Classpects: Unifying Aspect- and Object-Oriented Language Design

Hridesh Rajan Kevin Sullivan
Department of Computer Science, University of Virginia
{hr2j, sullivan}@cs.virginia.edu

Abstract

The contribution of this work is a compositional, modular building block for program design that unifies classes and aspects as they appear in today's most successful aspect-oriented languages. We call it the classpect*. We make three basic claims for this idea: first, it can be realized cleanly in an industrial-strength language; second, it improves the conceptual integrity of the programming model; third, its compositionality significantly expands the useful program design space. To test these claims we designed and implemented a language supporting this model. Eos-U extends C#, has all of the aspect-oriented capabilities of AspectJ, but also unifies classes and aspects. We then exhibit a layered architectural style for modular design of integrated systems, and we show that, unlike earlier aspects, classpects cleanly support this design style.*

1. Introduction

The most successful aspect-oriented (AO) languages today—the AspectJ-like languages [11]—support two related but distinct constructs for modular design: classes and aspects. Programs are designed in a two-layered style: a class layer is advised and extended by an aspect layer. Key properties of aspects make it hard to go beyond such a two-layered style. While aspects can advise classes, classes can't advise aspects, and aspects can't advise other aspects in full generality.

The main contribution of this work is a new module construct for program design, which we humorously call the *classpect.* The classpect simultaneously extends *classes* with the capabilities of aspects, and aspects with the compositionality of objects. We claim that this unification is achievable in a clean, industrial language; that it improves the conceptual integrity [4] of the design model; and that it expands the space of useful architectural styles available to the designer in valuable ways, with aspect composition as a new possibility.

To test feasibility, we designed and implemented a *classpect-oriented* language. *Eos-U* extends C#, has the aspect capabilities of *AspectJ,* and unifies classes and aspects. The compiler handles existing C# and fully supports classpects with an enhanced notion of *class*. To test the claim that the model expands the design space in useful ways, we exhibit a previously studied architectural style involving a separation of integration concerns in a layered style [20][22]. We show that it is not naturally expressible in AspectJ-like languages but is easily accommodated in our compositional setting.

The rest of the paper is organized as follows. Section 2 describes the current aspect language model. Section 3 discusses requirements for a unified model. Section 4 exhibits Eos-U as a proof of concept. Sections 5-8 present our evaluation of the enhanced compositionality of classpects. In particular, we show that they support a separation of integration concerns in a multi-layered architectural style. Section 9 discusses related work. Section 10 concludes.

2. Background

To make this work self-contained, we briefly review basic concepts in the dominant aspect-oriented 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 [12], and is crosscutting if it cannot be realized in traditional object-oriented designs except with scattered and tangled code. By scattered we mean not localized in a module but fragmented across a system. By tangled we mean intermingled with code for other concerns [10].

The driving problem for this work is the need to separate component integration concerns in complex systems. By integration we mean the co-ordination of component behaviors to automate interactions and to enforce constraints. Sullivan and Notkin [20][22] showed that integration is crosscutting in the sense that integration code is generally scattered and tangled.

Sullivan and Notkin then showed that integration relationships can be separated into *mediator* modules if the components to be integrated declare and announce integration-related events. An aspect-oriented critique of the mediator style is that it doesn't fully separate integration concerns, in that scattered and tangled event code is still needed. This paper and our recent works are driven by the hypothesis that aspects can improve the separation of integration concerns by serving as mediators, with *advice* replacing scattered event code.

The problem, as we show, is that current aspects are not entirely up to the task. Taking the separation of integration concern as a driving problem turns out to be a useful test of aspect language designs. It reveals shortcomings and leads to significant enhancements.

To make these points clear requires a more detailed treatment of aspect language design. Aspect languages add five key new constructs to the object-oriented model: join points, pointcuts, advice, introductions (not discussed in any detail in this paper), and aspects. We provide a simple example to make the points concrete.

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

A join point is a program execution event exposed, by the language definition, to behavioral modification by aspects. The execution of a method in the program in which the *Tracing* aspect appears is an example of a join point. A pointcut (lines 2-3) is a predicate that selects a subset of join points for such modification—here, any call to any method. An advice (see lines 4-6) serves as a *before, after,* or *around* method to effect such a modification 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 are class-like in supporting the procedural abstraction, encapsulation, and inheritance abilities of classes. However, they differ from classes in key ways. First, aspects can use pointcuts, advice, and introductions. In this dimension, aspects are strictly more expressive than classes. Second, manipulation of aspect instances under program control is severely constrained. Aspect instances are thus not first-class, and, in this dimension, classes are strictly more expressive than aspects. Third, although aspects can advise classes in many ways, they can advise aspect advice only in limited ways, making aspects not fully compositional under advising.

If aspects are to support modular representation of integration concerns in complex systems, aspects instances need to be first-class, and it must be possible to compose them in layered and hierarchical patterns. Two earlier works addressed the first-class-object issue [18][21], but left aspects and classes incomparable, and the non-compositionality problem, unresolved. This paper tackles this problem. In doing so, it achieves a unification of classes and aspects—of object- and aspect-oriented programming language models.

3. Conceptual Integrity in Design

In *The Mythical Man Month* [4], Brooks contends,

> ...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." (pp 42-44).

We contend that the conceptual integrity of the current language design model can be improved by our proposed unification. Eliminating non-orthogonal but incomparable classes and aspects in favor of a unified module construct achieves unity in design, so that every part of a program can "use the same techniques in syntax and analogous notions in semantics."

We see two basic requirements for a unified model. First, it should not weaken the expressiveness of earlier aspect or object languages. Second, it should have a single, first-class, class-like module construct, with a single method construct for procedural abstraction. The first requirement calls for the full expressive power of AspectJ-like languages; the second, for a compositional notion of *classes that can advise.* We claim that clean language designs meeting these requirements can be constructed. The next section supports this claim by presenting the Eos-U language as an example.

4. The Eos-U Model of Language Design

Eos-U is a *classpect*-oriented version of Eos [17], an aspect-oriented extension of C# [14], a.NET [15] language. Eos was the first AspectJ-like language with first-class aspect instances and instance-level advising.

```
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

Eos-U now unifies and generalizes aspects and objects in three important dimensions. The rest of this section presents the Eos-U language design model in detail.

4.1 Unifying classes and aspects

First, Eos-U unifies aspects and classes. A *class* in Eos-U supports the full *classpect* notion: all C# class constructs, all the capabilities of AspectJ-like aspects, and the extension to aspects needed to make them first class objects. Second, Eos-U unifies advice bodies and methods. The AspectJ *proceed* construct in *around* advice is eliminated by making the inner join point and its invocation explicit in an object-oriented style. Third, Eos-U provides a generalized and theoretically clean invocation model. Explicit invocation by procedure call and implicit invocation by advising are both first-class, general, and compositional invocation mechanisms.

Unifying advice and methods led to another elegant result. We make *join-point-advice* bindings separate and abstract. Architectural connection is made explicit in what amounts to the *event-method* binding constructs developed for implicit invocation systems [7][20][22]. In particular, Eos-U separates *crosscut specifications,* which express a *pointcut* and when advice executes (before, after or around), from advice bodies, which become normal methods. This separation allows one to reason separately about these 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,* in Figure 1 presents our *crosscut specification* construct. A *binding_declaration* has four parts. The first, *opt_static,* specifies whether a binding is static or not. A non-static binding is equivalent to instance-level advising [17].

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

Figure 2: An advice and equivalent binding

By instance-level advising, we mean selective advising of the join points of individual object instances. A static binding affects all instances of an advised class. The decision to depict instance/type level advising by keyword *static* was made to match the notion that static data members are class members, and non-static, instance members.

The second part of a binding (*after*/*before*/*around*) specifies whether the advising method executes after, before, or around selected join points. The third part, *pointcuts*, selects the join points at which an advising method executes. The final part specifies the advising method. A binding can provide a list of methods to execute at a join point *in the order specified*. The binding can also pass reflective information about the join point to the methods being invoked by binding method parameters to reflective information, using pointcut designators such as *this*, *target*, *args*, etc., just as in AspectJ [1].

Methods bound by binding declarations have to follow certain rules. First, a method has to be accessible in the class containing a binding declaration. 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 Baz.Bar())*, then it must return *int*.

The listing in Figure 2 shows an advice in the current aspect languages and equivalent method binding. The advice (lines 2-4) executes around the join point *execution(public int Baz.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).

4.2 New pointcut designators

To pass reflective information at a join point to a bound method, binding uses AspectJ-like pointcut designators such as *args*, *target* and *this*. Existing designators in Eos are incomplete in that not all the information available at the join point is exposed.

```
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

Any other information needed has to be marshaled from an *implicit argument* called *thisJoinPoint*. For example, to access the return value at a join point, one needs to call the method *getReturnValue* on the implicit argument. Eos-U adds new pointcut designators to fill the gap. 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 concepts such as implicit arguments to advice.

4.3 Around bindings

Around advice in AspectJ is executed *instead* of a join point, and can invoke the join point using *proceed*. In Eos-U a method bound *around* is similarly executed instead of the join point. If the method needs to proceed to execute the original join point, the method takes an argument of type *Eos.Runtime.AroundADP*. The class Eos.Runtime.AroundADP represents a delegate chain of original join points and other around method bindings, and provides a method, *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) as shown in the 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 poincut 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). Adding the new pointcut designator to the language eliminates the need for the special, implicit *proceed* construct of AspectJ.

The Eos-U design fulfills the requirements stated at the start of this section. First, there is only one unit of modularity, *class*, and one mechanism for procedural abstraction, *method*. Second, all the expressiveness of AspectJ-like languages is present in Eos-U, along with the extension that are needed for aspect instances to be first-class objects, as they must be for a unification of classes and aspects to be achieved.

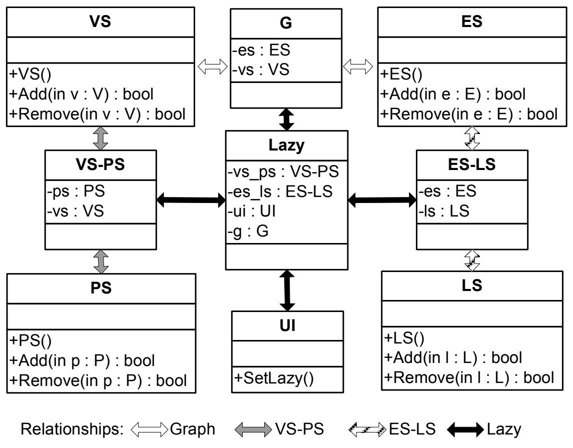

Figure 4. Graph System

In addition, join-point-to-method bindings are now separate, orthogonal, and abstract interface elements in Eos-U classes. Eos-U achieves a new unity of design—enhanced conceptual integrity—in the space of AspectJ-like languages.

The following sections evaluate our claim that the enhanced compositionality properties of Eos-U create valuable architectural possibilities for aspect-oriented design. We compare the abilities of AspectJ-like languages and Eos-U, respectively, to map a previously studied architectural design style having two key properties. First, integration relationships between objects are represented as separate mediator objects that are implicitly invoked by actions on the objects being integrated. We use aspect instances and advising for this purpose. Second, systems can be layered to an arbitrary depth. This is the property that will challenge the current aspect language model. The problem is that a commitment to use aspects to implement mediators constrains one to a two-layered design, because aspects can't advice other aspects with full generality. To make these points concrete, the next section starts with a specific challenge example from the mediator work of Sullivan and Notkin [25].

5. Composing Layered Integrated Systems

Our example system shown in Figure 4, consists of five components. The components; *VS*, *ES*, *PS* and *LS*, manage a set of vertices, a set of edges, a set of points on the user interface and a set of lines on the user interface respectively. The component *UI* presents a user interface, through which the user can create, insert, and remove points and lines. The user can create points and, given two points, a line. The user can also insert and delete points and lines into and from the point and line sets respectively.

```
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

A vertex (*V*) or point (*P*) stores a pair of coordinates and exports methods for creation, for getting and setting coordinates, and for testing equality. An edge (*E*) or line (*L*) stores references to two vertices and points, respectively, and provides methods for creation, getting the points, and for testing equality. The set types export methods for set creation, element insertion, deletion, and membership testing, and for getting an iterator that can return each element in turn.

There are some additional constraints on the system. First, the vertex and point set should be *consistent*. By consistent we mean that the system should maintain a 1-to-1 mapping between the elements of *VS* and *PS*. If a vertex is inserted into *VS*, a corresponding point should be inserted into *PS*. Similarly, the edge and the line set should be consistent. The consistency requirement between vertex & point set and edge & line set will be referred as *VS-PS* and *ES-LS* respectively.

Second, the vertex and the edge sets should form a *graph*. During the system operation it is possible to violate this requirement by inserting an edge into *ES* for which neither or only one of the vertices of the edge is in *VS* or by deleting a vertex from *VS* upon which one or more edges in *ES* is incident. We will refer to this requirement as *Graph*.

Third requirement is to support lazy as well as eager maintenance of *Graph* as well as consistency requirements *VS-PS* and *ES-LS* depending on a state maintained in the user interface component *UI*. When in *eager* mode the system will perform normally, maintaining the invariants dictated by the requirements *Graph*, *VS-PS* and *ES-LS*. It will make no compensating updates when in *lazy* mode. When toggling from *lazy* to *eager* mode the system will reestablish the invariants by adding or removing elements into or from the sets as necessary. We will refer to this requirement as *Lazy*.

This system is layered. Independent components, *VS*, *ES*, *PS*, *LS* and *UI* are in the first layer. *Graph*, *VS-PS,* and *ES-LS* are in the second. *Lazy* is in the third. Such a layered composition of independent parts can significantly ease the design and evolution of complex, integrated systems.

The requirements of this system that makes it a particularly interesting case study is that it requires separation of integration concerns at various levels of a multi-layered architecture. In the next section, we will describe how part of this separation is achieved by the existing aspect language model using aspect instances and instance level advising capabilities of Eos [17].

6. Separating Integration Concerns

Eos enabled an improved version of mediator-based design [20][22] by separating integration concerns as instance level aspects [17] that selectively advise the component instances to be integrated. The requirement *Graph* in this improved style is represented as an aspect, an instance of which integrates an instance of the vertex set *VS* and an instance of edge set *ES*. Similarly, the requirement *VS-PS* and *ES-LS* will be represented as aspects *VS_PS* and *ES_LS* integrating instances of VS & PS and ES & LS respectively. The logic to enforce desired constraints will be represented as advice constructs of these aspects that in turn will be invoked at the interesting events in the component execution.

The source code for the aspect G is listed in Figure 5. The constructor of the aspect *G* (lines 3-5) stores reference to the edge set and the vertex set instance being integrated. The first advice in *G* (lines 6-12) is invoked when the event "*Edge addition"* occurs during the execution of edge set *ES* and if an edge was successfully inserted it adds corresponding vertices to the vertex set. Second advice in G (lines 13-18) is invoked when the event "*Vertex removal"* occurs during the execution of vertex set VS and if a vertex was successfully removed it removes all incident edges from the edge set.

To impose the desired constraints on the edge set and vertex set, first advice (lines 6-12) of aspect G 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, return value and the argument of the method, is passed to the advice by binding 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.

```
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

The parameter *ret*, therefore represents successful addition of an edge *e* to the edge set *es*. 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) similarly removes every edge incident on the parameter vertex *v* if it was successfully removed from the vertex set *vs*.

The source code (details elided for presentation) for the aspect VS_PS is listed in Figure 6. The aspect represents the requirement "*Consistency between the Vertex set and Point set*". 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 the Edge set and Point set*" (ES-LS) can be implemented similarly.

The requirements *Graph*, *VS-PS* and *ES-LS* are fully modularized as aspects in the Eos implementation. However, *Lazy* is still not amenable to modularization using the current aspect language model. In the next section, we discuss this issue in detail and describe why the current model doesn't support a clean separation of integration concerns in layered systems.

7. Compositionality of Aspects

The requirement *Lazy* constrains the state and behavior of the components *Graph*, *VS-PS* and *ES-LS*. In the Eos implementation described in the previous section, these components are represented as aspects, instances of which integrate lower level components.

A naïve implementation of this requirement will add a mode bit (lazy) to each of the aspects *Graph*, *VS_PS* and *ES_LS*, a method to switch modes, and modify the aspect advice to handle lazy evaluation when the mode bit is set. This implementation technique though simple scatters the implementation of the concern *Lazy* over, and tangles it with, the implementation of the concerns *Graph*, *VS-PS* and *ES-LS*. The scattered and tangled code for the concern "*lazy evaluation*" makes design, understanding, and subsequent evolution more difficult. It is not possible to add or remove this feature without modifying the code for other components. Finally, in the current system, lazy evaluation isn't reusable.

An alternative approach is to implement the requirement as another aspect *Lazy*. This aspect will contain an advice that executes around the advice in aspects *G*, *VS_PS* and *ES_LS*. If we recall, by executing around the join point we mean that advice is executed instead of the join point and it may trigger the execution of the join point. In this case, our join points of interest are the advice constructs in the aspects *G*, *VS_PS* and *ES_LS* that maintain the invariants of the integration requirements.

In the eager mode, the advice in the aspect *Lazy* will simply trigger the execution of the join point i.e. let the advice constructs maintain the invariants. When in the lazy mode, it stores the reflective information about the join point, required to re-establish the invariants of the integration requirements, and skips the execution of the join point. The reflective information is stored in the order in which join point execution occurred. When the mode is toggled from lazy to eager, the aspect will read the saved information and re-establish the invariants of the integration requirements.

The alternative approach solves the problems of the naïve solution. The code for "*lazy evaluation*" 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 concern code.

The alternative solution looks promising but unfortunately; it can not be realized using the language model of current AspectJ-like languages including Eos. The key idea behind this solution is to represent the integration requirement as an aspect *Lazy* that uses advising as an invocation mechanism to integrate the component UI with components *G*, *VS_PS* and *ES_LS,* represented as aspects. However, in the current language model this structure is not achievable owing to a key restriction in AspectJ-like languages including our earlier work on Eos—while aspects can advise classes in many ways, they can advise other aspects only in restricted ways.

```
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 7. The Behavioral Relationship *Graph* in Eos-U

Aspects can advise methods in other aspects, but they can advise advice in other aspects in only limited ways. In the current model, individual advice constructs are not addressable because they don't have names. The pointcut designator *adviceexecution* selects all advice execution join points in the program. This selection can be narrowed down to all advice constructs in a given aspect by composing this pointcut designator with pointcut designator *within*. For example, the pointcut expression *adviceexecution() && within(G)* selects execution of every advice in the aspect G (Figure 5).

To implement the requirement *Lazy* for aspect G, we need to be able to address each advice in the aspect (Figure 5: lines 6-12 and 13-19) independently. In the current model, it is not possible to make such fine-grained selection. This restriction constrains application of advising as an invocation mechanism to two layered structures where classes at the bottom level are being advised by aspects 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. Here we see an example of hierarchical architecture where the requirements *G*, *VS-PS,* and *ES-LS* act as connectors to integrate two or more components and the requirement *Lazy* treats these connectors as components. Such requirements are commonplace and it has been shown that they are useful in designing systems for ease of evolution [20][22]. 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.

```
public class VS_PS {
      VS vs; PS vs;
       public VS_PS(VS vs, PS ps){
      this.vs = vs; this.ps = ps; ...
       }
public void AddPoint(bool ret, V v) {
       if(ret){
                /* Add a point in ps corresponding to v */
                }
       }
public void RemovePoint(bool ret, V v){
        if(ret){
                 /* Remove the corresponding point from ps */
                }
       }
/* Similar methods AddVertex and RemoveVertex */
after execution(public bool VS.Add(V)) && return(ret)
                    && args(v): call AddPoint (bool ret, V v);
after execution(public bool VS.Remove(V)) && return (ret)
                    && args(v): call RemovePoint(bool ret, V v);
/* Similar bindings for PS.Add and PS.Remove events */
}
```

Figure 8. Requirement VS-PS in Eos-U

8. Expanding the Design Space

Eos-U lifts these limitations by making the basic building block of program design fully compositional under both procedure call and advising. In Eos-U, both become first class invocation mechanisms. The only building block in Eos-U is the *classpect*. It has both join point advising and the compositional properties of classes, allowing for arbitrary multi-layered structures in which procedure call and advising are available.

To demonstrate full compositionality of the proposed model we re-implemented the graph system in Eos-U. Figure 7 shows the listing of the integration requirement *Graph* in Eos-U as a *classpect*. Similar to the listing in Figure 5, the constructor (lines 3-5) stores references to the *ES* and *VS* instances being integrated. A combination of join-point-to-method binding and methods replaces traditional advice. The first advice (lines 6-12) in Figure 5 is replaced by the binding (lines 17-18) and method *AddEnds* (lines 6-11) in Figure 7. The second advice (lines 13-18) in Figure 5 is replaced by the binding (line 19-20) and method *RemoveIncident* (lines 12-16).

The first method binding (lines 17-18) binds the join point execution of the method *public bool ES.Add(E e)* to the method *AddEnds* (lines 6-11). Similar to the body of the first advice in Figure 5, the method *AddEnds* adds the start and the end vertices to the vertex set if an edge was successfully added to the edge set. The second method binding (lines 19-20) binds the join point execution of the method *public bool VS.Remove(V v)* to the method *RemoveIncident* (lines 12-16). The listing in Figure 8 shows the implementation of the requirement *VS-PS* in Eos-U. The implementation of *ES-LS* is similar.

```
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 9. The Requirement Lazy in Eos-U

Eos-U is an advance over AspectJ-like languages, including Eos, in several dimensions. From a program understanding point of view, most of the code (lines 1-16) looks and works like a traditional object-oriented program. We believe that this symmetry can help make the transition from object-oriented to aspect-oriented design easier. The requirement "*Lazy*" is represented as a *classpect* that contains a constructor and two methods that perform the desired logic. An instance of this class can be created like a normal object-oriented class, passed as argument, returned as value etc.

From an evolution perspective, this implementation makes the integration logic, which was not addressable when represented as advice, fully addressable. Previously, we were unable to individually select the join point *advice execution* for advising. The unification of advice and methods moves the code that was earlier in the advice body in Figure 5 to normal methods eliminating this restriction.

By enabling unrestricted advising of methods that represent advices, the unified model has enabled unrestricted advising of *classpects*, representing traditional aspects, by other *classpects*. Unrestricted advising in turn enables construction of arbitrary multi-layered advising structures.

Earlier, we were not able to map the ideal design of the requirement "*Lazy*" directly to implementation due to lack of full compositionality. The advance in language design enables a clean separation of the concern as shown in Figure 9.

The Figure 9 shows the implementation of the requirement *Lazy* as *classpect Lazy*. The constructor (lines 4-7) stores references to the component instances being integrated. The idea behind the alternative implementation of the requirement was to select advice constructs in aspects *G*, *VS_PS* and *ES_LS* individually for advising. In Eos-U implementation, these advice constructs are represented as methods in the *classpects G*, *VS_PS* and *ES_LS*. The *classpect Lazy* provides methods to record invocation of these procedures. For presentation purposes, only two such methods, RecordAddEnds and RecordRemIncident (lines 8-17), are shown in Figure 9.

The mode is determined by a Boolean variable *lazy*. If *lazy* is true, these methods record necessary information to re-establish the invariants of integration requirements, else they invoke the inner delegate in the around delegate chain (lines 9 and 14).

The join-point-to-method bindings (line 18-24) binds the recording methods around the corresponding join points. They also expose the reflection information, arguments at the join points, using the pointcut expressions *args(ret), args(v)* and *args(e)* and the pointer to the around delegate chain using the pointcut expression *arountptr(p)*. The pointer to the around delegate chain is supplied to the methods so that they can invoke the join point if needed. The rest of the code (line 25-32) keeps the instance variable *lazy* consistent with the state in the component *UI*.

The *Lazy* constraint is now completely modularized as the *classpect Lazy*. To add or remove the feature from a set of objects, one just needs to add or remove an instance of this class. We have thus shown that the unification of the language model enables unrestricted composition of modules in an aspect-oriented program, which in turn enables effortless embodiment of layered, hierarchical or networked architectures with advising as an important invocation mechanism.

9. Related Work

AspectWerkz [2] is the design most closely related to our work. The aim of this project was to provide the expressiveness of AspectJ [1] without sacrificing Java and all the tools developed to support it. The solution is to use normal Java classes to represent both AspectJ-like classes and AspectJ-like aspects, with advice represented in normal Java methods, and to separate all join-point-advice bindings into either annotations in the form of comments, or into separate XML binding files.

AspectWerkz is a pragmatic and effective solution to the problem it addresses. As a side effect of meeting its design goals, it achieves a superficial unification of classes and aspects, in the sense that both are modeled as Java classes. However, the language does not achieve the unification that we seek.

First, classes and methods used in an aspect style are highly constrained. A class representing an aspect must have either no constructor or one with one of two predefined signatures. A method representing an advice body can have only one argument, of type *JoinPoint*. These limitations allow AspectWerkz to control aspect creation and advice invocation. Second, AspectWerkz lacks a unified language design. Rather, two languages are used, and a pre-processor resolves extra-linguistic binding constructs. Third, in AspectWerkz, reflective information is marshaled dynamically from *JoinPoint* arguments to advice methods, precluding static type checking of advice parameters.

Eos-U, by contrast, unifies aspects and classes in a single, fully general, first-class unit of modularity under program control. Eos-U has a unified programming language design. Eos-U supports static type checking across join-point-method bindings. Eos-U also supports full compositionality of these basic building blocks.

Aspect languages such as HyperJ [23][24] are, at a basic level, similar in having one unit of modularity, classes, with an extra-linguistic notation for expressing bindings. Nor do they support program control over aspects as first-class citizens. In practice, they are also often limited to using methods or program elements extracted to form methods as join points [9].

10. Conclusion and Future Work

In this paper, we present a novel modular building block, *classpect*, for program design. It unifies classes and aspects in both language design and in programs that need to use aspect- and object-oriented constructs. The unification improves the conceptual integrity of the current language design model with an expected but as yet untested improvement in ease of use. We further showed that this unification enables use of advising as for invocation in a unrestricted, compositional style. This advance expands the useful program design space to include important styles: in particular, one involving a separation of integration concerns in layered systems.

Eos-U inherits approaches to typing, subclassing, polymorphism, and the like, from the C# language. An area for future work is a semantics of inheritance for crosscut specifications. In Eos-U today, as in AspectJ, bindings are simply not inherited. The separate and abstract representation of crosscuts in Eos-U provides a new opportunity to investigate inheritance of binding in such languages.

11. Acknowledgements

This work was supported in part by the National Science Foundation under grant ITR-0086003.

12. References

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

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

[3] 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.

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

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

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

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

[8] 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.

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

[10] Kiczales, G., "The fun has just begun", Key note address of International Conference on Aspect-Oriented Software Development, Boston, MA, 2003.

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

[12] 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).

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

[14] Microsoft. C# Specification Homepage. http://msdn.microsoft.com/net/ecma/.

[15] Microsoft .Net Framework Developers Guide available at http://msdn.microsoft.com

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

[17] Rajan, H. and Sullivan, K., “Eos: Instance-Level Aspects for Integrated System Design”, *2003 Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 03)*, (Helsinki, Finland, Sept 2003).

[18] Rajan, H. and Sullivan, K., "Need for Instance Level Aspects with Rich Pointcut Language", In the proceedings of the Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT) held in conjunction with International Conference on Aspect-Oriented Software Development, Boston, Massachusetts, March 2003.

[19] Sakurai, K., Masuhara H., Ubayashi N., Matsuura, S., Komiya S., "Association Aspects", Proceedings of AOSD 2004. Lancaster, UK, March, 2004.

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

[21] Sullivan, K., Gu, L., Cai, Y., “Non-modularity in Aspect-Oriented Languages: Integration as a Crosscutting Concern for AspectJ”, Proceedings of Aspect-Oriented Software Design, 2002, pp. 19-26.

[22] 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 (short form: Proceedings of the 4th SIGSOFT Symposium on Software Development Environments, 1990, pp. 22–33).

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

[24] Tarr, P. and Ossher, H., “Hyper/J™ User and Installation Manual”, IBM Corporation.