Statement Annotations for Fine-Grained Advising

Marc Eaddy Alfred Aho

Department of Computer Science
Columbia University
New York, NY 10027
{eaddy,aho}@cs.columbia.edu

ABSTRACT

AspectJ-like languages are currently ineffective at modularizing *heterogeneous concerns* that are tightly coupled to the source code of the base program, such as logging, invariants, error handling, and optimization. This leads to complicated and fragile pointcuts and large numbers of highly-repetitive and incomprehensible aspects. We propose *statement annotations* as a robust mechanism for exposing the join points needed by heterogeneous concerns and for enabling declarative fine-grained advising.

We propose an extension to Java to support statement annotations and AspectJ's pointcut language to match them. This allows us to implement heterogeneous concerns using a combination of simple and robust aspects and explicit and local annotations. We illustrate this using a logging aspect that logs messages at specific locations in the source code. Statement annotations also simplify advising specific object instances, local variables, and statements. We demonstrate this using an aspect that traces method calls made to specific object instances and calls made from specific call sites.

Keywords

statement annotations, byte code annotations, fragile pointcut problem, logging problem, statement-level join points, instance-local advising

1. INTRODUCTION

Aspect-Oriented Programming (AOP) improves the separation of concerns by modularizing the code related to a concern that would otherwise be scattered throughout a program and tangled with the code related to other concerns. AspectJ-like languages are designed to modularize *homogeneous concerns*, which crosscut at module boundaries [17] and have a regular structure and common behavior [23].

1.1 Heterogeneous Concerns

Unfortunately, *heterogeneous concerns*, which exhibit irregular logic, are located at arbitrary places in the source code, and/or are highly coupled to the low-level structure of the code, are difficult to modularize using AspectJ-like languages [23]. Workarounds include creating complex and fragile pointcuts, writing a large number of highly-repetitive aspects, refactoring the base program to artificially expose the needed join points, or even abandoning AOP in favor of non-AOP solutions. The inability for AOP languages to effectively express heterogeneous concerns severely limits the potential for AOP to separate crosscutting concerns.

This problem was first observed by researchers working on refactoring programs to use aspects. Murphy et al. needed to identify join points "in the middle of methods" to refactor graphical user interface code [15]. Since AspectJ could not capture these join points directly, their solution was to insert *dummy method calls* "which exist solely to provide access to the desired join points." This workaround has become so common in the AspectJ community that it is considered a de facto aspect refactoring idiom [8] [13].

In another refactoring exercise, Sullivan et al. observed that the logging concern was scattered across 180 different locations in the HyperCast application [20]. The arbitrary locations of the log messages required 20-30 complicated pointcut declarations that could only approximate the actual locations. Furthermore, the pointcuts were highly

Reflection, AOP and Meta-Data for Software Evolution (RAM-SE 2006)

dependent on the structure of the underlying source code and would easily break when the code is modified. This is referred to as the *fragile pointcut problem* [14] [20]. Pointcut fragility in turn leads to the *AOSD-evolution paradox* [22], where programs written using AOP are actually harder to evolve, even though they are more modular. Other researchers have commented on the fragility of logging, optimization, and error handling aspects and the need for more robust and powerful pointcut languages [1] [5] [14] [17] [18] [19].

1.2 Our Approach: Statement Annotations

Statement annotations allow us to "name" any statement in a method body in a declarative fashion and attach arbitrary metadata. Statement annotations can be used to expose the join points needed to implement a heterogeneous concern or to perform fine-grained (instance- and statement-level) advising. This provides many benefits:

Pointcuts are simple and robust because they depend on semantically meaningful annotations instead of arbitrary program syntax or source locations [14] [3];

Advising is more fine-grained because advice can be applied to individual statements or object instances. While this is possible using other techniques, statement annotations provide a simple *declarative* way to perform fine-grained advising;

Advice is more reusable because it can access annotation metadata instead of hard-coding it;

Concerns can be easily integrated using a unified AOP solution instead of requiring a mixture of an AOP solution (for homogeneous concerns) and a non-AOP solution (for heterogeneous concerns);

The relationship between the base code and aspects is local and explicit, improving comprehensibility and maintainability [11].

Statement annotations must be invasively scattered throughout the base program and are thus crosscutting. We view this as an engineering tradeoff: we gain robustness, explicitness, and locality at the expense of obliviousness and modularity. Modularity is not sacrificed completely, however. Indeed, the parts of the concern that can be modularized *effectively* are specified in the aspect; the remainder is specified using annotations.

1.3 Outline

In Section 2 we describe statement annotations, show how pointcut matching in AspectJ-like languages can be easily extended to match them, and how they can be used to implement the heterogeneous logging concern. In Section 3 we describe fine-grained advising and how it can be used to enable instance- and statement-level tracing. We discuss the tension between locality/explicitness and modularity with respect to statement annotations in Section 4 and related work in Section 5. Section 6 concludes.

```
public class HelloWorld {
   public static void main(String args[]) {
      @State(State.Starting)
      System.out.println("Hello, World!");
   }
}
```

Listing 1. Statement annotation example.

2. STATEMENT ANNOTATIONS

Java and C#/.NET provide a flexible mechanism for allowing the programmer to attach annotations to program elements such as classes, methods, and fields. The annotations are stored in the class file as metadata and can be retrieved via reflection. However, these languages do not support annotating statements or byte code within the method body. This is unfortunate because statement annotations have the potential for a wide variety of uses:

Optimization – Statement annotations could potentially be used to enable OpenMP[1]-style parallel processing directives for Java. For example, some researchers have annotated *for-loops* to guide loop parallelization [2]. In addition, a compiler or other tool could annotate byte code with static analysis results to improve opportunities for optimization at JIT-time.

Bookkeeping – A form of byte code marking is used by AspectJ to prevent recursive weaving [10], by Steamloom to allow efficient enabling/disabling of advice at runtime [9], and by Spec#[2] to ignore injected code when performing static analysis. However, none of these systems allow arbitrary user-defined metadata to be associated with byte code.

Debugging and fault isolation – A debugger could selectively show or hide injected code based on the programmer's desired level of obliviousness. Injected byte code could be marked to indicate the source weaver or tool to improve fault isolation.

Listing 1 shows an example of what a statement annotation could look like in Java.[3]

[1] http://www.openmp.org

[2] http://research.microsoft.com/specsharp

[3] For the remainder of the paper we will use examples in AspectJ and Java. However, our extensions can be easily made to other languages.

2.1 Statement Annotation Matching

In AspectJ 5, pointcut expressions may contain annotation patterns, but they can only match regular Java annotations defined on methods, fields, etc. We extend the pointcut expression matching algorithm to match statement annotations, effectively extending the AspectJ join point model to include any annotated statement. This gives us fine-grained control over when advice is applied and can simplify pointcut expressions that have many special cases.

Our proposed extension is very simple: Any pointcut expression that allows an annotation pattern should consider statement annotations in addition to regular annotations. For example, if method foo has the method annotation MA, and the statement annotation SA is used at a particular call-site, then the following AspectJ pointcuts will match that call-site:

```
call(@MA * *(..))
call(@SA * *(..))
@annotation(MA)
@annotation(SA)
```

This matching algorithm allows us to use the same annotation at either method- or statement-level granularity. If in the future we decide we need to be able to distinguish the two, we can introduce a new pointcut that only matches statement annotations.

2.2 Statement Annotations Simplify Complicated Pointcuts

In the HyperCast paper, Sullivan et al. observed that while it is sometimes possible to create pointcuts that match specific statements, they can become quite complicated [20]:

```
pointcut HCLogicalAddrChanged(HC_Node node):
  set(I_LogicalAddress
    HC_Node.MyLogicalAddress)
  && (withincode(void
    HC_Node.messageArrivedFromAdapter(I_Messa
    ge))
  || withincode(void
    HC_Node.timerExpired(Object))
  || withincode(void
    HC_Node.resetNeighborhood()))
  && target(node);
```

Tourwé et al. observed that these kinds of complicated pointcuts arise because the pointcut language is too simplistic [22]. This hinders the evolvability of the base program because the pointcuts are likely to break if the underlying code changes.

This pointcut can be simplified if we are allowed to annotate the base program using method annotations:

```
pointcut HCLogicalAddrChanged(HC_Node node):
  set(I_LogicalAddress
    HC_Node.MyLogicalAddress)
  && withincode(@MyAnnotation * *(..))
  && target(node);
```

We can further simplify the pointcut by annotating the specific statements that require advising:[4]

```
pointcut HCLogicalAddrChanged(HC_Node node):
  set(@MyAnnotation I_LogicalAddress
    HC_Node.MyLogicalAddress)
    && target(node);
```

In general, annotations allow program elements to be explicitly named, eliminating the need for complicated pointcut expressions [3] or meta-programming. Annotations are explicit and collocated with the source code so they are more likely to be maintained as the program evolves. Pointcuts are only dependent on the annotations instead of the underlying structure of the code so they are more robust to changes and more reusable.

In the next section we show how to use a simple pointcut that matches statement annotations to implement the heterogeneous logging concern.

2.3 Heterogeneous Concern Example: Logging

Logging is the ability to record user-defined messages at specific points during the execution of the application. Logging is an example of a heterogeneous concern because its very nature is *ad hoc*—each log message is unique and often located at arbitrary points in the source code. Although both the tracing and logging concerns are complementary, and indeed often co-exist within the same application, logging is cumbersome to express using existing pointcut languages. This problem has become known colloquially as the *logging problem* [4] [12] [1] [21], however, it is a general problem that occurs anytime we try to capture a heterogeneous concern using AspectJ-like pointcut languages.

Listing 2 shows how the Note statement annotation can be used to expose specific join points within a method and attach arbitrary user-defined messages which can be logged by an aspect. When naming annotations, Kiczales and Mezini advice is to "choose a name that describes what is true about the points, rather than describing what a particular advice will do at those points." [11] Using their terminology, we chose an "annotation-property"-like name for the annotation, e.g., *Note*, as opposed to an "annotation-call"-like name, e.g., *Log*.[5]

---

[4] Sullivan et al. took a different approach to simplify the pointcut that requires the base programmer to update a finite state machine (FSM) [20]. We believe our approach is complementary. Indeed, statement annotations can be used to update the FSM.

[5] We thank Dean Wampler for suggesting this.

```
    ...several statements...
a.  @Note("Searching for plug-ins...")
    ...several statements...

    @Note("Entering very long, but hopefully not infinite, loop")
b.  while (true) { ... }
    @Note("Loop exited successfully")
```

a-b. Using statement annotations to expose interesting events.

```
aspect LogNotesAspect {
   before(Note noteAnnotation) : @annotation(noteAnnotation) {
      System.out.println(noteAnnotation.value() + " [" + thisJoinPoint + "]");
   }
}
```

c. Aspect for logging notes.

Listing 2. Logging using statement annotations.

2.3.1 Statement Annotations versus Procedure Calls and Macros

The statement annotations in Listing 2 could be replaced with plain old procedure/method calls or macros. However, advice methods are more powerful because they have access to richer join point context information and more evolvable because they can access this context implicitly. Another difference is that plain old method calls are always called, and therefore always incur some overhead, even if the aspect is disabled. While macros can be used to alleviate this overhead by expanding to nothing at compile time, they do not help us at runtime.

Finally, resorting to method calls and macros to implement a crosscutting concern represents a fundamental failure of our AOP language. Some crosscutting concerns are implemented using AOP and some using non-AOP techniques even though the concerns may be very similar (e.g., tracing versus logging, error detection versus error handling) and may even share a common base implementation.

2.3.2 Statement Annotations versus Dummy Method Calls

In Listing 2a, an Event annotation appears at an arbitrary location in a method body. Without the annotation it would be difficult or impossible to identify the join point using an AspectJ pointcut, and, even if we could, the resulting pointcut would be very fragile [20]. An alternative to using a statement annotation is to insert a dummy method call at this location, a common AspectJ programming idiom.

However, we find this unsatisfactory for several reasons. Dummy methods require a proliferation of empty methods which adversely affects design and code quality and can be confusing to the base programmer [15]. Statement annotations obviate the need for empty methods. Furthermore, statement annotations are less confusing because programmers familiar with annotation usage in Java and .NET are accustom to the interpretation of annotations at compile-time or postcompilation. Functionality that is woven into the base program as a result of the annotations will be less surprising, than, say, using around advice to replace empty method calls.

Unlike a statement annotation, a dummy method call is not directly associated with the next statement in the method body. Instead, it adds a new join point to the program. In the cases where the use of a statement annotation mirrors a method call or macro (aka an "annotation-call"), a dummy method call can be used instead. However, in the cases where a statement annotation is used to name and/or associate metadata with another statement (aka an "annotation-property"), dummy methods cannot be used. This makes dummy methods unsuitable for expressing some heterogeneous concerns such as local variable invariants and loop optimization hints [7] [6] and for performing fine-grained advising.

2.4 Statement Annotations Improve Statement-Level Pointcuts

Listing 2b shows statement annotations that bracket a *while-loop*. While AspectJ cannot match loops, some researchers have proposed extensions to the pointcut language to support matching statement-level join points of this kind [7] [6] [17]. While these proposed pointcuts make it easier to advise arbitrary statements, they are still fragile when they are used to identify specific statements.

Statement annotations allow individual statements to be discriminated without relying on the specific statement syntax. This means that if a *while-loop* is changed to a *for-loop* by the base programmer, the aspect will be unaffected. Furthermore, because statement annotations are explicit they are more

```
class BankAccount {
   public void transferFundsTo(float amount, BankAccount destination) {
      // Trace all calls made to the ar object
      @Trace AuthorizationRequest ar = new AuthorizationRequest(this, destination);

      // Trace a call made at a specific call site
      @Trace destination.deposit(amount);
      …
```

a. Statement annotations for tracing instances and specific method calls.

```
aspect TraceAspect {
   static Set instances = new HashSet();
   after() returning(Object o) : call(@Trace *.new(..)) {
      instances.add(o);
   }
   before(Object o) : call(* *(..)) && target(o) && if(instances.contains(o)) {
      System.out.println("Calling: " + thisJoinPoint);
   }
   before : call(@Trace * *(..)) {
      System.out.println("Calling: " + thisJoinPoint);
   }
}
```

b. Trace aspect using statement annotation matching.

Listing 3. Instance- and statement-level tracing. The statement annotations in **(a)** indicate that all calls to the `ar` instance should be traced as well as the `destination.deposit` method call The first advice in **(b)** adds instances created with the Trace annotation to the set. The second advice traces all calls to instances in the advised set. The third advice traces all calls originating from call-sites marked with the Trace annotation.

likely to be kept up to date when the base source code is modified.

3. DECLARATIVE FINE-GRAINED ADVISING

Statement annotations are the first technique we are aware of for supporting declarative (as opposed to programmatic) instance- and statement-level advising. Examples of systems that support programmatic advising are Steamloom [9] and Eos [16]. The AspectJ *aspectOf* construct supports this to a lesser extent. The benefit of using statement annotations is that they only declare the programmer's intention. The actual advising is done by the aspect, where low-level decisions such as *if* and *how* instances will be advised can be deferred, rather than scattering these decisions throughout the base program, thus improving the separation of concerns.

Tracing is the ability to record some or every method call[6] made during the execution of an application. Unlike logging, tracing is a homogeneous concern because it is highly structured in nature—the message format is consistent and the message is recorded at regular points in the execution of the application that are easily quantified using AspectJ-like pointcuts.

Annotations allow us to use tracing in a more heterogeneous fashion. For example, imagine we have a Trace annotation. We can use it as a method annotation and attach it to a particular method so that all calls to that method will be traced. Or we can use it as a statement annotation to trace calls made to a specific object instance or to advise specific call-sites. This would be hard to do using existing pointcut languages.

Listing 3 shows an example of *instance-level advising*. The Trace statement annotation marks the `ar` object instance which will cause all method calls to it to be traced. The TraceAspect has advice that matches constructor calls that are marked with the Trace annotation and adds the object instance to the set of advised instances. Another advice matches any method call to an instance in the advised set.

In their paper on optimization aspects, Siadat et al. needed to advise a specific method call out of multiple calls to that method in the method body, but observed that AspectJ-like pointcut languages did not support this level of granularity [19]. Listing 3 shows an example of how this *statement-level advising* can be achieved using statement annotations.

Notice that the Tracing aspect is a normal AspectJ aspect. Without statement annotation matching

[6] As well as other join points that can be easily expressed using AspectJ-like pointcut languages, such as field access.

support, the aspect will trace method calls to *any* instance of a class that has a Trace annotation on its constructor. With annotation support, we can narrow the focus to specific instances and call sites.

4. DISCUSSION

Looking at the examples, it may appear that the logging and tracing concerns have not been modularized at all. However, the parts of these concerns that can be modularized *effectively*, including how messages are formatted, what join point context is used, and where messages are sent, are modularized by the aspect. For example, it would be relatively easy to change the aspects to send messages to a different location or to support asynchronous logging and tracing.

The parts of these concerns that cannot be modularized effectively, namely the user-defined messages, the locations in code, and which object instances and statements to advise, are better captured using statement annotations. The explicitness and locality of statement annotations makes it less likely that changes to the source code will invalidate the aspects, and makes it unnecessary for the programmer to have global system knowledge to assure that the pointcuts match correctly [14].

5. RELATED WORK

Rho et al. present a fine-grained generic aspect language called LogicAJ2 [19]. Their pointcut language can match *arbitrary* declarations, statements, and expressions, and can bind to arbitrary join point context. The use of meta-variables in pointcuts, introductions and advice enables them to achieve heterogeneous, context-dependent effects. Unfortunately, identifying *specific* statements still requires referencing concrete base entities. This introduces dependencies which might break if base entities change.

By explicitly annotating the base code, using our statement annotations, for example, the base programmer could express these conceptual dependencies of aspects on base entities in a less fragile way at the expense of giving up obliviousness and introducing scattering in the base code. This remains interesting future work.

6. CONCLUSION

We showed that by combining statement annotations and pointcuts, we can support instance- and statement-level advising, simplify pointcut expressions to make them more robust and reusable, and elegantly express heterogeneous concerns such as logging. We did this by proposing a natural extension to the AspectJ pointcut matching algorithm that is consistent with AspectJ's overall language philosophy.

For identifying individual statements and code locations, statement annotations are more robust than using complex pointcuts or meta-programming, and more elegant and obvious than using dummy methods. For advising specific object instances and statements, statement annotations are simpler and more succinct than using programmatic advising.

Unfortunately, annotations require intrusive (nonoblivious) changes to the base program and do not completely modularize concerns. It remains our future work to develop an AOP solution that can completely modularize heterogeneous concerns, possibly involving aspect visualization or automatic aspect refactoring, in a way that is easy to understand, maintain, and evolve.

7. ACKNOWLEDGMENTS

We thank Boriana Ditcheva for creating a preprocessor for legalizing statement annotations for C#, Vibhav Garg for adding support for statement-level advising to Steamloom, and Kevin Sullivan and Yuanyuan Song for information about HyperCast. We also thank Matthew Arnold, Pascal Costanza, Günter Kniesel, and Dean Wampler for their feedback.

8. REFERENCES

[1] R. Bodkin, A. Colyer, and J. Hugunin. "Applying AOP for Middleware Platform Independence," Practitioner Report. In Proc. of Aspect-Oriented Software Development (AOSD'03), March 2003.

[2] W. Cazzola, A. Cisternino, D. Colombo. "[a]C#: C# with a Customizable Code Annotation Mechanism," In Proc. of the Symposium on Applied Computing (SAC'05), March 2005.

[3] V. Cepa and S. Kloppenburg. "Representing Explicit Attributes in UML," In Proc. of the Workshop on Aspect-Oriented Modeling (AOM'05), March 2005.

[4] J. W. Cocula. "Can AOP really solve the 'logging problem'?,". Online posting. AspectJ Users Discussion Forum. April 2003. http://dev.eclipse.org/ mhonarc/lists/aspectj-users/msg00383.html

[5] A. Colyer, A. Clement, R. Bodkin, and J. Hugunin, "Using AspectJ for component integration in middleware," Practitioner Report. In Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03), October 2003.

[6] B. Harbulot and J. Gurd. "Using AspectJ to Separate Concerns in Parallel Scientific Java Code," In Proc. of Aspect-Oriented Software Development (AOSD'04), March 2004.

[7] B. Harbulot and J. Gurd. "A Join Point for Loops in AspectJ," In Proc. of Aspect-Oriented Software Development (AOSD'06), March 2006.

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

[9] M. Haupt, M. Mezini, C. Bockisch, T. Dinkelaker, M. Eichberg and M. Krebs. "An Execution Layer for Aspect-Oriented Programming Languages," In Proc. of Virtual Execution Environments (VEE'05), June 2005.

[10] E. Hilsdale and J. Hugunin. "Advice weaving in AspectJ," In Proc. of Aspect-Oriented Software Dev. (AOSD'04), March 2004.

[11] G. Kiczales and M. Mezini. "Separation of Concerns with Procedures, Annotations, Advice and Pointcuts," In Proc. of the European Conference on Object-Oriented Programming (ECOOP'05), Springer LNCS, July 2005.

[12] G. Kiczales. "Can AOP really solve the 'logging problem'?" Online posting. AspectJ Users Discussion Forum, April 2003. http://dev.eclipse.org/ mhonarc/lists/aspectj-users/msg00390.html

[13] G. Kiczales. "General Best Practice Question," Online posting. AspectJ Users Discussion Forum, July 2003. http://dev.eclipse.org/mhonarc/ lists/aspectj-users/msg00726.html

[14] C. Koppen and M. Stoerzer. "PCDiff: Attacking the Fragile Pointcut Problem," In Proc. of the European Interactive Workshop on Aspects in Software (EIWAS'04), August 2004.

[15] G. C. Murphy, A. Lai, R. J. Walker, and M. P. Robillard. "Separating Features in Source Code: An Exploratory Study," In Proc. of the International Conference on Software Engineering (ICSE'01), May 2001.

[16] H. Rajan and K. Sullivan. "Eos: Instance-Level Aspects for Integrated System Design," In Proc. of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'03), September 2003.

[17] H. Rajan and K. Sullivan. "Generalizing AOP for Aspect-Oriented Testing," In Proc. of Aspect-Oriented Software Development (AOSD'05), March 2005.

[18] T. Rho, G. Kniesel, and M. Appeltauer. "Fine-Grained Generic Aspects," in Proc. of the AOSD Workshop on Foundations of Aspect-Oriented Languages (FOAL'06), March 2006.

[19] J. Siadat, R. J. Walker, and C. Kiddle. "Optimization Aspects in Network Simulation," In Proc. of Aspect-Oriented Software Development (AOSD'06), March 2006.

[20] K. Sullivan, W. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan, "Information Hiding Interfaces for Aspect-Oriented Design," In Proc. of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'05), September 2005.

[21] K. Sullivan. "Handling Logging and Tracing Concerns," Online posting. AOSD.NET Discussion Forum, July 2005. http://aosd.net/pipermail/discuss_ aosd.net/2005-July/001621.html

[22] T. Tourwé, J. Brichau, and K. Gybels. "On the Existence of the AOSD-Evolution Paradox," In Proc. of the AOSD 2003 Workshop on Software Engineering Properties of Languages for Aspect Technologies, March 2003.

[23] M. Trifu and V. Kuttruff, "Capturing Nontrivial Concerns in Object-Oriented Software," In Proc. of the Working Conference on Reverse Engineering (WCRE'05), November 2005.