Gang-of-Four Design Patterns: A Case Study of the Unified Model and the Eos Programming Language

Hridesh Rajan

Iowa State University
hridesh@cs.iastate.edu

Abstract. In earlier work, we showed that the AspectJ notions of aspect and class can be unified in a new module construct that we called the *classpect*, and that this new model is simpler and able to accommodate a broader set of requirements for modular solutions to complex integration problems. We embodied our unified model in the Eos language design. The main contribution of this paper is a case study, which considers the implementation of the Gang-of-Four (GOF) design patterns [1] in Eos to analyze the effect of new programming language constructs on these implementations. We also compare these implementations with the AspectJ's implementation. Our result shows that the Eos implementation showed improvement in 7 out of 23 design patterns, and are no worse in case of other 16 patterns. These improvements were mainly manifested in being able to realize the intent of the design patterns more clearly. The design structures realized in the Eos implementation provide supporting evidence for the potential benefits of the unified model.

1 Introduction

In prior work, we showed that the notions of aspect and class in the AspectJ language model [2] can be unified in a new module construct that we called the *classpect* [3]. We also showed that this new model is significantly simpler, more compositional, and able to accommodate a broader set of requirements for modular solutions to complex integration problems [4, 5].

We embodied our unified model in the Eos language design [3, 6–9], in which the basic unit of modularity is a *classpect*. We demonstrated the benefits of the unified language design in the context of small but representative examples and case studies [10]. These demonstrations did provide some basis for further speculations about the language model's potential to solve large-scale problems. The representative examples, however, does not provide direct evidence that the unified model is beneficial beyond the challenge problems and the case studies. The concerns to which we have applied the unified model are largely component integration concerns, in that they coordinate the behavior of two or more components. Component integration concerns are extremely important class of concerns; however, they are not the only crosscutting concerns aspect-oriented programming addresses.

In earlier work, Hannemann *et al.* [11] described the crosscutting concerns in the OO implementations of the Gang-of-Four (GoF) patterns and showed that an AspectJ-based solution modularizes these concerns. They argued that the concerns represented

by these patterns are scattered and tangled across several classes in the software system that play various roles in the patterns's implementation. Hannemann and Kiczales described an aspect-oriented implementation of these patterns with the intent to modularize these crosscutting concern.

The main contribution of this paper[1] is a case study in a similar vein, which considers the implementation of the Gang-of-Four (GoF) design patterns [1] in the Eos language to analyze the effect of new programming language constructs on these implementations. GoF design patterns are design structures commonly occurring in and extracted from real software systems. The benefits observed in the context of these models could be —to some extent— extrapolated to modularity benefits that might be perceived in real systems.

Three properties of this evaluation make it more valuable from an empirical standpoint. First, GoF design patterns are standard well-documented design structures. Selecting a standard problem for evaluation allows others to reproduce the results independently. Second, a prior implementation of these patterns in AspectJ is available. Having a prior independent implementation in the AspectJ language provides an opportunity to present a careful un-biased analysis. Third, we are not the first to argue that the AspectJ-based solution could be improved. Sakurai *et al.* [13] briefly observed that the type-level aspect-oriented implementation of the design patterns described by Hannemann *et al.* [11] exhibit the design problems and performance overhead of the form described by Rajan and Sullivan [7].

We evaluated the Eos implementations using the set of metrics proposed by Hannemann *et al.* [11] namely locality, reusability, composition transparency, and (un) pluggability. These metrics showed that Eos implementation has similar modularity properties that the AspectJ implementation enjoys. Our further evaluation also showed that for 7 out of 23 design patterns the Eos implementation was able to better realize the intent of the design patterns. Eos implementation was also more concise in terms of the line of code. For other 16 patterns, AspectJ and Eos implementation were the same. In other words, Eos constructs did not offer any additional improvements for these patterns.

The result that all GoF pattern implementations in Eos were at least as good as the pattern implementations in AspectJ, was not surprising since the Eos language model is a superset of the AspectJ language model. Moreover, the 7 GoF patterns, where the new Eos constructs did show benefits provides insight into their potential utility. Applying the language model to standard problems demonstrates that the benefits of the unified aspect language model are not limited to component integration concerns. Our assessments of the resulting designs provide evidence for the design structuring benefits of the Eos model, and the usability of the Eos language. In a nutshell, we contribute a demonstration of the immediate practical value of our conceptual work.

The rest of this paper is organized as follows. The next section gives background on aspect-oriented programming and the unified model. Section 3 describes the case study setting in detail. Section 4 describes the Eos implementation of the design patterns and compares it with the AspectJ implementation. Section 5 discusses related work, and Section 6 concludes.

[1] A previous version of this article appeared at PLOP 2007 [12].

2 Background

In this section, we briefly review the AspectJ[2] and unified language model embodied by Eos[3]. The focus is on their key differences. The AspectJ language model is described in detail by Kiczales *et al.* [2]. The unified language model is described in detail by Rajan *et al.* [3].

2.1 The AspectJ Language Model

In this subsection, we will review basic concepts in the AspectJ model. AspectJ [2] is an extension to Java [14]. Other languages using AspectJ's model include AspectC++ [15], AspectR [16], AspectWerkz [17], AspectS [18], Caesar [19], etc. While Eos [7] is not AspectJ-like, it is in the broader class of Pointcut-Advice-based AO languages [20]. The central goal of such languages is to enable the modular representation of crosscutting concerns, including the representation of concerns conceived after the initial system design. The programs in these languages are typically developed in two phases [21]. The concerns that can be modularized using the traditional object-oriented modularization techniques are put in classes. Aspects then modularize the crosscutting concerns by advising these classes. Differentiating between classes and aspects makes these languages asymmetric.

```
1  class TracedClass{
2    public void TracedMethod(){
3       System.out.println("In␣Traced␣Method");
4    }
5  }
6  aspect Tracing {  // An aspect
7   pointcut tracedExecution(): //A pointcut
8    execution(public void TracedMethod());
9   before(): tracedExecution() {//A before advice
10    System.out.println("Before␣Traced␣Method");
11  }
12  after(): tracedExecution() {//An after advice
13    System.out.println("After␣Traced␣Method");
14  }
15   Object around(): tracedExecution() {
16    System.out.println("Around␣Traced␣Method␣-␣1");
17    proceed(); //Proceeding the join point
18    System.out.println("Around␣Traced␣Method␣-␣2");
19  }
20 }
21 Output Trace:
22 Before Traced Method
23 Around Traced Method - 1
24 In Traced Method
25 Around Traced Method - 2
26 After Traced Method
```

Fig. 1. A Simple Example Aspect

[2] The language manual and compiler for AspectJ is available from http://www.eclipse.org/aspectj as of this writing.

[3] The language manual and the compiler for Eos is available from http://www.cs.iastate.edu/∼eos as of this writing

These languages add five key constructs to the object-oriented model: join points, pointcuts, advice, inter-type declarations, and aspects. A simple example is shown in Figure 1 to make these concepts concrete. The **`aspect`** `Tracing`, modifies the behavior of the **`class`** `TracedClass` *before*, *after*, and *around* certain selected execution events exposed to such modification by the semantics of the programming language. These events are called join points. The execution of the method `TracedMethod` in the **`class`** `TracedClass` is an example of a join point.

A pointcut is a predicate that selects a subset of join points for such modification — here, execution of the method `TracedMethod` in the **`class`** `TracedClass`. An advice is a special *method-like* construct that effects such a modification at each join point selected by a pointcut. An aspect is a class–like module that uses these constructs to modify behaviors defined by the classes of a software system. In the example, we have three different types of advice in the aspect: before, after, and around that affect the behavior of the method `TracedMethod` as shown in the output trace in the figure.

Like classes, aspects also support data abstraction and inheritance, but they do differ from classes in the following ways.

- Aspects can use pointcuts, advice, and inter-type declarations. In this sense, they are strictly more expressive than classes.
- 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.
- Although aspects can advise methods with fine selectivity, they can select advice bodies to advise only in coarse-grained ways.

2.2 The Unified Language Model

Rajan *et al.* addressed the limits of aspects in a new language model that unifies aspects and objects as follows[4] [3].

- It unifies aspects and classes as *classpects*. A *classpect* has all the capabilities of classes, all of the essential capabilities of aspects in AspectJ–like languages, and the extensions to aspects needed to make them first class objects.
- The unified model eliminates advice in favor of using methods only, with a separate and explicit join-point-method binding construct.
- It supports a generalized advising model. To the usual object-oriented mechanisms of explicit or implicit method call and overriding based on inheritance, the unified model adds implicit invocation using before and after advice, and overriding using around advice.

To make these points concrete we revisit the example presented in the previous section in Figure 2. A classpect (lines 1-8), similar to the aspect in the previous section, declares a pointcut (lines 2-3) to select the execution of any method, exactly as in AspectJ. It then composes the pointcut with the **`within`**`(Trace)` pointcut expression to exclude its own methods, to avoid recursion. A static binding (line 4)

[4] This language model has also been adopted by languages like Ptolemy [22, 23].

```
1 class Tracing {
2  pointcut tracedExecution():
3   execution(* *(..))&& !within(Trace);
4  static before tracedExecution(): Trace();
5  public void Trace() {
6   /* Trace the methods */
7  }
8 }
```

Fig. 2. A Simple Example Classpect

binds the method `Trace` (lines 5-7) to execute before all join points selected by the **`pointcut`** `tracedExecution`. Note that, by statically binding, join points in all instances are affected. A non-static binding would bind to instances selectively. The key difference in this implementation is that all concerns are modularized as classpects and methods. The crosscutting concerns, however, use bindings to bind the method containing the implementation of the crosscutting concerns to join points. In the next section, we demonstrate that this unification shows benefits in the Gang-of-Four design patterns.

3 Case Study Setting

The results described in this work are based on a case study that we conducted to compare the implementation of 23 GoF patterns in AspectJ and Eos. The AspectJ implementation that we compared against is described in detail by Hannemann and Kiczales [11]. The implementation provided by Hannemann and Kiczales uses these design patterns in the context of small examples. The first step was to translate the AspectJ implementation to Eos, adjusting for minor differences in underlying languages (Java and C#). In this step, no change in the Eos implementation was attempted. Eos language is a superset of AspectJ, which means that all constructs were available in the host language. Unlike the study setting by Garcia *et al.* [24], we did not modify the implementation examples. This translation was done by hand.

In the next step, patterns in Eos were improved by utilitizing the new mechanisms, namely: unified language model, instance-level advising, and first-class aspect instances.

In earlier work, we showed that translation of programs from the AspectJ language model to their equivalent in the proposed unified model is possible [25]. This translation does not require any non-local changes. The design space of the modular solutions using unified model is essentially a super set of the design space of the modular solutions in the AspectJ-like language model. Based on that observation one would expect to be able to translate the implementation of design patterns from AspectJ to Eos, which is indeed the case. The pattern implementations provided by Hannemann *et al.* can be translated to Eos without any non-modular changes. By non-modular changes we mean changes that are localized within a module boundary, but rather fragmented and spread across the system. The translation of pattern protocols is straightforward. The examples provided with the pattern implementation, however, are heavily dependent upon the platform. As a result, their translation is not quite direct owing to the host framework differences. AspectJ operates on Java Virtual Machine (JVM), whereas Eos operates on .Net Framework.

The translation rules from AspectJ model to Eos model do not require any non-modular changes, preserving the modularity of AspectJ based solution, and implying that the results in the Hannemann and Kiczales [11] may apply to the Eos implementation as well. To measure that we used the four metrics applied by Hannemann and Kiczales, namely: locality, reusability, composability, and (un)pluggability. We also applied some metrics used by Garcia *et al.* [24], such as Line of Code (LOC), to measure size. We also used an additional metric Close Match to Pattern Intent (CMPI) that evalutes to true for a pattern implementation, if the intent of the implementation closely matches the pattern specification.

4 Design Patterns

In this section, we will describe some patterns in which we noticed major improvements compared to other patterns. These improvements were in metric (CMPI) and (LOC), while other metric values largely remained unchanged.

4.1 Observer Pattern

The intent of the observer pattern is to define a one-to-many dependency between objects so that on an object's state change, all dependents are notified and updated automatically [1, p.293]. The AspectJ implementation divides the pattern implementation into two parts: parts that are common to all instantiations of the pattern and parts specific to an instantiation. The implementation abstracts the common part as a reusable **`aspect`** `ObserverProtocol` as shown in Figure 3. It provides an **`abstract pointcut`** `subjectChange` (lines 23–24) to represent observable state change of the subject. A concrete observer implementation defines this pointcut. The implementation also provides an abstract method; `update` (lines 24–25) to be redefined in concrete observers to implement the observer's logic.

The AspectJ language model does not fully support aspect instantiation and selective advising of object-instances [6]. In the Observer pattern, an instance of Observer needs to selectively advise instances of Subject. To emulate instance-level advising using type-level aspects, Hannemann and Kiczales's implementation of the Observer protocol needs to manipulate instances of participants. To be able to do so without coupling the `ObserverProtocol` with participants, it defines two new inner interfaces that are introduced by the concrete observers into participants so that `ObserverProtocol` can manipulate them. The pattern's implementation therefore modifies the implementation of the participants using `declare parents` constructs such that they not implement two new interfaces subjects (line 2) and observers (line 3). An example of the `declare parents` construct is shown in Figure 6.

The implementation also keeps a `HashMap` (line 4) of observers corresponding to an instance of the subject. It provides methods to `add` (lines 17–19) and `remove` (lines 20–22) observers corresponding to a subject. It also provides methods to retrieve observers for a subject (lines 5–16). The observer protocol logic is implemented by the advice (line 26–31). This advice is invoked by each instance of the class being advised, even if no observer is observing the instance. On being invoked, the advice

```
1 public abstract aspect ObserverProtocol{
2  protected interface Subject {}
3  protected interface Observer {}
4  private WeakHashMap perSubjectObservers;
5  protected List getObservers(Subject s){
6   if(perSubjectObservers == null){
7    perSubjectObservers = new WeakHashMap();
8    }
9   List observers =
10    (List)perSubjectObservers.get(s);
11    if ( observers == null ){
12     observers = new LinkedList();
13     perSubjectObservers.put(s, observers);
14     }
15    return observers;
16    }
17  public void addObserver(Subject s, Observer o){
18    getObservers(s).add(o);
19    }
20  public void removeObserver(Subject s,Observer o){
21    getObservers(s).remove(o);
22    }
23  protected abstract pointcut
24    subjectChange(Subject s);
25  protected abstract void
26    update (Subject s, Observer o);
27  after(Subject s): subjectChange(s){
28    Iterator iter = getObservers(s).iterator();
29    while ( iter.hasNext()){
30     update (s, ((Observer)iter.next()));
31     }
32   }
33 }
```

Fig. 3. Observer in AspectJ

looks up the invoking instance and retrieves the list of observers. It then iterates through the list to invoke each observer. In summary, the AspectJ implementation tangles the instance-level advising and instance-emulation concern [7] with the observer pattern concern. The need for roles and for maintaining a hash map are examples of design-time overheads incurred due to the asymmetry of the language model.

```
1 public abstract class ObserverProtocol {
2  protected abstract pointcut
3   subjectChange();
4  protected abstract void
5   update ();
6  after subjectChange(s): update(Subject s);
7 }
```

Fig. 4. Observer in Eos 78% Smaller

The AspectJ implementation of the observer pattern is localized, reusable, compositionally transparent, and (un) pluggable. The Eos implementation mimics the implementation strategy by similarly partitioning the pattern implementation into abstract classpect ObserverProtocol and concrete realization of observers inheriting from this classpect. The abstracted pattern is shown in Figure 4.

The Eos implementation does not tangle emulation concern with observer protocol concern. It clearly abstracts the behavior of the pattern. It clearly (and more concisely) conveys the intent of the pattern, which is to update an observer when a sub-

ject changes. The binding (Line 6) states that after the join points selected by the **abstract pointcut** SubjectChange, the method Update should be called. All the interfaces and additional code required to emulate instance-level behavior is not necessary in the Eos implementation. With respect to the metrics used by Garcia *et al.* [24], Eos implementation of the Observer pattern achieves nearly 78 percent reduction in LOC (Line of Code) of the observer protocol concern without increasing the complexity of the remaining concerns. Each line in Eos implementation corresponds to a line in AspectJ implementation. The Eos implementation, however, does not require the emulation code in the pattern implementation since it utilizes the instance-level constructs in the language. This implementation is also localized, reusable, compositionally transparent, and (un) pluggable. It also decreases the Number of Attributes (NOA) [24] of the Observer protocol concern to zero from one.

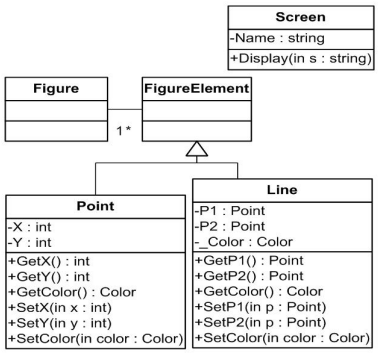

Fig. 5. The Figure Element System

```
1 public aspect ColorObserver
2   extends ObserverProtocol{
3  declare parents: Point implements Subject;
4  declare parents: Screen implements Observer;
5  protected pointcut subjectChange(Subject s):
6   call(void Point.setColor(Color))&& target(s);
7  protected void update(Subject s, Observer o){
8    ((Screen)o).display("Color->Screen.");
9  }
10 }
```

Fig. 6. Color Observer Implementation in AspectJ

```
1 public class ColorObserver
2   : ObserverProtocol{
3  Point p; Screen s;
4  public ColorObserver(Point p, Screen s){
5   addObject(p); this.p=p; this.s=s;
6  }
7  override pointcut subjectChange():
8   execution(void Point.setColor(Color));
9  public override void update(){
10  s.display("Color->Screen update.");
11 }
12 }
```

Fig. 7. Color Observer Implementation in Eos

Moreover, the composition of the participants into observing relationships becomes more intuitive in the Eos implementation. To illustrate let us consider the example system presented by Hannemann *et al.* shown in Figure 5. In this example, a figure element system, we have two potential subjects a point, a line, and an observer screen. Instances

of the **class** Screen observe change in the color and co-ordinates of instances of the **class** Point. A subject-observer relationship between Point and Screen in which Screen instance observes change in color of the Point instance is shown in Figure 6. The ColorObserver relationship implementation does not clearly communicate the specification that it involves two object instances, an observer instance and an observed instance. Instead, this part of the specification is hidden in the parent class, ObserverProtocol. Understanding the behavior of the parent class is necessary to deduce how to put two objects instances, a Screen and a Point into a color observing relationship. As a result, even though the pattern protocol achieves a physical separation of code, separation of concern between parent ObserverProtocol and the relationship ColorObserver is not achieved.

The implementation of the same ColorObserver relationship is shown in Figure 7. The implementation clearly represents the intent of the pattern. By declaring a constructor that takes a point and a screen as an argument, it depicts the observing relationship between these two entities. Compared to the AspectJ implementation where relationship instances are emulated implicitly using hash tables, in the Eos implementation one explicit instance of ColorObserver exists for each point and class instance that participate in the observing relationship.

The Eos implementation does not require code for instance-level weaving emulation [7]. It represents the ColorObserver as a class containing an instance variable screen to store reference to the observer Screen instance and the subject Point instance (line 3), a constructor (lines 4–6), definition of what it means for a subject to change (lines 7–8) and method to update the observer (Line 9–11). For comparison, the listing in Figure 8 shows the key parts of the client code. AspectJ code is preceded by the comment *AspectJ* and Eos code is preceded by the comment *Eos*.

```
/* Construct Point p and Screens s1, s2 here */
/* Begin AspectJ Code */
ColorObserver.aspectOf().addObserver(p, s1);
ColorObserver.aspectOf().addObserver(p, s2);
/* Begin Eos Code */
ColorObserver cobs1 = new ColorObserver(p, s1);
ColorObserver cobs2 = new ColorObserver(p, s2);
```

Fig. 8. Observer Clients in AspectJ and Eos

The client code in Figure 8 shows that Eos achieves modular component composition. As opposed to calling a special **aspectOf** method on ColorObserver module and then calling addObserver on that module, now subjects and observers are composed by creating new instances of observing relationship. In summary, Eos implementation of the Observer pattern closely mimics the design compared with the AO implementation of the Observer pattern. This straightforward mapping from the design to the implementation results from the instantiation and instance-level advising features of a *classpect*. A side effect of eliminating the need for additional design time overhead to emulate instantiation and instance-level weaving is a reduction in the size and complexity of the implementation.

4.2 Chain of Responsibility Pattern

The intent of the Chain of Responsibility pattern is to avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. The idea is to chain the receiving objects and pass the requests along the chain until an object handles it. [1, p.223].

Similar to the Observer implementation, the AspectJ implementation of the Chain of Responsibility pattern provided by Hannemann *et al.* divides the pattern implementation into two parts: parts that are common to all instantiations of the pattern and parts specific to an instantiation. The implementation abstracts the common part as a reusable **aspect** `CORProtocol` as shown in Figure 9. The aspect introduces two roles, `Handler` (line 2) and `Request` (line 3) corresponding to a handler and a request as interfaces. It further uses the inter-type declaration feature to introduce `acceptRequest` (lines 18–20) and `handleRequest` (line 21) methods into the **interface** `Handler` as default behavior. The aspect provides an **abstract pointcut** `eventTrigger` (lines 22–23) to represent the event that is to be handled by the chain. A concrete implementation defines this pointcut.

```
1 public abstract aspect CORProtocol{
2  protected interface Handler {}
3  protected interface Request {}
4  private WeakHashMap successors = new WeakHashMap();
5  protected void receiveRequest
6   (Handler h, Request r){
7   if(h.acceptRequest(r)){
8     h.handleRequest(r);
9    }else{
10    Handler s = getSuccessor(h);
11   if(s == null){
12     throw new CORException("End␣of␣chain␣reached)");
13    }else{
14     receiveRequest(s, r);
15    }
16   }
17  }
18  public boolean Handler.acceptRequest(Request r){
19   return false;
20   }
21  public void Handler.handleRequest(Request r){}
22  protected abstract pointcut eventTrigger
23   (Handler h, Request r);
24  after(Handler h, Request r):
25   eventTrigger(h, r){
26    receiveRequest(h, r);
27   }
28  public void setSuccessor
29   (Handler h, Handler s){
30    successors.put(h, s);
31   }
32  public Handler getSuccessor(Handler h){
33   return ((Handler) successors.get(h));
34   }
35 }
```

Fig. 9. Chain of Responsibility in AspectJ

The **aspect** `CORProtocol` keeps a `HashMap` (line 4) to keep track of the successors of a given handler. It provides methods to set the successor of a handler (lines 28–31) and to retrieve the successors of a given handler (lines 32–34). The main logic

of the chain of responsibility pattern is in method `receiveRequest` (lines 5–17). This method first checks whether a supplied handler can handle this request. If not, it tries the successors of the handlers. If there is a successor, it passes the request to the successor. Otherwise, it throws an exception to signify end of the chain (lines 11–13). The method `receiveRequest` is triggered by a delegating advice (line 24–27). This advice is triggered at the join points selected by the **`pointcut`** `eventTrigger` (line 22–23).

Similar to the AspectJ implementation, the Eos implementation shown in Figure 10 defines two roles, `Handler` (line 2) and `Request` (line 3) corresponding to a handler and a request as interfaces. It further uses the inter-type declaration feature to introduce the methods `acceptRequest` (lines 5–7) and `handleRequest` (line 8) into the interface `Handler` as default behavior. In addition, it also introduces a method `receiveRequest` to receive requests in all handlers (lines 9–16). In the AspectJ implementation the **`aspect`** `CORProtocol` had a similar method (lines 5–17 in Figure 9). The difference is now in the implementation technique that can be realized due to the new instance-level advising features in Eos [7]. The method `receiveRequest` in the Eos implementation first checks whether the current `Handler` object can handler the request, otherwise it throws an exception of type `COREception`.

```
1 public abstract class CORProtocol{
2  protected interface Handler {}
3  protected interface Request {}
4  introduce in Handler{
5  public bool acceptRequest(Request r){
6   return false;
7  }
8  public void handleRequest(Request r){}
9  public void receiveRequest
10  (Request r){
11   if (acceptRequest(r)){
12    handleRequest(r);
13    }else{
14    throw new COREception("End␣of␣chain␣reached");
15   }
16  }
17 after throwing(COREception)
18     execution(Handler.receiveRequest(..))
19     && args(r): receiveRequest(Request r);
20 public void setPredecessor(Handler h){
21  addObject(h);
22  }
23 }
24 protected abstract pointcut eventTrigger
25  (Handler h, Request r);
26 after eventTrigger(h, r):
27 TriggerRequest(Handler h, Request r);
28 public void TriggerRequest
29  (Handler h, Request r){
30    handler.receiveRequest(r);
31  }
32 public void setTrigger(Handler h){
33  addObject(h);
34  }
35 }
```

Fig. 10. Chain of Responsibility in Eos

In addition to the method `receiveRequest`, the classpect `CORProtocol` introduces a binding (lines 17–19) and another method `setPredecessor` (lines 20–22). This binding selects the join point execution of the method `Handler.receiveRequest` when an exception is being thrown and calls the method `receiveRequest` on the current `Handler` instance. Note that this binding is a non-static binding and that would not be easy to simulate in AspectJ. A non-static binding affects object instances selectively. The effect of this binding is to call the method `receiveRequest` on the current handler if the previous handler threw an exception. The method `setPredecessor` is provided to set a handler's predecessor. It calls the implicit method `addObject` to advise the predecessor instance. The effect of calling the implicit method `addObject` is to register the bound methods in the `Handler` instance with the join points in the object instance supplied as argument. The Eos implementation thus eliminates the need to keep a `HashMap` to represent the chain of responsibility, instead the chain is now implicit in the advising structure. The code for hash table lookup for each successor invocation is also eliminated. In addition, the Eos implementation also allows events to be triggered on an instance-level basis, which required complex emulation code when written in AspectJ. We will describe this difference in more detail in the context of a concrete example.

```
1 public aspect ClickChain extends CORProtocol{
2  declare parents: Frame implements Handler;
3  declare parents: Panel implements Handler;
4  declare parents: Button implements Handler;
5  declare parents: Click implements Request;
6  protected pointcut eventTrigger
7     (Handler h, Request r):
8    call(void Button.doClick(Click)) &&
9    target(h) && args(r);
10 public boolean Button.acceptRequest(Request r){
...
17 }
18 public void Button.handleRequest(Request r){
...
20 }
...
43 }
```

Fig. 11. Concrete Aspect ClickChain in AspectJ

To illustrate the difference in the implementation technique let us look at the example system presented by Hannemann et al. This example system has three type of GUI objects *Buttons*, *Panels*, and *Frames*. The objective is to handle the request `Button.Click` and propogate it through the chain Button-to-Panel-to-Frame if required. The concrete implementation of this example system in AspectJ declares another **`aspect`** `ClickChain` (Figure 11) that inherits from the **`aspect`** `CORProtocol`. This concrete aspect modifies the inheritance hierarchy of `Button`, `Panel`, and `Frame` to include the **`interface`** `Handler` and the inheritance hierarchy of the event `Click` to include the **`interface`** `Request`. It provides concrete implementation of the methods `acceptRequest` and `handleRequest` for these classes. The concrete aspect `ClickChain` also provides a concrete definition for the abstract pointcut `eventTrigger`. The effect of defining this pointcut is that for all instances of the button class, whenever the method `doClick` is called the

delegating advice (line 24–27 in Figure 9) will be invoked. However, this invocation is desired only when the method `doClick` is called in the context of a specific button instance that is the supplier of the request. Thus commitment to type-level advice invocation fails to achieve the desired objective in this case.

```
1 public class ClickChain : CORProtocol{
2  declare parents: Frame: Handler;
3  declare parents: Panel: Handler;
4  declare parents: Button: Handler;
5  declare parents: Click: Request;
6  protected pointcut eventTrigger
7   (Handler h, Request r):
8   execution(void Button.doClick(Click))&&
9    this(h) && args(r);
10 introduce in Button {
11  public bool acceptRequest(Request r){
...
18   }
19  public void handleRequest(Request r){
...
21   }
22  }
...
49 }/*6 more lines for inter-type declarations*/
```

Fig. 12. Concrete Classpect ClickChain in Eos

The Eos implementation (Figure 12) of this example system is similar. The concrete classpect `ClickChain` in Eos also modifies the inheritance hierarchy of `Button`, `Panel`, `Frame`, and `Click`, provides concrete implementation of the methods `acceptRequest` and `handleRequest` for these classes, and a concrete definition for the abstract pointcut `eventTrigger`. However, the effect of defining this pointcut is that for a specified instance of the button class, whenever the method `doClick` is called the delegating method `TriggerRequest` (line 24–27 in Figure 10) will be invoked. This instance is specified using the method `SetTrigger` defined in the abstract classpect `CORProtocol`. The client code for AspectJ and Eos shows this difference clearly. For comparison, the listing in Figure 13 shows the key parts of the client code. The common code is preceeded by *Common:*, the AspectJ code is preceded by *AspectJ:*, and the corresponding Eos code is preceded by *Eos:*.

```
/* Common Code Begins */
Frame frame = new Frame(...);
Panel panel = new Panel(...);
Button button1 = new Button(...);
Button button2 = new Button(...);
/* AspectJ Code Begins */
ClickChain.aspectOf().
   setSuccessor(button1, panel);
ClickChain.aspectOf().
   setSuccessor(panel, frame);
/* Eos Alternative Code Begins */
ClickChain chain = new ClickChain();
chain.SetTrigger(button1);
panel.SetPredecessor(button1);
frame.SetPredecessor(panel);
```

Fig. 13. Chain of Responsibility Clients in AspectJ and Eos

The client code creates an instance of `Frame`, an instance of `Panel`, and two instances of `Button`. In the AspectJ code, the instance `panel` is set as a successor to the `Button` instance `button1` and the `Frame` instance `frame` is set as a successor to the `Panel` instance `panel`. In this implementation, on each `button click`, complete chain of responsibility pattern is executed for both buttons because the aspect *ClickChain* ends up advising both instances of the button. The objective is to execute the chain of responsibility for only the `Button` instance `button1`.

In the Eos code, first an instance of the `ClickChain` is created. The method `SetTrigger` is called on this instance with the `Button` instance `button1` as argument to set the event *method call doClick on button1* as the request trigger. The `Button` instance `button1` is then set as predecessor of the `Panel` instance `panel` in the chain of responsibility by calling the method `SetPredecessor` on the `Panel` instance `panel`.

The effect is that the binding in the instance `panel` bounds the method `receiveRequest` to execute in the context of the instance `panel` whenever the method `receiveRequest` executing in the context of the instance `button1` throws an exception of type `CORException`, i.e. whenever `button1` is not able to accept a request. Similarly, the `Frame` instance `frame` is set as the predecessor of the `Panel` instance `panel` in the chain of responsibility. No instance is setting the `Frame` instance `frame` as a predecessor, as a result if this instance is unable to handle the request the exception `CORException` is finally thrown to denote unhandled requests.

The implementation strategy in Eos completely realizes the intent of the chain of responsibility pattern without the need to maintain the chain of successors in a `HashMap`. The chain is implicitly maintained in the recursive advising structure in the classpect `Handler`. A key property of this design structure is that an instance of a classpect advises another instance of the same classpect. This benefit is observed as a result of the generalized advising mechanism provided by our unified aspect language model [3]. In the AspectJ language model, realization of such design structures requires aspect-instance emulation and instance-level weaving emulation, adding addition complexity to the solution. In addition, the Eos implementation strategy also overcomes the problem with the AspectJ implementation where the intention is to advise one `Button` instance but the `ClickChain` aspect ends up advising all `Button` instances.

4.3 Mediator Pattern

The intent of the Mediator pattern is to define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently [1, p.273].

Similar to the Chain of Responsibility and the Observer pattern's implementation, the AspectJ implementation provided by Hannemann and Kiczales divides the pattern implementation into two parts: parts that are common to all instantiations of the pattern and parts specific to an instantiation. The implementation abstracts the common part as a reusable aspect `MediatorProtocol` as shown in Figure 14. It provides an abstract pointcut `change` (lines 14–15) to represent state change of the colleagues. A concrete mediator implementation defines this pointcut. The implementation provides an abstract method; `notify` (lines 19–20) to be redefined in concrete mediators to implement the

```
1  public abstract aspect MediatorProtocol{
2    protected interface Colleague{}
3    protected interface Mediator{}
4    WeakHashMap ColToMed = new WeakHashMap();
5    Mediator getMediator(Colleague colleague){
6    Mediator mediator = (Mediator)
7     ColToMed.get(colleague);
8    return mediator;
9   }
10  public void setMediator(Colleague colleague,
11    Mediator mediator){
12   ColToMed.put(colleague, mediator);
13  }
14  protected abstract pointcut
15   change(Colleague colleague);
16  after(Colleague c): change(c){
17   notify (c, getMediator(c));
18  }
19  protected abstract void
20   notify (Colleague c, Mediator m);
21 }
```

Fig. 14. Mediator in AspectJ

notification logic. The aspect `MediatorProtocol` also keeps a `HashMap` (line 4) to keep track of the colleague instances that are being mediated by a mediator instance. It provides methods to **`set`** (lines 10–13) and **`get`** (lines 5–9) mediator corresponding to a colleague.

This implementation does not work in cases where a colleague instance is participating in more then one mediating relationship. Let us assume a scenario where a colleague instance `c` is involved in two mediating relationships, `m1` and `m2`. To put the colleague in the mediating relationships the method `setMediator` will call with parameters `(c, m1)` and `(c, m2)` in any order. The method setMediator in turn will call the method `put` on `WeakHashMap ColToMed` with `Colleague c` as the key. When these calls are completed, the last mapping from colleague to mediator remains in the `WeakHashMap` as it replaces the value supplied in the old mapping.

```
1  public abstract class MediatorProtocol{
2   protected abstract pointcut
3    change();
4   after change() : notify();
5   protected abstract void
6    notify();
7  }
```

Fig. 15. Mediator in Eos, 66% Smaller.

Like the Observer pattern, the Eos implementation shown in Figure 15 does not tangle the emulation concern with the mediator protocol concern, resulting in a modular implementation of the mediator protocol. The implementation clearly represents the behavior of the pattern, decreasing the conceptual gap between the specification and implementation of the pattern. It clearly (and only) conveys the intent of the pattern. The intent of the pattern is that after change in a colleague mediator notifies the changes to other colleagues. The binding states that after the join points selected by the abstract pointcut `change`, the method `notify` should be called. No interfaces and additional code required to emulate instance-level advising is required.

4.4 Decorator

The intent of the decorator pattern is to attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality [1]. Decorator pattern implementation in AspectJ exposed the limitations of advising all instances of classes, a typical advising model in the AspectJ language. In the AspectJ implementation, shown in Figure 16 the decorator **aspect** `BracketDecorator` decorates all the call to the method `print` of the class `ConcreteOutput` such that a Bracket is printed around the string.

```
public aspect BracketDecorator {
 protected pointcut printCall(String s):
   execution(public void ConcreteOutput.print(String))
    && args(s);
 void around(String s): printCall(s) {
  s = "[" + s + "]";                                    // Decorates the string
  proceed(s);
 }
}
```

Fig. 16. Decorator in AspectJ

This version has two limitations. First, after this aspect is inserted in the system, all instances of the **class** `ConcreteOutput` are decorated. That may not be a desirable effect. One may want to decorate only certain instances of ConcreteOutput. Second, this version does not allows for dynamic composition of decorators, which is one of the most useful uses of the decorator pattern.

```
public class BracketDecorator {
 public BracketDecorator(){}
 ...
}
ConcreteOutput o = new ... //Creating object
BracketDecorator dec = new ... //Creating decorator
dec.addObject(o); //Adding decorator to object
dec.removeObject(o);//Removing decorator from object
```

Fig. 17. Decorator in Eos (Pointcut and advice similar to Figure 16)

On the other hand, the Eos decorator implementation uses instance-level advising to affect only the specified instances of `ConcreteOutput`. In the construct, the classpect `BracketDecorator` starts decorating the `ConcreteOutput` instance by calling the implicit method `addObject` on that instance and stops decorating by calling complementary method `removeObject`. Eos version thus does allow for dynamic composition of decorators.

4.5 Bridge Pattern

It is worth discussing and analyzing a pattern, where we did not get any design structuring benefits from using Eos. In this case, AspectJ's implementation also did not have any design structuring benefits. The intent of this pattern is to separate an abstraction

from its implementation, so that they can be varied independently. The implementation strategy followed by the AspectJ (and that we followed in Eos as well) is to separate out common members from various versions of abstraction into a separate aspect, represent the abstraction as an interface, specify the variations as concrete classes that implement that interface, and have the aspect insert the commonalities into theses classes. Similar, design structuring benefits would have been obtained by representing common parts of the abstraction as an abstract OO class, therefore, the AO solution did not advance over the OO solution. Eos solution was also similar to AspectJ Solution.

4.6 Other Patterns

We observed similar design structuring benefits in the Eos implementation of the Composite, Command, Strategy, and Singleton patterns. For example, in case of the Composite pattern, the AspectJ implementation represents the Composite-Child containment relationship using a visitor pattern that triggers operations on the components of a composite. In the Eos implementation, this pattern is not needed. Instead, the Composite-Child containment relationship is implicitly represented as an advising relationship. In addition, the emulation code in the CompositeProtocol aspect to keep a list of children is not needed. Instead, first class aspect instances model the composition relationships in the system, which more clearly represents the design intent in the runtime structure.

Similarly, in case of the Strategy pattern the emulation code to store the relationship between a strategy and it's context is replaced by implicit instance-level advising relationship. In case of other 16 design patterns, the Eos implementation was the same as the AspectJ implementation, so we are no worse off then before in these cases as well.

4.7 Analysis

The implementation of design patterns in Eos showed that the unified language model eliminates the need for emulation strategies in 7 patterns making the resulting implementation much simpler. The simplification in these cases is the result of including instantiation and instance-level advising as language features, unifying aspect and class, and unifying method and advice. The composition of participants and patterns is also much more intuitive now.

Figure 18 shows the results of the case study for the patterns that we considered previously in the work and others. The rows in the figure represent the patterns and the column represents the metrics that we considered for this case study.

In particular, we considered four metrics used by Hannemann *et al.*namely: locality, reusability, composition transparency, and (Un) pluggability. They used these metrics to measure the modularity properties of the pattern implementations. Informally, a pattern implementation has locality if the code for the pattern is syntactically localized in a module. It is reusable, when most of the implementation can be imported in other application without much effort. It is composition transparent, if using one pattern implementation in an application does not hinder the option to use other patterns and finally, if it (Un) pluggable, if the pattern implementation can be put in the application and taken out with ease.

	Modularity Properties				CMPI	Size	Constructs responsible for major improvements					
Pattern Name	Locality	Reusability	Composition Transparency	(Un) pluggability		LOC Ratio (Eos / AspectJ)	after advice	around advice	inter-type declarations	declare parents	first-class aspect instances	instance-level advising
Facade	Similar for Java, AspectJ and Eos				A/E	1	-	-	-	-	-	-
Abstract Factory	Similar for Java, AspectJ and Eos				A/E	1	-	-	-	-	-	-
Bridge	Similar for Java, AspectJ and Eos				A/E	1	-	-	-	-	-	-
Builder	Similar for Java, AspectJ and Eos				A/E	1	-	-	-	-	-	-
Factory Method	Similar for Java, AspectJ and Eos				A/E	1	X	-	-	-	-	-
Interpreter	Similar for Java, AspectJ and Eos				A/E	1	-	-	-	-	-	-
Template Method	X	-	-	X	A/E	1	-	-	-	-	-	-
Adapter	X	-	X	X	A/E	1	-	-	-	X	-	-
State	X	-	-	X	A/E	1	X	-	-	-	-	-
Decorator	X	-	X	X	E	1	X	X	-	-	X	X
Proxy	X	-	X	X	A/E	1	-	X	-	X	-	-
Visitor	X	X	X	X	A/E	1	-	-	X	X	-	-
Command	X	X	X	X	E	0.31	X	-	-	X	X	X
Composite	X	X	X	X	E	0.82	-	-	X	X	X	-
Iterator	X	X	X	X	A/E	1	-	-	-	-	-	-
Flyweight	X	X	X	X	A/E	1	-	-	X	X	-	-
Memento	X	X	X	X	A/E	1	-	-	-	X	-	-
Strategy	X	X	X	X	E	0.58	-	X	-	X	X	X
Mediator	X	X	X	X	E	0.34	X	-	-	X	X	X
Chain of Responsibility	X	X	X	X	A/E	0.84	X	-	X	X	-	X
Prototype	X	X	X	X	A/E	1	-	-	X	X	-	-
Singleton	X	X	-	X	A/E	1.41*	-	X	-	X	X	-
Observer	X	X	X	X	E	0.22	X	-	-	X	X	X

X in a cell represents a property holds for Eos Implementation of the Pattern.
CMPI means close match to pattern intent. A/E means that both AspectJ and Eos implementation match the intent.
Either A or E means that only the AspectJ or Eos's implementation closely matches a pattern's intent.
*: an alternative implementation of singleton utilizing first-class aspect instances.

Fig. 18. Qualitative Analysis Results for Java, AspectJ [11] and Eos

We also studied the size of the pattern implementation in both Eos and AspectJ. In particular, we report on the size of the reusable part of the pattern in Figure 18. All lines of code were measured after ignoring comments and ignore closing braces that often consitute a single line by themselves.

The third metrics we considered is Close Match to Pattern Intent (CMPI). This metric can take true or false value. By close match to pattern intent, we mean whether the implementation in the language closely matches the original intent of the pattern. For example, the intent of Class Adapter pattern is to convert the interface of a class into another interface clients expect so that classes can work together that couldn't otherwise because of incompatible interfaces. The implementation of the Class Adapter pattern in a language without multiple inheritance (such as Java) will have CMPI value of false. In other words, CMPI demonstrate the naturalness of the pattern representation in the language.

Finally, we also studied the aspect-oriented language constructs and their role in enabling better representation of GoF patterns. In particular, we looked at the role of AspectJ specific constructs around advice, before advice, inter-type declaration, and

declare parent construct. Eos also has these constructs. In addition, Eos has features for creating aspect instances, and for advising other object instances on a selective basis.

The key observations from our case study are as follows:

- Implementations for Facade, Abstract Factory, Bridge, Builder, and Factory method remained almost the same for Java, AspectJ and Eos. This is because the primary purpose for these patterns is to provide a passive abstraction. There implementation is already well-modularized.
- The declare parents constructs played a key role in both AspectJ's and Eos's AO implementation of GoF patterns. This construct allows one to impose another type-hierarchy on an existing type hierarchy. In the implementation of GoF patterns, it was mostly used to impose roles on pattern participants.
- Most usages of the advice construct in the GoF pattern implementation turned out to be for creating relationships between objects rather than for creating relationship between classes. In order to mimic the relationships between objects, AspectJ implementation used a hash map to store and retrieve these relationships.
- Eos implementation and the instance-level advising feature was able to avoid the need to mimic the relationships between objects. Instead, it had a direct representation such that the design mirrored the runtime struture.
- In some cases, savings in terms of code size (LOC) were as much as 78 % in case of Eos.
- Eos implementation for six patterns namely, decorator, command, composite, strategy, mediator, and observer closely followed the original intent of the paer, whereas the AspectJ implementation did not. This is primarily because AspectJ did not provide a mechanism to efficiently represent relationships between pattern participant instances as advising relationships. The advising relationships between participant instances had to be emulated on top of the advising relations between participant classes.
- In most cases, improvement in Eos's implementation was due to the combination of instance-level advising and first-class aspect instances e.g. decorator, command, strategy, etc, although in some cases, e.g. composite and singleton, design structuring benefits were largely due to first-class aspect instances.

5 Related Work

Most closely related to this work is the evaluation of aspect-oriented implementation of the Gang-of-Four design patterns [1] conducted by Hannemann and Kiczales [11] and Garcia *et al.* [24]. Hannemann and Kiczales compared the object-oriented implementation in Java with aspect-oriented implementation in AspectJ using qualitative metrics. Garcia *et al.* [24] used a set of quantitative metrics to compare the object-oriented and aspect-oriented implementations. Our work compares the design structures realized in an aspect-oriented implementation in Eos with another aspect-oriented implementation in AspectJ.

The subject of this evaluation, the unified aspect language model [3], is related to AspectJ [2], AspectWerkz [17], and Caesar [19]. In at least one early version of

AspectJ, there was no separate aspect construct. In this version, the class was extended to support advice. To the best of our knowledge, the synthesis of OO and AO techniques achieved by our unified model was not present there. Advice bodies and methods were still separate constructs; and it is unclear to what extent advising as a general alternative to method invocation and implicit invocation was supported. In addition, flexible aspect instantiation and instance-level weaving were not supported. Rajan and Sullivan showed that first-class aspect instances and instance-level advising improved the modularization of integration concerns [4, 7]. This work reinforces our earlier findings.

Another closely related design is that of AspectWerkz [17]. The aim of AspectWerkz is to provide the expressiveness of AspectJ [2] without sacrificing pure Java and the supporting tool infrastructure. The solution is to use normal Java classes to represent both classes and AspectJ-like aspects, with advice represented in normal methods, and to separate all join-point-advice bindings either into annotations in the form of comments, or into separate XML binding files. AspectWerkz provides a proven solution to the problem of AspectJ-like programming in pure Java, but it does not achieve the unification that we have pursued.

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

The design of Caesar [19] is also closely related to our approach. The aim of Caesar is 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 the advice constructs couple crosscut specifications with advice bodies. Consequently, as in AspectJ, advice bodies are still not addressable as individual entities. They can be advised as a group using an advice-execution pointcut. In Caesar, as in Eos, advice can be bound statically or dynamically; however, aspects in Caesar cannot directly advise individual objects on a selective basis.

Aspect languages such as HyperJ [26, 27] 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 [28].

Several others have evaluated aspect-oriented programming techniques on different benchmarks. Early assessments were conducted by Mendhekar *et al.* [29], Kersten and Murphy [30], Walker *et al.* [31], etc. Mendhekar *et al.* [29] used RG, an environment for creating image processing systems to evaluate aspect-oriented program-

ming. Kersten and Murphy [30] used Atlas, a web-based learning environment to evaluate aspect-oriented programming. Walker *et al.* [31] also conducted an initial assessment of aspect-oriented programming. These assessments describe the performance of an aspect-oriented approach in isolation on unique problems; our approach compares two different aspect-oriented models using standard problems. Compared to Dyer *et al.* [32,33]'s work that compares AO interface features, we compare language features.

6 Conclusion

In this paper, we described a case study that analyzed the Eos implementation of 23 GoF design patterns. This implementation showed improvement in the case of 7 out of 23 design patterns compared to the AspectJ implementation. The implementation was no worse in other 16 patterns. A successful demonstration of the capabilities of the language model on standard, broadly utilized, design structures inspires confidence in its potential and practical utility. In most cases, these benefits emerged from the ability to model relationships between participant instances as implicit advising structures. The unification in Eos thus allowed new type of design structures, for example, the reverse chain of predecessors in the Chain of Responsibility pattern, to emerge. A new set of patterns of advising structures is perhaps around the corner, waiting for the wider adoption and use of aspect-oriented programming mechanisms. We also contribute an analysis of the language constructs that were most useful during the GoF design pattern implementation, which provides insight into the design and use of AO languages constructs. These results should also apply to languages inspired by Eos, e.g. Ptolemy [22,23].

7 Acknowledgments

The discussions with participants of the seminar Com S 610-HR, in particular, with Gary T. Leavens were very helpful. Thanks to Kevin Sullivan, Bill Griswold, Mary Lou Soffa and John C Knight for comments on an early version. Thanks to Ademar Aguiar, Atul Jain, Ralph Johnson, Berna L. Massingill, Mark Mahoney and Miguel P. Monteiro for their valuable comments during PLoP 2007 writer's workshop. Rajan was supported in part by the NSF grant CNS-0627354, and by NSF grant CNS-07-09217. This work was also supported in part by NSF grants ITR–0086003 and FCA–0429947.

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1995)
2. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In Knudsen, J.L., ed.: ECOOP 2001 — Object-Oriented Programming 15th European Conference. Volume 2072 of Lecture Notes in Computer Science. Springer-Verlag, Budapest, Hungary (June 2001) 327–353

3. Rajan, H., Sullivan, K.J.: Classpects: unifying aspect- and object-oriented language design. In: ICSE '05: Proceedings of the 27th international conference on Software engineering, New York, NY, USA, ACM Press (2005) 59–68
4. Sullivan, K., Gu, L., Cai, Y.: Non-modularity in aspect-oriented languages: integration as a crosscutting concern for aspectj. In: AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development, New York, NY, USA, ACM Press (2002) 19–26
5. Sullivan, K.J., Notkin, D.: Reconciling environment integration and software evolution. ACM Transactions on Software Engineering and Methodology **1**(3) (July 1992) 229–68
6. Rajan, H., Sullivan, K.: Need for instance level aspect language with rich pointcut language. In Bergmans, L., Brichau, J., Tarr, P., Ernst, E., eds.: SPLAT: Software engineering Properties of Languages for Aspect Technologies. (mar 2003)
7. Rajan, H., Sullivan, K.: Eos: instance-level aspects for integrated system design. In: ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, New York, NY, USA, ACM Press (2003) 297–306
8. Rajan, H., Sullivan, K.: Generalizing AOP for aspect-oriented testing. the proceedings of the Fourth International Conference on Aspect-Oriented Software Development (AOSD 2005) (2005) 14–18
9. Rajan, H., Sullivan, K.J.: Unifying aspect- and object-oriented design. ACM Transactions on Software Engineering and Methodology (TOSEM) **19**(1) (August 2009)
10. Rajan, H., Sullivan, K.J.: Classpects in practice: A test of the Unified Aspect Model. Technical Report 00000383, Iowa State University, Department of Computer Science (September 2005)
11. Hannemann, J., Kiczales, G.: Design pattern implementation in Java and AspectJ. In: Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications, ACM Press (2002) 161–173
12. Rajan, H.: Design pattern implementations in Eos. In: Proceedings of the 14th Conference on Pattern Languages of Programs, ACM (2007)
13. Sakurai, K., Masuhara, H., Ubayashi, N., Matsuura, S., Komiya, S.: Association aspects. In: AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development, New York, NY, USA, ACM Press (2004) 16–25
14. Gosling, J., Joy, B., Steele, G.L.: The Java Language Specification. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1996)
15. Spinczyk, O., Gal, A., Schroeder-Preikschat, W.: AspectC++: an aspect-oriented extension to the c++ programming language. In: CRPITS '02: Proceedings of the Fortieth International Confernece on Tools Pacific, Darlinghurst, Australia, Australia, Australian Computer Society, Inc. (2002) 53–60
16. Bryant, A., Feldt, R.: AspectR - simple aspect-oriented programming in Ruby (Jan 2002)
17. Bonér, J.: What are the key issues for commercial AOP use: how does AspectWerkz address them? In: AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development, New York, NY, USA, ACM Press (2004) 5–6
18. Hirschfeld, R.: Aspects - aspect-oriented programming with squeak. In: NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World, London, UK, Springer-Verlag (2003) 216–232
19. Mezini, M., Ostermann, K.: Conquering aspects with caesar. In: AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development, New York, NY, USA, ACM Press (2003) 90–99

20. Masuhara, H., Kiczales, G.: Modular crosscutting in aspect-oriented mechanisms. In Cardelli, L., ed.: ECOOP 2003—Object-Oriented Programming, 17th European Conference. Volume 2743., Berlin (July 2003) 2–28
21. Sullivan, K.J., Griswold, W.G., Song, Y., Cai, Y., Shonle, M., Tewari, N., Rajan, H.: Information hiding interfaces for aspect-oriented design. In: The Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005). (Sept 2005) 166–175
22. Rajan, H., Leavens, G.T.: Ptolemy: A language with quantified, typed events. In: ECOOP '08: 22nd European Conference on Object-Oriented Programming. (July 2008)
23. Rajan, H., Leavens, G.T.: Quantified, typed events for improved separation of concerns. Technical Report 07-14d, Dept. of Computer Science, Iowa State University (2008)
24. Garcia, A., Kulesza, U., Sant'Anna, C., Lucena, C., Figueiredo, E., von Staa, A.: Modularizing design patterns with aspects: a quantitative study. In: AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, New York, NY, USA, ACM Press (2005) 3–14
25. Rajan, H., Sullivan, K.J.: On the expressive power of Classpects. Technical Report CS-2005-14, University of Virginia, Department of Computer Science (2005)
26. Tarr, P., Ossher, H.L., Harrison, W.H., Sutton, Jr., S.M.: N degrees of separation: Multi-dimentional separation of concerns. In: Proceedings of the 21st International Conference on Software Engineering. (May 1999)
27. Ossher, H., Tarr, P.: Multi-dimensional separation of concerns using hyperspaces. IBM Research Report 21452, IBM (April 1999)
28. Harrison, W., Ossher, H., Tarr, P.: Asymmetrically vs. symmetrically organized paradigms for software composition. In Bergmans, L., Brichau, J., Tarr, P., Ernst, E., eds.: SPLAT: Software engineering Properties of Languages for Aspect Technologies. (March 2003)
29. Mendhekar, A., Kiczales, G., Lamping, J.: RG: A case-study for aspect-oriented programming. Technical Report SPL97-009 P9710044, Xerox PARC (February 1997)
30. Kersten, M., Murphy, G.C.: Atlas: a case study in building a web-based learning environment using aspect-oriented programming. In: OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (1999) 340–352
31. Walker, R.J., Baniassad, E.L.A., Murphy, G.C.: An initial assessment of aspect-oriented programming. In: ICSE '99: Proceedings of the 21st international conference on Software engineering, Los Alamitos, CA, USA, IEEE Computer Society Press (1999) 120–130
32. Dyer, R., Rajan, H., Cai, Y.: Language features for software evolution and aspect-oriented interfaces: An exploratory study. Transactions on Aspect-Oriented Software Development (TAOSD): Special issue, best papers of AOSD 2012 **10** (2013) 148–183
33. Dyer, R., Rajan, H., Cai, Y.: An exploratory study of the design impact of language features for aspect-oriented interfaces. In: AOSD '12: 11th International Conference on Aspect-Oriented Software Development. (March 2012)