Design Patterns: A Canonical Test of Unified Aspect Model

Hridesh Rajan
Dept. of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA, 50010, USA
hridesh@cs.iastate.edu

Kevin Sullivan
Dept. of Computer Science
University of Virginia
151 Engineer's Way
Charlottesville, VA, 22904, USA
sullivan@cs.virginia.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 work is an analysis of the design structuring benefits, the usability, and the practical utility of the Eos language beyond integration problems. To that end, we present a comparative analysis of the implementations of the Gang-of-Four design pattern in AspectJ and Eos. Our result shows that the Eos implementation is better in 7 out of 23 design patterns, and no worse in case of other 16 patterns. The design structures realized in the Eos implementation turned out to be better then the AspectJ version, presenting supporting evidence for the potential benefits of the unified model.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*patterns, information hiding, and languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*patterns, classes and objects*

General Terms

Design, Languages

Keywords

Design Patterns, Classpect, Unified Aspect Language Model, Binding, Eos

1. INTRODUCTION

In prior work, we showed that the notions of aspect and class in the AspectJ language model [13], can be unified in a new module construct that we called the *classpect* [22]. 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 [27, 29].

We embodied our unified model in the Eos language design [19], in which the basic unit of modularity is a *classpect*; and we realized the model in a concrete and usable form in the Eos compiler. We demonstrated the benefits of the unified language design in the context of small but representative examples and case studies [21]. 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 co-ordinate 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.*[9] described the crosscutting concerns in the Gang-of-Four (GOF) patterns and showed that an AspectJ-based solution modularizes these concerns. The main contribution of this paper is an evaluation of the Eos component model, language, and compiler [19, 22] using the Gang-of-Four (GOF) design patterns [6] as test-cases. These patterns are design structures commonly occurring in and extracted from real software systems. The benefits observed in the context of these models—to some extent—could be 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.* [24] briefly observed that the type-level aspect-oriented implementation of the design patterns described by Hannemann *et al.* exhibit the design problems and performance overhead of the form described by Rajan and Sullivan [19].

Our evaluation shows that for 7 out of 23 design patterns the Eos implementation was able to realize better design

Conference '06 Some Place, Some Country

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

Figure 1: A Simple Example Aspect

```
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 }
```

Figure 2: A Simple Example Classpect

structures. For other 16 patterns, both implementation are the same with respect to the metrics documented by Hannemann *et al.*[9] and Garcia *et al.*[7]. The result is not surprising as the Eos language model is essentially a superset of the AspectJ language model. 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, the usability of the Eos language, and the practical utility of our language implementation in the Eos compiler. 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 Eos implementation of the design patterns and compares it with the AspectJ implementation. Section 4 describes some limitations of the approach discovered by the evaluation. Section 5 discusses related work and Section 6 concludes.

2. BACKGROUND

In this section, we briefly review the AspectJ and unified language model embodied by Eos. The focus is on their key differences. The AspectJ language model is described in detail by Kiczales *et al.* [13]. A complete language manual and compiler is available from the AspectJ web site [1]. The unified language model is described in detail by Rajan *et al.* [22] and the compiler is available from the Eos web site [5].

2.1 The AspectJ Language Model

In this subsection, we will review basic concepts in the AspectJ model. AspectJ [13] is an extension to Java [8]. Other languages using AspectJ's model include AspectC++ [25], AspectR [3], AspectWerkz [2], AspectS [11], Caesar [16], etc. While Eos [19] is not AspectJ-like, it is in the broader class of Pointcut-Advice-based AO languages [14]. 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 [28]. 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.

These languages add five key constructs to the object-oriented model: join points, pointcuts, advice, inter-type declarations, and aspects. (For the purpose of this paper, inter-type declarations are not relevant as they remain unchanged in the unified model.) A simple example is shown in Figure 1 to make these concepts concrete. The aspect (lines 1-7), modifies the behavior of a program *before*, *after*, or *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 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, execution of any method outside the Tracing aspect. An advice (see lines 4-6) is a special *method-like* construct that effects 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.

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

2.2 The Unified Language Model

Rajan *et al.* addressed the limits of aspects in a new language model that unifies aspects and objects in three ways [22]. First, 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. Second, the unified model eliminates advice in favor of using methods only, with a separate and explicit join-point-method binding construct. Third, 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) 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, uses 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. DESIGN PATTERNS

The Gang-of-Four (GoF) design patterns [6] identify common design structures in software systems; and provide mechanism and sample implementation strategies for efficiently tackling them. Initial implementation strategies are oriented towards object-oriented paradigm, however, it has been shown that the implementation language affects the use and implementation strategies [4] [31] [17] [26].

Inspired by these studies, Hannemann *et al.* [9] presented implementation of design patterns in AspectJ and showed that it is better with respect to a set of criteria. Sakurai *et al.* [24] briefly observed that the type-level aspect-oriented implementation of the design patterns described by Hannemann *et al.* exhibit the design and performance overhead of the form described by Rajan *et al.* [19]. This section looks at the implementation of some of the design patterns. We present a realization of design patterns using the unified language model. We also compare the new implementation with that of Hannemann *et al.*. Note that Hannemann *et al.*'s implementation uses the AspectJ language.

In earlier work, we showed that translation of programs from the AspectJ language model to their equivalent in the proposed unified model is possible [23]. 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. 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 [9] as well as those by Garcia *et al.* [7] apply to the Eos implementation as well. For 7 out of 23 patterns, however, expressiveness of the unified language model allowed realizations of even better design structures. Due to the space limitations, we only describe three of these patterns in detail in the rest of this section. The subsection 3.2 to 3.3 describe the Chain of Responsibility, the Observer and the Mediator pattern respectively. Similar design structuring benefits were observed in case of the Composite pattern, Command pattern, Singleton pattern and Strategy pattern. To determine whether an implementation strategy is better then other we consider the mapping from the design of the pattern to implementation as a key factor.

3.1 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 [6, 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 ObserverProtocol aspect 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.

```
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);
24 protected abstract void
25     update (Subject s, Observer o);
26 after(Subject s): subjectChange(s) {
27  Iterator iter = getObservers(s).iterator();
28  while ( iter.hasNext() ) {
29    update (s, ((Observer)iter.next()));
30    }
31  }
32 }
```

Figure 3: Observer in AspectJ

The AspectJ language model does not fully support aspect instantiation and selective advising of object–instances [20]. 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 interfaces that are introduced by the concrete observers into participants so that ObserverProtocol can manipulate them. The pattern's implementation therefore superimposes two roles, subjects(line 2) and observers (line 3), on participants.

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 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 [19] with the observer pattern concern. The need for roles and for maintaining a HashMap 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 }
```

Figure 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 improves the modularity of the pattern. It does not tangle emulation concern with observer protocol concern. It clearly abstracts the behavior of the pattern in a much nicer way. It clearly (and only) conveys the intent of the pattern, which is to update an observer when a subject 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 gone.

With respect to the metrics used by Garcia *et al.* [7], it achieves nearly 78 percent reduction in LOC (Line of Code) of the observer protocol concern without increasing the complexity of the remaining code. Each line in Eos implementation corresponds to a line in AspectJ implementation. The Eos implementation, however, eliminates all the emulation code from the pattern implementation. This implementation is also localized, reusable, compositionally transparent, and (un) pluggable. It also decreases the Number of Attributes (NOA) [7] of the Observer protocol concern to zero from one.

Moreover, the composition of the participants into the 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 Point. A subject-observer relationship between Point and Screen in which Screen instance observes change in color of the Point instance is shown on the left hand side of Figure 6. The ColorObserver relationship implementation does not clearly communicate the

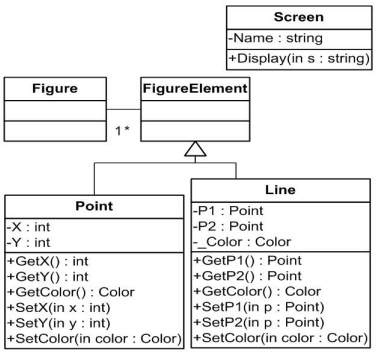

Figure 5: The Figure Element System

AspectJ

```
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 }
```

Eos

```
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 }
```

Figure 6: The Color Observer in AspectJ and Eos

specification that it involves two object instances, an observer instance and an observed instance. 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 on the right hand side of Figure 6. 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 [19]. 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 7 shows the key parts of the client code. AspectJ code is preceded by *AspectJ:* and Eos code is preceded by *Eos:*. The client code in Figure 7 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 mim-

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

Figure 7: Observer Clients in AspectJ and Eos

ics 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.

3.2 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. [6, 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 8. 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.

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 9 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 8). The difference is now in the implementation technique that can be realized due to the new instance-level advising features in Eos [19]. The method *receiveRequest* in the Eos implementation first

```
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 handler, Request request){
7   if (handler.acceptRequest(request)){
8     handler.handleRequest(request);
9    } else {
10   Handler successor = getSuccessor(handler);
11  if (successor == null){
12        throw new CORException("End of chain reached)");
13   } else {
14    receiveRequest(successor, request);
15    }
16   }
17  }
18 public boolean Handler.acceptRequest(Request request){
19  return false;
20  }
21 public void Handler.handleRequest(Request request) {}
22 protected abstract pointcut eventTrigger
23      (Handler handler, Request request);
24 after(Handler handler, Request request):
25       eventTrigger(handler, request){
26  receiveRequest(handler, request);
27  }
28 public void setSuccessor
29      (Handler handler, Handler successor){
30  successors.put(handler, successor);
31  }
32 public Handler getSuccessor(Handler handler){
33  return ((Handler) successors.get(handler));
34  }
35 }
```

Figure 8: Chain of Responsibility in AspectJ

checks whether the current Handler object can handler the request, otherwise it throws an exception of type *CORException*.

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 requires complex emulation code when written in AspectJ. We will describe this difference in more detail in the context of a concrete example.

To illustrate the difference in the implementation tech-

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

Figure 9: Chain of Responsibility in Eos

nique 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 10) 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* as *protected pointcut eventTrigger(Handler handler, Request request): call(void Button.doClick(Click)) && target(handler) && args(request)*. 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 8) 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.

The Eos implementation (Figure 11) 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

```
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 handler, Request request):
8            call(void Button.doClick(Click)) &&
9            target(handler) && args(request);
10  public boolean Button.acceptRequest(Request request){
...
17  }
18  public void Button.handleRequest(Request request){
...
20  }
...
43}
```

Figure 10: Concrete Aspect ClickChain in AspectJ

```
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 handler, Request request):
8           execution(void Button.doClick(Click)) &&
9           this(handler) && args(request);
10  introduce in Button {
11  public bool acceptRequest(Request request){
...
18  }
19  public void handleRequest(Request request){
...
21  }
22 }
...
49} /* 6 more lines due to inter-type declaration syntax */
```

Figure 11: Concrete Classpect ClickChain in Eos

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 9) 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 12 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: Frame frame = new Frame(...);
Common: Panel panel = new Panel(...);
Common: Button button1 = new Button(...);
Common: Button button2 = new Button(...);
AspectJ: ClickChain.aspectOf().setSuccessor(button1, panel);
AspectJ: ClickChain.aspectOf().setSuccessor(panel, frame);
Eos: ClickChain chain = new ClickChain();
Eos: chain.SetTrigger(button1);
Eos: panel.SetPredecessor(button1);
Eos: frame.SetPredecessor(panel);
```

Figure 12: 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 *Panel* 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 [22]. 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.

3.3 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 [6, 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 *MediatorProtocol* aspect as shown in Figure 13. 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 also provides an abstract method; *notify* (lines 19–20) to be redefined in concrete mediators to implement the notification logic. The *MediatorProtocol* aspect also keeps a

```
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 }
```

Figure 13: Mediator in AspectJ

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 with the new one.

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

Figure 14: Mediator in Eos, 66% Smaller.

Like the Observer pattern, the Eos implementation shown in Figure 14 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.

3.4 Other Patterns

We observed similar design structuring benefits in the

Eos implementation of the Composite, Command, Strategy, and Singleton patters. 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. 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.

3.5 Summary

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.

4. LIMITATIONS

Using design patterns as a test of the unified aspect language model exposed a key limitation of our binding mechanism. In the current version of Eos [5] the syntax of a binding defined as shown in Figure 15. The emphasized symbols denote non-terminals. A binding specifies a pointcut expression and a list of method bindings. A method binding is defined as an identifier that denotes the name of the bound method and the list of parameters that the method expects. The methods specified by the method bindings are bound at the join points selected by the pointcut expression. A method specified in the binding must be defined in the lexical scope of the binding.

```
binding_declaration
    :  opt_static after pointcut: method_bindings;
    |  opt_static before pointcut: method_bindings;
    |  opt_static type around pointcut: method_bindings;
    ;
method_bindings
    :  method_bindings, method_binding
    |  method_binding
    ;
method_binding
    : Id ( opt_formal_parameter_list)
    ;
opt_static
    :  EMPTY | static
    ;
```

Figure 15: Syntax of the Eos Binding Declaration

Some use cases encountered in the design pattern implementation suggested that just specifying the method name and the formal parameters may not be sufficient to handle all usage of bindings. Some mechanism to optionally specify the instance on which the bound method is called is also required. For example, in the implementation of the chain of responsibility pattern in Eos (Figure 9), the binding on lines 26–27 binds the method *TriggerRequest* to execute at the join points selected by the pointcut *eventTrigger* (lines 24–25). For the reader's convenience the relevant part of the implementation is reproduced in Figure 16. The method *TriggerRequest* calls the method *receiveRequest* on the supplied *Handler* instance passing it the supplied *Request* instance as an argument. Ideally, the handler of this binding (lines 26–27) is the method *receiveRequest*. The handler instance and the arguments are also available in the binding through the pointcut expression, however, the syntax limitation of the binding mechanism does not allows the simpler representation similar to that shown in Figure 17. The workaround requires the use of the delegation pattern as shown in the Figure 16.

```
1 public abstract class CORProtocol{
...
24 protected abstract pointcut eventTrigger
25      (Handler handler, Request request);
26 after eventTrigger(handler, request):
27    TriggerRequest(Handler handler, Request request);
28 public void TriggerRequest
29      (Handler handler, Request request){
30    handler.receiveRequest(request);
31  }
...
35}
```

Figure 16: Snippet from the Chain of Responsibility Protocol Implementation in Eos

```
24 protected abstract pointcut eventTrigger
25      (Handler handler, Request request);
26 after eventTrigger
27      (Handler handler, Request request):
28       handler.receiveRequest(request);
```

Figure 17: Ideal Representation of the Binding

In future, we will address this limitation of the binding mechanism and investigate the possibility and challenges associated with specifying the target instance as part of the binding specification.

5. RELATED WORK

Most closely related to this work is the evaluation of aspect-oriented implementation of the Gang-of-Four design patterns [6] conducted by Hannemann and Kiczales [9] and Garcia *et al.* [7]. Hannemann and Kiczales compared the object-oriented implementation in Java with aspect-oriented implementation in AspectJ using qualitative metrics. Garcia *et al.* [7] 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 [22], is related to AspectJ [13], AspectWerkz [2], and Caesar [16]. 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 [19, 27]. This work reinforces our earlier findings.

Another closely related design is that of AspectWerkz [2]. The aim of AspectWerkz is to provide the expressiveness of AspectJ [13] 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 [16] 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 [30, 18] 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 [10].

Several others have evaluated aspect-oriented programming techniques on different benchmarks. Early assessments were conducted by Mendhekar *et al.* [15], Kersten and Murphy [12], Walker *et al.* [32], etc. Mendhekar *et al.* [15] used RG, an environment for creating image processing systems to evaluate aspect-oriented programming. Kersten and Murphy [12] used Atlas, a web-based learning environment to evaluate aspect-oriented programming. Walker *et al.* [32] 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.

6. CONCLUSION

In this paper, we showed that the Eos implementations of 7 out of 23 design patterns was better than AspectJ implementation, and 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.

7. ACKNOWLEDGMENTS

This work was supported in part by NSF grants ITR–0086003 and FCA–0429947. The comments from William G. Griswold were very helpful in deciding the scope of this evaluation. The discussions with Gary T. Leavens were very helpful in the presentation of the chain of responsibility pattern.

8. REFERENCES

[1] AspectJ programming guide. http://www.eclipse.org/aspectj/.

[2] J. Bonér. 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*, pages 5–6, New York, NY, USA, 2004. ACM Press.

[3] A. Bryant and R. Feldt. AspectR - simple aspect-oriented programming in Ruby, Jan 2002.

[4] C. Chambers, B. Harrison, and J. Vlissides. A debate on language and tool support for design patterns. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 277–289, New York, NY, USA, 2000. ACM Press.

[5] Eos web site. http://www.cs.virginia.edu/ eos.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[7] A. Garcia, U. Kulesza, C. Sant'Anna, C. Lucena, E. Figueiredo, and A. von Staa. Modularizing design patterns with aspects: a quantitative study. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 3–14, New York, NY, USA, 2005. ACM Press.

[8] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[9] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 161–173. ACM Press, 2002.

[10] W. Harrison, H. Ossher, and P. Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. In L. Bergmans, J. Brichau, P. Tarr, and E. Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, Mar. 2003.

[11] R. Hirschfeld. 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*, pages 216–232, London, UK, 2003. Springer-Verlag.

[12] M. Kersten and G. C. Murphy. 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*, pages 340–352, New York, NY, USA, 1999. ACM Press.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, Budapest, Hungary, June 2001.

[14] H. Masuhara and G. Kiczales. Modular crosscutting in aspect-oriented mechanisms. In L. Cardelli, editor, *ECOOP 2003—Object-Oriented Programming, 17th European Conference*, volume 2743, pages 2–28, Berlin, July 2003.

[15] A. Mendhekar, G. Kiczales, and J. Lamping. RG: A case-study for aspect-oriented programming. Technical Report SPL97-009 P9710044, Xerox PARC, Feb. 1997.

[16] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM Press.

[17] P. Norvig. Design patterns in dynamic programming. In *Object World 96*, Boston, MA, May 1996.

[18] H. Ossher and P. Tarr. Multi-dimensional separation of concerns using hyperspaces. IBM Research Report 21452, IBM, Apr. 1999.

[19] H. Rajan and K. Sullivan. 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*, pages 297–306, New York, NY, USA, 2003. ACM Press.

[20] H. Rajan and K. Sullivan. Need for instance level aspect language with rich pointcut language. In L. Bergmans, J. Brichau, P. Tarr, and E. Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, mar 2003.

[21] H. Rajan and K. J. Sullivan. Classpects in practice: A test of the Unified Aspect Model. Technical Report 00000383, Iowa State University, Department of Computer Science, Sept. 2005.

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

[23] H. Rajan and K. J. Sullivan. On the expressive power of Classpects. Technical Report CS-2005-14, University of Virginia, Department of Computer Science, 2005.

[24] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 16–25, New York, NY, USA, 2004. ACM Press.

[25] O. Spinczyk, A. Gal, and W. Schroeder-Preikschat. AspectC++: an aspect-oriented extension to the c++ programming language. In *CRPITS '02: Proceedings of the Fortieth International Confernece on Tools Pacific*, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[26] G. T. Sullivan. Advanced programming language features for executable design patterns. Lab Memo AIM-2002-005, MIT Artificial Intelligence Laboratory, 2002.

[27] K. Sullivan, L. Gu, and Y. Cai. 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*, pages 19–26, New York, NY, USA, 2002. ACM Press.

[28] K. J. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. 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)*, pages 166–175, Sept 2005.

[29] K. J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–68, July 1992.

[30] P. Tarr, H. L. Ossher, W. H. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimentional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.

[31] D. von Dincklage. Making patterns explicit with metaprogramming. In *GPCE '03: Proceedings of the second international conference on Generative programming and component engineering*, pages 287–306, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[32] R. J. Walker, E. L. A. Baniassad, and G. C. Murphy. An initial assessment of aspect-oriented programming. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 120–130, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.