Classpects: Unifying Aspect- and Object-Oriented Language Design

Hridesh Rajan
Department of Computer Science, University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, Virginia 22904-4740, USA
+1 434 982 2296
hr2j@cs.virginia.edu

Kevin J. Sullivan
Department of Computer Science, University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, Virginia 22904-4740, USA
+1 434 982 2206
sullivan@cs.virginia.edu

ABSTRACT
The contribution of this work is the design, implementation, and early evaluation of a programming language that unifies classes and aspects. We call our new module construct the *classpect*. We make three basic claims. First, we can realize a unified design without significantly compromising the expressiveness of current aspect languages. Second, such a design improves the conceptual integrity of the programming model. Third, it significantly improves the compositionality of aspect modules, expanding the program design space from the two-layered model of AspectJ-like languages to include hierarchical structures. To support these claims, we present the design and implementation of Eos-U, an AspectJ-like language based on C# that supports classpects as the basic unit of modularity. We show that Eos-U supports layered designs in which classpects separate integration concerns flexibly at multiple levels of composition. The underpinnings of our design include support for aspect instantiation under program control, instance-level advising, advising as a *general* alternative to object-oriented method invocation and overriding, and the provision of a separate join-point-method binding construct.

Categories and Subject Descriptors
D.1.5 **[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 **[Language Constructs and Features]**: Classes and objects; Modules, packages

General Terms
Design, Human Factors, Languages.

Keywords
Aspect-oriented, classpect, join point-method binding.

1. INTRODUCTION
Aspect-oriented programming languages and methods have begun to attract significant attention from industry and from the research community. The most visible languages to date are AspectJ and related languages [3][4][8][12][15][19]. Early in the design of AspectJ, tradeoffs were made against generality, orthogonality, and other such design principles, in favor of adoptability, which, among other things, was seen as a key to a convincing empirical evaluation of the chief concepts of aspect-oriented design.

Although the jury is still out, aspect-oriented views of modularity, the new and in some cases radical capabilities of aspect languages, and trends toward industrial adoption combine to warrant research on the design, implications, and uses of such languages and methods. In this paper, we reexamine one of the most fundamental decisions made early in the design of AspectJ: to support separate but closely related *class* and *aspect* module constructs. We also study the commitment that this decision entailed to a two-level program design style, with systems organized as object-oriented *base* layers advised by superimposed *aspects*. Kiczales reports that the decision was based requests from users, who wanted to be able to easily see and control uses of the new mechanisms [13].

Our motivation rests on two observations. First, separating classes and aspects reduces the conceptual integrity of the programming model [6], arguably making it harder in the long run for programmers to understand and use aspect-oriented programming. Second, the asymmetry of classes and aspects complicates system composition and ultimately harms modularity. Asymmetries occur in two areas. First, while aspects can advise classes, classes cannot advise aspects, and aspects cannot advise aspects as flexibly as they can advise classes. Second, aspects instances cannot be created or manipulated under program control in the same ways as class-based objects [24][27]. In practice, these asymmetries constrain designers to the two-layer architectural style that we mentioned. Hierarchical layering of aspects is difficult at best.

In this work, we present a model that unifies the capabilities of classes and aspects in a single, more expressive construct that we call the *classpect*. We claim and show this unification is possible without reducing the expressiveness of AspectJ-like languages; that it improves the conceptual integrity of the design model; that it creates valuable new possibilities for program design, with hierarchical aspect composition as a practical new possibility; and that it ultimately enables the modularization of what we call *higher-order* crosscutting concerns.

To evaluate the feasibility of our ideas and to support evaluation, we designed and implemented a *classpect-oriented* language called *Eos-U*. Eos-U extends C#, has the aspect capabilities of *AspectJ,* and unifies classes and aspects. Our compiler handles existing C# programs and supports classpects with an enhanced notion of *class*. There is no longer a separate *aspect* construct.

The rest of the paper is as follows. Section 2 describes the AspectJ model and our requirements for a unified model. Section 3 present Eos-U. Sections 4–8 discuss separation of integration concerns and support our claims for the improved compositionality of Eos-U. Section 9 assesses the nature and potential importance of our results. Second 10 discusses related work. Section 11 concludes.

2. BACKGROUND AND MOTIVATION

To make this work self-contained, we briefly review the AspectJ model. The central idea is that aspects are class-like constructs that enable the modular representation of crosscutting concerns. A concern is a dimension in which a design decision is made [17], and it is crosscutting if its realization in traditional object-oriented designs leads to scattered and tangled code. Scattered means not local to a module but fragmented across a system. Tangled means intermingled with code for other concerns [14].

Aspect languages add five key constructs to the object-oriented model: join points, pointcuts, advice, inter-type declarations, and aspects.[1] We provide a simple example.

```
1 aspect Tracing {
2   pointcut tracedCall():
3       call(* *(..));
4   before(): tracedCall() {
5       /* Advice: Trace before each call matched by tracedCall */
6   }
7 }
```

A join point is a point in program execution exposed by the semantics of the language to possible modification by aspects. The execution of a method is an example of a join point. A pointcut (lines 2-3) is a predicate that selects a subset of join points for modification: here, any call to any method. An advice (lines 4-6) serves as a *before, after,* or *around* method to effect such an extension at each join point selected by a pointcut. An aspect (lines 1-7) is a class-like module that uses these constructs to modify behaviors defined by the classes of a software system.

Aspects also support the data abstraction and inheritance abilities of classes, but they do differ from classes. First, aspects can use pointcuts, advice, and inter-type declarations. In this sense, they are strictly more expressive than classes. Second, instantiation of aspects and binding of advice to join points are wholly controlled by the Aspect language runtime. There is no *new* for aspects. Aspect instances are thus not first-class, and, in this dimension, classes are strictly more expressive than aspects. Third, although aspects can advise methods with fine selectivity, they can select advice bodies to advise only in coarse-grained ways.

In earlier work [24][27], we addressed the limits of aspects with respect to instantiation and join point binding under program control, but we left aspects and classes separate and incomparable, and the resulting compositionality problems unresolved. We now tackle this problem, leading to a design in which advising emerges as a general alternative to overriding or method invocation.

We see two basic requirements for a unified, more compositional model. First, a new model should preserve the expressive power of AspectJ. This constraint rules out the use of languages with much more limited join point and pointcut models. Second, a unified design should be based on a single, first-class, class-like module construct, and a single method construct for procedural abstraction (whereas AspectJ has both methods and advice).

3. EOS-U: THE UNIFIED DESIGN

Eos-U provides a proof of concept. Eos-U is a *classpect*-oriented version of Eos [23], which is itself an AspectJ-like extension of C# [20][21]. Eos fully supports instance-level aspects, which means that it provides first-class aspect instances (*new*) and instance-level advice binding under program control. The rest of this section presents the Eos-U language design in more detail.

3.1 Unifying Classes and Aspects

Eos-U unifies aspects and objects in three ways. First, it unifies aspects and classes. A *class* in Eos-U supports the full *classpect* notion: all C# class constructs, all of the essential capabilities of AspectJ aspects, and the Eos extensions to aspects needed to make them first-class objects. Second, Eos-U eliminates advice in favor of using methods only, with a separate and explicit join-point-method binding construct. Third, Eos-U supports a generalized advising model. To the usual object-oriented mechanisms of explicit or implicit method call and overriding based on inheritance we add implicit invocation using *before* and *after* advice, and overriding using *around* advice, respectively.

3.2 Crosscut Specification

Eliminating anonymous advice in favor of named methods led us to make *join-point-method* bindings separate and abstract, in a style similar to the *event-method* binding constructs of implicit invocation systems [9][26][28]. Eos-U separates what we call *crosscut specifications* from advice bodies (now just methods). A crosscut specification 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, *binding_declaration* (Figure 1) presents our *crosscut specification* construct. A *binding_declaration* has four parts. The first, *opt_static,* specifies whether a binding is static. Non-static bindings result in instance-level advising [23]: selective advising of the join points of individual object instances. Static bindings 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, *pointcuts*, selects the join points at which an advising method executes. The final part specifies the advising method.

[1] Eos-U supports inter-type declarations, also called *introductions,* but they are not essential to or discussed further in this paper.

```
binding_declaration
    : opt_static after pointcuts : call method_bindings;
    | opt_static before pointcuts : call method_bindings;
    | opt_static type around pointcuts : call method_bindings;
    ;
method_bindings
    : method_binding , method binding
    | method_binding
    ;
method_binding
    : IDENTIFIER(opt_formal_parameter_list)
    ;
opt_static
    : Empty
    | static
    ;
```

Figure 1. Syntax of the binding declaration

```
 1  Eos/AspectJ:
 2  int around():execution(public int *.Bar()){
 3      /* Foo code */
 4  }
 5
 6  Eos-U:
 7  int Foo(){
 8      /* Foo code */
 9  }
10 static int around execution(public int *.Bar()): call Foo();
```

Figure 2: An advice and equivalent binding

```
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): call Cache(Eos.Runtime.AroundADP d);
```

Figure 3. A Method Bound Around

A binding provides a list of methods to execute at a join point, in the order specified. 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. As with *around* advice in AspectJ, an Eos-U method bound *around* is allowed to return a value, so *around* bindings must be declared with return types.

Methods subject to binding have to follow certain rules. First, a method must be accessible in the class declaring a binding. Second, a method bound *before* or *after* a join point can have only *void* as a return type. Third, a method bound *around* a join point must have a return type that matches the return type at the join point. For example, if method *Foo* is bound *around* the execution join point *execution(public int *.Bar())*, then it must return *int*.

The listing in Figure 2 presents an advice construct as it would appear in current aspect languages and the equivalent method binding in Eos-U. The advice (lines 2-4) executes around the join point *execution(public int *.Bar())*. The binding separates the advice body (lines 3-4) from the crosscut specification (line 2). The advice body becomes the body of the method *Foo* (lines 7-9). The crosscut specification becomes part of the binding (line 10).

3.3 Around Bindings

Around advice in AspectJ is executed *instead* of a join point, and can invoke the join point using *proceed*. In essence, the around method overrides the join point method, with calls to *proceed* being analogous to delegating calls to *super*. In Eos-U a method bound *around* is also executed instead of the join point. If the method might need to call the overridden method, the first method takes an argument of type *Eos.Runtime.AroundADP*. This class 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 3).

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 *InnerInvoke* on *d* (line 3).

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. Making inner join point invocations explicit in an object-oriented style eliminates this special case. The Eos-U *InnerInvoke* method removes any such special cases from the language design.

A current limitation of our language implementation is that the return type of the *InnerInvoke* method is *object,* precluding static type checking. Method return values could be statically typed to be the same as the surrounding around method using generics. We plan to address this issue as soon as .NET supports generics.[2]

3.4 Additional Power of Overriding

In AspectJ-like languages, there are two different ways to *override* a method: by object-oriented inheritance and by aspect-oriented *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.

It might appear that this redundancy compromises the conceptual integrity of our design. The key insight is that these mechanisms differ fundamentally, and in a way consistent with the nature of aspect-oriented programming: not in their effect on runtime behavior, but rather on the design structure.

Consider two analogies. In object-oriented systems that support implicit invocation [9], 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

[2] As of this writing, Eos-U is build upon .NET Framework 1.1. Generics are not supported in version 1.1. Full support for generics is expected in .NET Framework 2.0.

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; aspect-oriented *around* advising does not.

3.5 New Pointcut Designators

To pass reflective information at a join point to a bound method, a binding uses AspectJ-like pointcut designators such as *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-U removes this special case by binding method arguments to the required reflective information in the *crosscut specification* 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-U 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 enhance readability by eliminating implicit arguments to advice.

Eos-U fulfills the requirements we laid out for a unified model. There is one unit of modularity, *class*, and one mechanism for procedural abstraction, *method*. All of the essential expressiveness of AspectJ-like languages is present in Eos-U, along with the extensions needed for aspects 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-U. Eos-U thus does appear to achieve a novel unity of design.

4. INTEGRATION CONCERNS

The rest of this paper is concerned with supporting our claim that the enhanced compositionality of classpects creates valuable new possibilities for aspect-oriented program design. Our evaluation rests on a comparative analysis of the abilities of AspectJ-like languages and Eos-U to achieve a clean *separation of integration concerns* in hierarchical or layered designs. We show that Eos-U is significantly more expressive in this context.

By an *integration concern* we mean a requirement for the co-ordination of the behaviors of a given subset of components in a system. In general, different and potentially overlapping subsets of objects are subject to different integration concerns. Sullivan and Notkin [22][24][28] showed that integration is crosscutting in the sense that integration code is generally scattered across the classes of the objects to be integrated. They also showed that integration concerns can be separated out as *mediator* classes that use explicit and implicit invocation in a particular way.

The idea is to represent each *kind* of integration concern as a corresponding mediator class, and to represent each *instance* of such a concern, for a given subset of objects, as an instance of the mediator class. Each mediator object observes events announced by the objects it integrates, modifies its state, and calls their methods to coordinate their states and behaviors. Objects can thus be integrated without their classes being either coupled or encapsulated; and the code for each integration concern is separately modularized.

Sullivan and Notkin went on to show that complexly integrated but still evolvable systems, such as integrated radiation treatment planning systems, could be composed from decoupled objects in a *layered* mediator style [29]. In this style, a mediator at one level serves both to integrate its subjects and as a subject for a mediator at the next level up.

The question we ask in this paper is whether aspects can be used as mediators in this style, with advising replacing implicit invocation. In earlier work [27], we showed that AspectJ aspects cannot serve as mediators without costly workarounds, because they cannot be instantiated or bound to selected object instances under program control. We then showed that the instance-level aspects of Eos enabled the use of aspects as mediators without workarounds [23]. However, we did not address the question of layered compositions. To use aspects as mediators in such a style demands that aspects be able to advise other aspects in full generality. Such a style is inconsistent with the properties and accepted usage of AspectJ-like languages.

To support this point and to show that Eos-U solves the problem, we compare AspectJ-like approaches and Eos-U against this style with a small example. The example is artificial, but is derived from and representative of structures used in real systems.

5. LAYERED INTEGRATED CONCERNS

Our example, presented in Figure 4, employs five basic classes. An instance of the class *VS* manages a set of vertices; and *ES,* a set of edges. An instance of *PS* manages a set of points displayed on a user interface; and *LS,* a set of connecting lines. A vertex (*V*) or point (*P*) stores a pair of coordinates and exports constructors and methods for getting and setting coordinates and for testing equality. An edge (*E*) or line (*L*) stores references to two vertices or points, respectively, and provides constructor, accessor, and equality methods. The set types export methods for creation, element insertion, deletion, and membership testing, and for getting an iterator that can return each element in turn. An instance of *UI* presents a user interface, through which one can create points and lines and insert them into, and delete them from, given *PS* and *LS* objects, respectively. The idea is that instances of the additional mediator constructs will keep systems of these objects consistent as changes occur. For example, as points are added to *PS,* corresponding vertices should be added to *PS.*

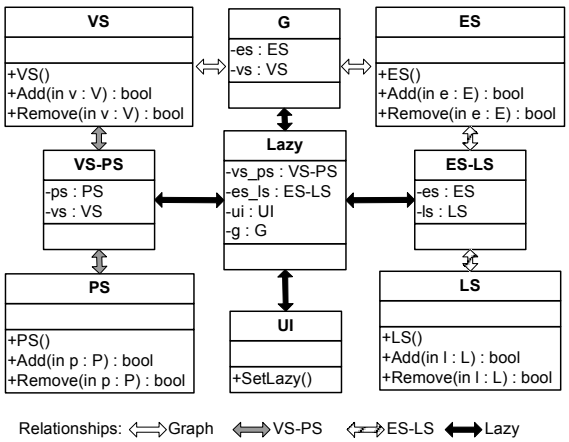

Figure 4. Graph System

Our example already exhibits a separation of integration concerns. *VS-PS* represents a class of relationships, an instance of which satisfies a requirement for given objects of the vertex set and point set classes to be kept consistent in a one-to-one relationship, as either of the two objects is changed. *ES-LS* is a similar relationship for objects of the *LS* and *ES* classes.

Our example also shows how objects can participate in several relationships. *G* (for *Graph*) represents a class of relationships, an instance of which ensures that a given edge set, *es,* remains consistent with a vertex set, *vs*, in the sense that the sets continue to represent a graph. An instance *g* of *G* could preserve this invariant in the face of deletion of a vertex, *v,* from *vs,* by deleting from *es* each edge incident on *v*, for example. The vertex set, *vs*, could thus be related by *g* to the edge set, *es,* and by an instance of *VS-PS* to *ps.* In this case, deleting a vertex would result in updates to *ps*, to *es,* and indirectly to any line set, *ls* integrated with *es* by an object of class *ES-LS.* Figure 5 presents a snippet of code for an implementation of G using the original Eos language.

So far we have seen a design with a first layer of *base* components (the sets), and a second layer of aspect-like mediators. The last class in our example, *Lazy,* is really the key. It is a relationship-maintaining object *at the next level up*. See Figure 4. Its purpose is to coordinate the behaviors of the mediators at the first level. In a system in which many updates will be made to a component in a short time, it might be useful to temporarily turn off the updating of related (e.g., user interface) objects, and to cache updates for flushing at an appropriate time. *Lazy* serves this purpose in our example. More importantly, it illustrates the idea of multi-layered, aspect-like mediation.

The state of an object, *l,* of class *Lazy* dictates how *l* coordinates the activities of associated mediators of classes *VS-PS*, *ES-LS,* and *Graph.* When in *eager* mode, *l* allows the *Graph*, *VS-PS* and *ES-LS* objects to operate as already described. However, when in *lazy* mode, *l* prevents them from propagating effects immediately, and instead caches the updates for later flushing. When toggling from *lazy* to *eager* mode, *l* reestablishes the invariants by adding or removing elements into or from the sets as necessary. *Lazy* thus separately represents a performance-related caching concern.

```
1   public instancelevel aspect G {
2       ES es; VS vs;
3        public G(ES es, VS vs){
4       this.es = es; this.vs = vs; …
5       }
6   after(bool ret, E e):execution(public bool ES.Add(E))
7                           && return(ret) && args(e){
8        if(ret){
9               vs.Add(e.GetStart());
10              vs.Add(e.GetEnd());
11             }
12       }
13  after(bool ret, V v): execution(public bool VS.Remove(V))
14                          && return(ret) && args(v){
15       if(ret){
16              /* Remove all edges incident on v */
17             }
18       }
19  }
```

Figure 5. Implementation of *Graph* in Eos

The property that makes this example useful in this paper is that it involves a *layered* separation of integration concerns. The next section shows how our earlier work on Eos and instance-level aspects [23] enables the effective use of aspect modules and aspects instances to realize the first level of mediators. The section after that shows why instance-level aspects alone are not enough, and why we needed a deeper unification of classes and instances for satisfactory composability of aspects in multi-layer structures.

6. MEDIATORS AS ASPECTS

Eos enabled an improved version of mediator-based design [26][28] by separating integration concerns as instance level aspects [23]. The basic idea is to use separate aspect instances as mediators that selectively advise their subject instances, with advising in place of scattered event (implicit invocation) code. The relationship *G* is thus represented as an aspect module, an instance of which integrates an instance of the vertex set *VS* and an instance of edge set *ES*. *VS-PS* and *ES-LS* are represented as aspects in the same way. The logic to enforce desired constraints is represented as advice that is invoked when the subjects engage in potentially invariant-violating action (e.g., vertex deletion).

Figure 5 presents source code for the *G* aspect. The constructor (lines 3-5) stores reference to the edge set and the vertex set objects to be integrated. The first advice in *G* (lines 6-12) is invoked when the event "*Edge addition"* occurs during the execution of edge set, and if an edge was successfully inserted it adds corresponding vertices to the vertex set.

This advice thus selects the join point "*execution of the method ES.Add*" using the pointcut expression *execution(public bool ES.Add(E))*. The reflective information about the join point, the return value and the argument of the method are passed to the advice by binding the advice parameters *ret* and *e* to reflective information using the pointcut expressions *return(ret)* and *args(e)*. The method *ES.Add* returns *true* in case of a successful addition. The parameter *ret*, therefore represents successful addition of an edge. When *ret* is true, the start and the end vertices of second parameter edge *e* are added to the vertex set *vs*.

The second advice (lines 13-18) is invoked and operates analogously when a "*Vertex removal"* event occurs on the vertex set. If a vertex was successfully removed, the advice removes all incident edges from the edge set.

```
1   public instancelevel aspect VS_PS {
2       VS vs; PS vs;
3       public VS_PS(VS vs, PS ps){
4       this.vs = vs; this.ps = ps; …
5       }

6   after(bool ret, V v):execution(public bool VS.Add(V))
7                           && return(ret) && args(v){
8       if(ret){
9               /* Add a point in ps corresponding to v */
10              }
11      }

12  after(bool ret, V v): execution(public bool VS.Remove(V))
13                          && return(ret) && args(v){
14      if(ret){
15              /* Remove the corresponding point from ps */
16              }
17      }
18  /* Similar advice for PS.Add and PS.Remove events */
19  }
```

Figure 6. Implementation of VS-PS in Eos

Figure 6 presents source code (with details elided) for the aspect VS_PS. The aspect represents the requirement "*Consistency between the Vertex set and Point set*" (VS-PS). The first advice (lines 6-11) executes when the event "*Vertex Addition*" occurs during the execution of the vertex set; and on successful vertex addition, adds the corresponding point to the point set. Similarly, the second advice (lines 12-18) removes the corresponding point after a successful vertex removal. The requirement "*Consistency between Edge and Point set*" (ES-LS) is implemented similarly.

The *Graph*, *VS-PS* and *ES-LS* concerns are fully modularized as aspects in the Eos program design. However, *Lazy* is still not easily modularized. The next section analyzes the problem.

7. THE COMPOSITIONALITY PROBLEM

The *Lazy* concern constrains the behavior of *Graph*, *VS-PS* and *ES-LS*. In our Eos implementation, these components are represented as instance-level aspects that advice base objects.

A typical implementation of the *Lazy* requirement would add a mode bit (lazy) to each of *Graph*, *VS_PS* and *ES_LS*, a method to switch modes, and would modify the aspect advice to handle lazy evaluation when the mode bit is set. This implementation scatters code for the *Lazy* concern over and tangles it with code for the *Graph*, *VS-PS* and *ES-LS* concerns. This scattering makes it harder to design, understand, selectively deploy, reuse, and change the *Lazy* concern.

An alternative would be to implement *Lazy* as a second-level aspect, advising the first-level *G*, *VS_PS* and *ES_LS* aspects. In particular, such a *Lazy* aspect could use *around* advice to override advice execution in the first-level aspect. When in *eager* mode, *Lazy* would just delegate back to the overridden advice. In *lazy* mode, it would cache the required reflective information and skip execution of the overridden method join point. The reflective information would be stored in the order in which join point executions occurred. When toggled from *lazy* to *eager*, the aspect would use the saved information to re-establish the invariants.

This alternative would solve the problem of the first, non-modular solution. The code for "*lazy evaluation*" would be in a separate, modularized, and reusable aspect. To add or remove this feature from the system, we would just add or remove an aspect instance.

Unfortunately, this approach cannot be realized *satisfactorily* using the model of current AspectJ-like languages, including the original Eos. The problem is in the asymmetric capabilities and treatments of classes and aspects. In particular, in AspectJ-like languages, including the original Eos, aspects can advise methods selectively, but they can advise advice only in quite limited ways.

In the current model, individual advice bodies are anonymous, and so they cannot be selected based on their names in pointcut descriptors. The pointcut designator *adviceexecution* thus selects all advice execution join points in the program. This selection can be narrowed down to all advice in a given aspect by composing this pointcut designator with the pointcut designator *within*. For example, the pointcut expression *adviceexecution() && within(G)* selects execution of every advice in the aspect G (Figure 5).

To implement a *Lazy* aspect, we would have to be able to address each advice in *G* independently (Figure 5: lines 6-12 and 13-19). The current languages do not support such fine-grained selection. This restriction compromises the compositionality of aspects. One of its effects is to constrain applications to the two-layered structures that we have discussed, where object-oriented code is advised by first-level aspects, but no higher-level advising occurs.

A workaround that springs to mind is to have advice delegate to corresponding aspect methods and to advise these methods. There are two problems with this idea. The first, and less important, is that the need for such a hack is evidence that there is something wrong in the current design. Second, the work-around is deeply unsatisfactory, in general.

First, it requires either ubiquitous up-front use, 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, delegating 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 methods, namely *proceed*.

All such parameters have to be passed to the delegate methods, incurring additional design- and run-time costs and risk 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 7 presents an example (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 (Lines 9-11). *Proceed* is allowed only in advice bodies, not in methods, in the current languages. *Proceed* will thus have to be passed from the advice body to the delegatee as a closure, perhaps using the *worker object* pattern of Laddad [16]. The work-around is both complicated and incurs the need for scattered changes, undermining the purpose of aspects.

The lack of full aspect-aspect compositionality complicates the use of advising as a general mechanism in important architectural styles, including layered systems. There is real value in being able to structure systems in such ways. In particular, the layered mediator style has been shown to be valuable in the design of evolvable integrated systems [26][28]. The inability to support such styles effectively appears unnecessarily to restrict the use of aspect technology for separating concerns—and integration concerns, in particular—in a natural, compositional style.

```
1   // Original advice
2       void around(): <pointcut> {
3           If(ShallProceed)proceed();
4       }
5   // Workaround applied to advice above
6       void around():<pointcut> {
7           OriginalAdviceCodeInMethod();
8       }
9       void OriginalAdviceCodeInMethod() {
10          If(ShallProceed) proceed();
11      }
```

Figure 7. Workaround applied to an around advice

```
1   public class G {
2       ES es; VS vs;
3        public G(ES es, VS vs){
4       this.es = es; this.vs = vs; …
5       }
6       public void AddEnds(bool ret, E e) {
7        if(ret){
8                vs.Add(e.GetStart());
9                vs.Add(e.GetEnd());
10              }
11       }
12      public void RemoveIncident (bool ret, V v) {
13       if(ret){
14               /* Remove all edges incident on v */
15              }
16       }
17      after execution(public bool ES.Add(E)) && return(ret)
18                  && args(e): call AddEnds (bool ret, E e);
19       after execution(public bool VS.Remove(V)) && return(ret)
20                  && args(v): call RemoveIncident(bool ret, V v);
21  }
```

Figure 8. Implementation of *Graph* in Eos-U

8. MODULARIZING HIGHER-ORDER CROSSCUTTING CONCERNS IN EOS-U

Eos-U provides a solution to these problems in the *classpect,* a fully compositional basic building block for unified object- and aspect-oriented program design. Eos-U supports advising as a first-class, general alternative to explicit and implicit method invocation and to overriding by way of inheritance. A classpect combines the advising capabilities of aspects with the first-class status and compositionality properties of class-based objects, allowing for the design of arbitrary structures with the flexibility to choose between object- and aspect-oriented mechanisms.

As a proof-of-concept test of this idea, we re-implemented our example system using Eos-U classpects. Figure 8 presents the classpect for the graph-maintaining mediator, *G*. The constructor (lines 3-5) stores references to the *ES* and *VS* objects to be integrated. A combination of join-point-to-method binding and methods replaces traditional, AspectJ-like advising. Compare this code with that in Figure 5. The first advice in Figure 5 (lines 6-12) is replaced in Figure 8 by a binding (lines 17-18) and the method *AddEnds* (lines 6-11). The second advice in Figure 5 (lines 13-18) is replaced by a binding (line 19-20) and the method *RemoveIncident* (lines 12-16). The first binding (lines 17-18) binds the execution join point of *public bool ES.Add(E e)* to the method *AddEnds* (lines 6-11), which adds the start and the end vertices to the vertex set if an edge was successfully added to the edge set. The second binding (lines 19-20) binds the execution join point of *public bool VS.Remove(V v)* to *RemoveIncident* (12-16).

```
1   public class VS_PS {
2       VS vs; PS vs;
3        public VS_PS(VS vs, PS ps){
4       this.vs = vs; this.ps = ps; …
5       }
6   public void AddPoint(bool ret, V v) {
7        if(ret){
8                /* Add a point in ps corresponding to v */
9               }
10       }
11  public void RemovePoint(bool ret, V v){
12       if(ret){
13                /* Remove the corresponding point from ps */
14              }
15       }
16  /* Similar methods AddVertex and RemoveVertex */
17  after execution(public bool VS.Add(V)) && return(ret)
18                  && args(v): call AddPoint (bool ret, V v);
19  after execution(public bool VS.Remove(V)) && return (ret)
20                  && args(v): call RemovePoint(bool ret, V v);
21  /* Similar bindings for PS.Add and PS.Remove events */
22  }
```

Figure 9. Implementation of VS-PS in Eos-U

Figure 9 presents the corresponding Eos-U code for *VS-PS*. The implementation of *ES-LS* is similar.

The real advantages of our approach emerge in relation to the implementation of *Lazy*, presented in Figure 10. The constructor (lines 4-7) stores references to the instances being integrated: in this case, aspect-oriented mediators. The *Lazy* class now simply overrides the advice-like methods of *G*, *VS_PS* and *ES_LS* with implementations that cache their invocations based on the state of the given *Lazy* object. We present two examples: RecordAddEnds and RecordRemIncident (lines 8-17 in Figure 10). The mode is determined by the Boolean *lazy*. If true, these methods record necessary information to re-establish the invariants of integration requirements; otherwise they invoke the inner delegate in the around delegate chain (lines 9 and 14).

The join-point-to-method bindings (lines 18-24) bind these methods around the corresponding join points. The bindings also provide the arguments at the join points to the bound methods using the pointcut expressions *args(ret), args(v)* and *args(e),* and the pointer to the around delegate chain using *arountptr(p)*. The pointer to the around delegate chain is supplied to the methods so that they can invoke the join point if needed. Eos-U has no *proceed*. The rest of the code (lines 25-32) keeps the instance variable *lazy* consistent with the state in the component *UI*.

The *Lazy* concern, which cuts across the lower-level crosscutting integration concerns, is now modularized as the *classpect Lazy*, at a second level in the advising hierarchy. To add or remove the feature from a set of objects, one just needs to add or remove an instance of this class.

9. DISCUSSION

We have claimed that a unification of object- and aspect-oriented constructs is possible and have developed a unified design with improved compositionality properties. We have tested these claims and found them supported by exhibiting a language design and compiler, and by a comparative analysis of the ability of AspectJ-like and Eos-U-like languages to preserve, at the code level, the modular structure of a specification that featured multi-level integration concerns.

One way to try to account for these improvements is by appeal to the idea of conceptual integrity in design. Brooks wrote,

> *...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, then to have one that contains many good but independent and uncoordinated ideas. … Simplicity and straightforwardness proceed from conceptual integrity. Every part must reflect the same philosophies and the same balancing of desiderata. Every part must even use the same techniques in syntax and analogous notions in semantics. Ease of use, then, dictates unity of design, conceptual integrity." [6](pp 42-44).*

The additional expressive and compositional power of classpects emerged when we enforced the kind of design unity that Brooks advocates. It forced aspect-like constructs to support *all* of the capabilities of classes—notably *new*. It forced classes to support aspect-oriented advising as a *generalized* alternative to traditional invocation and overriding. By driving out anonymous advice in favor of methods as the sole mechanism for procedural abstraction, it also pushed a previously submerged but important abstraction to the fore: the join-point-method binding.

We hypothesize that, to the extent that aspect-oriented methods turn out to be beneficial, a unified model is likely to be even more so. First, we hypothesize that the unified model will improve the *ease-of-learning* and *ease-of-use* of aspect-oriented methods. From an understanding point of view, most of the code in Figure 8, lines 1-16, looks and works like a traditional object-oriented program, for example. We believe that this symmetry could well make the transition from object-oriented to aspect-oriented design easier.

Second, we hypothesize that a unified model can further improve our ability to modularize systems, with benefits in evolvability, understandability, communication overheads in development, and parallel development. In particular, abstracting from the example of the last section, we see that the unified design supports, as a practical possibility, what we might call *modularization of higher-order crosscutting concerns*. Whereas the first-level mediators modularized integration concerns that cut across the base objects, the *Lazy* mediator modularized a concern that cut across the first-level integration concerns. The AspectJ de facto commitment to a two-level structure with one base level and one aspect level makes it hard to cleanly modularize such towers. In effect, higher-order concerns are all squashed down into—and consequently scattered across and tangled into—the single available aspect layer.

We finish this section by returning to the AspectJ rationale for separating classes and aspects. The goal was *adoptability*. Users asked for the separation because they wanted to be able to see and control a risky new construct in their systems. The non-integration of classes and aspects was thus an early tradeoff against unity for a better chance at adoption. Current AspectJ-like language designs thus still reflect that evolutionary path. Now that these languages are achieving significant adoption, with concomitant reductions in perceived risk, designers should revisit deep tradeoffs made to surmount early adoptability barriers. Ideally, reconsideration would occur before strong adoption creates irreversible lock-in. Our work thus presents a *timely* analysis of a potentially important alternative language design and program structuring philosophy.

```
1  using Eos.Runtime;
2  public class Lazy {
3        VS_PS  vs_ps; ES_LS es_ls; G g; UI ui; bool lazy = false;
4         public Lazy(VS_PS vs_ps, ES_LS es_ls, G g, UI ui){
5        this.vs_ps = vs_ps; this.es_ls = es_ls;
6        this.g = g; this.ui = ui; …
7        }

8  public void RecordAddEnds(bool ret, E e, AroundADP p) {
9            if(!lazy) p.InnerInvoke();
10           else {  /* Record invocation of G.AddEnds */ }
11       }

12 public void RecordRemIncident(bool ret, V v, AroundADP p){
13         if(!lazy) p.InnerInvoke();
14         else { /* Record invocation of G.RemoveIncident */}
15       }
16 /* Similar methods to record method invocations of
17    VS_PS and ES_LS */

18 void around execution(public void G.AddEnds(bool, E))
19    && args(ret) && args(e) && aroundptr(p):
20    call RecordAddEnds (bool ret, E e, AroundADP p);

21 void around execution(public void G.RemoveIncident(bool, v))
22    && args(ret) && args(v) && aroundptr(p):
23    call RecordRemIncident (bool ret, V v, AroundADP p);
24 /* Similar bindings for methods in VS_PS and ES_LS */

25 void SetLazy() { lazy = true; }

26 after execution(public void UI.SetLazy()):call SetLazy();

27 void ResetLazy(){
28     /* For each recorded invocation of AddEnds invoke
29          the method AddEnds on g. Similarly invoke other
30          appropriate methods for other recorded invocations*/
31    }

32 after execution(public void UI.ResetLazy()): call ResetLazy();
33 }
```

Figure 10. The Requirement Lazy in Eos-U

10. RELATED WORK

AspectJ [1], AspectWerkz [4], and Caesar [19] are all related to our work. Kiczales reports [13] that in at least one early version of AspectJ, there was no separate *aspect* 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-U. 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 [4] is the design most closely related to our work. The aim of this project was to provide the expressiveness of AspectJ [1] without sacrificing pure Java and all the surrounding tools. 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.

First, and crucially, the system does not support the concept of aspects as objects under program control. Instead, the use of Java classes as aspects is highly constrained so that the runtime system can maintain control. 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, albeit with significant adoptability benefits. Third, AspectWerkz lacks static type checking of advice parameters. Rather, reflective information is marshaled from the *JoinPoint* arguments to advice methods.

The design of Caesar [19] 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-U, advice can be bound statically or dynamically; however, aspects in Caesar cannot directly advise individual objects on a selective basis. Both first class aspect instances and instance-level advising are essential for expressing integration concerns as aspects [24][27].

Aspect languages such as HyperJ [30][31] have 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 [11].

11. CONCLUSION

The main contribution of this work is a novel synthesis of object- and aspect-oriented programming language constructs and design methods, including the Eos-U language, a compiler able to handle production code, and evidence that suggests that this synthesis has potentially significant benefits in aspect-oriented program design. In particular, we showed that the classpects provides a new level of support for modularizing what we identified and characterized as *higher-order* crosscutting concerns. This work creates a timely opportunity to rethink the two-level ontology for aspect-oriented programs that the original separation of aspects and objects entailed.

12. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grants ITR-0086003 and CCF-0429786. We thank the anonymous reviewers for their helpful comments and Gregor Kiczales for recounting the origin of aspects as separate from objects.

13. REFERENCES

[1] AspectJ : http://eclipse.org/aspectj

[2] AspectC++, http://www.aspectc.org.

[3] AspectR: “Simple Aspect Oriented Programming in Ruby,” http://aspectr.sourceforge.net/.

[4] AspectWerkz: http://aspectwerkz.codehaus.org/

[5] Aldrich, J., "Open Modules: A Proposal for Modular Reasoning in Aspect-Oriented Programming.", In the Proceedings of the *Workshop on Foundations of Aspect Languages (FOAL ’04)*, March 2004.

[6] Brooks, F. P. Jr., "The Mythical Man-Month: Essays on Software Engineering", Addison-Wesley, 1975.

[7] Dijkstra, E. W., "The Humble Programmer", *Communications of the ACM*, Vol 15, No: 10, pp. 859-866, 1972.

[8] Eos: http://www.cs.virginia.edu/~eos

[9] Garlan, D., and Notkin, D., “Formalizing Design Spaces: Implicit Invocation Mechanisms”. *VDM '91: Formal Software Development Methods*, Oct. 1991.

[10] Filman, R. E., and Friedman. D. P., "Aspect oriented programming is quantification and obliviousness", In *OOPSLA 2000 Workshop on Advanced Separation of Concerns*, Minneapolis, MN, Oct. 2000.

[11] Harrison W., Ossher H., and Tarr P., “Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition”, *IBM Research Report* RC22685 (W0212-147) December 30, 2002.

[12] Hirschfeld, R., "AspectS -- Aspects in Squeak", *ECOOP'2002 Workshop on Generative Programming*, Jun 2002.

[13] Kiczales, G., “Personal Communication with Kevin Sullivan”, Jan 2005.

[14] Kiczales, G., “The fun has just begun”, Key note address of 2nd *International Conference on Aspect-Oriented Software Development*, Boston, MA, 2003.

[15] Kiczales, G.., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J., “Aspect-oriented programming,” *Proceedings of the European conference on object-oriented programming (ECOOP)*, Springer-Verlang, Lecture Notes on Computer Science 1241, June 1997.

[16] Laddad, R., “AspectJ in Action: Practical Aspect-Oriented Programming”, Manning publications, 2004.

[17] Lamping, J., “The role of the base in aspect-oriented programming”, *First Workshop on Multi-dimensional separation of concerns in object-oriented systems* (at OOPSLA '99).

[18] MacLennan, B. J., “Principles of Programming Languages: Design, Evaluation, and Implementation”, 3rd Edition, Oxford University Press, 1999.

[19] Mezini, M., and Ostermann, K., “Conquering Aspects with Caesar”, *Proceedings of the 2nd international conference on Aspect-oriented software development (AOSD 03)*, Mar 2003, Boston, MA, USA, pp. 90-100.

[20] C#: http://msdn.microsoft.com/net/ecma/.

[21] .Net Framework: http://msdn.microsoft.com

[22] Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules", *Communications of the ACM*, 15(12):1053–1058, Dec 1972.

[23] Rajan, H. and Sullivan, K., “Eos: Instance-Level Aspects for Integrated System Design”, *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software*

engineering (ESEC/FSE 03), Helsinki, Finland, Sep 2003, pp 291-306.

[24] Rajan, H. and Sullivan, K., "Need for Instance Level Aspects with Rich Pointcut Language", *Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT)* held in conjunction with 2nd *international conference on Aspect-oriented software development*, Boston, MA, USA, Mar 2003.

[25] Sakurai, K., Masuhara H., Ubayashi N., Matsuura, S., Komiya S., "Association Aspects", *Proceedings of the 3rd international conference on Aspect-oriented software development (AOSD 04),* Lancaster, UK, Mar 2004, pp. 16-25.

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

[27] Sullivan, K., Gu, L., Cai, Y., "Non-modularity in Aspect-Oriented Languages: Integration as a Crosscutting Concern for AspectJ", Proceedings of the *1st international conference on Aspect-oriented software development (AOSD 02)*, Enschede, The Netherlands, Apr 2002, pp. 19-26.

[28] Sullivan, K. and Notkin, D., "Reconciling environment integration and software evolution", *ACM Transactions on Software Engineering and Methodology* 1, 3, July 1992, pp. 229–268.

[29] Sullivan, K., Kalet, I., Notkin, D., "Evaluating the mediator method: Prism as a case study", *IEEE Transactions on Software Engineering*, Vol. 22, No. 8, August 1996. pp. 563-579.

[30] Tarr, P. and Ossher, H., "Multi Dimensional Separation of Concerns using Hyperspaces", *IBM Research Report* 21452, April, 1999.

[31] Tarr, P. and Ossher, H., "Hyper/J™ User and Installation Manual", IBM Corporation.