Unifying Aspect- and Object-Oriented Design

HRIDESH RAJAN
Iowa State University
KEVIN J. SULLIVAN
University of Virginia

The contribution of this work is the design and evaluation of a programming language model that unifies aspects and classes, as they appear in AspectJ-like languages. We show that our model preserves the capabilities of AspectJ-like languages, while improving the conceptual integrity of the language model and the compositionality of modules. The improvement in conceptual integrity is manifested by the reduction of specialized constructs in favor of uniform orthogonal constructs. The enhancement in compositionality is demonstrated by better modularization of integration and higher-order crosscutting concerns.

Categories and Subject Descriptors: D.2.10 [**Software Engineering**]: Design; D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.2.2 [**Design Tools and Techniques**]: Modules and interfaces, Object-oriented design methods; D.2.3 [**Coding Tools and Techniques**]: Object-Oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms: Design, Human Factors, Languages

Additional Key Words and Phrases: Classpect, Unified Aspect Language Model, Binding, Eos, Aspect-Oriented Programming, Instance-Level Advising, First Class Aspect Instances

1. INTRODUCTION

The programming models of aspect-oriented (AO) languages [Kiczales et al. 1997] in the style of AspectJ [Kiczales et al. 2001], support the *aspects* as a separate module constructs distinct from the *class*, and *advice* as a procedural abstraction mechanism distinct from the *method*. Aspects are promoted as, and have demonstrated to some extent the ability to modularize concerns that cut across traditional abstraction boundaries [Sabbah 2004; Colyer and Clement 2004]. Other languages that explicitly make such distinctions include AspectC++ [Spinczyk et al. 2002], AspectR [Bryant and Feldt 2002], AspectWerkz [Bonér 2004], AspectS [Hirschfeld 2003], Caesar [Mezini and Ostermann 2003], and others.

The work described in this article is based on an idea described in the paper entitled "Classpects: Unifying Aspect- and Object-Oriented Language Design," published in the *Proceedings of the 27th International Conference on Software Engineering* (May 2005) and in the paper entitled "Eos: Instance-level Aspects for Integrated System Design," published in the *Proceedings of the 2003 Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering* (September 2003). This research was supported in part by NSF grants ITR-0086003 and FCA-0429947.

Authors' address: H. Rajan, hridesh@cs.iastate.edu, Computer Science, Iowa State University, Ames, IA 50010. K. Sullivan, sullivan@cs.virginia.edu, Computer Science, University of Virginia, Charlottesville, VA 22903.

These syntactic distinctions have influenced software development activities beyond just programming, in so far as they encourage a sharp conceptual separation of world into traditional and crosscutting modules.

The problem that we identify and address in this paper is an unnecessary lack of orthogonality, compositionality, and conceptual integrity in the design of this class of languages, due to the distinctions between class and aspect modules, and between methods and advice. By conceptual integrity we refer to the notion promoted by Brooks [Brooks 1995], that "conceptual integrity is the most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas."

The contribution of this work is a new language model that solves these problem to a significant degree with a novel module construct that unifies classes and aspects, and methods and advice. We call this construct the *classpect*. Classpects improve orthogonality and conceptual integrity by replacing separate but closely related constructs with a straightforward, unified construct. They improve compositionality by enabling better separation of integration and higher-order concerns.

The rest of this paper is organized as follows. The next two sections explain our motivation for, and describe, our approach. Sections 4 and 5 report on our tests to validate our claims. Section 4 discusses validation of the claim that classpects improve the separation of integration concerns. Section 5 presents our data in support of the claim that our language design improves the separation of higher-order concerns. Section 6 and 7 discuss related work and conclude.

2. NON-ORTHOGONALITY AND ASYMMETRY

In this section,with respect to three principles of programming language design proposed by MacLennan [MacLennan 1986], we reexamine one of the most fundamental decisions made early in the design of AspectJ [Kiczales et al. 2001]: to support separate but closely related class and aspect module constructs. The *orthogonality principle* suggests that "independent functions should be controlled by independent mechanisms." The *regularity principle* suggests that "regular rules, without exceptions are easier to learn, use, describe, and implement." The *simplicity principle* suggests that "a language should be as simple as possible and there should be a minimum number of concepts with simple rules for their combination." In what follows, we analyze the design of the class of AspectJ-like languages with respect to these criteria, taking AspectJ as the most prominent and well-developed exemplar of this class.

AspectJ is an extension of Java [Gosling et al. 1996]. The central goal of this language is to enable the modular representation of crosscutting concerns, including the representation of concerns conceived after the initial system design. Programs in these languages are typically developed in two phases [Sullivan et al. 2005]. Concerns that can be modularized using traditional object-oriented modularization techniques are expressed as classes. Concerns that crosscut traditional module boundaries are subsequently expressed as aspect modules that advise these so called base modules.

AspectJ adds five key constructs to the object-oriented model: join points, pointcuts, advice, inter-type declarations, and aspects. Such an aspect modifies the state space and the behavior of a program *before*, *after*, or *around* certain selected execution events (join

```
1  aspect Tracing {
2     pointcut tracedExecution(): execution(* *(..)) && !within(Tracing);
3     before(): tracedExecution() { /* Trace the methods */ }
4  }
```

Fig. 1. A Simple Example Aspect

points) exposed to such modification by the semantics of the programming language. Figure 1 presents a simple example. The `Tracing` aspect performs a tracing action immediately before any join point selected by the pointcut `tracedExecution`. The pointcut (line 2) is a predicate that selects a subset of join points for such modification — here, execution of any method outside the `Tracing` aspect. The advice (line 3) is a special, implicitly invoked, method-like construct that effects such a modification at each join point selected by the designated pointcut descriptor. An inter-type declaration (not shown here) statically introduces members such as fields or methods in other types. The aspect (lines 1-4) is a class-like module that uses these constructs to modify behaviors defined elsewhere in a software system.

2.1 Aspects and Classes

The motivation for the unification of aspects and classes rests on two observations[1]. First, separating classes and aspects reduces the conceptual integrity of the programming model, arguably making it harder in the long run for programmers to understand and use AOP. Second, the asymmetry of classes and aspects complicates system composition, and as we show, can actually harm modularity. Asymmetries occur in two areas. First, while aspects can advise classes, classes cannot advise aspects, and aspects cannot advise other aspects in full generality. Second, aspect instances cannot be created or manipulated under program control in the same ways as class-based objects. In practice, these asymmetries constrain the architectural styles realizable using advising as an invocation mechanism. For example, hierarchical layering of aspects is difficult at best.

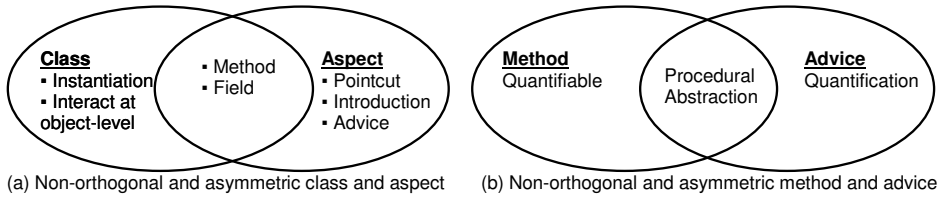


Fig. 2. Non-Orthogonality and Asymmetry in AO Languages

The root of these asymmetries lies in non-orthogonality of aspects and advice constructs in AspectJ-like languages. By non-orthogonality we mean that the language design does not comply with the MacLennan's orthogonality principle [MacLennan 1986]. These languages introduce the aspect as a separate abstraction mechanism and advice as a separate procedural abstraction mechanism. The new aspect construct and old class construct are

[1]These problems were discussed previously in [Rajan and Sullivan 2003b; 2003a; 2005b]

non-orthogonal. As presented in Figure 2-a, aspects overlap classes in supporting data abstraction and inheritance. They are more expressive than classes in their support for pointcuts, advice, and inter-type declarations. However, they are less expressive than classes in that aspect instances are not first class objects. Rather, AspectJ embraces a *module-like* semantics of aspect instantiation and advising.

2.2 Aspect Instances are not First-class Objects

An AspectJ aspect is a module-like construct in the following sense. First, that it is (in most cases) treated as a singleton. Second, aspect instantiation is not under program control. There is no general-purpose mechanism such as `new` for aspect instances. Rather, a single global instance is created by the language runtime on the first reference to an aspect. The interaction with the rest of the system is static in the sense that an aspect modifies the behaviors of the classes that it advises, and thus all instances of a given class in a given system are so modified.

There are mechanisms for associating instances of a given aspect with instances of a given instance on a case-by-case basis. For example, if an aspect definition includes a `perthis` modifier, an instance of the aspect is created for each object. If a `pertarget` modifier is used, an instance of the aspect is created for each object that is the target object of the join points e.g. target object of a call. Similarly, when `percflow` and `percflowbelow` modifiers are used, an aspect instance is created for each control flow. These mechanisms show that there are use cases where support for aspect instantiation is needed. However, current language design is contrary to the MacLennan's regularity principle in that the supporting mechanisms are limited and ad hoc.

In a nutshell, aspect instances are thus not first-class in two different ways. First, programs cannot manage their creation explicitly. Rather the language runtime manages aspect instantiation. Second, aspects cannot generally advise individual object instances. Rather, instance-level advising has to be emulated by class-level advice, complicating program design. This emulation can have significant costs in program design complexity and runtime performance.

Every instance of an advised class in a system has to pay such a performance penalty, even if the intention is to advise only some objects of that class. Furthermore, the cost imposed on each instance, as we will show, increases with the number of aspects that advise this class. Such a structure is not scalable in general. For example, in the case of generic classes such as collection classes, instances of which are used in many different ways throughout a system, such a situation is likely to be unacceptable.

To understand the source of the performance penalty and to explore possible optimization techniques that can be applied, consider a typical AO compilation strategy as presented in Figure 3. This strategy is used by the current compilers for AspectJ language. Implementation mechanisms have changed slightly in newer releases, e.g. recent releases operate at the byte code level instead of source code level, but basic strategy remains the same in that a call to advice is statically inserted at the join point.

The aspect is translated to an object-oriented class. The advice in the aspect is translated to a method with an automatically generated name. The advice invocation is implemented by statically inserting a method call to the generated method at the join point. Reflective information about the join point is also constructed, if necessary. The runtime construction of reflective information and the method invocation are the primary sources of the performance penalties for individual objects. These costs are incurred at join points whether or

```
public class Trace{
 public static Trace aspectInstance;
 static Trace(){
  aspectInstance = new Trace();
 }
 public static Trace AspectOf(){
  return Trace.aspectInstance;
 }
 public static bool HasAspect(){
  return Trace.aspectInstance!=null;
 }
 public void advice__0(){//Advice
  /* ... */
 }
}
```

```
class Sensor{
  boolean Signal;
  public void Detect(){
    Signal = true;
    /** Invoking an after advice **/
    Trace.aspectInstance.advice__0();
    }
  }
```

Fig. 3. A Typical Underlying Representation of an Aspect (left) and an After Advice Invocation (right)

not that object need to be advised. As we will discuss below, emulating selective instance-level advising would require a further lookup cost to determine whether an object is subject to advising. This cost is generally incurred on the advice side.

A clever compiler implementation might be able to optimize away some of these overheads in the same vein as Aotani and Masuhara [Aotani and Masuhara 2007] were able to optimize some conditional pointcuts that can be statically evaluated. Additional optimization strategies based on dynamic analysis are also applicable. For example, a compiler implementation that switches implementation strategies based on the nature of advising relationships in the system by looking at its execution profile may be able to provide the right implementation for the advising scenario.

Two possible optimization strategies seem possible: first, a compiler may collect the execution trace of the system beforehand and optimize the advising relationships according to the trace. Second, a compiler may add additional infrastructure to dynamically monitor and switch the implementation as needed at runtime. The first optimization strategy will incur a one-time cost of profiling the system. In this case, however, if the actual system behavior deviates from the profile some optimizations might have to be revisited in the light of the new execution profile. The second optimization strategy will incur a time and space overhead for including the monitoring infrastructure, for monitoring the program execution, and for switching the implementation strategy. Its not clear how such optimizations would perform. At a minimum they would significantly complicate language implementation.

Note that the design decision made by initial AspectJ language designers to commit to a non-object-oriented, namely static module-based, view of aspects may have been justified at the time. However, abandoning the key idea in object-oriented programming that running systems are composed of object instances, has real opportunity costs.

2.3 Advice and Methods

The advice and method constructs are also non-orthogonal and asymmetric. As shown in Figure 2-b, advice and method both support procedural abstraction. Advice is more expressive than method. It can quantify, i.e. it can use pointcuts to bind to join points [Filman and Friedman 2000]. Advice in other dimensions is less expressive than method. They are anonymous therefore it is not possible to distinguish between two advice constructs in a pointcut expression, instead the pointcut descriptor `adviceexecution` selects all advice constructs in a program. Note that pointcuts use lexical pattern matching on names to select join points. As a result, although an advice can advise methods with fine selectivity,

they can select advice bodies to advise only in coarse-grained ways. The granularity of selection is limited to all advice constructs in an aspect [2].

One may argue that splitting advice into two parts, a delegating advice and a method, copying the original advice body into the method body, and replacing the advice body with the call to the new method will solve the problem. In limited cases, it does work; however, for a large class of advice constructs, namely around advice that use proceed and before and after advice that use reflective information this approach doesn't work and more complicated workarounds are needed. We will revisit this workaround again in detail in Section 5. Furthermore, this workaround suggests a natural separation between advice bodies (what) from advice declaration (when). As we will describe in the next section, this intuition plays a significant role in our unification of advice and method.

This restriction that advice bodies can be selected for advising only in coarse-grained ways constrains application of advising as an invocation mechanism to two-layered structures. Here, methods at the bottom level are being advised by advices at top level. It also results in the lack of full aspect-aspect compositionality in the language model.

The lack of full aspect-aspect compositionality precludes use of advising as an invocation mechanism in a range of architectural styles including layered, hierarchical, and networked systems [Garlan and Shaw 1993]. Rajan and Sullivan [Rajan and Sullivan 2005b] have previously presented a case study of hierarchical architectures in which connectors integrate two or more components, and another level of connector treats two or more connectors as components. Another case study is discussed in Section 5. Both case studies demonstrate the restrictive compositionality. Sullivan and Notkin showed that such requirements are useful in designing systems for ease of evolution [Sullivan and Notkin 1990; 1992]. The inability to support such styles without workarounds–and the tacit constraints to two-layered designs, restricts natural use of aspect technology for separating concerns in a fully compositional style.

The next section describes the unified language design in which advising emerges as a general alternative to overriding or method invocation. There are two basic requirements for a unified, more compositional model.

—A new model should preserve the capabilities of AspectJ. This constraint rules out the use of languages with much more limited join point and pointcut models.

—A unified design should be based on a single unit of modularity (whereas AspectJ-like languages have both aspect and class), first-class notion of instances, and a single method construct for procedural abstraction (whereas AspectJ has both method and advice).

3. A UNIFIED LANGUAGE MODEL

In this section, we describe the design and implementation of a programming language model that unifies classes and aspects into a new module construct that we call *classpects*. We also present the design and implementation of a programming language Eos that supports classpect. Eos is an extension of C#. Similar extensions for Java are also possible. The underpinnings of the language design include:

—support for instantiation under program control,

[2] A pointcut such as `adviceexecution()&& within(TypePattern)` will suffice for such selection.

—instance-level advising,

—advising as a general alternative to method invocation and overriding, and

—the provision of a separate join-point-method binding construct.

3.1 Unifying Aspect and Classes

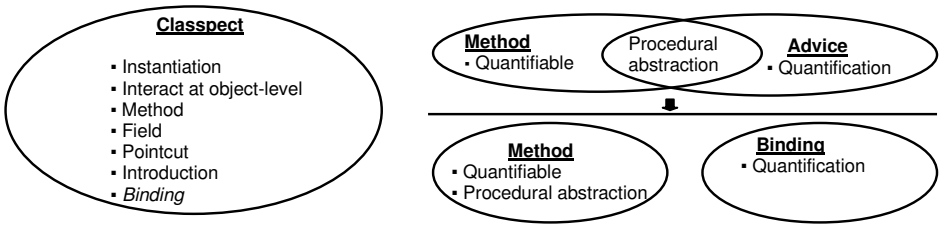
Eos unifies aspect- and object-oriented language design in three ways (See Figure 4-a).

—It unifies aspects and classes. A classpect supports: all C# class constructs, all essential capabilities of AspectJ aspects, and extensions to make aspect instances first-class.

—It eliminates advice in favor of using methods only, with a separate and explicit join-point-method binding construct.

—It supports a generalized advising model. To the object-oriented mechanisms of method invocation and overriding based on inheritance, we add implicit invocation using **`before`** and **`after`** bindings, and overriding using **`around`** bindings, respectively.

A classpect is declared using the keyword **`class`** for backward compatibility, i.e all C# classes are legal classpects.



(a) Unifying aspect and class as classpect (b) Unifying method and advice, binding as distinct construct

Fig. 4. A Unified Aspect Language Model

The unification proposed in this work also breaks the commitment to the static, module-based semantics of aspects. It provides a more general instantiation model in which classpects can be instantiated just like OO classes using the operator **`new`**. Classpects may also provide constructors like classes.

3.2 Binding Declarations

Eos eliminates anonymous advice in favor of named methods. It also makes join-point-method bindings separate and abstract, in a style similar to the event-method binding constructs of implicit invocation systems [Garlan and Notkin 1991; Sullivan and Notkin 1990; 1992]. Eos separates what we call crosscut specifications from advice bodies, which are now just methods, as presented in Figure 4-b. A binding declaration defines both a pointcut and when given advice should run: before, after or around. This separation allows one to reason separately about binding issues and to change them independently; and it supports advice abstraction, overloading, and inheritance based on the existing rules for methods.

The grammar production, ⟨bdecl⟩ (Figure 5) presents the binding declaration. A binding declaration has four parts. The first, optional modifier **`static`**, specifies whether a binding is static. The default modifier value is nonstatic, i.e. a binding only affects join points

⟨bdecl⟩ ::= [**static**] **before** ⟨pointcut⟩ **:** ⟨handler⟩ **;**
| [**static**] **after** ⟨pointcut⟩ **:** ⟨handler⟩ **;**
| [**static**] ⟨type⟩ **around** ⟨pointcut⟩ **:** ⟨handler⟩ **;**
⟨handler⟩ ::= ⟨identifier⟩ **(** ⟨form⟩* **)**

Fig. 5. Syntax of the Eos Binding Declaration in BNF, [] represent optional

of selected object instances. Rajan and Sullivan have called it instance-level advising [Rajan and Sullivan 2003a]. A static binding affects all instances of advised classes. The second part of a binding (**after**/**before**/**around**) states when the advising method executes: *after, before, or around*. The third part, **pointcut**, selects the join points at which an advising method executes. These join points are called *subjects* of the binding. The final part ⟨handler⟩ specifies an advising method. The advising method is called *handler* of the binding. If several methods are to execute at the join points selected by the binding, the handler could call a sequence of other methods in its body.

A binding can also pass reflective information about the join point to the methods invoked, by binding method parameters to reflective information using the AspectJ pointcut designators such as **this**, **target**, **args**, etc in a standard fashion. The handler methods have to follow certain rules. First, a handler method must be in the lexical scope of a binding. Second, a handler bound before or after a join point can have only void as a return type. Third, a handler bound around a join point must have a return type that is a subtype of the return type at the join point. For example, if the handler `Foo` is bound around the execution join point **execution**(**public int** `*.Bar()`), then it must return int. These design decisions are made to preserve the underlying language semantics for method calls. The methods may produce return value only when there is an explicitly specified consumer.

```
1   /* Advice */
2   before():execution(Sensor.detect()){
3     ...
4   }
```

```
1   /* Binding */
2   before():execution(Sensor.detect()):
3     beforeDetect();
4   /* Method Equivalent to Advice Body*/
5   void beforeDetect(){ ... }
```

Fig. 6. An AspectJ Advice and an Equivalent Eos Binding Declaration

The listing in Figure 6 presents an advice construct as it would appear in current aspect languages and the equivalent method binding in Eos. The advice executes before the join point **execution**(`Sensor.detect()`). The binding separates the advice body from the crosscut specification. The advice body becomes the body of the method `beforeDetect`. The crosscut specification becomes part of the binding (top right).

3.3 Around Bindings

An **around** advice in AspectJ is executed instead of a join point, and can invoke the join point using **proceed**. In essence, the around advice overrides the join point, with calls to **proceed** being analogous to delegating calls to **super**. In Eos, a method bound around is also executed instead of the join point. Unifying around advice and methods poses a question: whether to allow **proceed** in all methods. Allowing **proceed** in methods that are bound around but not in other methods introduces a special case. We instead choose

to make inner join point invocations explicit in an object-oriented style, which eliminates this special case making the language design more orthogonal.

In Eos, if an around-bound method might need to call the overridden join point, it takes an argument of type `AroundADP`. This type represents a delegate chain including the original join point and other around method bindings, and it provides a method called `innerInvoke` to invoke the next element in the delegate chain. The argument to the method is bound to the delegate chain at the join point using the pointcut designator **`aroundptr`** (line 6 in Figure 7). Note that C# programmers are already familiar with the notion of delegates as it has been part of the language since its inception. `AroundADP` is a type of delegate. Therefore, it does not add any new concept. There has been discussion about adding closures to Java and C#, which will further simplify the language design.

```
1   void cache (Eos.Runtime.AroundADP d){
2      if( /* need to invoke inner join point */)
3         d.innerInvoke();
4      }
5   static void around execution(public void SomeClass.someMethod())
6       && aroundptr(d): cache(Eos.Runtime.AroundADP d);
```

Fig. 7. A Method Bound Around

The binding (lines 5-6) binds the method `cache` around the execution of `SomeClass.someMethod` and exposes the around delegate chain at the join point using the pointcut expression **`aroundptr`**`(d)`, which binds the reference to the delegate chain to the argument `d` of the method `cache` (lines 1-4). The method `cache` can invoke the inner delegate in the chain by invoking the method `innerInvoke` on `d` (line 3).

A limitation of our current language implementation is that the return type of the method `innerInvoke` is `object`, precluding static type checking. This method's return type could be statically set to the common subtype of the overridden join points using generics.

3.4 New Pointcut Designators

To pass reflective information at a join point to a bound method, a binding uses AspectJ-like pointcut designators **`args`**, **`target`** and **`this`**. In AspectJ-like languages, three special variables are visible within the bodies of advice: **`thisJoinPoint`**, `thisJoinPointStaticPart`, and `thisEnclosingJoinPointStaticPart`. These variables can be used to explicitly marshal reflective information at a join point. For example, to access the return value at a join point, one calls the method `getReturnValue` on the variable **`thisJoinPoint`**.

Unifying advice and methods poses another question: whether to allow these special variables in all methods. Allowing these variables in methods that are bound before, after or around, but not in other methods introduces a special case. Eos removes this special case by requiring that the all required reflective information about the join point is explicitly supplied. The method arguments are bound to the required reflective information in the binding construct using pointcut designators.

The pointcut designators in the original Eos are incomplete for this purpose, in that not all the information available at join points is exposed. Other information, marshaled

earlier from the three special variables, might be needed. For example, to access the return value at a join point, one calls the method `getReturnValue` on the implicit argument. Eos adds new pointcut designators to fill the gap. For example, the pointcut designator **`return`** exposes the return value at the join point. The pointcut designator `joinpoint` exposes all information about the join point by exposing an object of type `Eos.Runtime.JoinPoint`. These designators allow previously implicit arguments to advice to be passed as explicit arguments to the method bound at the join points.

Eos fulfills the requirements laid out for a unified model. There is one unit of modularity, classpect, and one mechanism for procedural abstraction, method. All of the essential expressiveness of AspectJ-like languages is present in Eos, along with the extensions needed for aspect instances to work as first-class objects, as they must in a unified model. In addition, join-point-to-method bindings are separate, orthogonal, abstract interface elements in Eos. Eos thus does appear to achieve a novel unity of design in the programming model with respect to the family of AspectJ-like languages.

3.5 Additional Power of Overriding

In AspectJ-like languages, there are two different ways to override a method: by object-oriented inheritance and by AO around advice. A consequence of replacing advice bodies with methods is that methods that serve as advice can be overridden in either of these ways. These mechanisms differ fundamentally, and in a way consistent with the nature of AOP: not in their effect on runtime behavior, but rather on the design structure.

Consider two analogies. In object-oriented systems that support implicit invocation [Garlan and Notkin 1991], there are two ways for an invoker to invoke an invokee: explicit call or implicit invocation. The runtime result is the same, but the design-time structures are different. Having both mechanisms gives the designer the flexibility to shape the static structure independently of the runtime invocation structure. Inter-type declarations in AspectJ-like languages provide a similar capability for class state and behavior. They allow a third-party aspect to change the members of a class without the involvement of the class itself. The runtime effects are again the same, but the resulting architectural properties are different. Supporting inheritance and around advising as two mechanisms for overriding methods that serve as advice bodies provides just such architectural flexibility with respect to advice overriding. Object-oriented overriding demands an inheritance relation; AO around advising does not [Rajan and Sullivan 2005b].

3.6 Static vs. Non-Static Binding

A binding can be static or nonstatic. The (static binding, method) pair is equivalent to AspectJ advice. The nonstatic binding allows selective instance-level advising. Figure 8 provides an example usage of these constructs. Lines 2-4 in the left and right columns show an example of static and nonstatic binding respectively. The only syntactic difference is that the static binding construct is declared by putting the modifier **`static`** before it. The effect of static binding is to execute the method `trace` after the execution of the method `detect` for all sensor instances.

As the output at the bottom of the left column in the figure shows, after every call to the method `detect` the trace method prints the line `Before detect` on the console. On the other hand, the effect of nonstatic binding is to execute the method `trace` after the execution of the method `detect` for sensor instances selected for advising. As the output at the bottom of the right column in the figure shows, in case of nonstatic binding, the trace

```
1 class Trace{
2  static before execution(
3  public void Sensor.detect()):
4   trace();
5  public void trace(){
6   Console.WriteLine("Before detect");
7  }
8  public static void Main(..){
9   Sensor s1 = new Sensor();
10  Sensor s2 = new Sensor();
11  Sensor s3 = new Sensor();
12  Console.WriteLine("Setting Sensor 1");
13  s1.detect();
14  Console.WriteLine("Setting Sensor 2");
15  s2.detect();
16  Console.WriteLine("Setting Sensor 3");
17  s3.detect();
18 }
19 }
Output (All detect calls are traced):
Setting Sensor 1
Before detect
Setting Sensor 2
Before detect
Setting Sensor 3
Before detect
```

```
1 class SelectiveTrace{
2  before execution(
3   public void Sensor.detect()):
4   trace();
5  public void trace(){
6   Console.WriteLine("Before detect");
7  }
8  public SelectiveTrace(Sensor s){
9   addObject(s);
10 }
11 public static void Main(..){
12  Sensor s1 = new Sensor();
13  Sensor s2 = new Sensor();
14  Sensor s3 = new Sensor();
15  SelectiveTrace t = new SelectiveTrace(s1);
16  Console.WriteLine("Setting Sensor 1");
17  s1.detect();
18  Console.WriteLine("Setting Sensor 2");
19  s2.detect();
20  Console.WriteLine("Setting Sensor 3");
21  s3.detect();
22 }
23 }
Output (Selective detect calls traced):
Setting Sensor 1
Before detect
Setting Sensor 2
Setting Sensor 3
```

Fig. 8. Example Bindings: Static (left) and nonstatic (right): output presented in gray

method prints the line `Before detect` only when the method `detect` is called on sensor instance `s1`. Here the sensor instance `s1` is selected for advising by the classpect `SelectiveTrace`. The static binding has the usual semantics of AspectJ advice, in the rest of this subsection we will look at the semantics of nonstatic binding in more detail.

A classpect provides the ability to selectively advise object instances via nonstatic bindings and **`new`** for instantiation. An aspect, on the other hand advises a class and thus all objects of that class and does not provide a general mechanism for instantiation. Here the classpect `SelectiveTrace` binds the method `trace` to execute before the execution of the method `Sensor.detect` using a binding. The Eos compiler implementation reads the absence of static modifier as a hint to allow instance-level weaving. The Eos compiler delays binding of join point to methods until runtime. At compile time, it attaches event stubs at the matched join points, and generates implicit methods **`addObject`** and **`removeObject`** methods in the classpect `SelectiveTrace` to enable runtime registration and deregistration. In the next section, we will provide more detail.

The effect of calling **`addObject`** with an object as argument is to register all bound methods to be implicitly invoked. These bound methods are invoked at join points matched by the pointcut expression declared in bindings. In Figure 8, right column, calling **`addObject`** on line 9 registers `trace` to execute before the join point **`execution`**`(`**`public void`** `Sensor.detect())`. As a result, when the `detect` method is called on `s1`, the bound method `trace` is called on `t` before the execution of actual join point. The method **`removeObject`** deregisters all bound methods from all matched join points for that classpect instance.

3.7 Weaving of Static and Non-Static Bindings

The Eos compiler performs source-leve weaving to process the bindings and to generate appropriate stubs. In the rest of this subsection, we will discuss this process for both kinds of bindings. We will use the example described in the previous section to illustrate it. Note that this is just one realization of the unified model. Other implementations such as in a virtual machine [Dyer and Rajan 2008] are also possible.

For classpects containing only static bindings, the weaving process is the same as the AspectJ aspects as presented earlier in Figure 3. The Eos compiler inserts a static instance, a static constructor to initialize this instance, a method to retrieve this static instance and a method to check if the classpect contains any bindings. The compiler also inserts a call to the handler method for the static bindings at the join points. The handler methods are invoked on the static instance of the classpect. As a result, the Eos implementation does not incur any additional space and time overhead due to the weaving process when compared to the AspectJ implementation for the static binding case.

Figure 9 shows the compiler-generated code for `Sensor` and `SelectiveTrace` classpects. Note that `SelectiveTrace` uses nonstatic binding. The additional code inserted by compiler is marked. Two synthetic methods are added to a classpect containing nonstatic bindings, e.g. **`addObject`** on lines 18–28 and **`removeObject`** on lines 29–37 in the classpect `SelectiveTrace` in Figure 9.

```
1  class Sensor{
2   boolean signal;
3   public void detect(){
4    if(ADP_Detect!=null)
5     ADP_Detect.Invoke();
6     Signal = true;
7    }
8    public Eos.Runtime.ADP ADP_Detect;
9   }
10 using Eos.Runtime;
11 class SelectiveTrace{
12  public void trace(){
13   Console.WriteLine("Before ...");
14  }
15  public static void Main(String[] args){
16   ...
17  }
18  public void addObject(object obj){
19   if(obj == null)return;
20  /* An alternative is to use a polymorphic
21   implementation of addObject */
22   if(obj is Sensor){
23    Sensor cobj = ((Sensor)(obj));
24    cobj.ADP_Detect = ADP.Combine(
25     cobj.ADP_Detect,
26     ADP.Create( this,"trace"));
27    }
28  }
29  public void removeObject(object obj){
30   if(obj == null)return;
31   if(obj is Sensor){
32    Sensor cobj = ((Sensor)(obj));
33    cobj.ADP_Detect = ADP.Remove(
34     cobj.ADP_Detect,
35     ADP.Create( this,"trace"));
36    }
37  }
38 }
```

Fig. 9. Underlying implementation of selective trace: compiler-generated code presented in gray

For each join point that is the subject of any nonstatic binding in the system, a synthetic field of type `Eos.Runtime.ADP` (line 8) is inserted in the classpect. The type `Eos.Runtime.ADP` is an Eos specific implementation of the .NET delegates. The synthetic field is named to avoid conflicts. Here, the field `ADP_Detect`[3] is inserted in the `Sensor` corresponding to the join point "before the execution of the detect method". For

[3]For presentation purposes the original name of the field `ADP_Eos_before_execution_detect` is shortened to `ADP_Detect`.

each (join point, binding) pair such that the join point is a subject of the binding, a delegatee corresponding to the handler of the binding is added to this field. The location of the join point is instrumented to invoke this delegate, if it is not **`null`**(left column).

The generated method **`addObject`** takes an instance of type `object` as argument. The selective weaving process is applied to this instance. The **`addObject`** method first checks if the supplied instance is **`null`**. If it is **`null`** the method returns immediately. An **`if`** statement block is generated in the method **`addObject`** for each type that contains any subject join points for a nonstatic binding in the classpect. The **`if`** statement block checks if the argument `obj` matches any subject type, and casts it into a properly typed object accordingly. An alternative is to generate customized **`addObject`** methods for each subject type.

Here, the only subject join point is "execution of the method Detect" and it is contained in the classpect `Sensor`, hence only one **`if`** statement block is generated for type `Sensor`. After casting, a delegatee for the method `trace` is created by calling the operation `Create` of the runtime type `Eos.Runtime.ADP`. The operation `Create` takes an instance and a method name as argument and creates a delegate to call the named method on the supplied instance. The delegate is then combined with the delegate `ADP_Detect` of the `Sensor` instance `cobj` using the operation `Combine` of the runtime type `Eos.Runtime.ADP`. The operation `Combine` takes two delegate instances and combines their delegate chains such that the delegates in the chain of the second delegate instance are appended at the end of the delegate chain of the first delegate instance. Duplicates are not eliminated. The method **`removeObject`** works similarly.

In our implementation strategy, for a nonstatic binding, every instance of an advised classpect incurs a constant time overhead of a simple **`null`** check to determine if it is being advised. This overhead is very small compared to the cost of an advice invocation in case of the workaround. Only those instances that are actually advised by calling **`addObject`** invoke the delegate chain to run the handler methods. The space overhead for each instance of a classpect is an additional field of type `Eos.Runtime.ADP` corresponding to each subject join point in the classpect. The space overhead for classpects that also have bindings are two additional methods **`addObject`** and **`removeObject`**. The implementation strategy illustrates that in usecases where there is a need to emulate AspectJ-like aspects, a nonstatic binding can be used without incurring any additional overhead with respect to the AspectJ implementation. In usecases where there is a need to selectively advise object instances, nonstatic bindings can be used. The only additional overhead for every advised instance is a null check. These overheads are analyzed in detail in the next subsection.

3.8 Performance Analysis of Static vs. Non-Static Bindings

The provision of static and nonstatic bindings naturally leads to the question: When should a static binding be used as opposed to nonstatic binding?. This subsection analyzes the system structures and their fit, performance-wise, with either of these two levels of granularity of bindings, and offers guidelines for organizing bindings. In this analysis, relevant factors are total number of instances of a class and the subset of these instances that are affected by a binding. Let us assume that a set of instances of a classpect C_i is N_i and a set of instances of classpects C_i that are affected by a binding B_j is M_{ij}. The fraction of the instances of class C_i being affected by binding B_j is $\frac{|M_{ij}|}{|N_i|}$, where $|N_i|$ denotes the cardinality of the

set N_i and $|M_{ij}|$ denotes the cardinality of the set M_{ij}. Depending on the value of this fraction, different implementation strategies are warranted.

Consider the following five cases:

(1) $\frac{|M_{ij}|}{|N_i|} = 1$,

(2) $\frac{|M_{ij}|}{|N_i|} \rightarrow 1$,

(3) $1 - \delta > \frac{|M_{ij}|}{|N_i|} > \delta$, where $\delta \rightarrow 0$,

(4) $\frac{|M_{ij}|}{|N_i|} \rightarrow 0$,

(5) $\frac{|M_{ij}|}{|N_i|} = 0$.

For the first case, the fraction is one (i.e., the total number of instances of the class is equal to the number of instances that should be advised). These bindings should always be static. For the bindings in the fifth partition, the fraction is zero (i.e., no instance is being affected). For the second, third, and fourth case, the fraction is between 0 and 1 (i.e., some instances are being affected). In the second case, the fraction tends to 1, in other words nearly all instances are being affected but not all. In the fourth case, the fraction tends to 0, which means that a very small fraction is being affected.

Ideally, bindings in the second, third, and fourth cases need to be nonstatic or in other words these bindings should be selective instance-level. If the language does not support selective instance-level bindings, it can be implemented in at least two different ways. In the first implementation strategy, the method bound by B_j keeps a list of instances (M_{ij}) being advised and then looks up the invoking instance in that list. This implementation strategy will incur lookup overhead and method invocation overhead. Let us assume the method invocation overhead to be O_{invoke} and the constant lookup overhead of looking up an instance in the list of instance (M_{ij}) to be O_{lookup}(assuming an O(1) lookup algorithm). Total overhead of this implementation technique is:

$$\sum_j \sum_{\forall i(where |M_{ij}|>0)} ((|N_i| - |M_{ij}|) * O_{invoke} + |N_i| * O_{lookup}).$$

This overhead has two parts: the overhead incurred by each instance that should not be affected by the binding and the overhead of looking up an instance to determine whether it should be affected by the binding. The second part of the overhead is constant. For the second case, the first part of this overhead is not significant, whereas for bindings in the fourth partition, the first part of the overhead is significant. For bindings in the third partition, as the fraction of instances affected decreases, this overhead will increase. It might be possible, however, for a language implementation to employ static or dynamic analysis techniques such as profiling to optimize these overheads as previously discussed.

In the second implementation technique, an object instance will keep a list of bindings affecting it and invoke all bound methods in the list one by one or depending on some precedence. There is no lookup overhead in this case; however, a zero check will be necessary to determine whether the list is empty. This condition check will only be necessary for the join points that are potentially affected by the binding. Let us assume constant overhead of this condition check is O_{zero}. The total overhead in this case will be:

$$\sum_{\forall i \exists j(|M_{ij}|>0)} |N_i| * O_{zero}.$$

Given that $O_{zero} \ll O_{lookup}$ the overhead in first case will be significantly larger than second case. The Eos compiler implements the second technique (to achieve minimal overhead) without introducing design dependence.

3.9 Summary

In this section, we showed that the unification of aspects and classes in a language design is possible. The unification brings conceptual unity to the programming model. In the new language design, aspect-like constructs support all of the capabilities of classes-notably *new*. Classes support AO advising as a generalized alternative to traditional invocation and overriding. Supporting the new operator eliminates the need for the irregular per* constructs from the language design. The anonymous, asymmetric, and non-orthogonal advice was replaced in favor of methods as the sole mechanism for procedural abstraction. The asymmetric and non-orthogonal aspects were eliminated and quantified binding surfaced as the central notion of AOP.

We presented the design and implementation of Eos, an extension of C# that embodies the unification. In Eos, classpects are the basic unit of modularity. The classpect instances are first class in all respects: they can be created at will, passed as parameters, returned as values, etc. Besides OO method invocation, classpects also offer AO method-join point binding as a generalized invocation mechanism.

4. SEPARATION OF INTEGRATION CONCERNS

Component integration creates value by automating the costly and error-prone task of imposing desired behavioral relationships on components manually. A problem is that straightforward software design techniques map integration requirements to scattered and tangled code, compromising modularity in ways that dramatically increases development and maintenance costs. This section presents the first data point in the overall evaluation of our approach. It validates the claim that the unified model improves the separation of integration concerns. It compares the implementation of a simple but representative integration scenario using AspectJ and Eos.

This section is organized as follows. First, we present a representative example system and its simple object-oriented implementation. The latter demonstrates that component integration is indeed fragmented, scattered, and tangled with the component code. Secondly, we briefly describe mediator-based design style [Sullivan and Notkin 1992] to separate integration concerns and present the implementation of our representative system using this technique. This design style largely modularizes integration concerns, but leaves event declarations, announcements, and registrations scattered and tangled, which suggests that AO languages could be a good candidate to fully modularize this concern. Finally, we discuss and analyze AO implementations of our example system using AspectJ and the unified model of Eos and compare these two versions to verify the claim that the unified model improves modularization of integration concerns.

4.1 A Running Example

A simple but representative example system is shown in Figure 10. This system has components of two types: `sensors` (black boxes) and `cameras` (grey boxes). There are two sensors `s1`, `s2`, and two cameras `c1` and `c2` in the system. These components are required to behave together such that whenever sensor `s1` detects a signal, the cameras `c1` and `c2` take a picture and whenever sensor `s2` detects a signal only camera `c2` takes a picture.

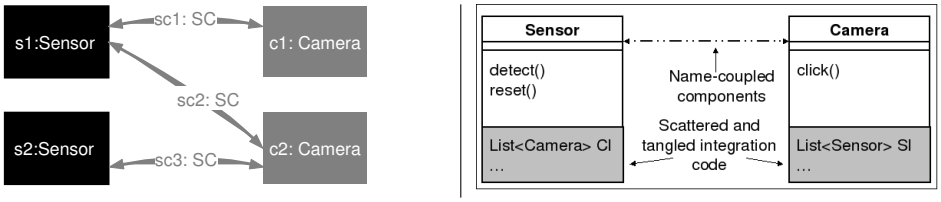


Fig. 10. Sensor System and a Simple Object-oriented Design

Furthermore, when the cameras are done taking pictures, a flag in corresponding sensor is reset so that it can sense again. These relationships coordinate the control, actions, and states of subsets of system components to satisfy overall system requirements. They are also called *behavioral relationships* [Sullivan and Notkin 1992].

From here on, we will use the terms behavioral relationship and integration concern interchangeably. The behavioral relationship "whenever a sensor detects a signal, the camera takes a picture" is also called the sensor-camera integration concern. Figure 10 shows the behavioral relationships between components as double-headed arrows. The integration relationship between a sensor and a camera is labeled `SC` and shown as a grey arrow. There are three instances of this relationship `SC`, `sc1` between sensor `s1` and camera `c1`, `sc2` between sensor `s1` and camera `c2`, and `sc3` between sensor `s2` and camera `c2`.

This example system is a representative of a broad class of systems called integrated systems. An integrated system is one in which logically unrelated objects, such as compilers, editors, debuggers, or any other kinds of discrete components must interact dynamically to meet system-level requirements (e.g., that the editor must automatically open the right file and scroll to the right line when the debugger encounters a breakpoint). The key ideas that this example demonstrates are, first, a component type may be integrated using more than one kind of integration relationship, and second, an instance of a component might participate in more than one instance of any given kind of relationship. The next subsection looks at a simple implementation of this system using object-oriented programming techniques.

4.2 Simple Object-Oriented Integration

In this implementation strategy, components are represented as instances of object-oriented classes. Figure 10 (right) shows classes implementing `sensor` and `camera` components. Realizing desired behavioral relationships between components requires `sensors`, and `cameras` to keep track of the other component instances with which they are integrated. One way to do so is to keep a list of other component instances with which a given component is integrated. As shown in Figure 10, the `sensor` class keeps a list of `cameras` with which it is integrated and vice-versa, thereby referring to these components.

Components to observe the desired behavior will need to invoke each other, which will be achieved by calling each other and thus there will a name dependence between these components resulting in coupling and preventing their separate compilation, link, test, use, etc. For example, in Figure 10, sensor `s1` will invoke camera `camera c1`, `c2` and `s2` will invoke camera `c2` to take pictures. Cameras, when done taking pictures, will invoke respective sensors to reset them. Figure 11 shows the object-oriented implementation of the class `Sensor` to make the points concrete.

In this implementation, the code that implements the sensor-camera integration concern is scattered across the sensor and the camera implementation. Let us assume that the sensor

```
class Sensor{
 boolean signal;
 // Sensor-Camera Integration Concern
 List cameraList = new LinkedList();
 public void addCamera(Camera camera){
  cameraList.add(camera);
  }
 public void invokeCameraClick(){
  Iterator iter = CameraList.iterator();
  while (iter.hasNext())
   ((Camera)iter.next()).click();
  }
 boolean sense = true;
 public void detect(){
  if ( sense ) {/*Recursion Guard*/
   sense = false;
   signal = true;
   if(cameraList.iterator().hasNext())
    invokeCameraClick();
    }
  }
 public void reset(){ sense = true; }
 }
```

Fig. 11. Scattered and tangled integration concerns in OO implementation of sensor - the integration concern is marked in gray. In this figure a simple reset mechanism is added to prevent mutually recursive calls between detect and click. From here onwards, we will elide this mechanism.

component was also integrated with another component type (say display). The code that would have implemented the sensor-display integration would also have been scattered across sensor and the display components. Moreover, this code would have been tangled with the sensor concern and the sensor-camera integration concern in the method `detect` of the class `Sensor`.

4.3 Mediator-Based Design

The fragmented, scattered, and tangled integration code increases the probability of faulty software due to inconsistencies and leads to costly time-consuming development making software evolution hard. The mediator-based design style [Sullivan and Notkin 1990; 1992] was introduced to ease the design and evolution of integrated systems. As previously described, integrated systems are a very broad class of systems in which objects have to work together to achieve system objectives. This design style advocates structuring designs as behavioral entity relationships (ER) models and preserving the modular structure of these designs in programs. The key idea is to modularize behavioral relationships or integration concerns as separate mediator modules. An implementation of the sensor-camera system using this design style is described and analyzed in this subsection.

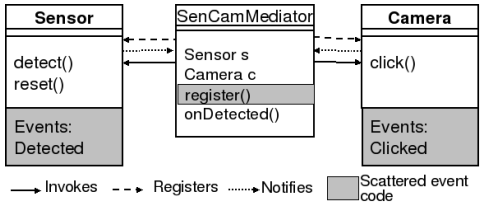


Fig. 12. Mediator-based Design of the System

The programming solution using the mediator approach involves two ideas. First, the entities and relationships are now represented as abstract behavioral types (ABTs) instead of abstract data types (ADTs) [Liskov and Zilles 1974], as in OO languages. The abstract behavioral type is an extension of the abstract data type (ADT). ADTs define abstractions in terms of method interfaces; ABTs, in contrast, also include explicitly exported events. An ADT defines a class of objects in terms of operations that can be applied to an object of that class. An ABT defines a class of objects in terms of the operations that can be

```
class Sensor{
 boolean signal;
 List dList = new LinkedList();
 public void registerDetect
  (Mediator mediator){
   dList.add(mediator);
   }
 public void invokeAfterDetect(){
  Iterator iter = dList.iterator();
  while (iter.hasNext())
   ((Mediator)iter.next()).onDetect();
  }
 public void detect() {
  signal = true;
  if(dList.iterator().hasNext())
   invokeAfterDetect();
  }
}
```

```
class SenCamMediator implements Mediator {
Sensor s; Camera c;
public SenCamMediator(Sensor s, Camera c){
this.s = s; this.c = c;
}
s.registerDetect(this);
c.registerClick(this);
}
public void onDetect(){c.click();}
public void onClick(){s.reset();}
}
```

Fig. 13. Scattered Event Code in Mediator-Based Implementation (shown in gray)

applied to an object of that class and 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. For example, consider a simple ABT `Sensor`, which provides operation to convey detection of a signal. Apart from this operation, the `Sensor` ABT also defines an event `Detected` that is announced whenever any Sensor instance detects a signal.

Second, entities and relationships are mapped to corresponding ABT-based objects in a way that would avoid crosscutting implementations of behavioral relationships. For example, the behavioral integration relationship Sensor-Camera, will be modeled as a separate ABT, `SenCamMediator`. An instance, `sc1`, of this ABT `SenCamMediator` will register with events announced by sensor `s1` and camera `c1`. When sensor `s1` announces the event `Detected`, the mediator `sc1` will invoke the method `click` on `c1`. On the other hand, when camera `c1` announces the event `Clicked`, the mediator `sc1` will invoke the method `reset` on `s1`. Other behavioral relationships will be modeled similarly. The integration concern is thus largely modularized in the `SenCamMediator`.

At the source code level, in a mediator-based implementation of the sensor system (see Figure 13), components are represented as instances of object-oriented classes `Sensor` and `Camera`. The classes representing components expose events as well as methods in their interfaces. Objects announce events to notify registered mediators of occurrences. The behavioral relationships between component instances are represented as instances of separate classes, called mediators. Mediator instances function as observers that effect component integration upon notification. Here the sensor-camera integration concern is represented as the `SenCamMediator` mediator. A `SenCamMediator` instance would maintain references (`s`, `c`) to `Sensor` and `Camera` instances to be integrated; implement method `onDetect` and `onClick` that explicitly invokes `c.click` and `s.reset` respectively; and, registers `onDetect` and `onClick` to be implicitly invoked by `s.Detected` and `c.Clicked` events.

Now when the sensor detects a signal, the mediators respond by clicking the cameras and vice versa. Yet the representations of the sensor and camera are not statically tied to each other or to the mediator. The sensor and camera components also remain uncomplicated by integration code and the behavioral relationships are largely modularized. Behavioral integration is thus reconciled with component independence, resulting in a modular struc-

ture that supports the independent implementation, testing, use, and evolution of the sensor and camera components, and easier evolution of the now separately abstracted integrating behavioral relationship.

However, at least one problem remains. Components to be integrated have to declare and announce events sufficient to meet the needs of observing mediators. There is thus design dependence between components and mediators, even though there is no naming dependence: Component designers have to be aware of requirements for event notifications imposed by extant or admitted mediator types; and unanticipated changes in that set can require changes in mediated component types. Filman and Friedmann would argue that the components are not oblivious to mediators [Filman and Friedman 2004]. These changes may not even be possible to controlled source code or to components in a third party supplied library. Kiczales would argue that the behavioral relationships are not entirely modularized, insofar as the required event code has to be produced and maintained by the component developers [Kiczales et al. 1997]. Moreover, the event declaration, announcement and registration code is fragmented, scattered, and tangled with the component code as shown in the Figure 13. Thus, adding a new behavioral relationship and corresponding mediator class can require changes to multiple component classes–in the worst case, across a whole system.

Aspect-oriented programming (AOP) has emerged to address precisely the problem of scattered and tangled concerns. A crucial difference between the mediator-based design style and AOP is in the underlying event model. The mediator approach assumes explicit event declaration, announcement, and registration. On the other hand, in the aspect-oriented programming model, language semantics makes a subset of events in program execution available as implicitly declared join points. Pointcut expressions are provided to register with a set of these implicit join points.

Aspects lead to an idea for improving mediator-based design: implement mediators as aspects, and use join points and pointcuts in place of explicit events. The solution for our example simultaneously preserves the name independence of the components being integrated by providing a mechanism that enables one component to invoke another without naming it and eliminates the need for fragmented, scattered, and tangled event concern. Sullivan et al. [Sullivan et al. 2002] investigated the mapping of mediators to aspects in AspectJ-like languages. They showed that mediators can be implemented as aspects in current AspectJ-like languages, with one caveat. Most current, major aspect languages, including AspectJ and HyperJ, suffer from two shortcomings with respect to the mediator style. First, aspects are essentially global modules, rather than class-like constructs supporting first-class instances under program control. Second, aspects advise entire classes, not object instances. The next subsection summarizes and significantly extends the results presented there.

4.4 Mediators as Aspects

In the Mediator-based design style, behavioral relationships between component instances are represented as instances of separate classes, informally called mediators. These mediator instances then register selectively with component instances to receive event notifications. From the description of the design style, two feature requirements emerge for any abstraction that is used to represent mediators. First, it should have instantiation capabilities. Second, it should be able to create associations selectively with component instances.

```
class Sensor{
 boolean signal;
 public void detect() {
  signal = true;
  }
 /* reset mechanism elided */
}
class Camera{
 boolean clicked;
 public void click(){
  clicked = true;
  }
}
class SCMediator implements Mediator {
 Sensor s; Camera c;
 public SCMediator(Sensor s, Camera c){
  this.s = s; this.c = c;
  }
 public void onDetect(){ c.click(); }
 public void onClick(){ s.reset(); }
}
```

```
aspect SenCamMediatorModule {
 static WeakHashMap map;
 static{map = new WeakHashMap();}
 public void connect(Sensor s, Camera c){
  SCMediator sc =
   new SCMediator(s, c);
  map.put(sc, s); map.put(sc, c);
 }
 before():execution(void Sensor.detect()){
  Sensor s = (Sensor)
  thisJoinPoint.getThis();
  SCMediator sc =
   (SCMediator)map.get(s);
    if(sc!= null)sc.onDetect();
 }
 before():execution(void Camera.click()){
  Camera c = (Camera)
  thisJoinPoint.getThis();
  SCMediator sc =
  (SCMediator)Map.get(c);
  if(sc!= null) sc.onClick();
 }
}
```

Fig. 14. An AO Implementation: Emulating Instance-level advising using Hashmaps

These two requirements translate to two key features in the AO world – the ability to arbitrarily instantiate aspects and the ability to selectively advise object instances.

Unfortunately, most major current aspect-languages lack these features. A survey of a subset of AO languages and approaches is presented in Section 6. The survey showed that the combination of features required for mediator-based design style is not present in any of the extant languages and approaches.

The mediator approach requires instance-level weaving and first-class aspect instances because it requires that each type of behavioral relationship be represented as a mediator class, with class instances representing relationship instances. Moreover, the class instances register with the events of the instances to be integrated. Recall, for example, that in our sensor system we have three instances of the `SenCamMediator`, one connecting the sensor `s1` to the camera instance `c1`, second connecting the sensor `s1` to the camera instance `c2` and the third connecting the sensor `s2` to the camera instance `c2`. Each `SenCamMediator` instance registers with the detected event of the sensor instance, and with the clicked event of its particular camera instance.

Mediators cannot be mapped directly to aspect instances in AspectJ-like languages because aspects cannot be instantiated in a general way, nor can they selectively advise instances of other classes. Work-arounds are possible in AspectJ, but even the best ones known incur unnecessary, non-negligible costs in performance or design complexity. The next subsection looks at two such work-arounds.

4.5 Work-Arounds and Their Costs

There are at least two basic work-arounds consistent with the use of aspects as behavioral relationship modules. In both cases, behavioral relationship types are mapped to aspect modules programmed to simulate first-class aspect instances and instance-level advising.

To simulate instances, the aspect provides methods to create, delete, and manipulate instances implemented as records. The difference is in the simulation of instance-level advising. In the first approach shown in Figure 14, the aspect advises relevant join points

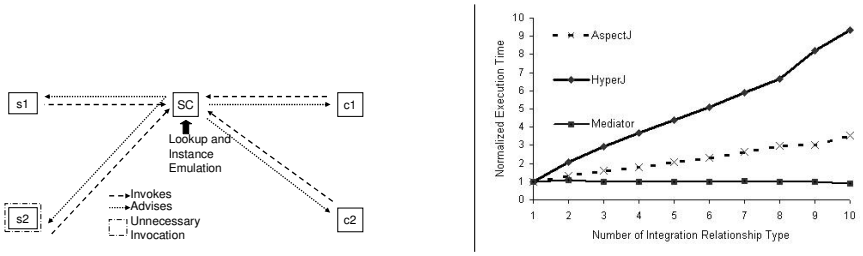


Fig. 15. Runtime Object Structure for an Aspect-based implementation (left) and Performance Curve for AspectJ, HyperJ, and Mediator-Based Design (right)

```
class Sensor{
  boolean signal;
  public void detect() {
    signal = true;
    }
  /* reset mechanism elided */
  }
class Camera{
  boolean clicked;
  public void click(){
    clicked = true;
    }
  }
class SCMediator implements Mediator {
  Sensor s; Camera c;
  public SCMediator(Sensor s, Camera c){
    this.s = s; this.c = c;
    s.registerDetect(this);
    c.registerClick(this);
    }
  public void onDetect(){ c.click(); }
  public void onClick(){ s.reset(); }
  }
```

```
aspect SensorExtension{
 introduce in Sensor {
   List dList = new LinkedList();
   public void registerDetect(Mediator mediator){
    dList.add(mediator);
    }
   public void invokeAfterDetect(){
    Iterator iter = dList.iterator();
    while (iter.hasNext())
     ((Mediator)iter.next()).onDetect();
     }
    }
  }
 after(Sensor s):
  execution(public void Sensor.detect())
    &&this(s){
    s.invokeAfterDetect();
   }
}
aspect CameraExtension{
/* Similar to SensorExtension introduce
  the list MediatorListClick, the method
  RegisterClick for mediators to register,
  and the method InvokeAfterClick for
  advice to invoke mediators.*/
}
```

Fig. 16. Alternative AO Implementation: Emulating Instance-level advising using Introductions

of the classes whose instances are to be integrated. All instances invoke the aspect at each such join point. The aspect maintains tables recording the identities of objects to be treated as advised instances. When the aspect is invoked, it looks up the invoker to see if it is such an instance. If so, the aspect delegates control to a simulated advice method, if not it returns immediately (See Figure 15 (left)).

This work-around works in the sense that it both modularizes the behavioral relationship code, data, and invokes relations, and relieves the developer of having to work with explicit events. However, the approach is less than ideal for several reasons. First, it is awkward to have to simulate instances in an otherwise object-oriented language. Second, such programs are actually harder to understand: the aspects read as advising classes, when, in fact the intent is to advise instances. Third, the approach adds unnecessary design complexity and thus cost and undependability with simulation implementations. Fourth, the workaround adds runtime overhead in two dimensions. In particular, at each join point, each instance of an advised class has to invoke each advising aspect, if only to have it re-

turn upon failing the check for a simulated advised instance. The sensor instance marked with unnecessary invocation in Figure 15 (left) is one such instance.

To understand the performance impact of the work-around, we did a study of the penalties associated with implementing mediators using AspectJ and HyperJ. The HyperJ implementation is not discussed here but described in detail elsewhere [Rajan 2005]. Figure 15 (right) presents the results. The X-axis shows the number of different mediators or aspects that advise the `Detected` event of the simple `Sensor` type. The Y-axis shows the normalized execution time of invoking the `detect` member function 10^8 times averaged over 15 runs. The normalization factor was the absolute execution time in the case of a single integration relationship. These experiments were conducted on Sunfire v210 workstation with Dual 1.0 GHz UltraSparc IIIi and 2GB RAM. The cost per invocation clearly rises, as expected, with the number of aspects advising the type–and will do so for every instance of the type. The advice code in this case did nothing but return immediately. Having additional code in the advice will affect the study in two ways. First, the fraction (overhead-time / advice-execution-time) will decrease if the advice-execution time increases. Second, additional lookup code will have to be added to execute the advice only when required. Note that the results presented here represent the state of the implementation of AspectJ and HyperJ compilers and not of the language model. Also note that it might be possible for the compiler implementation to optimize some of these overheads but no compiler implementation as of this writing appears to provide these optimizations, nor is it entirely clear that effective optimization is even technically f easible.

By contrast, as the chart shows, a mediator style of integration using plain Java imposes a very small, constant overhead. Event code needs to be present in the class `Sensor` to notify registered mediators, but if none are registered, the cost is constant, a single *zero* check, to see if any mediators are registered to be invoked. A slightly larger cost is paid only for mediators registered with specific object instances. It does not matter how many mediator types might advise a given object, but only how many mediator instances actually do so. It does not matter how many join points are advisable.

There are situations in which the cost of this work-around might be unacceptable. An example would be the case of a mediator implemented as an aspect that has to respond to insertions on just one instance of a widely used, basic List class. It would be unreasonable, and is unscalable, for all clients of all instances of List to have to pay a price for one, isolated client.

The second work-around uses AspectJ-like introductions to extend the classes to be integrated with explicit event interfaces and code (See Figure 16). The aspect also advises the join point at which the event is to be announced. The advice announces the event. The objects registered are not themselves aspects but ordinary mediators. The effect is to implement a traditional mediator design, but with explicit events modularized with the mediators that need them.

This work-around works, too, achieving integration without loss of modularity. It relieves the component developer of having to anticipate the events that mediators might need. In that sense, it arguably improves on the original mediator style; however, it also incurs the performance overhead of the first work-around. The additional advice call, which in turn calls `invokeAfterDetect`, is made for all sensor instances. Finally, the approach has two other problems. First, it does not really implement mediators as aspects at all, but only modularizes the explicit events that mediators need. It misses the point:

we want to use join points rather than explicit events to invoke mediators. Having join points invoke advice that announce events that invoke mediators is at best a complex, relatively costly approach, using redundant mechanisms (events, join points). Second, if several mediators need to respond to the same event, each introduces its own event code and interface–bloating the code–rather than using the same event or join point.

4.6 The Conceptual Gap

In his famous 1968 letter to the editors of the Communications of the ACM, *Go To Statement Considered Harmful* [Dijkstra 1968], Edsger Dijkstra wrote:

> We should do . . . our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Dijkstra's argument in the context of conceptual gap between code and runtime structure is equally applicable to the mapping between specification and runtime structure. In essence, Dijkstra argues that the runtime conceptual model of the system should in fact be very close to any static model of the system including its specification. MacLennan's structure principle [MacLennan 1986] of programming language design similarly suggests that "the static structure of the program should correspond in a simple way to the dynamic structure of the corresponding computations".

Figure 19 (top) shows the ideal object structure and Figure 19 (bottom-right) shows the actual AO-mapping from specification of the sensor system to its runtime structure. The conceptual model of the system at runtime is far removed from that at specification time in two significant ways: first, instances of the behavioral relationships in the specification are represented by one global aspect-instance at runtime, and second, associations between component instances and behavioral relationship instances are mangled in the runtime structure. The gap between these two models leads to unnecessarily hard to understand programs, as well as design time complexity. This conceptual gap can be traced back to the static module-based view of aspects.

To recall, Eos unifies aspects and classes in favor of a more compositional block of program design, namely classpects. Like classes, classpects are first class (i.e. they can be instantiated). Similar to aspects, classpects can advise using bindings. In addition to advising at the type-level, classpects can also selectively advise instances. Eos has a rich join point model and expressive pointcut language. These features promise to fill the gap in the realization of the mediator-based design style using AOP techniques. The next subsection shows the implementation of the sensor system in Eos.

4.7 Mediator as Classpects

Figure 17 shows the implementation of `Sensor-Camera` integration concern using classpects. The implementations of `Sensor` and `Camera` components are elided for presentation. The implementations of these concerns, however, remain unchanged, separate from the implementation of the integration concerns and from each other. A modularization of component concerns is thus achieved in this implementation.

Mediator is declared as classpect using the keyword **`class`** (line 1). The implicit method **`addObject`** is used on line 5 to selectively advise objects `s` and `c`.

```
1  class SenCamMediator{
2    Sensor s; Camera c;
3    public SenCamMediator(Sensor s, Camera c){
4      this.s = s; this.c = c;
5      addObject(s); addObject(c);
6      }
7    after():execution(public void Sensor.detect()):onDetect();
8    public void onDetect(){c.click();}
9    after():execution(public void Camera.click()):onClick();
10   public void onClick(){s.reset();}
11   }
```

Fig. 17. Classpect-Based Implementation of the Integration Concern

```
1   public static void Main(string[] arg){

3     Sensor s1 = new Sensor();
4     Sensor s2 = new Sensor();
5     Camera c1 = new Camera();
6     Camera c2 = new Camera();
7     ...
8     SenCamMediator sc1 = new SenCamMediator(s1,c1);
9     SenCamMediator sc2 = new SenCamMediator(s1, c2);
10    SenCamMediator sc3 = new SenCamMediator(s2, c2);
11    ...
12   }
```

Fig. 18. Modular Composition of Components and Connectors

The classpect `SenCamMediator` declares two nonstatic bindings (lines 7 and 9). To recall, nonstatic bindings allow selective advising of object instances. The first binding on line 7 binds the method `onDetect` to execute after the execution of the method `detect` in `Sensor`. The second binding on line 9 is similar. The methods `onDetect` and `onClick` in the classpect `SenCamMediator` encapsulate the integration logic. The solution therefore achieves modularization of integration logic. The join point/pointcut model of AO languages is used instead of explicit event declaration, announcement, and registration code modularizing the scattered and tangled event concern as well. This solution thus achieves a complete modularization of the integration concern.

Due to selective instance-level advising, the method `onDetect` is only invoked when the method `detect` is called on sensor instance `s`. Unlike type-based aspect implementation shown in Figure 14 and 16, there is no need to maintain a hash-table in the classpect-based implementation to emulate instance-level advising resulting in a significant decrease in code complexity. There are no unnecessary invocations so the implementation does not exhibit performance overheads such as those shown in Figure 15.

The sensor system is now composed naturally with component and connector instances as shown in Figure 18. The main routine constructs component instances and connector instances and connects component instances using connector instances.

The runtime structure of the system shown in Figure 19 (bottom-right) now mirrors the specification of the system, eliminating the conceptual gap between the static program structure and dynamic program structure. The behavioral relationships that were hidden until runtime by the instance-emulation and instance-level weaving emulation code are now explicit in the design and the implementation. In summary, the integration using

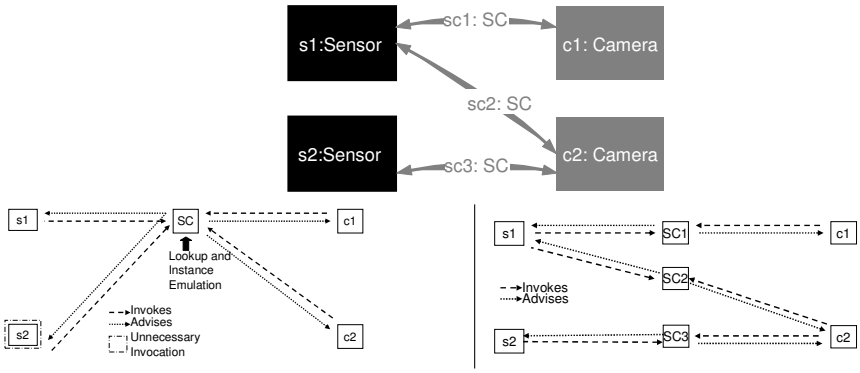


Fig. 19. Runtime Object Structure for Aspect (bottom-left) and Classpect Based (bottom-right) Implementations of the Sensor Camera System (top)

classpects achieves a natural mapping from specification and design to implementation without resorting to unnecessary design complexity and performance overhead.

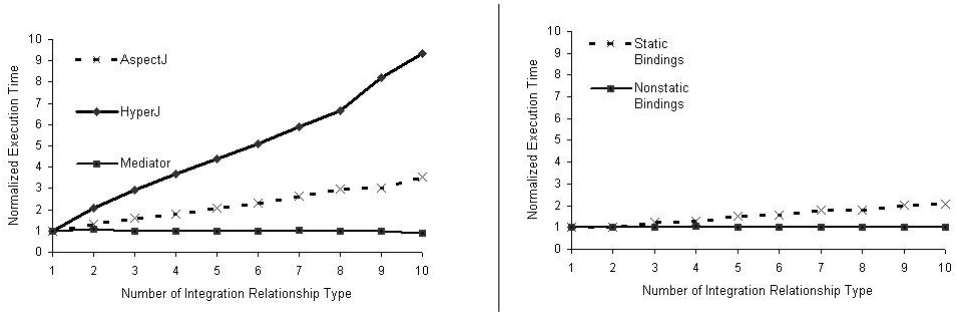


Fig. 20. Increasing overhead with number of integration relationships in the system, **left:** AspectJ, HyperJ, and plain Java implementation of Mediator, **right:** Eos static bindings and Eos instance-level bindings

We measured the performance of selective instance-level bindings using the benchmarks described in Section 4.5. The slight complication in the comparison due to differences in host languages (Java, C#) was resolved by comparing only the normalized execution time as discussed previously. We substituted type-level Eos bindings for AspectJ aspects for this comparison. Figure 20 (right) shows that Eos type-level bindings replicate the degrading performance of AspectJ aspects, while Eos selective instance-level bindings indeed exhibit the constant overhead of mediator-based designs.

Figure 21 shows the side-by-side SeeSoft view of all implementations of the sensor-camera integration concern discussed so far. The first, second, third, and fourth columns show the mediator based implementation, the AspectJ first work-around, the AspectJ second work-around, and the Eos implementation respectively. The first box in all four columns shows the implementation of the sensor concern. The second box shows the implementation of camera concern, and the third box shows the implementation of the sensor-camera integration concern. As can be observed, in the mediator-based style sensor and camera concerns are complicated by the event code. The first work-around in AspectJ

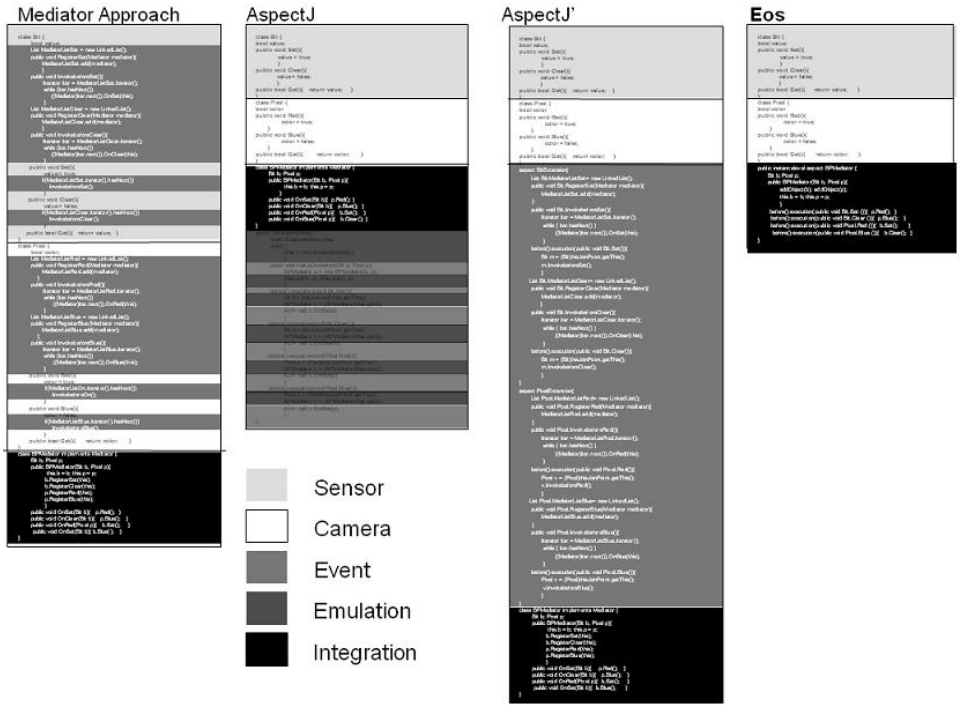


Fig. 21. A Comparative View of Implementations

replaces explicit events with implicit join points, but the additional design complexity to emulate aspect instantiation and selective weaving complicates the integration concern. The second work-around emulates mediator-based design, but without modifying the actual components, bringing back the event code. The Eos version, however, is free of both explicit event-related code and additional design complexity. The length of the columns also suggests that the Eos implementation is shorter compared to other implementations. Eos clearly achieved a significant improvement in the separation of integration concerns.

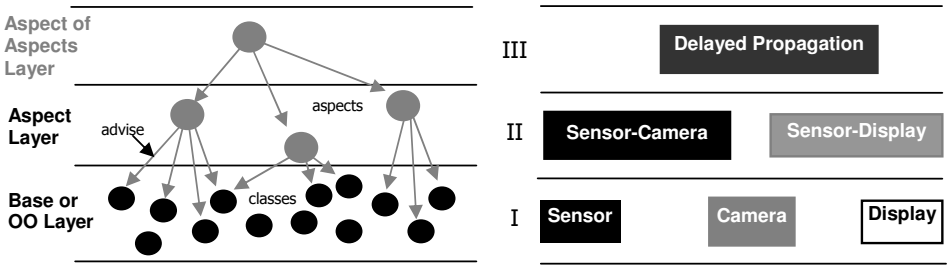
4.8 Summary

In this section, we analyzed the ability of type-based aspects to modularize a broad class of concerns known as integration concerns using a simple but representative example. The comparative analysis of various possible design structures revealed that for improved modularization of this class of concerns, the key requirements for aspects are generalized instantiation and ability to selectively advise instances. In the absence of these features, workarounds are needed that add unnecessary design-time and runtime complexity in design structures. Usefulness of a classpect-based language design over aspect-based language design in improving the modularization of integration concern was manifested in precisely these dimensions. These capabilities of classpects also enable improved modularization of behavioral design patterns [Gamma et al. 1995] as shown by our closely related work [Rajan 2007]. In general, the class of crosscutting concerns where the behavior to be modularized is instance-specific, and/or where a separate copy of the state need

to be maintained as part of the modularized behavior for each participant in the behavior or a combination thereof, is likely to benefit from classpects.

5. SEPARATION OF HIGHER-ORDER CONCERNS

AOP is a relatively new paradigm. The adoptability and interest in the current language models and approaches shows promise. The interest of the developer community makes it important for researchers to investigate the new technology by applying it to new application areas. In the last section, we described separation of integration concerns as a challenge problem for AspectJ-like languages. The experiments revealed an important shortcoming largely due to the commitment to have aspects as separate abstraction mechanism different from classes. The unified model of Eos significantly improved the modularization of integration concerns. This section provides the second data point of the overall evaluation of our approach. It supports the claim that the unified model improves the modularization of higher-order crosscutting concerns.



(a) Layered advising to modularize higher-order concerns (b) Need for layered advising in the example system

Fig. 22. Constructing Hierarchical Systems Using Advising as an Invocation Mechanism

To recap, a concern is a dimension in which a design decision is made and it is crosscutting if its realization using prevailing decomposition techniques leads to scattered and tangled code. A crosscutting concern is *higher-order* if its realization would be scattered and tangled across the implementations of other crosscutting concerns. For example, in Figure 22 (a) the aspect in the top layer modularizes concerns that were scattered and tangled across the second layer of aspects. A simple higher-order concern, `Delayed Propagation` is introduced in the next subsection in the context of the Sensor system discussed in Section 4. The rest of the section analyzes the attempts to modularize this simple *higher-order concern* using prevalent AOP techniques.

5.1 Delayed Propagation Concern

Let us consider an evolution scenario for the sensor system discussed in Section 4. A new type of component `Display`, a new behavioral relationship `sensor-display integration`, and a new higher-order crosscutting concern `Delayed Propagation` is introduced in the system as shown in Figure 22-b. The new behavioral relationship requires sensor and display component instances to behave together such that whenever the sensor instance detects a signal, the display instance is

updated and vice versa. The implementation of this concern is similar to that of sensor-camera integration concern discussed in Section 4.

The requirements for the higher-order concern `Delayed Propagation` dictates that the system should provide means to control the propagation of behavioral relationships related updates. Delayed propagation can be switched on or switched off. When the delayed propagation is turned on, all updates required by the behavioral relationships, to coordinate the control, actions, and states of subsets of system components to satisfy overall system requirements, are cached until delayed propagation is turned off. Turning off delayed propagation causes all cached updates to be flushed to restore the system state and for the subsequent updates to be handled immediately.

This feature is representative of caching feature commonly found in software systems. A common use-case is to avoid updating other views of a model, while the user is editing in one view to avoid screen flickering. For example, in Galileo [Sullivan et al. 1997] that provides a graphical and a textual view of a fault tree, updates of the graphical view could be avoided, when the fault-tree model is being edited in the textual view. The next subsection describes a simple implementation of the delayed propagation concern.

5.2 Simple Realization of the Delayed Propagation Concern

Figure 23 shows a straightforward realization of the delayed propagation concern using the AspectJ language model. The implementation of this concern crosscuts the aspects `SenCamMediatorModule` and `SenDispMediatorModule` that in turn are themselves representations of crosscutting integration concerns. Following three changes are made to each aspect representing an integration concern. First, a Boolean flag `delay` and a mutator/accessor pair are added to each aspect to store and change the state of delayed propagation. Second, each advice is modified to handle the delayed propagation. If the delayed propagation flag is **`false`**, the advice works normally. If it is **`true`**, the advice caches the update and skips the update.

This implementation strategy results in a scattered and tangled implementation of the delayed propagation concern. The integration logic that was well modularized before in advice constructs is now coupled with the delayed propagation concern. The implementation also does not provide a single interface to switch delay propagation on/off; instead, delayed propagation has to be toggled for each mediator. An alternative solution is to extract the common functionality related to the delayed propagation concern as an abstract aspect. The concrete aspects `SenCamMediatorModule` and `SenDispMediatorModule` inherit from this abstract aspect. This solution is an advance over the simple solution in that the common functionality is being reused. However, much of the delayed propagation concern still remains scattered and tangled with the advice constructs in aspects `SenCamMediatorModule` and `SenDispMediatorModule`. AOP aims to solve precisely this problem. The next subsection analyzes how delayed propagation can be modularized using aspects.

5.3 Modularizing The Delayed Propagation Concern

An alternative strategy is to implement delayed propagation as an aspect. This aspect can provide a single interface to turn delayed propagation on and off. It can override the execution of the advice constructs in the `SenCamMediatorModule` and `SenDispMediatorModule` aspects using around advice. The around advice will call

```
aspect SenCamMediatorModule {
 boolean delay;  /*Delay concern begin*/
 int detectCount = 0;
 int clickCount = 0;
 public boolean getDelay(){return delay;}
 public void setDelay(boolean value){
  delay = value;
  if(value==false){
   for(int i=detectCount; i>0; i++)
   // Flush cached detects;
   for(int j=clickCount; j>0; j++)
   //Flush cached clicks;
   }  /*Delay concern end*/
    ...
 after():execution(* Sensor.detect()){
  if(delay) /* Delay concern */
   detectCount++; /*Delay concern*/
  else {
    ...
   }
 }
 after():execution(* Camera.click()){
  if(delay)/* Delay concern*/
   clickCount++;/* Delay concern*/
  else {
    ...
   }
 }
}
```

```
aspect SenDispMediatorModule {
 bool delay;  /*Delay concern end*/
 int detectCount = 0;
 int updateCount = 0;
 public boolean getDelay(){return delay;}
 public void setDelay(boolean value){
  delay = value;
  if(value==false){
   for(int i=detectCount; i>0; i++)
   // Flush cached detects;
   for(int j=updateCount; j>0; j++)
   //Flush cached updates;
   }  /*Delay concern end*/
    ...
 after():execution(* Sensor.detect()){
  if(delay)/* Delay concern */
   detectCount++;/* Delay concern */
  else {
    ...
   }
 }
 after():execution(* Display.update()){
  if(delay)/* Delay concern */
   updateCount++;/* Delay concern */
  else {
    ...
   }
 }
}
```

Fig. 23. Scattered Implementation of the Delayed Propagation Concern (in gray)

proceed if delay is false, so that the updates in the mediators proceed as usual, or else, cache the updates and omit original join point execution.

This solution approach promises to modularize the scattered and tangled delayed concern. It also satisfies the requirement to provide a single interface to turn delayed propagation on and off. This approach solves the problems of the simple solution discussed in previous section. The code for delayed propagation is now a separate, modularized, and reusable aspect. To add or remove this feature from the system, we just need to add or remove the aspect. The component code is now independent of the integration code. Unfortunately, this solution cannot always be easily realized using the AspectJ-like language model due to a key restriction–while aspects can advise classes in many ways, they can advise other aspects only in restricted ways.

In the current model, individual advice bodies are anonymous, therefore pointcut expressions cannot select a subset of them based on their names. The pointcut designator **`adviceexecution`** selects all advice-execution join points in the program. One can narrow down this selection by composing the `adviceexection` pointcut with the **`within`** pointcut. For example, the pointcut expression **`adviceexecution`**`()` `&&` **`within`**`(SenCamMediatorModule)` selects execution of every advice in the **`aspect`** `SenCamMediatorModule` (Figure 14). However, to implement delayed propagation for sensor-camera and sensor-display integration concerns, addressing each advice in the aspect `SenCamMediatorModule` and `SenDispMediatorModule` independently is necessary. Current model does not allow such fine-grained selection over advice bodies.

A workaround discussed in Section 2 was to have advice delegate to corresponding aspect methods and to advise these methods. The need for such a workaround is an evidence

of the limitation of the current language design. Moreover, the workaround itself is unsatisfactory in general. First, it requires either ubiquitous up-front use of the delegation pattern, or–contrary to the central purpose of aspect-orientation–that scattered changes be made to aspect modules whenever any of their advice bodies become subject to advising. Both approaches require source code, which is not always available. Second, delegation is not entirely straightforward. Advice bodies have to be analyzed to determine whether or not they use implicitly declared reflective information, such as **thisJoinPoint** or implicit constructs, e.g. **proceed**.

```
1   void around(): <pointcut> {
2      if(shallProceed)proceed();
3      }
```

```
1   void around():<pointcut> {
2      originalAdviceCodeInMethod();
3      }
4   void originalAdviceCodeInMethod(){
5      if(shallProceed) proceed();
6      }
```

Fig. 24. An Example Around Advice (left), and the Delegation Work-Around Applied to it (right)

Passing all such parameters to the delegate methods incurs additional design-time and runtime costs and the risks of error. The situation is even more complicated in cases of around advice bodies, which execute instead of the original join point and which can call the original join point using proceed. Figure 24 presents an example (left column, lines 1-4): if `shallProceed` is true, the original join point is invoked. Applying the workaround results in the proceed call being moved to a delegatee (right column, lines 4-6). In current languages, **proceed** is not allowed in methods bodies. A closure to the **proceed** expression will thus have to be passed from the advice body to the delegatee, perhaps using the worker object pattern of Laddad [Laddad 2003]. The work-around is both complicated and incurs the need f or scattered changes, undermining the purpose of aspects.

The unified model replaces non-orthogonal and asymmetric aspect and class as well as advice and method by symmetric classpects, bindings, and methods. We claimed that this reorganization of language constructs improves the compositionality of the resulting language model under advising as an invocation mechanism and improves the modularization of higher-order concerns. To validate these claims, the next subsection presents the classpect-based implementation of the delayed propagation concern.

5.4 Delayed Propagation as Classpect

A key advance that the classpect-based implementation of mediators makes over an aspect-based implementation is that the integration logic is now modularized in a named construct, method. Figure 17 shows the representation of integration logic as normal methods. These methods can be selected individually using existing pointcut designators and patterns. The refactoring of an advice construct into binding and method thus solves the problems encountered by the second solution, making the complete modularization of the delayed propagation concern feasible without work-arounds.

Figure 25 shows the delayed propagation concern as a classpect. The `Delay` classpect declares four methods to cache the sensor detects, camera clicks, and display updates. It binds these methods to the execution of methods in the `SenCamMediator` and `SenDispMediator`. For example, the first binding binds the method `cacheDetect`

```
1  class Delay {
2   static void around execution(SenCamMediator.onDetect())
3     && this(sc) && aroundptr(p): cacheDetect(SenCamMediator sc, AroundADP p)
4   public void cacheDetect(SenCamMediator sc, AroundADP p){
5     if(delay) detectCache.enQueue(sc); /* cache detects */
6     else p.innerInvoke(); /* Equivalent to proceed call */
7    }
8   /* Similarly for SenCamMediator.onClick, SenDispMediator.onUpdate
9      SenDispMediator.onDetect there are separate around
10     bindings and around-bound methods. */
11  ...
12  Queue detectCache; /* A FIFO data structure to model cache */
13  bool delay;
14  public bool Get(){return delay;}
15  public void Set(bool value){
16    delay = value;
17    if(value==false)
18      /* Propagate updates */
19      while(detectCache.next())((SenCamMediator)(detectCache.current())).onDetect();
20      /* Similarly for other cached updates */
21    }
22  }
```

Fig. 25. Delayed Propagation Concern as a Classpect

to execute around the method `SenCamMediator.onDetect`. The effect of this binding is that whenever `SenCamMediator.onDetect` is called, instead of executing its body, control is transferred to the method `cacheDetect`. The method `cacheDetect` either caches the call or allows it to proceed as usual. Caching in this simple case, only keeps track of the `SenCamMediator` instance in a FIFO queue. When the delayed propagation is turned off each instance is retrieved from the queue and the method `onDetect` is called on the mediator instance. This solution can be further optimized by only applying the updates that do not cancel each other but for simplicity we omit that.

Note that the method `SenCamMediator.onDetect` is called whenever a `Sensor` instance detects a signal, to propagate the change to the `Camera` instance. So caching its execution is equivalent to caching the propagation of the sensor-camera integration concern and replaying its execution is equivalent to flushing all updates, precisely the requirement of the delayed propagation concern. This solution thus shows that classpects were able to improve the modularization of the higher-order concern without the need for workarounds. Moreover, the classpect `Delay` remains amenable to be advised by another layer of classpects, demonstrating the full compositionality of the unified language model.

5.5 Summary

In this section, we validated the claim that the unified model improves the modularization of higher-order concerns. We demonstrated and compared the implementation of a simple but representative concern using aspects and classpects. The aspect based implementation required crosscutting use of delegation pattern to expose the right set of join points. The need for such work-around also demonstrated that in the current language model aspect-aspect compositionality is restrictive. The classpect-based implementation was able to modularize the delayed propagation concern completely, showing improvement in the modularization of higher-order concerns and the compositionality of the language model. Getting rid of aspect and advice as a separate abstraction mechanism and including bindings thus opens up new architectural possibilities using advising as an in-

vocation mechanism. Widespread adoption of AOP has just begun, and we are starting to see a number of crosscutting concerns. With the continued adoption, it is likely that more higher-order crosscutting concerns will be discovered that will benefit from the new architectural possibilities that classpects enable.

6. RELATED WORK

The description of related ideas is categorized along two main themes, unification of aspects and classes and techniques that allow instances of be advised selectively.

6.1 Unification of Aspects and Classes

AspectJ [Kiczales et al. 2001], AspectWerkz [Bonér 2004], and Caesar [Mezini and Ostermann 2003] are all related to our work. In at least one early version of AspectJ, aspect was not a separate construct. Rather, the class was extended to support advice. No evidence indicates, however, that those early designs achieved the synthesis of OO and AO techniques of Eos. Advice bodies and methods were still separate; it is unclear to what extent advice could be advised at all; and there was no support for flexible aspect instantiation.

AspectWerkz [Bonér 2004] is the design most closely related to our work. The aim of this project was to provide the expressiveness of AspectJ [Kiczales et al. 2001] without sacrificing pure Java and the supporting tool infrastructure. The solution is to use normal Java classes to represent both classes and AspectJ-like aspects, with advice represented in normal methods, and to separate all join-point-advice bindings either into annotations in the form of comments, or into separate XML binding files. AspectWerkz provides a proven solution to the problem of AspectJ-like programming in pure Java, but it does not achieve the unification that we have pursued. Spring Framework is similar [Johnson and et al. 2007]. JBoss AOP framework [Khan et al. 2007] is also similar, except that it also provides capabilities of aspect instantiation.

First, and crucially, these approaches do not support the concept of aspects as objects under program control; rather they are really an implementation of the AspectJ model. Instead, the use of Java classes as aspects is highly constrained so that the runtime system can maintain control. For example, in AspectWerkz, a class representing an aspect must have either no constructor or one with one of two predefined signatures, and a method representing an advice body has one argument of type JoinPoint. AspectWerkz uses this interface to manage aspect creation and advice invocation. AspectWerkz also lacks a single-language design, in that it uses both Java and XML binding files. Third, AspectWerkz lacks static type checking of advice parameters. Rather, reflective information is explicitly marshaled in advice methods.

The design of Caesar [Mezini and Ostermann 2003] is also closely related to our approach. The aim of Caesar was to decouple aspect implementation and the aspect binding with a new feature called an aspect collaboration interface (ACI). By separating these concepts from aspect abstraction, Caesar enables reuse and componentization of aspects. This approach is similar to ours and to AspectWerkz in that it uses plain Java to represent both classes and aspects; however, it represents advice using AspectJ like syntax. Methods and advices are still separate constructs, and advice constructs couples crosscut specifications with advice bodies. Consequently, as in AspectJ, advice bodies are still not addressable as individual entities. They can be advised as a group using an advice-execution pointcut. In Caesar, as in Eos, advice can be bound statically or dynamically; however, aspects in Caesar cannot directly advise individual objects on a selective basis. Previously, we showed

that both first-class aspect instances and instance-level advising are essential for expressing integration concerns as aspects [Rajan and Sullivan 2003b; Sullivan et al. 2002].

HyperJ [Tarr et al. 1999; Ossher and Tarr 1999] has one unit of modularity, classes, with a separate notation for expressing bindings. However, they do not support program control over aspects as first-class objects, and to date the join point models that they have implemented have been limited mainly to methods [Harrison et al. 2003]. Object team [Herrmann 2003], an approach for collaboration-based designs, is related in that they also provide a separate dimension for decomposing crosscutting functionalities. Object teams are instatiable collection of roles, unlike hyperslices in HyperJ and aspects in AspectJ, and similar to classpects. Compared to the unification proposed by our approach, object teams are separate and distinct entities from classes. Moreover, their join point model is also limited mainly to methods as pointed out by the author himself [Herrmann 2003].

Tucker and Krisnamurthi [Tucker and Krishnamurthi 2003] consider making both pointcuts and advice first-class entities in the functional language context. Their approach builds upon a base language that already provides functions as first-class entities expressing pointcuts and advices as a function easy. On the other hand, in languages such as Java and C# functions are not first-class entities making the design of aspect languages difficult.

Suvée et al. [Suvée et al. 2003] in their work *JAsCo* also raise the question of dynamic deployment of aspects and propose a solution based on a new concept that they call *aspect beans*. Their aspect beans utilize an enhanced component model that provides mechanisms for dynamic component adaptation model that is clearly superior to AspectJ-like languages. In essence, interesting events in the components are exposed so that aspects can be deployed against them. JAsCo also allows separate pointcut and advice specifications grouped inside the *hook* mechanisms. However, JAsCo still maintains the conceptual separation between a base and an aspect model and does not address the issues of selective adaptation of component instances.

Finally, our previous work on unified AO model [Rajan and Sullivan 2005a] and instance-level aspects [Rajan and Sullivan 2003b; 2003a] forms the basis of this work. This work significantly expands on the previous work describing the performance tradeoff, interesting compilation techniques, etc. The notion of association aspects developed by Sakurai et al. [Sakurai et al. 2004] is also related. Association aspects addressed the limitation of instance-level aspects that they always select by the target object; however, association aspects also maintain the conceptual distinction between aspects and classes and therefore share the difficulties of modularizing higher-order concerns with AspectJ.

6.2 Affecting Behaviors of Instances Selectively

The idea to affect behaviors of instances selectively is not new. It has appeared in many forms including, the observer-pattern based design style in which observers selectively registers with objects to observe and affect their behavior. Many AO approaches, e.g. AspectS [Hirschfeld 2003], Composition Filters [Aksit et al. 1994], AMF [Constantinides and Elrad 2001], OIF [Filman et al. 2005], and EAOP [Douence and Südholt 2002], that take a wrapper-based approach, in which messages sent to and from components are intercepted for processing by aspect wrapper objects, also have the ability to selectively affect behaviors of instances. The wrapper-based approach has appeared in many forms over the decades: from Common Lisp, to tool integration frameworks. It has been picked up and given an AOSD interpretation by efforts such as Sina/st [Koopmans 1995] and ility-insertion [Filman et al. 2005].

Many such languages are called AO; however, by taking the wrapper-based approaches they generally give up on the richness of the join point model and limit themselves to what has been called operational-level composition [Harrison et al. 2003]. They provide a limited or non-existent pointcut language. Other AspectJ-like languages, such as AspectC++ [Gal et al. 2001] and AspectR, on the other hand, do not support first-class aspect instances and instance-level advising. Languages such as HyperJ [Tarr et al. 1999] and DJ [Marshall et al. 1999], in which the pointcut concept does not apply, also lack the instance-level capabilities.

Table I. A Characterization of a Subset of AO Languages and Approaches

Abbreviations for the table
IL: Instance-level aspect weaving
FI: First-class Aspect Instances
CL: Call, execution and return join point
FS/FG: Field set and get join points
E: Exception handling join points
OI: Object initialization join points
M: Messages as join points
D: Dynamic deployment

Language	IL	FI	CL	FS/FG	E	OI	M	D
AspectJ [Kiczales et al. 2001]			X	X	X	X		
HyperJ [Tarr and Ossher 2000]			X					
EAOP [Douence and Südholt 2002]	X	X	X					X
Composition filter model [Bergmans and Akşit 2005]	X	X	X				X	X
Aspect Moderator Framework [Constantinides and Elrad 2001]	X	X	X				X	
Object Infrastructure Framework [Filman et al. 2005]	X	X	X				X	X
DJ [Marshall et al. 1999]			X					
AspectC# [Kim 2002]			X					
Claw			X					
AOP#			X					
AspectR [Bryant and Feldt 2002]			X	X	X	X		
AspectC++ [Gal et al. 2001]			X	X				
AspectS [Hirschfeld 2003]	X	X	X				X	
AspectWerkz [Bonér 2004]			X	X	X			
JBoss AOP [Khan et al. 2007]	X		X	X		X		X
Spring Framework [Johnson and et al. 2007]			X					X
Caesar [Mezini and Ostermann 2003]			X					X
JAsCo [Suvée et al. 2003]		X	X			X		X
CARMA [Gybels and Brichau 2003]	X		X	X		X		X
Eos [Rajan and Sullivan 2003a]	X	X	X	X	X	X		X

As part of this research, a survey of a range of AOP languages was conducted. The survey showed that none except Eos support first-class aspect instances and generalized weaving model. The required language should have support for the following:

—*Rich join point model*: the approach defines join points well beyond function call and return. Important join points include field get and set, exception and message passing, and object creation and deletion.

—*Expressive pointcut language*: the approach permits declarative expression of join points to be advised by an aspect. The ability to *quantify* [Filman and Friedman 2004] join

points across a program is a key enabler of modular representation of far-reaching cross-cutting concerns, such as call tracing.

—*Instance-level weaving*: the approach allows advice to be woven with selected object instances. Instance level weaving is different from runtime aspect weaving. The former allows the developers to say, *Weave this aspect to only these instances of that class*, whereas the later allows them to say either explicitly or implicitly *When the control reaches this point of execution weave this aspect and when it reaches that point unweave it*. An implementation of instance level aspect weaver may provide additional abilities to dynamically attach and detach aspects from base object at runtime or the ability to instantiate an aspect but by definition it is not required to do so.

—*First-class aspect instances*: First-class objects of given aspect types are typically used to maintain separate aspect state for separately advised objects. The ability to create an instance of an aspect using a general-purpose mechanism is available.

Table I presents the state of the art in aspect language design as reflected in a broad survey of extant aspect languages. Rows name languages; columns, features. For languages such as HyperJ, the table reflects current implementation status, rather than hypothesized capabilities. The compiler for other languages, such as Sina/St is not available for experimentation. In these cases, the table reflects the language designs as described in published works. The detailed survey of aspect-languages in Table I show that the combination of features is indeed non-existent in current AOP languages and approaches.

AspectJ supported instance-level advising, but without first-class aspect instances through version 0.5. The AspectJ mailing list records discussions on the need for `addObject`, with several use cases presented by users. The current form provides constructs like per this and per target for associating aspect-instances and object-instances. These constructs are inadequate. First, per this and per target aspect instances are not under program control but are associated automatically with all instances of advised types, and each instance is associated with just one other object. AspectJ, to the best of author's knowledge, never supported both instance-level advising and first-class aspect instances. This is the combination needed for an adequate generalization to the instance level.

7. CONCLUSION

The main contribution of this work is a demonstration that it is both possible and useful to eliminate the conceptual distinction between classes and aspects in AspectJ-like languages and programming design methods, in favor of a single generalized module construct supporting both class-like and aspect-like composition. Our unification is based on a reasonably elegant new module construct that we have called the *classpect.*

This unification is not just a matter of programming mechanism design, however. It also suggests that we can do without the two-level class/aspect ontology that decomposes system design in two separate layers and phases: an object-oriented base subject to transformation by subsequently developed aspect modules. This thus also work provides a way to reconcile the differences between asymmetric and symmetric language models. In particular, our approach has restored symmetry to the domain of AspectJ-like languages without fundamentally altering the language model in the way that, for example, HyperJ does.

To test the potential practicality and utility of our ideas, we have developed, implemented and described a substantial language supporting classpects, a compiler that imple-

ments the language, and a number of design experiments that support the claims that we have made for the unification we have proposed. In particular, we showed that separate aspect and advice are not essential; instead, quantified binding is at the core of AOP.

Several questions remain for future work. Is there merit in exploring language designs in which bindings, pointcut descriptors, and advices are first-class objects? Does the unification we report on here provide a better starting point for developing a better semantics of aspect languages? How might a classpect-based language be extended to model crosscut programming interfaces (XPIs) explicitly [Sullivan et al. 2005]? We believe that this XPI question merits particular attention, and that the Eos language and compiler provide a solid launching pad for the research needed to answer the question.

8. ACKNOWLEDGEMENTS

This work was supported in part by NSF grants ITR-0086003, FCA-0429947, CNS-0627354, and CNS-0709217. The comments from Jack Davidson, David Evans, William Griswold, John Knight, Mary Lou Soffa, and anonymous referees were very helpful.

REFERENCES

AKSIT, M., WAKITA, K., BOSCH, J., BERGMANS, L., AND YONEZAWA, A. 1994. Abstracting Object Interactions Using Composition Filters. In *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, R. Guerraoui, O. Nierstrasz, and M. Riveill, Eds. Vol. 791. Springer-Verlag, London,UK, 152–184.

AOTANI, T. AND MASUHARA, H. 2007. SCoPE: an AspectJ compiler for supporting user-defined analysis-based pointcuts. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*. ACM Press, New York, NY, USA, 161–172.

BERGMANS, L. AND AKŞIT, M. 2005. Principles and design rationale of composition filters. Addison-Wesley, Boston, 63–95.

BONÉR, J. 2004. What are the key issues for commercial AOP use: how does AspectWerkz address them? In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*. ACM Press, New York, NY, USA, 5–6.

BROOKS, F. P. 1995. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*, Second ed. Addison Wesley, Reading, Mass.

BRYANT, A. AND FELDT, R. 2002. AspectR - simple aspect-oriented programming in Ruby.

COLYER, A. AND CLEMENT, A. 2004. Large-scale AOSD for middleware. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*. ACM Press, New York, NY, USA, 56–65.

CONSTANTINIDES, C. A. AND ELRAD, T. 2001. Composing concerns with a framework approach. In *Proc. Int'l Workshop on Distributed Dynamic Multiservice Architectures (ICDCS-2001), Vol. 2*, Z. Choukair, Ed. IEEE Computer Society, Washington, DC, 133–140.

DIJKSTRA, E. W. 1968. Go to statement considered harmful. *Communications of the ACM 11,* 3 (Mar.), 147–148.

DOUENCE, R. AND SÜDHOLT, M. 2002. A model and a tool for event-based aspect-oriented programming (EAOP). Tech. Rep. 02/11/INFO, Ecole des Mines de Nantes.

DYER, R. AND RAJAN, H. 2008. Nu: a dynamic aspect-oriented intermediate language model and virtual machine for flexible runtime adaptation. In *AOSD '08: Proceedings of the 7th International Conference on Aspect-oriented Software Development*. ACM, New York, NY, USA.

FILMAN, R. E., BARRETT, S., LEE, D. D., AND LINDEN, T. 2005. Inserting ilities by controlling communications. Addison-Wesley, Boston, 283–295.

FILMAN, R. E. AND FRIEDMAN, D. P. 2000. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*.

FILMAN, R. E. AND FRIEDMAN, D. P. 2004. Aspect-oriented programming is quantification and obliviousness. In *Aspect-oriented Software Development*. Addison-Wesley Professional, 21–35.

GAL, A., SCHRÖDER-PREIKSCHAT, W., AND SPINCZYK, O. 2001. AspectC++: Language proposal and prototype implementation. In *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (OOPSLA 2001)*, K. De Volder, M. Glandrup, S. Clarke, and R. Filman, Eds.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

GARLAN, D. AND NOTKIN, D. 1991. Formalizing design spaces: Implicit invocation mechanisms. In *VDM '91: Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development-Volume I*. Springer-Verlag, London, UK, 31–44.

GARLAN, D. AND SHAW, M. 1993. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, V. Ambriola and G. Tortora, Eds. Vol. 1. World Scientific Publishing Company, 1–40.

GOSLING, J., JOY, B., AND STEELE, G. L. 1996. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

GYBELS, K. AND BRICHAU, J. 2003. Arranging language features for more robust pattern-based crosscuts. In *Second International Conference on Aspect-oriented Software Development (AOSD)*.

HARRISON, W., OSSHER, H., AND TARR, P. 2003. Asymmetrically vs. symmetrically organized paradigms for software composition. In *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, L. Bergmans, J. Brichau, P. Tarr, and E. Ernst, Eds.

HERRMANN, S. 2003. Object teams: Improving modularity for crosscutting collaborations. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*. Springer-Verlag, London, UK, 248–264.

HIRSCHFELD, R. 2003. AspectS - Aspect-Oriented Programming with Squeak. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*. Springer-Verlag, London, UK, 216–232.

JOHNSON, R. AND ET AL. 2007. The spring framework - reference documentation 2.03.

KHAN, K., BURKE, B., RAINONE, F., PEDERSEN, S., FLEURY, M., BROCK, A., HUSSENET, C., AND CULPEPPER, M. 2007. JBoss AOP reference documentation.

KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. 2001. An overview of AspectJ. In *ECOOP 2001 — Object-Oriented Programming 15th European Conference*, J. L. Knudsen, Ed. Lecture Notes in Computer Science, vol. 2072. Springer-Verlag, Budapest, Hungary, 327–353.

KICZALES, G., LAMPING, J., LOPES, C. V., MAEDA, C., MENDHEKAR, A., AND MURPHY, G. 1997. Open implementation design guidelines. In *Proceedings of the 19th International Conference on Software Engineering*. IEEE, Boston, Massachusetts, 481–90.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C. V., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *ECOOP'97: Proceedings of the European Conference on Object-Oriented Programming*. Springer-Verlag, 220–42.

KIM, H. 2002. AspectC#: An aosd implementation for c#. Tech. Rep. TCD-CS-2002-55, Department of Computer Science, Trinity College, Dublin.

KOOPMANS, P. 1995. Sina user's guide and reference manual. Tech. rep., Dept. of Computer Science, University of Twente.

LADDAD, R. 2003. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning.

LISKOV, B. AND ZILLES, S. 1974. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*. 50–59.

MACLENNAN, B. J. 1986. *Principles of programming languages: design, evaluation, and implementation (2nd ed.)*. Holt, Rinehart & Winston, Austin, TX, USA.

MARSHALL, J., ORLEANS, D., AND LIEBERHERR, K. J. 1999. DJ: Dynamic structure-shy traversal in pure Java. Tech. rep., Northeastern University. May.

MEZINI, M. AND OSTERMANN, K. 2003. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*. ACM Press, New York, NY, USA, 90–99.

OSSHER, H. AND TARR, P. 1999. Multi-dimensional separation of concerns using hyperspaces. IBM Research Report 21452, IBM. Apr.

RAJAN, H. 2005. Unifying aspect- and object-oriented program design. Ph.D. thesis, The University of Virginia, Charlottesville, Virginia.

RAJAN, H. 2007. Design patterns in Eos. In *PLoP '07, Conference on Pattern Languages of Programs*.

RAJAN, H. AND SULLIVAN, K. 2003a. Eos: instance-level aspects for integrated system design. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM Press, New York, NY, USA, 297–306.

RAJAN, H. AND SULLIVAN, K. 2003b. Need for instance level aspect language with rich pointcut language. In *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, L. Bergmans, J. Brichau, P. Tarr, and E. Ernst, Eds.

RAJAN, H. AND SULLIVAN, K. 2005a. Aspect language features for concern coverage profiling. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*. ACM Press, New York, NY, USA, 181–191.

RAJAN, H. AND SULLIVAN, K. J. 2005b. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*. ACM Press, New York, NY, USA, 59–68.

SABBAH, D. 2004. Aspects: from promise to reality. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*. ACM Press, New York, NY, USA, 1–2.

SAKURAI, K., MASUHARA, H., UBAYASHI, N., MATSUURA, S., AND KOMIYA, S. 2004. Association aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*. ACM Press, New York, NY, USA, 16–25.

SPINCZYK, O., GAL, A., AND SCHROEDER-PREIKSCHAT, W. 2002. AspectC++: an aspect-oriented extension to the c++ programming language. In *CRPITS '02: Proceedings of the Fortieth International Confernece on Tools Pacific*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 53–60.

SULLIVAN, K., GU, L., AND CAI, Y. 2002. Non-modularity in aspect-oriented languages: integration as a crosscutting concern for aspectj. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*. ACM Press, New York, NY, USA, 19–26.

SULLIVAN, K. J., DUGAN, J. B., KNIGHT, J., ET AL. 1997. Galileo: An advanced fault tree analysis tool.

SULLIVAN, K. J., GRISWOLD, W. G., SONG, Y., CAI, Y., SHONLE, M., TEWARI, N., AND RAJAN, H. 2005. Information hiding interfaces for aspect-oriented design. In *The Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*. 166–175.

SULLIVAN, K. J. AND NOTKIN, D. 1990. Reconciling environment integration and component independence. *SIGSOFT Software Engineering Notes 15,* 6 (Dec.), 22–33.

SULLIVAN, K. J. AND NOTKIN, D. 1992. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology 1,* 3 (July), 229–68.

SUVÉE, D., VANDERPERREN, W., AND JONCKERS, V. 2003. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*. ACM Press, New York, NY, USA, 21–29.

TARR, P. AND OSSHER, H. 2000. Hyper/J user and installation manual. Tech. rep., IBM T. J. Watson Research Center.

TARR, P., OSSHER, H., HARRISON, W., AND STANLEY M. SUTTON, J. 1999. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*. IEEE Computer Society Press, Los Alamitos, CA, USA, 107–119.

TUCKER, D. B. AND KRISHNAMURTHI, S. 2003. Pointcuts and advice in higher-order languages. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*. ACM Press, 158–167.