# Non-Modularity in Aspect-Oriented Languages: Integration as a Crosscutting Concern for *AspectJ*

**(revised and extended, June 6, 2002)**

Kevin Sullivan
University of Virginia
Department of Computer Science
151 Engineer's Way
P.O. Box 400740
Charlottesville, VA 29903, USA
+1 804 982 2206
sullivan@virginia.edu

Lin Gu
University of Virginia
Department of Computer Science
151 Engineer's Way
P.O. Box 400740
Charlottesville, VA 29903, USA
+1 804 982 2200
lg6e@virginia.edu

Yuanfang Cai
University of Virginia
Department of Computer Science
151 Engineer's Way
P.O. Box 400740
Charlottesville, VA 29903, USA
+1 804 982 2200
yc7a@virginia.edu

## ABSTRACT
Aspect-oriented (AO) methods and languages seek to enable the preservation of design modularity through mappings to program structures, especially where common (object-oriented) languages fail to do so. The general claim is made that AO approaches enable the modularization of crosscutting concerns. The problem that we address is that it is unclear to what extent such claims are valid. We argue that there are meaningful bounds on the abilities of past, present, and future languages to succeed in this regard—bounds that we need to understand better. To make this idea concrete we exhibit a significant bound: Component integration (Sullivan & Notkin 1992, 1994) is not adequately modularizable in *AspectJ*.

## Keywords
Aspect, non-modularity, integration

## 1. INTRODUCTION
Aspect-oriented languages aim explicitly to enable the preservation of modularity in design where existing programming languages and methods fail. The problem we address in this paper is that we don't yet understand the bounds of the validity of this claim. For what important modular design structures, if any, do prominent aspect-oriented languages provide no modular representations? We show one interesting bound by example. We also present a critical analysis of some basic terms of AO programming. In a nutshell, we find that *aspects* are relative: Whether a module is an aspect in one language depends on whether the concern it represents has no modular representation in another. Our analysis puts AO languages in a broader context of advances, dating to 1972 [10], concerned with preserving modular designs structures in corresponding program representations. Finally, we suggest that the intentional search for *non*-modularity-preserving properties of prevailing languages and program design methods is a good way to make progress. This paper illustrates the application of this approach to modularity-preserving AO languages.

The rest of this paper is organized as follows. Section 2 addresses modularity in design; its preservation—or not—through mappings of designs to programs; how mapping problems have driven innovation; and what aspects really are. Section 3 views behavioral relationships—protocols that integrate objects into systems—as aspectual for standard OO languages and methods. Section 4 presents the language of abstract behavioral types as one that *is* modular for behavioral relationships. Section 5 shows that *AspectJ* is *not* modular for such integration concerns. Section 6 concludes.

## 2. WHAT ARE ASPECTS?
The task of a *software architect* is, for given requirements, to devise a design structure having the required runtime properties and a modular structure that maximizes the value of the design in the assumed environment [14]. Modularity can add value to a system in the form of reduced cost of comprehension, real options to vary and change the system, improved time to market through parallelism in development, component reusability, and so forth.

Devising a design structure involves the selection of design parameters, the structuring of dependences among their values, and the choice of a value for each. A good modular design is one that abstracts key design decisions as explicit design parameters and in which dependences among design parameter values are such that the values of certain key parameters can be chosen—i.e., design decisions made or changed—largely independently of others.

The task of a *program designer*, by contrast, is to represent a given design structure in a corresponding program structure. A program structure, expressed in a programming language, is subject to the constraints and possibilities inherent in the language, and to additional constraints, e.g., as imposed by style rules.

For the benefits of modularity in design to be realized, the program that represents a design must preserve its modular structure. We will say that a program *preserves modularity* if independent parameters in the design are represented by independent constructs in the program. When modularity is not preserved, independent design parameters are coupled though coupled representations.

There are many ways in which a program can fail to preserve modularity. One special case occurs when each of several independent design parameters is represented by a corresponding program construct, but where the representation of some other design parameter is scattered across and is merged into the previous program constructs.

The representation of the latter parameter is said to *crosscut* the other representations.[1] The design parameters are coupled indirectly in the code, and the corresponding design decisions can therefore no longer be made or changed entirely independently.

In structuring software, a good architect seeks to abstract and decouple the key dimensions (design parameters) in which it is worthwhile to be able to vary or change a system independently. This is Parnas's information hiding criterion [10]. A key goal of a program designer then is to choose a program structure that preserves the modular structure of the abstract design.

Yet there are bounds on the abilities of programming languages, subject to the additional constraints of design methods, styles, and conventions,[2] to preserve such modular structures. If a design exceeds the bounds of a language, we will say that the design is *not modularizable in the language*, and that the language is *not modular for the design*. In this case, any program representation of the design in the language will exhibit undesirable coupling, such as crosscutting implementation artifacts, that complicate software design and evolution, increasing costs and complexity, reducing dependability, and so forth.

A problem arises when, for one reason or another, a programmer decides to use a language that is not modular for otherwise valuable design structures. An even worse problem arises when architects think in the terms of such languages. Then they might not even conceive of adequately modular designs. These phenomena rise to the level of major problems in practice and theory when prevailing language paradigms are non-modular for important classes of well modularized designs. Such paradigms strongly influence software engineers to select inadequate design and program structures.

What is needed in such a case is a new language in a new paradigm. One desirable property of such a language is that it be modular for valuable designs that were modular in the old language: Nothing should be lost. The new language should also be modular for valuable but previously non-modularizable designs.

One key driver of the evolution of language paradigms is the discovery of valuable new ways of structuring abstract designs. Prevailing languages are sometimes found to be non-modular for such designs. New languages that are modular are then developed.

For example, Parnas saw a valuable new class of designs in which data-structure-valued design parameters are kept independent. He argued that the prevailing top-down, structured programming approach was not modular for such designs because it called for procedures to communicate through shared data structures, coupling the designs of procedures through the data structures. He then showed how the introduction and proper use of abstract data type interfaces could preserve the modularity of designs in which concrete data structure design decisions were decoupled from other design decisions. His work thus helped to establish the next paradigm: object-orientation.

Subsequent experience has taught that object-orientation is not ideal. There are valuable design structures for which standard object-oriented languages and design method are non-modular. The problem is not in the concept of information hiding, but in the commitment to the abstract type interface as the key mechanism for supporting the information hiding design strategy. The discovery of non-modularity properties of prevailing language paradigms for important design structures continues to drive such innovation.

New languages are emerging to accommodate innovative modular design structures for which traditional object-oriented languages are non-modular. Such new languages are called *aspect-oriented* [2]. These languages, which include *AspectJ* [1] and *HyperJ* [9], are described using relatively new terms: notably *aspect* and *crosscutting implementation*.

The preceding discussion positions us to analyze these terms to better understand what they mean. What we will find, in a nutshell, is that, under one common set of definitions of the terms, *aspects are relative.* We now explain what we mean by this statement.

Kiczales et al. [6] define an aspect to be "a modular unit of crosscutting implementation." We try to make this idea precise in the following terms, distinguishing between two separate ideas: when a program structure can be said to be an aspect, and when a design parameter can be said to be an aspect.

Kiczales defines an aspect as a program structure—a certain kind of module. However, in our experience, we have found the term *aspect* also widely used to refer to design parameters that are hard to express in modular form in traditional languages. We need to distinguish these usages of the term and make their definitions somewhat more precise. We start with the design-level meaning of the term.

Suppose *D* is design structure; *I*, a design parameter in *D*; and *New* and *Old*, two languages. At the design level, *I* can be said to be an *aspect relative to New and Old* if and only if *Old* is not modular for *I*, and *New* is. For example, a *tracing policy*—a common example of a design parameter with no satisfactory modular representation in typical object-oriented languages—could be called an aspect relative to an OO language and an aspect-oriented language. It is a design-level aspect because it cannot be modularized well in the old language but it can be modularized in the new language.

An alternative design-level definition—and perhaps the most common—is that *I* is an *aspect relative to a language L* if *L* is non-modular for *I*. Thus tracing could be said to be an aspect for a typical object-oriented language.

Finally, we could define a design parameter *I* to be an *essential* aspect if there is no language in which I could possibly have a modular expression. We will return to the issue of accidental and essential aspects at the end of this paper.

Turning to the program representation level, a representation *P* of *I* in the language *New* is said to be an *aspect relative to languages New and Old* if and only if *I* is an aspect relative to *New* and *Old*, and *P* is actually represented as a module in the *New* language. Thus the representation of a tracing design parameter as a modular expression in a language such as *AspectJ*, or perhaps in a language

[1] In the AOSD community there is no apparent consensus on the definitions of the terms *crosscut* and *tangle*. In this paper, we use these terms more or less interchangeably. An alternative definition uses *crosscut* to refer to inherent intermingling of *behavior* related to different concerns, and *tangle* to refer to non-modular source code representations of such behavior. By this alternate definition, an *aspect* is a modular representation of a crosscutting concern.

[2] Henceforth, for brevity, we will simply say *languages*. By this term we mean both the programming language itself and additional style and usage rules or conventions that constrain the expression of concepts using the programming language.

with object-oriented reflection [7], can be said to be an aspect precisely because its representation is modular in the new language whereas there was no available modular representation in the old (e.g., object-oriented) language. The modular expression of the design parameter is thus a modular unit of (otherwise) crosscutting implementation. The term *crosscutting* here tacitly assumes the old OO language, in which the implementation of the design parameter is necessarily crosscutting. The tracing design parameter that is an aspect for the old language is not for the new language. It is thus, under one common set of definitions, that *aspects are relative*.

## 3. INTEGRATION IS AN ASPECT, FOR OO

In earlier work [11][13] Sullivan and Notkin identified a class of design structures for integrated systems in which the objects to be integrated and the *behavioral relationships* that integrate them are conceived and structured as separate, explicit design parameters. They showed that these design structures have the potential to significantly ease the design and evolution of integrated systems, but that standard OO languages and methods are not modular for these structures. Such languages unavoidable tangle object and integration concerns, causing major problems for the design, development, and evolution of integrated systems. In other words, integration is aspect for standard OO languages.

To address this problem, Sullivan and Notkin developed the abstract behavioral type (ABT) as a language mechanism, and showed that ABT-based languages are modular for such designs. They coined the term *mediator* to mean a modular, ABT-based representation of an otherwise crosscutting behavioral relationship. In an experimental systems style they tested the hypothesis that structuring designs this way and preserving their structures in programs would ease the design, development, and evolution of integrated systems [12].

In this section, we review in more detail why integration is an aspect for OO, using an archetypal scenario for the design, program representation, and evolution of integrated systems. In the next section, we show that ABT's are modular for integration concerns. In brief, behavioral relationships have modular representations in ABT terms. Thereafter we exploit the example that we develop here to test whether integration remains an aspect for OO languages augmented with the novel aspect-oriented constructs of *AspectJ*.

Suppose that you are asked to construct a system that integrates the behaviors of several binary digit, or *Bit*, objects. The initial system configuration has two *Bits*, *b1* and *b2,* each of which can be viewed and manipulated directly by clients of the system (such as other objects in the system). The state space of a *Bit* is a single Boolean digit. The applicable operations include *Set* and *Clear,* which set the state of a *Bit* respectively to *1* and *0*. The *Get* operation returns the current state of *Bit*. A standard OO approach would be to define a *Bit* class and to create the running system in part by instantiating two objects of this class, *b1* and *b2*.

Suppose now that the system integration requirements call for *b1* and *b2* to work together as follows: if any client *Set's* (or *Clears*) either *Bit*, the other must be *Set* (*Cleared*). The behaviors of the *Bits* have to be integrated by a behavioral relationship, which we will call *Equality*, that maintains a state-equality constraint. The diagram in *Figure 1* illustrates a design structure in which the behavioral relationship is conceived as a separate design parameter. Sullivan and Notkin called this kind of design structure a *behavioral entity-relationship model*. The key is that the constraint pertains to (crosscuts) several objects in the system.

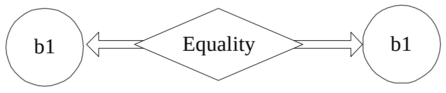

Figure 1:Design with two objects and a behavioral relationship.

Now consider a step in the evolution of the design—an augmentation step. Two more *Bit* objects, *b3* and *b4,* are added to the system. *Bit b3* is required to work with *b2* in a second instance of the *Equality* behavioral relationship. Thus, if any of the first three bits are *Set* or *Cleared*, the others will be, too.

*Bit b4* is required to work with *b3* in a different relationship, which we call *Trigger*: If any client *Sets b3,* then *b4* must be *Set,* too. In the resulting system, *b4* can be *Set* and *Cleared* with no effect on the other *Bits*. If *b3* is *Cleared,* there is no effect on *b4* (but *b2* will have to be cleared and then *b1*). However, if *b3* is *Set*, the *Trigger* relationship requires that *b4* be *Set*, too.

One reason that structuring designs this way is valuable is that such structures are easy to extend locally with system-wide behavioral effects—the property we want in an integrated system. The augmented design structure is illustrated in Figure 2:

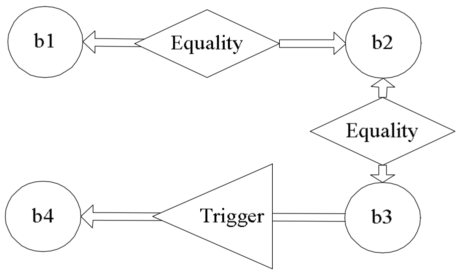

Figure 2: The augmented design structure.

When we try to map these design structures into object-oriented programs in standard ways we find that the behavioral relationships end up expressed in crosscutting implementations. Standard OO languages and methods are not modular for designs in which behavioral relationships are independent design parameters. We now make this point concrete.

A straightforward OO approach would start by mapping *Bits* to instances of a *Bit* class, and the relationships to code that is tangled in the *Bit* class methods. Within the implementation of *Set*, for example, code might appear to find and update related *Bit* objects.

Consider, for example, the design in which *b1* and *b2* are integrated by *Equality*. The *Bit* class would have instance variables to represent the value of the bit and the identity of any related *Bit* objects. The code implementing *Set* and *Clear* would also implement the propagation of effects to related *Bits*. The representation of the relationship is tangled with and scattered across the representations of the objects that are related. Standard OO approaches do not preserve the modularity of even simple integrated system designs in this style.

The consequences of the loss of modularity are significant. First, the lack of an abstract representation of the *Equality* relationship makes the system unnecessarily hard to understand. Second, we cannot easily remove the behavioral relationship from the system because its representation is hardwired into the *Bit* code. Third, we cannot easily reuse the *Bit* class because each object assumes a corresponding object and an *Equality* relationship with it. Fourth, we cannot develop the programming code for the *Bit* and *Equality* abstractions independently. Fifth, we cannot test and debug their implementations independently. Sixth, we cannot reuse the relationship code independently; and so on. In other words, all of the well known costs of a serious loss of modularity and abstraction are incurred by the standard OO representation in this case.

More sophisticated OO methods do not adequately resolve these problems. One idea is be to use Gamma's *mediator design pattern* [3] to represent the *Equality* relationship as a separate object.[3] In this design, *Equality* is abstracted as a class *Equality* containing the code for updating one bit when the other changes. The problem is that *Bit* objects have to call this mediator, directly or indirectly; so the *Bit* class still ends up coupled either directly to the definition of the Gamma-style mediator.

This design is better in that the relationship now has a first-class abstract representation, making the code easier to understand; but several representations are still coupled: relationship-related calls are hardwired into the *Bit* code. Reusing *Bit* is precluded because its objects assume the presence of external mediators. Nor can we develop the programming code for the *Bit* independently of that for the *Equality* abstraction because the *Bit* code uses the *Equality* code substantively. We cannot test and debug the *Bit* implementation independently. We cannot reuse the relationship code because it assumes the presence of two *Bit* objects; and so on. This solution is somewhat better, but not genuinely modular.

Nor does using implicit invocation by itself (event notification, or the Gamma-style *Observer* pattern [3]) solve these problems. It is not much better for each *Bit* to *register* with the other to receive updates than for each *Bit* to *call* the other. We see this problem in the Model-View-Controller pattern (see [3]). Here, views register with models to be notified of relevant changes and implement the "update" code. The code to manage model-view relationships thus crosscuts the code for the views.

All of these problems are vastly complicated when designs start to evolve. Consider what happens when we augment the design with *b3* and the *Equality* relationship linking it to *b2.* To represent this design change in the program, we might complicate the *Bit* class to enable *Bits* to participate in multiple relationships (e.g., *b2* with *b1* and *b3*). Alternatively, we might produce two different *Bit* classes, the first designed for objects that participate in one *Equality* relationship, such as *b1* and *b3,* and the second for objects that participate in two (*b2*). Code implementing the pair of *Equality* relationships is now merged into the code for the second *Bit* class.

When we add *b4* and a *Trigger* relationship linking *b4* to *b3*, things go from bad to worse. Although we can make *b4* an instance of the *Bit* class, we need a new class for *b3*: one that implements *Bits* that participate in both the *Equality* and *Trigger* relationships. The representations of multiple design parameters are truly tangled in the representation of *b3*. It jumbles code for the *Bit* abstraction, and for the *Equality* and *Trigger* relationships.

[3] Gamma's *Mediator* pattern is related to but not the same as ours. In addition to representing relationships as separate objects, as in Gamma's pattern, our mediators employ implicit invocation (also known as the *Observer* pattern) to decouple the representations of the objects related from the mediator representing the relationship.

## 4. ABTS: MODULAR FOR INTEGRATION

The problems in our example scale into to major difficulties in the design and evolution of real systems, such as integrated software engineering and other kinds of environments. As the number of relationships in a design increases, the structure of any standard OO program representation degrades rapidly. The upshot is that standard OO methods are not scalable for integrated system design.

Sullivan and Notkin [11][13] introduced mediator-based design as a partial solution. The challenge was to extend standard OO methods to make them modular for designs in the form of behavioral entity relationship models. We give an example of what this would mean.

In our design example, the *Bit* parameters would be represented in a program by objects of a self-contained class, *Bit,* with simple *Get, Set,* and *Clear* methods. The *Equality* and *Trigger* relationships would be represented as objects of corresponding *Equality* and *Trigger* classes. These objects (*mediators*) would be connected to *Bit* objects at runtime to make them work together. The initial design would be mapped to a program that creates *Bits b1* and *b2* and *Equality* object *e(b1,b2).* Any call to *b1.Set* would activate *e,* which would call *b2.Set* in turn—with *e* preventing recursion.

One simple approach for preventing unbounded recursion is to have the mediator maintain state that encodes whether it is already in the midst of updating one *Bit* as a result of an action of the other. In this case, any call to *b1.Set* would activate *e,* which would check its *busy bit*, return if it indicates an update is already in progress; and otherwise set it, call *b2.Set*, and then clear it before returning.

The approach preserves the modularity of the design. When the design is extended locally, for example, the required program changes are also localized: The program is extended to create two more *Bits, b3* and *b4*, an *Equality* mediator *e2(b2,b3*), and a *Trigger* mediator *t(b3,b4).* Now *b3.Set* actives both *t* and *e2; t* calls *b4.Set; e2* calls *b2.Set;* that call activates *e1,* and *e1* calls *b1.Set.*

Sullivan's and Notkin's aim was to ease the design and evolution of integrated systems—a very broad class of systems, given that objects always have to work together to achieve system objectives. The solution had two parts: structure designs as behavioral entity relationship models; and preserve their modular structures in programs. Learning to design using behavioral ER models took effort. Figuring out how to preserve their modularity in practical programs required novel programming constructs and methods.

The programming solution, in turn, involved a combination of two ideas. The first idea was to represent entities and relationships not with abstract data types, as in OO languages, but in terms of abstract behavioral types (ABT's). The second idea was to map entities and relationships to corresponding ABT-based objects in a way that would avoid crosscutting implementations of behavioral relationships. We now explain these two ideas in greater detail.

An ABT defines a class of objects not just in terms of operations that can be applied to an object of the class but also in terms of *events* that such an object can *announce.* Announcing an event invokes (meta-level) operations implemented by other objects that have *registered* to receive such events from a given object.

Figure 3 illustrates a *Bit* ABT with *Get, Set,* and *Clear* operations and *JustSet* and *JustCleared* events. Mechanisms are provided for objects to register for such events. The implementations of the operations are responsible for announcing the events.

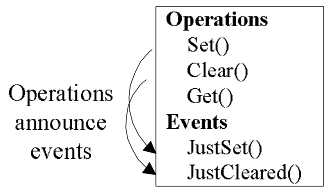

Figure 3: A *Bit* ABT

The key observation is that the "language" of ABT's is modular for designs in the form of behavioral entity-relationship models. There are modularity preserving mapping from such designs to modular program structures. Each behavioral relationship is represented as mediator ABT without crosscutting the representations of the objects to be integrated or those of other relationships.

The only problem is to ensure that relevant mediator operations are invoked when integration actions need to be taken. Events serve this function. A mediator implements relationship-maintaining operations and registers these operations so that they are invoked when objects signal possible needs for integration actions.

No mediator-specific code need be embedded in the objects to be integrated. The representations of the objects are decoupled (for compile-time, link-time, and, in general, run-time dependencies) from the representations of the mediators that integrate them. Each relationship in the design structure also has its own modular representation in the program. The bottom line is that entity and relationship representations no longer have to crosscut each other.

We now illustrate these ideas in terms of our running example. Consider how mediators provide a modularity-preserving representation of our integrated system of *Bits.* To represent the *Equality* relationship, we define a mediator ABT *Equality.* The ABT defines operations *Bit1JustSet, Bit2JustSet, Bit1JustCleared,* and *Bit2JustCleared* that implement responses to the first and second *Bit* respectively being *Set* or *Cleared.* The mediator is given references to the *Bit* objects, *Bit1* and *Bit2*, it is to integrate. It uses these references to manipulate each *Bit* in response to the event announcements of the other. *Bit1JustSet*, for example, calls *Bit2.Set.* A mediator constructor or initialization operation takes the references to two *Bit* objects and registers the mediator's response operations with the events of these *Bit* objects.

Figure 4 illustrates the mapping for the two-*Bit* case. The design structure is depicted above; the mapping to the program structure, in the middle; and the program structure, below. The program preserves the modularity of the design. Among other things, the mediator is defined to reference *Bit* objects (to register with their events and to call their operations), but the definition of *Bit* remains independent. Thus, the modular extensibility properties of the design are maintained: design extensions map to local program extensions without loss of modularity.

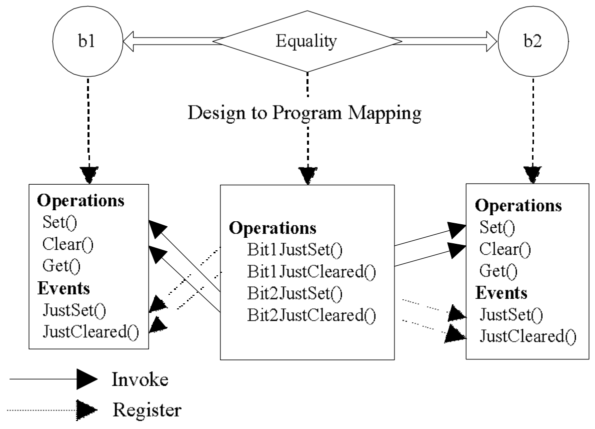

Figure 4: A modular representation of an integrated system design.

ABT's treat events in object interfaces at the same level as operations. It is possible to emulate multiple-event interfaces in a range of languages [8]. In Java, for example, each logical event to be exposed can be represented by three operations implemented by an underlying event-valued instance variable of the object. The three operations allow clients to *Register* and *Unregister* for event notifications, and for the object itself to *Announce* the event. Each underlying event object implements the *Observer* pattern. This approach makes the kinds of events that an object can announce explicit in its advertised interface, rather than implicit in the parameter values sent on a single *Subject-to-Observer* channel.

## 5. INTEGRATION: ASPECT FOR ASPECTJ

We can now reformulate and give a somewhat surprising answer to the question at the beginning of the paper. Aspect-oriented programming languages appear provide new possibilities for the modular representation of important kinds of design structures. Yet, we don't understand for what valuable design structures they languages remain non-modular. What are the bounds?

We now show one somewhat surprising bound. *AspectJ* is not fully modular for behavioral relationships—the interaction protocols that integrate objects into desired systems. We believe, but do not argue in this paper, that other prominent aspect-oriented languages, including *HyperJ*, share this non-modularity property.

We refine the question further. Are there design structures (1) that are known to be valuable in practice, (2) that are not modularizable in standard OO languages, (3) that conceivably or demonstrably are modularizable in *some* practical programming language, (4) for which one can reasonably expect OO languages extended with prominent AO mechanisms to be modular, (5) but that can be shown to be non-modularizable in these languages? If we can exhibit such cases, then we have discovered interesting bounds on the claimed modularity properties of the languages. Discovering these bounds can help show the way to future progress.

We have shown that the integrated systems design structures discussed now satisfy properties (1)–(3): Designs with independent behavioral relationships have value; they are not modularizable in standard OO languages; and they are modularizable, using mediators, in a practical extension to OO, the language of ABT's.

In this section, we identify *AspectJ* as a language satisfying conditions (4) and (5). We might reasonably expect that the powerful AO extensions provided by *AspectJ* would make it modular for integration concerns; but they don't. The problem, it turns out, is in *AspectJ's* limited model of aspect instances.

Starting with point (4), we might reasonably expect *AspectJ* to be modular for integration concerns because a mediator, as a modular representation of an integration concern, can clearly be seen as a special and precursor case of the more general *aspect* construct of the *AspectJ* language. It would be surprising for a language that is intended to address the general case not to be able to address an important special case. We now argue that mediators are special case of aspect modules as defined by AspectJ.

First, mediators meet the definition of *aspect*: they are modular units of crosscutting implementation (relative to OO languages). This point is reinforced by the clear mapping of mediator-related constructs to fundamental elements of an aspect-oriented language. An *event* is a *join point*—a point in program execution to which meta-level actions can be attached. Meta-level mediator methods that are registered to be invoked when events are announced are *advice* constructs. A mediator is a special case *aspect* module.

Second, at first glance, the ABT language is strictly less expressive than *AspectJ*. *AspectJ* implicitly defines a very broad range of join points, relieving the program designer of having to declare them explicitly (as with events). *AspectJ* also provides the powerfully expressive mechanism of the *pointcut* for concisely identifying sets of join points of interest, and for attaching advice to all join points in a given pointcut. Finally, *AspectJ* has other mechanisms with no analogs in the language of ABTs, including mechanisms to add state to class definitions and for short-circuit parameter passing.

Thus, we might reasonably expect that *AspectJ* is strictly better for representing behavioral relationships as aspects. We hypothesized that we could use *AspectJ* join points and pointcuts to avoid having to explicitly define and announce events and to register for all events in given pointcut equivalents. We tested this hypothesis by exploring a number of approaches to representing mediators as aspects in *AspectJ*.

We now turn to point (5). Our exploration led us to reject the hypothesis. *AspectJ* aspects cannot fully emulate ABT mediators. We trace the problem to limitations of the *AspectJ* model of *aspect instances*. This result suggests that future *AspectJ* versions might profitably incorporate a more flexible model of aspect instances.

Our idea was to represent behavioral relationships as aspects using join points, rather than as mediators using event notification. To make this idea concrete and to show that it works in a simple case, we present an *AspectJ* programs for the case of two *Bit's* integrated in an *Equality* relationship. This program starts with the *Bit* class presented in Figure 5.

```
Program 1
public class Bit {
  boolean value;
  public Bit(){value=false;}
  public void Set(){value=true;}
  public void Clear(){value=false;}
  public boolean Get(){return value;}
}
```

Figure 5: A simple Bit class in Java.

The aspect in Figure 6 implements an *Equality* relationship for two *Bits*. It stores references to two *Bit* objects. The *map* function expects a reference to one of the *Bits* as an argument, and it returns the other. When the aspect advice is activated by one of the *Bits* being *Set* or *Cleared* the aspect calls *map* to get a reference to the other *Bit* and then updates it. The pointcut specifications enable attachment of advice to the appropriate points in program execution: where calls to *Bit.Set* and *Bit.Clear* occur.

The first advice runs after *Bit.Set* is called. To prevent unbounded recursion, as in the mediator solution, the code checks a guard bit and does nothing if it is set. Otherwise the code obtains the *Bit* to be updated, sets the guard bit, updates the other *Bit*, clears the guard bit, and returns. The second advice does the same for *Clear*.

The attractiveness of *AspectJ* for the design and evolution of integrated system implementations using a mediator-like approach is clear. We have succeeded in integrating two *Bit's* in a behavioral relationship without having to compromise modularity. The *Bit* code remains unadulterated and we have a modular representation for the otherwise crosscutting *Equality* relationship. This design also creates the options to add, remove or change the behavior of the relationship independently.

Unfortunately, although this simple case works, it does not scale. This program is limited to integrating two *Bits* with a single global instance of the *Equality* aspect. To accommodate the addition of a third and fourth *Bit* and the *Trigger* relationship requires merging their code and data into this single aspect module. Independent concerns in the design thus crosscut each other in the program.

Figure 7 presents a flawed attempt at a better solution. The fault in the program points to the problem with the idea of implementing mediators as aspects. We discuss the flaw momentarily.

This program represents in a single aspect zero or more pairs of *Bits* related by *Equality*. The aspect uses *introduction declarations* to cause state and behavior needed to represent *Equality* on a per *Bit*-pair basis to be appended to the *Bit* class. Each *Bit* object is thus imbued with the means to participate in one *Equality* relationship.

A program can now create two *Bits* and place them in an *Equality* relationship by calling *Bit.relate* (introduced by the aspect into the *Bit* class) with the identity of the other *Bit* as a parameter. The introduced *inRelation* variable is set to true in each *Bit*, and the value of each *Bit's peer* is set to refer to the other. Aspect advice that runs after *Bit.Set* and *Bit.Clear* checks whether the invoking *Bit* is in a relationship (*inRelation* is true), and, if so, updates the other *Bit* as appropriate. The aspect uses the same *busy* construct to terminate the otherwise unbounded recursion.

The fault in this program has to do with this *busy* variable. Before we address that problem, however, we note that a more obvious problem with this design is that each *Bit* is limited to being in at most one *Equality* relationship. An improvement is to inject into the *Bit class* an instance variable storing a *List of Bits:* to hold references to the zero or more *Bits* to which a given *Bit* is related. The advice would traverse this list, updating peers, taking appropriate measures to avoid unbounded recursion.

```
Program 2
aspect Equality {
  static boolean busy;
  Bit b1;
  Bit b2;

Equality(Bit bit1, Bit bit2){
    b1 = bit1;
    b2 = bit2;
  }

  Bit map(Bit b){
    if (b == b1) return b2
    else return b1;
  }

  pointcut callSet():
    call(void Bit.Set());

  pointcut callClear():
    call(void Bit.Clear());

  after(): callSet(){
    if (!busy) {
      Bit peer = map(
        (Bit)thisJoinPoint.getTarget());
      busy = true;
      peer.Set();
      busy = false;
    }
  }

  after(): callClear(){
    if (!busy) {
      Bit peer = map(
        (Bit)thisJoinPoint.getTarget());
      busy = true;
      peer.Clear();
      busy = false;
    }
  }
}
```

Figure 6: A mediator-like aspect for *Equality*.

Finally, we note that yet another solution is available. The lists of *Bits* could be removed from individual *Bit* objects to be unified in a single relational table maintained by the aspect. This table would contain an entry for each pair of *Bits* related by *Equality*. The aspect would provide a function for putting pairs of *Bits* in or remove them from the relation. The *map* function would look up the set of *Bits* related to a given *Bit*, namely *this* (the *Bit* that caused the advice to be run). In response to a *Bit.Set*, for example, the advice would use *map* to compute the image of the *Bit*, and iterate over all the related *Bits*, updating their states as necessary.

```
Program 3
aspect Equality {
  static boolean busy;
  public boolean Bit.inRelation = false;
  public Bit Bit.peer;
  public void Bit.relate(Bit b){
      this.inRelation = true;
      this.peer= b;
      this.peer.inRelation = true;
      this.peer.peer = this;
  }
  pointcut callSet(Bit b):
    target(b) && call(void Set());
  pointcut callClear(Bit b):
    target(b) && call(void Clear());
  after(Bit b): callSet(b) {
    if (b.inRelation == true){
      if (!busy) {
        busy = true;
        b.peer.Set();
        busy = false;
      }
    }
  }
  after(Bit b): callClear(b) {
    if (b.inRelation == true) {
      if (!busy) {
        busy = true;
        b.peer.Clear();
        busy = false;
      }
    }
  }
}
```

Figure 7. A (faulty) aspect solution using *introduction.*

Each of these solutions has problems. The first is limited to one relationship instance for the whole system, and it does not scale as more entities and relationships are added to the design. The second limits each *Bit* to participating in one *Equality* relationship. That is not adequate to support our design scenario, in which *b2* eventually participates in two such relationships, with *b1* and *b3.* The third and fourth solutions solve this problem by allowing each *Bit* to be related to any number of other peer *Bits* in equality relationships.

The third and fourth solutions are attractive, too, in providing a solution for representing the *Trigger* relationship. A second aspect is introduced for this purpose. It is closely modeled on the *Equality* aspect. We have thus succeeded in representing *Bits* without crosscutting relationship code, and in representing two kinds of relationships without crosscutting each other. What remains?

The problem is highlighted by the fault involving *busy*. In our third solution (Figure 7), the *busy* variable belongs to the aspect *type*. The problem is that we need a *busy* variable per relationship *instance*. Consider how the extended system with four *Bits* fails. Suppose *b1* is *Set,* The *Equality* advice *after-callSet* is invoked. It finds *busy* clear, sets it, and then *Sets* b2. The *after-callSet* advice is invoked recursively for *b2* (which is correct). It finds *busy* on, and so returns immediately. The setting of *b2* should have resulting in the subsequent setting of *b3.* The second instance of the *Equality* relationship, linking *b2* to *b2,* was not handled properly.

The basic problem that this example reveals has two parts. First, each behavioral relationship *instance* can have an arbitrarily complex and stateful behaviors. Our example exhibits this point in the simplest non-trivial way: The one-bit *busy* variable represents the dynamic state of a behavioral relationship instance. Second, by default, there is only one instance of an aspect module per system.

The correct mapping of behavioral relationships in design to aspects now becomes clear. An aspect represents a type of behavioral relationship, and has to emulate OO-like creation, manipulation, and deletion of instances.

The fourth and final program, in which the aspect maintains a table of *Bit* pairs, can be repaired to provide a workable solution in this style. Without details, the key is to associate per-instance state with each pair of *Bits* in an aspect's table, and to provide means, e.g., in the form of static aspect methods, to get and set the state on a per-pair basis. One solution would be to reify each behavioral relationship as a record containing references to the objects it integrates and the required state components. The aspect would maintain a table of these tuples—a relation keyed by object pairs.

We can now characterize the modularity properties of *AspectJ* with respect to design structures containing behavioral relationships as independent parameters. *AspectJ* is not fully modular insofar as the relationship instances of a given type are not represented as abstract first-class objects in the program. Rather, their representations are merged together (a kind of crosscutting implementation) and are implicit in the state of a single aspect module. We have lost the mediator-like mapping of behavioral relationships in design to corresponding first-class objects in the program structure.

Second, a correct aspect-oriented solution could incur a significant performance overhead for the table lookups required to retrieve and manipulate the representations of behavioral relationship instances. Using a hash table to store the relation would maintain constant time in the number of instances, but at a significant cost in space, and still a significant cost in time relative to direct access to objects.

The root of the problem is that the *AspectJ* model of *aspect instances* is too limited to fully preserve the abstract structure of our designs in terms of the modular constructs of the programming language. This is the bound that we promised to exhibit. *AspectJ* remains not fully modular for component integration concerns that are abstracted as independent parameters in design.

There is one final detail. *AspectJ* does have a more general model of aspect instantiation than we have discussed so far. The *AspectJ* technical documentation states the following:

> If an aspect A is defined `perthis(`*`Pointcut`*`)`, then one object of type A is created for every object that is the executing object (i.e., "this") at any of the join points picked out by *`Pointcut`*. The advice defined in A may then run at any join point where the currently executing object has been associated with an instance of A…. Similarly, if an aspect A is defined `pertarget(`*`Pointcut`*`)`, then one object of type A is created for every object that is the target object of the join points picked out by *`Pointcut`*.

In other words, it is possible to have more than one aspect instance of a given type. Can we not use multiple instances in this way to implement mediators? The short answer is, no. There are two reasons. First, in our design an object can participate in several instances of any given behavioral relationship type. Our *b2* is linked to both *b1* and *b3* by separate *Equality* relationships. The *per this* and *per target* constructs of AspectJ, however, are limited to attaching *at most one* instance of an aspect of a given type to any given object. Second, these instances are limited to being attached to just one object, but what we need to represent are relational structures that connect co-equal participants in a relationship.

A final possible "hack" that we haven't yet evaluated adequately is to abuse the *per* constructs as follows. Define a *proxy* Java class in *AspectJ* and an aspect as being *per this* for each proxy instance. Proxy objects serve no purpose other than to circumvent the intent of the designers of *AspectJ's* to deny the programmer access to the runtime system for aspect instantiation.

We leave open the question of whether this "hack" can be exploited to emulate mediators with aspect instances. In any case, it so violates the intended style constraints that we consider it to be not properly in the realm of the *AspectJ* language and *style*. Our conclusions appear to stand for the intended usage of the language.

## 6. CONCLUSION

Significant advances in programming languages and design methods have been driven by the discovery of potentially valuable design structures for which existing approaches are not modular. The emergence of aspect-oriented programming is an example: it responds to the problem that OO languages, in particular, cannot preserve the modular structures of valuable, feasibly modularizable designs—that OO representations of such structures cannot avoid crosscutting representations of independent design parameters.

This perspective sheds some new light on some of the terminology being used to describe AO. In particular, an *aspect* at the program representation level is defined as a modular unit of crosscutting implementation. We can now rephrase this: An aspect is a modular representation of an independent design parameter in a "new" language, for which there is no modular representation in some other, often tacitly assumed, "old" language or language paradigm.

Looking back, we find that, by this definition, new languages have been aspect-oriented all along. In this paper, we have seen just two examples. Parnas found a new kind of design structure in which valuable dimensions of change and variation are represented as independent parameters in design. In his seminal paper he focused on variation of data structure choices in particular. He showed that the prevailing structured languages and related design methods were not modular for such design structures: procedure implementations were crosscut by assumptions about concrete data structures. He then introduced an information hiding style based on the use of ADT interfaces. In 1972, these were aspects: modular units of previously crosscutting implementation.

Similarly, Sullivan and Notkin saw that behavioral relationships, interactions protocols by which object behaviors are composed into systems, can profitably be abstracted as independent parameters in design. They then showed that such protocols are not modularizable in standard OO languages, but that they are modularizable in some practical languages: e.g., using ABTs to represent behavioral relationships as mediators. In this paper, we have shown that mediators are aspects not only in being modular units of otherwise crosscutting implementation, but also in having a direct mapping to the terms of modern AO language design: join points and advice. *Aspect-Oriented* languages are the latest (and perhaps the greatest) in a line of "aspect-oriented" languages stretching back more than thirty years.

Looking back, at each turning point someone had an insight that there is some kind of important design parameter that remains an aspect: for which the prevailing best language remains non-modular. With such an insight in hand, the task of inventing the next mechanism for preserving design modularity in programs can begin in earnest, and a new kind of modular program structure can be developed that can reasonably called an aspect relative to the previous language. When such mechanisms become common, what at first was seen as special, an *aspect,* becomes the new baseline against which new non-modularity properties are found.

Looking forward, then, the question for today is the old one: What kinds of design parameters remain aspects for today's best languages? For what important kinds of design parameters do the languages remain non-modular? In this paper, we ask the question, what remains an aspect for AspectJ?. We showed one such bound: that integration remains an aspect, to some degree, for *AspectJ.*

We do not suggest in this paper how to solve this problem. It is clear from earlier work on mediators and OO reflection that "weaving aspects dynamically" to modularize integration concerns is feasible. The real question is what are the tradeoffs? *AspectJ* is one a number of languages, including *HyperJ,* in which reflective behaviors are set up statically. These languages reap advantages from this constraint. One example is the pointcut language of *AspectJ*? Another is the performance gain of being able to compile away meta-level indirection. To what extent can the pointcut language and performance be preserved, while permitting runtime creation and binding of aspect instances? It is not clear.

Finally, we end with a bigger question—really the one we asked at the beginning. Brooks has distinguished between accidental and essential difficulties in software development. Each advance in aspect-oriented design shows that some previously intractable difficulty with crosscutting implementation was accidental: an artifact of a now evidently solvable problem in language design. Some bounds on the abilities of languages to preserve modularity in design are accidental.

The question is, What are the *essential* bounds on the modularity of program code, if any? For what kinds of concerns, abstracted as independent parameters, can no practical programming language ever be modular? Are there essential aspects? It appears that many hard-to-achieve (so called *non-functional*) properties of programs, such as dependability, might be inherently aspectual. Unreliability, for example, arises from faults that are by their very nature scattered about the code of a system. To modularize reliability as an aspect would appear to require the localization of all of the faulty code in a system—surely an absurd hope. What realistically can we hope for? What are the bounds of reasonable expectation?

## ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under grants CCR-9804078 and ITR-0086003.

## REFERENCES

[1] Xerox Corporation, AspectJ Team, *The AspectJ Programming Guide*, 2001, available at URL http://www.aspectj.org/ as of this writing.

[2] Elrad, T., R.E. Filman and A. Bader, guest editors, *Communications of the ACM 44*, 10, Special Issue on Aspect-Oriented Programming, October 2001.

[3] Gamma, Helm, Johnson and Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1994.

[4] Kalet, I.J., J.P. Jacky, M.M Austin-Seymour, S.M. Hummel, K.J. Sullivan and J.M. Unger, ``Prism: a New Approach to Radiotherapy Planning Software," International Journal of Radiation Oncology, Biology and Physics, 36, 2, 1996, pp. 451--461.

[5] Kiczales,G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier and J. Irwin, "Aspect-oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming* (ECOOP), Springer-Verlang, *Lecture Notes on Computer Science 1241*, June *1997.*

[6] Kiczales,G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W.G. Griswold, "An overview of AspectJ," in *Proceedings of the European Conference on Object-Oriented Programming* (ECOOP), 2001.

[7] Maes, P., "Concepts and experiments in computational reflection. Proceedings of OOPSLA'87, *ACM SIGPLAN Notices*, vol 22, 1987, pp 147-155.

[8] Notkin, D., D. Garlan, W.G. Griswold, and K. Sullivan, "Adding Implicit Invocation to Languages: Three Approaches," *Proceedings of the JSSST International Symposium on Object Technologies for Advanced Software*, (November 1993). (Also appears as a Springer-Verlag Lecture Notes in Computer Science volume \#742.)

[9] Ossher. H. and P. Tarr: Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development. Kluwer, 2000.

[10] Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM, 14(1): 221-227, 1972

[11] Sullivan, K.J., and D. Notkin, "Reconciling environment integration and software evolution," *ACM Transactions on Software Engineering and Methodology 1*, 3, July 1992, pp. 229–268 (short form: *Proceedings of the 4th SIGSOFT Symposium on Software Development Environments*, 1990, pp. 22–33).

[12] K.J. Sullivan, I.J. Kalet, and D. Notkin, "Evaluating The Mediator Method: Prism as a Case Study," IEEE *Transactions on Software Engineering*, 22, 8, August, 1996, pp. 563--579.

[13] Sullivan, K.J., *Mediators: Easing the Design and Evolution of Integrated Systems,* Ph.D. dissertation, University of Washington, 1994

[14] Sullivan,K.J., W.G. Griswold, Y. Cai and B. Hallen, "The structure and value of modularity in software design," Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 2001, Vienna, pp, 99–108.