *  element by dx units and the y-coordinate
     *  by dy units. */
    public void move( int dx, int dy ) {
        setX( getX() + dx );
        setY( getY() + dy );
    }

    /* ... */
}
```

**Figure 1: A Java module declaring an abstract class**

```
package aosd03;

public class Rectangle extends FigureElement {
    private int _height, _width;

    /** Initializes this rectangle. */
    public Rectangle(int x, int y, int h, int w) {
        super(x,y);
        _height = h;
        _width = w;
    }

    public int getArea() {
        return _height * _width;
    }

    /* ... */
}
```

**Figure 2: A Java module declaring a class**

```
package aosd03;

aspect MoveLimiting {

    pointcut setPC(int z):
        args(z) && call(void set*(int));

    /** Throws IllegalArgException if argument is
     *  negative. */
    before(int z): setPC(z) {
        if (z < 0) {
            throw new IllegalArgException();
        }
    }

    pointcut move(FigureElement fe, int dx, int dy):
        target(fe) && args(dx, dy)
        && call(void move(int,int));

    /** Throws IllegalArgException if allowing
     *  move will make either coordinate
     *  negative. **/
    before(FigureElement fe, int dx, int dy):
        move(fe, dx, dy) {
        int nextX = fe.getX() + dx;
        int nextY = fe.getY() + dy;
        if (nextX < 0 || nextY < 0) {
            throw new
                IllegalArgException();
        }
    }

    /* ... */
}
```

**Figure 3: An AspectJ module providing advice for** `FigureElement`

## 2. PROPOSED FEATURES

We have shown that AspectJ in general does not support modular reasoning; the behavior of a module can only be determined by a whole-program analysis. In this section we describe some language features that are sufficient to support modular reasoning and separate compilation in an aspect-oriented language. For concreteness we describe these features as extensions to AspectJ.

The key feature to support modular reasoning in our proposal is to divide aspects into two sorts: assistants and spectators. "Spectators" are limited in that they may not change the behavior of the modules they apply to (in a way to be made concrete later), "assistants" are not limited in this way. Since spectators do not change behavior, they preserve modular reasoning even when applied without explicit reference by the modules they view. Hence spectators preserve most of the flexibility of the current version of AspectJ. Because assistants can change the behavior of the modules to which they apply, to maintain modular reasoning they can only be applied in modules that explicitly reference them. Section 2.1 describes assistants and Section 2.2 describes spectators. Section 2.3 describes briefly how this categorization of aspects can provide for separate compilation.

### 2.1 Assistants

We call aspects that can change the behavior of a module *assistants*. The `MoveLimiting` aspect of Figure 3 is an assistant. 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:

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

#### 2.1.1 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 [16, 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.

To automate this hyper-cutting pattern, and to avoid potentially 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 [11, §6.5]. Multiple accept clauses may appear in a single module, following any import clauses. For example, the `FigureElement` module could accept the `MoveLimiting` assistance by declaring:

```
accept MoveLimiting;
```

We will generalize this idea with aspect 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 sorts of join points that roughly simulate this behavior. Advice on join points described via `call` pointcuts is woven into all client code that is compiled together with that advice. Advice on join points described via `execution` pointcuts is woven into the implementation code (assuming the implementation and advice are compiled together). 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.

In general a module may accept assistance from multiple assistants and both a client and an implementation module may accept assistance. The composition of assistant and implementation code is formed respecting the following (symmetric) order at each join point:

1. Apply any before-advice accepted by the client module in the order that it is accepted.
2. Apply the "before-part" (i.e., the code preceding a proceeds expression) of any around-advice accepted by the client module in the order that it is accepted.
3. Apply any before-advice accepted by the implementation module in the order that it is accepted.
4. Apply the before-part of any around advice accepted by the implementation module in the order that it is accepted.
5. Execute the implementation module code.
6. Apply the "after-part" (i.e., the code following a proceeds expression) of any around advice accepted by the implementation module in the reverse order from which it is accepted.
7. Apply any after-advice accepted by the implementation in the reverse order from which it is accepted.
8. Apply the after-part of any around advice accepted by the client module in the reverse order from which it is accepted.
9. Apply any after-advice accepted by the client module in the reverse order from which it is accepted.

This ordering ensures, for example, that the first assistance accepted by the client is "nearest" to the client and that the last assistance accepted by the implementation is nearest to the implementation on any control flow path. Multiple applicable advice bodies in a single assistant are accepted in the order given in the assistant's declaration, or in the reverse order for after-advice and the after-part of around-advice. Inherited advice is considered to appear at the end of the inheriting aspects; this respects the ordering for inherited advice defined for AspectJ [15, §3.5]. However, the ordering of all advice is underspecified in AspectJ. Our symmetric total ordering differs from the asymmetric ordering of advice implemented in the current version of ajc [16, p. 182].

In the current version of AspectJ the ordering of advice from unrelated aspects is determined by the ordering of arguments to ajc, Xerox's AspectJ compiler, along with any dominates-declarations. A dominates-declaration allows the programmer to specify that one aspect is to be executed before another named aspect. Dominates-declarations are limited in their expressiveness compared to explicitly accepted assistance, which gives finer control to the programmer. Explicitly accepted assistance also eliminates the dependence on command-line arguments to specify ordering, moving this specification into the language itself.

(For simplicity and modularity we have decided for the present to confine acceptance of assistance to the module in which it is explicitly accepted. For example, assistance accepted by `FigureElement` is not automatically applied to invocations of methods declared in `Rectangle`. On the other hand, if for a particular method `Rectangle` does not override `FigureElement`'s implementation, then the inherited method carries with it the assistance accepted in the `FigureElement` module. This approach also provides flexibility since the programmer can always add an accept clause to the subclass module or override a superclass method; this gains assistance in the first case and "shadows" assistance acceptance in the second. Similar considerations apply for assistance accepted by a superclass module of a client class. Also for simplicity we do not allow interfaces to accept assistance. Future work may reevaluate these decisions.)

### 2.1.2 Aspect Maps

Modular reasoning in aspect-oriented programming languages can be achieved if we require modules to explicitly accept assistance. But some assistants are applicable to code through-out an entire package or program, for example, exception handlers. It would be inconvenient (to say the least) to include accept clauses for these assistants in every module, and error prone to have to remember to add accept clauses for these assistants to every new module.[3]

We introduce aspect maps to avoid these problems. An *aspect 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. Each Java package may contain at most one aspect map. The aspect map for a package is given in a file named `package.map` stored in the directory containing the package source code. The syntax for an aspect map is given in Figure 4. An example aspect map is given in Figure 5.

[3]Such accept clauses would also represent code tangling, though not as severely as duplicating the advice code in every advised method.

| | | |
|---|---|---|
| *AspectMap* | ::= | *PackageDecl ImportDecls*$_{opt}$ *MappingList*$_{opt}$ |
| *PackageDecl* | ::= | `package` *Identifier* `;` |
| *MappingList* | ::= | *Mapping MappingList*$_{opt}$ |
| *Mapping* | ::= | *TypePat* `{` *AcceptList*$_{opt}$ `}` |
| *AcceptList* | ::= | *AcceptClause AcceptList*$_{opt}$ |
| *AcceptClause* | ::= | `accept` *Identifier* `;` |

**Figure 4: The syntax of aspect maps; *TypePat* refers to type patterns in [2, Appendix A], *ImportDecls* refers to regular Java import declarations [11, §7.5]**

```
package aosd03;

* {
  accept MoveLimiting;
}

Rectangle {
  accept AreaStretching;
}
```

**Figure 5: An example aspect map**

The type pattern `*` (preceding the '{' in the example) says that all types in the `aosd03` package accept the `MoveLimiting` assistant. (We do not allow aspect maps to specify fully qualified names in type patterns; instead we implicitly prepend the package name of the map's package to the given pattern. 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 the example says that, in addition to the `MoveLimiting` assistant, `aosd03.Rectangle` also accepts the `AreaStretching` assistant.[4] As with accept clauses in modules, the identifier in an accept clause of an aspect map is subject to Java's usual namespace rules for packages and imports.

One can think of aspect maps as a kind of introduction; they add accepts clauses to modules in the local package and subpackages.

The assistance accepted via aspect maps still allows modular reasoning. To wit, the package clause at the beginning of a module names all the possible locations where an aspect 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 aspect map. More specifically, the assistance accepted by a given module consists of:

- all assistants named in accept clauses in the module,
- all assistants to which the module is mapped by the `package.map` file for the module's package, and
- all assistants to which the module is mapped by any `package.map` files in packages containing the module's package.

This recursive search for acceptance of assistance in the module's package and outer packages allows the programmer to specify program-wide assistance in the root package

[4]The `AreaStretching` assistant is omitted from this paper for space reasons.

```
package aosd03;
spectator aspect DistanceTracking {

    /** Tracks total distance moved by all figure
     * elements. */
    private double _distance;

    before(FigureElement fe, int dx, int dy):
        MoveLimiting.move(fe, dx, dy)
    {
        _distance += Math.sqrt( Math.pow(dx,2) +
                                Math.pow(dy,2) );
        System.err.println("Total distance: " +
                           _distance);
    }
}
```

**Figure 6: A spectator aspect**

of a hierarchy, package-specific assistance in a single package, and module specific assistance in a single module. Future work will evaluate inheritance mechanisms for aspect maps that would allow finer grained control than the simple unioning of maps from outer packages presented here.

## 2.2 Spectators

Explicitly accepted assistance supports modular reasoning. Aspects maps give the programmer flexibility in accepting assistance. But what about "development aspects"[14, 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 aspect 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 behavior of any other module. Because it does not change the behavior, we will say that a spectator *views* (rather than "advises") methods.

In concrete terms, a spectator may only mutate the state that it owns (in the sense of alias control systems like [22, 24]) and it must not change the control flow to or from a viewed method. In addition to mutating owned state it seems reasonable to allow spectators to change accessible global state as well, since a Java module cannot rely on that state not changing during an invocation (modulo synchronization mechanisms). The term "spectator" is intended to connote the hands-off role of these aspects.

For example, Figure 6 gives a spectator called `DistanceTracking`. The `spectator` modifier (in the second line) declares that this aspect must not change the behavior of any other module. This spectator mutates its own state by incrementing `_distance` and mutates the global state by printing to `System.err`. However, it does not change the behavior of `FigureElement`'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 results are 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 [10, pp. 293–303].

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.

The primary challenge of implementing this part of our proposal lies in determining whether a given aspect is really a spectator. We envision a static analysis that conservatively verifies this. This analysis 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, we can restrict the sort of control flow constructs allowed in spectators to achieve an approximate solution. We propose that in spectators:

- before-advice must not explicitly throw an 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 explicitly throw an exception on any control flow path.

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. A more conservative solution to control flow might disallow loops, method calls, and synchronized code within a spectator's advice. A completely conservative solution is not possible in a Java-like language since executing any advice in a spectator requires more memory than just executing the viewed method. This additional memory usage could result in an `OutOfMemoryError` that prevents control flow from continuing to the advised method. Because of this, and the draconian nature of the more conservative solution, our approximate solution that disallows all explicitly thrown exceptions in the advice and handles any checked exceptions in methods called by the advice seems reasonable.[5]

Additionally, the checks for "spectatorness" must 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 mutation analysis for spectators is closely related to the problem of verifying frame axioms [4]. In fact we can think of spectators as having an implicit frame axiom that prevents modification of locations that are relevant to the receiver, the arguments, or the value returned or exception thrown by the viewed method. (Intuitively the relevant locations are those that, if changed, would change the abstract state of the object [22, 23].)

The main difficulty with statically verifying this lack of relevant mutations is how to deal with aliasing. For example, suppose we have a logging spectator that uses an array

[5]We imagine that, in many cases, program verification techniques could be used to prove termination and that no unchecked exceptions are thrown.

to track the elements added to some `Set` object. Suppose `Set` uses an array for its representation. If the spectator's array and the `Set`'s array are aliased, we might end up with an element being added to the array twice—possibly violating `Set`'s invariant and changing its behavior. There is a substantial body of work on alias control that may be useful in attacking this [22, 24].

Another practical difficulty is related to library methods used in a spectator. Consider the before-advice of `DistanceTracking` in Figure 6. The arguments to the advised method, `dx` and `dy`, are passed to the library method, `Math.pow()`. Now since `dx` and `dy` are of type `int` it is not possible for `Math.pow()` to mutate them, so `DistanceTracking` is clearly a spectator. But suppose `dx` were of a reference type. Then if `dx` were passed to a library method we would need to verify that the library method did not mutate the object referred to by `dx`. A mechanism like the `pure` modifier in JML could be used to annotate libraries for this purpose [18]. Library methods would also have to be annotated based on whatever alias control mechanism was chosen for the language.

## 2.3 Separate Compilation

Because of the generality of aspects without our restrictions and limitations of the target Java Virtual Machine (or JVM) [20], AspectJ currently requires whole-program compilation [15]. In our proposal, because assistance is explicitly accepted, it is a simple matter to support separate compilation for modules that accept assistance; the compiler just weaves it into the accepting modules.

On the other hand, spectators present an interesting challenge for separate compilation. On the surface, since spectators do not change the behavior of other modules, it should be possible to separately compile them. And indeed this is true—except for the issue of dispatching to spectators. The generality of spectator application means that they can potentially be dispatched to from any join point.

A second challenge for separate compilation of spectators is related to a capability that we would like spectators to have. Because spectators cannot change the behavior of other modules we would like to be able to apply them to a program that is already running, for example to diagnose a problem in a long-running server application.

An aspect-oriented virtual machine could address both of these concerns by including facilities for applying and removing spectators from already running programs and for dispatching to spectators at the appropriate join points.[6] Others have suggested that separate compilation for AspectJ is possible using techniques such as modified virtual machines [15, p. 343]. However, with this approach dynamic dispatch would be needed for all assistants. In our proposal the explicit acceptance of assistance allows the separate compilation of assistants by compile-time weaving. Thus the less-efficient dynamic dispatch is only needed for spectators. Furthermore, spectators that were always used could be named in accept clauses (or aspect maps) to gain the efficiency of compile-time weaving.

[6]From a goal of generating initial acceptance for aspect-oriented languages, using the ubiquitous JVM as a target platform is a reasonable choice. But when anticipating future directions in language design we should be free of this constraint.

# 3. EVALUATION

This section briefly evaluates the practical consequences of our proposal. Our evaluation is limited to a review of existing programs. We first consider the aspect-oriented programming guidelines suggested in the ATLAS case study [13]. Then we survey the example aspects from the AspectJ Programmers Guide [2] and Kiselev's book [16].

## 3.1 ATLAS Case Study

In the ATLAS case study [13], the authors proposed several guidelines to make working with aspects easier. These were proposed since they had discovered that (p. 346):

> The extra flexibility provided by aspects is not always an advantage. If too much functionality is introduced from an aspect it may be difficult for the next developer—or the same developer a few months later—to read through and understand the code base.

One of Kersten and Murphy's suggestions is to limit coupling between aspects and classes to promote reuse. Specifically, they suggest that one should avoid the case where an aspect explicitly references a class and that class explicitly references the aspect, since then the class and aspect are mutually dependent. Such mutual dependencies prevent independent reuse. Is this suggestion problematic for our requirement that modules explicitly accept assistance? No, because explicit acceptance does not necessarily imply mutual dependence between aspects and classes. Suppose an implementation module, $M$, accepts assistance from an assistant, $A$, and $A$ is applicable to $M$. If $A$ explicitly references $M$, then the modules are mutually dependent. However, if $A$ only applies to $M$ because of wildcard-based pattern matching and does not explicitly reference $M$, then the modules are not mutually dependent. Another way to avoid this dependence is to have $M$ accept the advice of an abstract assistant that $A$ extends.

Client acceptance provides another way to avoid mutual dependence. Suppose a client module, $C$, accepts assistance from an assistant, $A'$, and $A'$ only changes the behavior of modules referenced by $C$, but does not change $C$'s behavior. In this case $A'$ and $C$ are not mutually dependent. In sum, programmers can reduce mutual dependency by having clients accept assistance or by limiting explicit references to classes from assistants.

Kersten and Murphy also suggest using aspects as factories by having them provide only after-returning advice on constructors. This after-returning advice mutates the state of every object instantiated to change its default behavior. Limiting the aspects in this way restricts the scope of object–aspect interaction. In our proposal a simple assistant can fill the role of a factory aspect.

For aspects that do not act as factories Kersten and Murphy propose three style rules that restrict the use of aspects (pp. 349–350):

> Rule #1: Exceptions introduced by a weave must be handled in the code comprising the weave. . . . Rule #2: Advise [sic] weaves must maintain the pre- and post-conditions of a method. . . . Rule #3: Before advise [sic] weaves must not include a return statement.

These rules are essentially equivalent to our definition of spectators in that they prevent aspects from changing the

**Table 1: Example aspects and their categories (from the examples directory of the Version 1.0.6 release of AspectJ, available from `http://www.aspectj.org`)**

| *Example* | *Category* |
|---|---|
| telecom/TimerLog | spectator |
| tjp/GetInfo | spectator |
| tracing/lib/AbstractTrace | spectator |
| tracing/lib/TraceMyClasses | spectator |
| tracing/version1/TraceMyClasses | spectator |
| tracing/version2/Trace | spectator |
| tracing/version2/TraceMyClasses | spectator |
| tracing/version3/Trace | spectator |
| tracing/version3/TraceMyClasses | spectator |
| bean/BoundPoint | client utility |
| introduction/CloneablePoint | client utility |
| introduction/ComparablePoint | client utility |
| introduction/HashablePoint | client utility |
| observer/SubjectObserverProtocol | client utility |
| observer/SubjectObserverProtocolImpl | client utility |
| spacewar/Display.DisplayAspect | client utility |
| spacewar/Display1.SpaceObjectPainting | client utility |
| spacewar/Display2.SpaceObjectPainting | client utility |
| telecom/Billing | client utility |
| telecom/Timing | client utility |
| spacewar/EnsureShipIsAlive | impl. utility |
| spacewar/GameSynchronization | impl. utility[a] |
| spacewar/RegistrySynchronization | impl. utility[a] |
| spacewar/Registry.RegistrationProtection | impl. utility |
| coordination/Coordinator | assistant[b] |
| spacewar/Debug | combined |

[a] extends the abstract `coordination/Coordinator` assistant
[b] refers only to abstract pointcuts

behavior of the advised method. However, we propose elevating these style rules to the level of statically checked restrictions.

## 3.2 Impact of Restrictions

To better understand how our restrictions might limit the practical expressiveness of AspectJ, we review several examples from two separate sources.

### 3.2.1 AspectJ Programming Guide

We use the examples in the AspectJ Programming Guide to see if our restrictions prohibit any recommended idioms. The programming guide's examples can serve this purpose since they "not only show the features [of AspectJ] being used, but also try to illustrate recommended practice" [2] (from Preface). We separate the example aspects into categories based on how we would implement them with our restrictions. Table 1 lists the examples by category; we describe the categories here.

**Spectators** Many of the example aspects clearly meet our definition of spectator. To satisfy our restrictions these would only require the `spectator` annotation.

**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, i.e., aspects declared inside class declarations. These are similar in spirit to explicitly excepted assistance. We may wish to consider nested aspects to be implicitly accepted by their containing modules.

Other example aspects that could be implemented as assistants might be considered *implementation utilities*. These assistants encapsulate some unit of cross-cutting concern related to a single module, for example, 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.

The `Coordinator` aspect of the `coordination` package is abstract. This abstract aspect modifies the behavior of the modules to which it refers, making it an assistant in our terminology. However, `Coordinator` only refers to abstract pointcuts. Thus, for the advice in `Coordinator` to be applicable to any module a concrete aspect extending `Coordinator` would have to be declared. This concrete aspect would be an assistant and would need to be accepted per our design. In fact, the two "synchronization" implementation utilities listed in Table 1 are concrete assistants extending `Coordinator`.

**Combined** To satisfy our restrictions one example aspect, the `Debug` aspect of the `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`, that provided methods which allowed 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 menu.

### 3.2.2 Kiselev

While the AspectJ Programming Guide provides many small examples demonstrating recommended idioms, Kiselev's book *Aspect-Oriented Programming with AspectJ* [16] provides an extensive case study. It is the only freely available case study (that we are aware of) written in AspectJ that is not the implementation of another aspect-oriented programming language. The aspects given in the book in chapters 5-8 are all related to this case study, which concerns a web service that is supposed to store and retrieve users stories (an "e-zine"). Table 2 gives a summary of these aspects and how they relate to our categories.

Kiselev categorizes his examples as "development", "production", or "runtime" aspects [16, Chapter 9]. It is useful to discuss how these categories relate to our division of aspects into spectators and assistants.

The development aspects (`Logger`, `Tracer`, `Profiler` and `CodeSegregation`) are "used as development aids" (p. 115), but are not useful during production use of the system. Since these are optional aspects we would hope that they are spectators in our categorization. This is clearly the case for

**Table 2: Main example aspects from chapters 5-8 of Kiselev's book [16]; subheadings give Kiselev's categorization of the aspects**

| *Example* | *Category* |
|---|---|
| *Development Aspects* | |
| Logger | spectator |
| Tracer | spectator |
| Profiler | spectator[a] |
| CodeSegregation | not defined[b] |
| *Production Aspects* | |
| Authentication | client utility[c] |
| Exceptions | client utility |
| NullChecker | spectator[a] |
| *Runtime Aspects* | |
| OutputStreamBuffering | impl. utility |
| Pooling | impl. utility |
| ConnectionChecking | impl. utility |
| ReadCache | impl. utility |
| *not categorized* | |
| NewLogging | client utility |
| PaidStories | spectator |

[a]Minor changes are needed to make this aspect a spectator.
[b]The `CodeSegregation` aspect introduces warnings and errors, which are outside the scope of the current work.
[c]This aspect includes some features (parent declarations) that are outside the scope of the current work.

`Logger` and `Tracer`. The `Profiler` aspect would be considered a spectator in our categorization—except for one issue. `Profiler` declares before- and after-advice that can change the control flow by explicitly throwing an exception when an I/O error occurs while writing profiling data to disk. We could categorize `Profiler` as a spectator and use a root-level aspect map to apply it to the entire project. However, `Profiler` is a development aspect; we would like to be able to switch it off and on without editing the source code. We can resolve this difficulty by simply changing `Profiler` to swallow I/O errors and report the problem to the developer (via `System.err`, for example). This would make `Profiler` a spectator, as intended. Since it would be a spectator the developer could remove it from the running virtual machine, fix the file system problem that precipitated the exception, and add `Profiler` back into the running virtual machine, all without affecting the behavior of the core application. The `CodeSegregation` uses AspectJ's `declare error` and `declare warning` constructs to introduce additional compile-time checks. These constructs are outside the scope of the current work, but are discussed briefly in Section 4.

Kiselev's production aspects (`Authentication`, `Exceptions`, and `NullChecker`) are "absolutely essential to the application" (p. 115). Since they are absolutely essential it is reasonable to include them in the appropriate aspect maps knowing that these references will not have to be changed. These aspects do have some interesting properties *vis-à-vis* our categorization. The `Authentication` aspect is, at its core, a client utility. It is applied to objects which render web pages and manage the current user session to ensure that the user of the session is validated. We say that this is a client utility because it may be the case that some clients of these objects may wish to allow unauthenticated access to some pages. In Kiselev's example the `Authentication` aspect is applied broadly. This is easily implemented using a root-level aspect map. The `Authentication` aspect also uses introduction, via AspectJ's `declare parents` construct, to add additional methods to the classes it assists. Introductions are also beyond the scope of the current work, but are discussed briefly in Section 4.

The `Exceptions` aspect is a client utility in our categorization, a fact emphasized by Kiselev when he argues for using a `call` join point instead of an `execution` join point by saying that, with the `execution` join point, "some other application would not be able to utilize this class without the `Exceptions` aspect attached to it" (p. 76).

We would argue that Kiselev's third production aspect, `NullChecker`, should actually be considered a development aspect, since it is an aid to contract enforcement that might not be included in a production system. In Kiselev's code `NullChecker` throws an exception and so would need to be modified in a similar way as `Profiler` to meet be a spectator in our categorization.

Kiselev's runtime aspects ( `OutputStreamBuffering`, `Pooling`, `ConnectionChecking`, and `ReadCache`) are all "useful but not critical" [16, p. 115]. These are units of cross-cutting concern that might not be part of an initial implementation, but once added to the system they are likely to remain part of it. They are all implementation utilities that apply to a single class. Under our proposal, as these aspects were written and added to the application an accept clause would be added to the advised class. To use this approach the `call` join points in each of these aspects would be changed to `execution` join points.

The last two aspects in the table (`NewLogging` and `PaidStories`) are not categorized by Kiselev, but should be considered production aspects. The are used to change the behavior of the application without invasive editing. Under our proposal these aspects would be accepted using a root-level aspect map. `PaidStories` is a code example of code that qualifies as a spectator but would be named in an aspect map to gain the efficiency of compile-time weaving.

To summarize, except for the `CodeSegregation` aspect and a portion of the `Authentication` aspect, both of which use AspectJ features that are outside the scope of the current work, all the aspects in Kiselev's case study can be easily implemented under our proposal .

# 4. DISCUSSION

This section briefly discussion several interesting aspects of the current work.

Explicit acceptance of assistance interacts in interesting ways with `call` and `execution` pointcuts. Consider the examples from Section 1. If `FigureElement`'s module accepted the `MoveLimiting` assistant, but no client did, then the advice in the assistant would only apply to the calls to `setX` and setY within the body of the `move` method. This is because the assistant only uses `call` join points. Invocations of `FigureElement`'s `setX`, `setY`, or `move` methods from client code would not be advised because we are assuming that no client code accepts the assistance. But if the assistant instead substituted `execution` join points for the `call` join points, then having the `FigureElement` module accept the assistance would be sufficient to have the advice apply to invocations of `setX`, `setY`, or `move` from all client modules.

One might suppose that we could eliminate `execution` join points and just use `call` join points in their place, re-

lying on the explicit acceptance of assistance to determine where to weave the advice code. But where should the advice go if an assistant is accepted by both a client and an implementation module? The compiler cannot modularly know where all accept clauses in a program might appear, and so there is no modular answer to the question. Thus both `call` and `execution` join points are required in the language.

This distinction also impacts the distinction between client utilities and implementation utilities discussed in Section 3.2. An assistant using `call` join points is not a viable implementation utility. Conversely, an assistant using `execution` join points is not a viable client utility. To write an assistant that could fill either role one would have to write pointcuts that used a combination of `call` and `execution` pointcuts, along with the dynamic context pointcut `cflowbelow` to prevent duplicate application of the advice. Something like

```
( call(P) || execution(P) )
&& !cflowbelow(call(P))
```

might suffice. It may be reasonable to define a syntactic sugar for such pointcuts. However, the `cflowbelow` pointcut requires runtime checks whereas `call` and `execution` do not. Thus, restricting a given assistant to being exclusively a client utility (using `call`) or an implementation utility (using `execution`) is likely to be more efficient.

It seems that ordering advice based on the ordering of accept clauses might eliminate the need for AspectJ's dominates declarations, which specify aspect precedence. While technically this seems to be the case, we are not claiming that relying on the ordering of accept clauses is any less error prone than relying on dominates declarations to control the order of interacting aspects. On the one hand, it is impossible to know when writing an aspect all the potential other aspects that it should dominate. On the other hand, it would be quite easy to accidentally misorder the acceptance of two pieces of interacting advice in an aspect map or the accept clauses for a module. Compared to dominates declarations, the use of explicit acceptance also spreads and makes less obvious the kind of decisions that dominates declarations record.

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 considered an assistant in our categorization. But suppose an aspect introduced a new method to a class such that the method did not override an inherited method. Since no other code could have called that new method, this introduction should not change the behavior of existing code. So perhaps such an introduction should be allowed in a spectator. This case is similar to the introduction of external generic functions via MultiJava's open class mechanism [8, 6]. Introduction involves subtle modularity issues, particularly for avoiding runtime ambiguities. These issues are made more complex by the possibility that the introduced methods might be advised by pre-existing aspects.

The current work also does not address AspectJ's `declare error` and `declare warning` constructs. But these constructs to 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 at compile-time, then an error or warning should be issued. It seems that these constructs should be allowed in spectator aspects. It is likely that aspects containing these constructs would be broadly applied using aspect maps.

An aspect that used the `declare soft` construct would clearly change the control flow of a program to which it was applied. Such an aspect would be an assistant in our categorization.

# 5. RELATED WORK

Katz and Gil [12] 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. (Bougé and Francez give an approachable introduction to superimposition [5].) Katz and Gil briefly sketch a categorization of aspects into three categories. Their "spectative" aspects match 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 aspect maps and they do not talk about how one might enforce that declared spectators have no observable side effects. Their suggestion regarding mining the work on superimposition in developing aspect-oriented programming seems to be a reasonable one. Much of the work on superimposition is concerned with proving properties of distributed algorithms, or adding additional provable properties to distributed algorithms without disturbing other underlying properties. Our work can be viewed as a beginning at extending these more theoretical ideas into practical language designs.

We know of no other published work that attempts to restore modular reasoning to AspectJ.

# 6. CONCLUSIONS

To summarize, we have shown that with a few additional language features are sufficient to support modular reasoning in a language like AspectJ. Our proposal separates aspects into two categories, assistants and spectators, which provide complementary features. Assistants have the full power of AspectJ's aspects, but to maintain modular reasoning 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 without requiring spectators to be explicitly excepted.

Our proposal introduces aspect maps to allow acceptance of assistance without scattering many duplicate accept clauses throughout a program.

Explicit acceptance of assistance allows separate compilation to weave advice declared in assistants into the modules accepting that assistance. We have also suggested that an aspect-oriented virtual machine would allow efficient dispatch to spectators. Such a virtual machine would also permit spectators to be added to and removed from running programs.

Our evaluation showed that for all cases studied (except perhaps for those using language features related to introduction, which are outside the scope of this work), our modifications to the language provide sufficient flexibility.

The major challenge for our proposal is checking that aspects declared as spectators meet our definition, as discussed

in Section 2.2. We have specified constraints on spectators that allow modular reasoning about their (lack of) impact on control flow. A type system that restricts aliasing and mutation should allow modular reasoning about spectators (lack of) impact on the relevant state of the modules they view. The next steps in this area are:

- developing an aspect-oriented calculus for investigating these ideas in a formal setting (perhaps building on [29, 25, 17]), and
- developing and proving sound a type-system for the calculus that statically enforces our proposed restrictions on spectators.

We are also interested in demonstrating the utility and effectiveness of our ideas by:

- further developing and implementing our proposal using the type system mentioned above,
- programming non-trivial systems with it, and
- investigating the use of an aspect-oriented virtual machine to support adding and removing spectators from running programs.

In this paper we have focused on adding support for modular reasoning to the AspectJ language. Future work will also investigate the relevance of our proposal to other aspect-orientation programming languages and techniques, such as composition filters [3], adaptive methods [19], and multidimensional separation of concerns as embodied by Hyper-J [26, 28].

## 7. ACKNOWLEDGMENTS

We would like to thank Yoonsik Cheon, Todd Millstein, Markus Lumpe, and Robyn Lutz, for their helpful comments on an early version of this work [7]. We would also like to 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. The work of both authors was supported in part by a grant from Electronics and Telecommunications Research Institute (ETRI) of South Korea, and in part by the US National Science Foundation under grants CCR-0097907 and CCR-0113181.

## 8. REFERENCES

[1] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language Third Edition.* Addison-Wesley, Reading, MA, third edition, 2000.

[2] AspectJ Team, the. The AspectJ programming guide. Available from http://aspectj.org/doc/dist/progguide/index.html, Feb. 2002.

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

[4] A. Borgida, J. Mylopoulos, and R. Reiter. '... and nothing else changes': The frame problem in procedure specification. In *Proceedings Fifteenth International Conference on Software Engineering, Baltimore*, May 1993. Preliminary version obtained from the authors.

[5] L. Bougé and N. Francez. A compositional approach to superimposition. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 240–249. ACM Press, 1988.

[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, Minneapolis, Minnesota*, volume 35(10) of *ACM SIGPLAN Notices*, pages 130–145, Oct. 2000.

[9] L. Friendly. The design of distributed hyperlinked programming documentation. In S. Fraïssè, F. Garzotto, T. Isakowitz, J. Nanard, and M. Nanard, editors, *Proceedings of the International Workshop on Hypermedia Design (IWHD'95), Montpellier, France, 1–2 June 1995*, pages 151–173. Springer, 1995.

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

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition.* The Java Series. Addison-Wesley, Boston, Mass., 2000.

[12] S. Katz and Y. Gil. Aspects and superimpositions. In *Int'l Workshop on Aspect-Oriented Programming (ECOOP 1999)*, June 1999.

[13] M. A. Kersten and G. C. Murphy. Atlas: A case-study in building a web-based learning environment using aspect-oriented programming. In *Proceedings of the 1999 ACM Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA '99)*, volume 34(10) of *ACM SIGPLAN Notices*, pages 340–352, Denver, CO, November 1999. ACM.

[14] 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.

[15] 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.

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

[17] R. Lämmel. A semantical approach to method-call interception. In G. Kiczales, editor, *Proc. 1st Int' Conf. on Aspect-Oriented Software Development*

*(AOSD-2002)*, pages 41–55. ACM Press, Apr. 2002.

[18] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06q, Iowa State University, Department of Computer Science, June 2002. See `www.jmlspecs.org`.

[19] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. *Commun. ACM*, 44(10):39–41, Oct. 2001.

[20] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publishing Co., Reading, MA, second edition, 2000.

[21] B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, NY, 1992.

[22] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. The author's PhD Thesis. Available from http://www.informatik.fernuni-hagen.de/import/pi5/publications.html.

[23] 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.

[24] 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.

[25] D. Orleans. Incremental programming with extensible decisions. In G. Kiczales, editor, *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, pages 56–64. ACM Press, Apr. 2002.

[26] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Commun. ACM*, 44(10):43–50, Oct. 2001.

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

[28] 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.

[29] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. 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 1–8. Department of Computer Science, Iowa State University, Apr. 2002.