Aspectual Reflection

SERGEI KOJARSKI* KARL LIEBERHERR* DAVID H. LORENZ* ROBERT HIRSCHFELD†

AOSD'03 SPLAT Workshop

Abstract

With most of today's aspect-oriented language extensions, developers have both aspectual and core reflection mechanisms available to them. From a software engineering point of view, these mechanisms serve different purposes in different application areas. This paper explores to what extent aspectual and core reflection overlap. Interactions of aspectual and core reflection are discussed based on practical observations in concrete examples.

1 Introduction

Aspect-oriented programming (AOP) is in essence a computational reflection mechanism [15]. The join point model reflects a program's behavior: a join point provides the ability to introspect; advice provides the intercession (manipulation) capability. Aspect-oriented languages (e.g., AspectS [8] and AspectJ [13, 10]) are typically built on top of a base object-oriented programming (OOP) language, which has native support for reflection. Consequently, in addition to its aspectual reflection mechanism (ARM), the aspect-oriented language also supports the underlying OOP core reflection mechanism (CRM). AOP programmers thus have two reflective mechanisms to their disposal.

From a software engineering point of view, the two reflective mechanisms serve different purposes, with different application areas. In AspectJ, for example, Java Core Reflection is mainly used for introspection on structure [4], while aspects are mainly used for intercession on behavior. However, an aspect could just introspect the behavior without inflecting any functional affect on the program; and instantiating objects or invoking methods using reflection does affect the program's run-time structure and behavior.

This paper explores to what extend reflective properties of ARM and CRM overlap. We illustrate how ARM and CRM may interact in beneficial and unexpected ways, resulting in improvement of the software engineering abilities of each other. The paper addresses the fundamental question of whether or not one mechanism subsumes the other.

A contribution of the paper is in giving concrete examples of how the two mechanisms interact, and making practical observations based on those examples. Using Smalltalk and AspectS, we illustrate that traditional AOP semantics can be achieved by reflection; using Java and AspectJ, we illustrate that traditional reflection semantics can be simulated by AOP; and we give examples on how the two can interact collaboratively.

The rest of the paper is organized as follows. Section 2 discusses the implementation of AOP features in AspectS on top of OOP-level Smalltalk reflection. Section 3 describes the implementation of a reflection application programming interface on top of AspectJ without using Java Core Reflection. Section 4 describes a practical software engineering perspective on AOP and Reflection interaction in AspectJ. In Section 5 we conclude by mapping the space of AOSD languages with respect to aspectual and reflective features.

Throughout the paper we use RI and AI as abbreviations for the interfaces provided by reflective and aspectual features, respectively. We denote by RI^R and AI^R the CRM-based implementation, and by RI^A and AI^A the ARM-based implementation, of RI and AI, respectively.

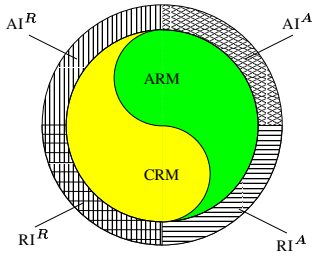

Figure 1: Aspectual Reflection

*College of Computer & Information Science, Northeastern University, 360 Huntington Avenue 161 CN, Boston, Massachusetts 02115 USA. {kojarski,lieber,lorenz}@ccs.neu.edu

†Future Networking Lab, DoCoMo Communications Laboratories Europe, Landsberger Strasse 308-312, D-80687 Munich, Germany. hirschfeld@docomolab-euro.com

2 AOP on Top of Reflection

Historically, the seeds of AOP are founded in metaobject protocols (MOPs) [6] and open implementations (OIs). This fact gave rise to the point of view that reflection is the underlying basis for AOP. Under this view, AOP facilitates disciplined meta-programming, utilizing MOPs [17]. Indeed, AOP can be implemented as an interface to the underlying MOP functionality (AI^R) as exemplified by AspectS.

AspectS is a framework for general-purpose AOP in the Squeak/Smalltalk environment [9]. AspectS supports AOP by means of general Smalltalk meta-programming without any language extensions. Based on concepts of AspectJ, AspectS extends the Smalltalk MOP to accommodate the aspect modularity mechanism.

2.1 Pointcut designators over MOP

AspectS makes use of the full Smalltalk MOP to enumerate join-point descriptors. In the following advice method, the pointcut is determined by asking class `Morph` for all of its subclasses including itself (`#withAllSubclasses`), selecting the ones that have implemented a method named `#mouseEnter:` (`#includesSelector:`), and then collecting join-point descriptors referring to the class implementing `#mouseEnter:` and the `#mouseEnter:` selector itself.

```
AsMorphicMousingAspect≫
adviceMouseEnter

↑ AsBeforeAfterAdvice
 qualifier: (AsAdviceQualifier attributes:
          { #receiverClassSpecific. })
 pointcut: [
  Morph withAllSubclasses
   select: [:each |
    each includesSelector: #mouseEnter:]
   thenCollect: [:each |
    AsJoinPointDescriptor
     targetClass: each
     targetSelector: #mouseEnter:]]
 beforeBlock:
  [:receiver :arguments :aspect :client |
   self showHeader: '>>> MouseENTER >>>'
   receiver: receiver
   event: arguments first]
```

Not only is the Smalltalk MOP expressive enough to describe the pointcut, but for the Smalltalk reader the description is also very clear and precise. In comparison, even the experienced Java programmer may find it difficult to always predict the declarative meaning of a pointcut designator in AspectJ [16].

2.2 Advice over MOP

AspectS makes use of block method wrappers, special wrappers that allow to plug-in block contexts for additional behavior. For each kind of advice there is a matching method wrapper implementation. Method wrappers [3] allow the introduction of code that is executed before, after, or instead of an existing method [1].

According to the attributes stated in an advice qualifier, a method wrapper is configured with one or more activation blocks. Each activation block (closure) is provided with the aspect instance associated with the wrapper, and the base level activation context (base sender) that allows access to not only the receiver of the message, but to the whole chain of activation contexts (Smalltalk's stack).

For example, the following method returns an activation block from the AspectS framework that evaluates to the `true` object if the base-level receiver is an instance previously made known to the aspect:

```
AsMethodWrapper class≫
receiverInstanceSpecificActivator

↑ [:aspect :baseSender |
| result |
result ← aspect
 hasReceiver: baseSender receiver.
aspect ← baseSender ← nil.
result] copy fixTemps
```

Method wrappers override `#valueWithReceiver: arguments:` from `CompiledMethod`. Each wrapper class provides an individual implementation. The following listing shows that of `AsHandlerWrapper`, a wrapper that will be placed by a handler advice in front of another compiled method to add additional computation to be executed if exceptions signalled:

```
AsHandlerWrapper≫
valueWithReceiver: anObject
arguments: anArrayOfObjects

 | client active |
client ← thisContext baseClient.
active ← self isActive.
↑ [
 self clientMethod
  valueWithReceiver: anObject
  arguments: anArrayOfObjects.
] on: self exception do: [:ex |
 active
  ifTrue: [self handlerBlock copy fixTemps
   valueWithArguments: {
    anObject.
```

[1] A method wrapper replaces an entry in a class' method dictionary (a compiled method or another method wrapper), adds behavior to the method invocation, and eventually invokes the wrapped method itself.

```
    anArrayOfObjects.
    self aspect.
    client.
    ex. }]
  ifFalse: [ex pass]]
```

2.3 Trade-offs of using MOP

The less invasive a new feature is, the less difficult its interaction with other features. The benefit of implementing AOP on top of reflection (AI^R) is that it is does not entail changes at the language level, not even changes to the standard lookup process. There is no language extension required to introduce the aspect modularity construct. The aspect modularity construct can be provided via an API to a framework for coordinated meta-programming.

By design, AspectS prevents from changing Smalltalk. Instead of introducing new language constructs, AspectS utilizes the expressiveness of Smalltalk itself as a pointcut language. As an alternative to modifying Squeak's standard lookup process, AspectS employs method wrappers to change the objects that the lookup mechanism returns. Unfortunately, this in not possible with Java/AspectJ.

AI^R also allows more flexible control over aspects. In contrast to systems like AspectJ, weaving and unweaving in AspectS happens dynamically at runtime, on-demand, employing metaobject composition.

However, even if a language posses a rich MOP, certain limitations remain. While AspectS provides AspectJ-like concepts, it is limited due to the limitations of Smalltalk reflection. AspectS was designed to allow the expression of aspects with join-points that match the design center of Smalltalk: objects and message sends between objects. Smalltalk, however, is not solely built on the message send metaphor, but also allows return statements and direct variable access: access to temporary, instance, class, and class instance variables. While Smalltalk's MOP makes it easy to affect message sends, manipulating variable access is impossible just through metaobject composition.

There are also performance issues related to efficiency [7]. Although, mixins in Smalltalk [1], optimistic optimization in Java [17], and other techniques can alleviate this problem to some extent.

Not all languages support a MOP rich enough to enable AOP. Java Core Reflection, for example, is not a MOP and therefore AspectJ cannot be implemented by a Java library as AspectS is implement in Squeak. AspectJ uses preprocessing techniques, which is not strictly reflection-based RI [2].

[2] But the generated code uses RI.

3 Reflection on top of AOP

In this Section we underscore the observation that AOP can be the basis for RI. Even if the base OO language does not provide RI^R, the AOP language can still build one on top of its ARM (RI^A). We illustrate this first for object-level and then for class-level structural reflection.

3.1 Object-level reflection

Consider the manner in which one can access the field `fname` of an object `host` using Java Core Reflection:

```
try {
  host.getClass()
    .getDeclaredField(fname).get(host);
} catch(IllegalAccessException iae) {
  System.out.println("Illegal field access
      : "
    + iae.getMessage());
} catch(NoSuchFieldException nsfe) {
  System.out.println("Field not found: "
    + nsfe.getMessage());
}
```

We can access the field using AOP *without* using reflection by, e.g.,

```
FieldInspector.aspectOf(host).get(fname);
```

where reflection is simulated with the `FieldInspector` aspect:

```
import java.util.HashMap;
aspect FieldInspector
  pertarget(initialization(
    (!FieldInspector).new(..))) {
  before(Object newValue): set(* *.*)
    && args(newValue) {
    fields.put(thisJoinPoint
      .getSignature().getName(),newValue);
  }
  public Object get(String name) {
    return fields.get(name);
  }
  private HashMap fields = new HashMap();
}
```

In the above aspect, **pertarget** guarantees that every object ever created is associated on initialization with an instance of the `FieldInspector` aspect. `FieldInspector.hasAspect(target)` is always true, and `aspectOf(host)` is always defined. For simplicity, `FieldInspector` assumes field names to be unique (and ignores the issue of overridden fields) and (similar to RI^R) boxes primitive values in objects (ignoring default value initialization).

The `FieldInspector` example illustrates that it is possible to provide reflective information about every field of every object without having core reflection. This example can be extended to provide complete structural reflection on the object-level resulting in the AOP-based RI (RI^A). RI^A could even expose structural information beyond what RI^R normally offers. For example, run-time profiling information (e.g., how many times a particular method was executed) is not a part of RI^R, but can be easily provided using AOP.

3.2 Class-level reflection

To complete the picture, we need to also explore reflection over a class structures. Consider the manner in which one can find a superclass of a class `C` using Java Core Reflection API (RI^R):

```
C.class.getSuperclass();
```

Similar to the case with object-level reflection, we can also simulate class-level reflection. We can find the superclass using AOP *without* using RI^R by, e.g.,

```
ClassHierarchy.getSuperclass(C.class)
```

where the class-level structural reflection is simulated with the aspect `ClassHierarchy`:

```
import java.util.HashMap;
aspect ClassHierarchy
 percflow(call(
   (!ClassHierarchy).new(..))) {
 before(): initialization(
   (!ClassHierarchy).new(..)) {
   Class descClass =
       thisJoinPointStaticPart
     .getSignature().getDeclaringType();
   hierarchy.put(descClass,superClass);
   superClass = descClass;
 }
 private static HashMap hierarchy =
   new HashMap();
 public static Class getSuperclass(
   Class desc) {
   return (Class)hierarchy.get(desc);
 }
 private Class superClass;
}
```

The `ClassHierarchy` aspect maintains a hash-map `hierarchy`, which reflects all the classes instantiated directly or indirectly. In comparison, RI^R reflects on classes that were loaded to the JVM [3] therefore accounting for a larger set of classes. Once AspectJ allows byte-code level weaving, advising the `java.lang.ClassLoader` class would eliminate this minor distinction (e.g., using an around advice on the `loadClass` method).

Another approach is to extend existing AOP model with a mechanism of statically executable advice [11]. Such a mechanism would furnish a complete picture of the class structure statically. RI^A could then expose more structural meta information than RI^R does today.

3.3 Tradeoffs

In the introspection sense, RI^A is as complete as RI^R, although the lack of statically executable advice in current AOP implementations limits their reflective capabilities. RI^R and RI^A have performance-space tradeoffs. MOP and Reflection usually entail considerable performance overhead [7] since they handle (traverse and convert) the internal representation of a meta-information. RI^A displays improved performance by directly accessing the meta-information via the hash-map. For the same reasons, however, RI^A is less economical than RI^R, since RI^A caches everything, it actually duplicates the internal information.

With RI^A, the overhead is adjustable: we have control over what to reflect, i.e., we can reflect only on selected join points (classes, objects, etc). RI^R is not configurable. RI^R is available regardless of whether or not it is used. It always uses one source of meta-information (its internal representation) and is tightly coupled to its implementation [**?**]. In contrast, aspect-based RI^A can be (un)pluggable allowing better composability. Furthermore, it gives opportunities to increase compile and runtime efficiency by unplugging RI^A when not needed or not in use.

The join point model provided by AspectJ puts certain limitations on AspectJ-based RI^A. The join point interface does not provide comprehensive static information (e.g., poor reflective abilities over Java interfaces). Furthermore, the `org.aspectj.lang.reflect.MethodSignature` join point interface (which corresponds to the `java.lang.reflect.Method` class in Java) lacks certain meta-method capabilities, e.g., invocation-by-signature (which is provided in RI^R). Nevertheless, the gap between RI^A and RI^R is bridgeable. Statically executable advice along with improved lexical a join point model would eliminate these problems.

4 AOP and Reflection Interaction

From a software engineering perspective, AspectJ has limited aspectual (incomplete static information) and reflective capabilities (bound by Java Core Reflection), which limits its area of applicability. In this Section,

[3] With Java Core Reflection, one cannot reflect on classes until they are loaded to the JVM.

we explore opportunities to overcome these limitations by combining aspectual and reflective features of AspectJ in unexpected ways.

4.1 Reflection within advice

Compile-time weaving promoted by AspectJ in contrast with AspectS dynamic weaving approach gives better run-time performance though less flexibility. Using RI^R within AspectJ's advice could improve aspects run-time flexibility.

An example below illustrates this idea. Aspect `Notifier`, shown at the listing invokes static no-argument methods (listeners) using RI^R method-invocation technique. The basis for a method selection is equality between method name and the name of the join point target class. Notice, that aspect `Notifier` knows nothing about `ClassListeners` class (or even if such a class exists).

Further development of this approach would allow dynamically (de)attach functionality to advised join points providing limited run-time aspects weaving [5, 2].

However, it is possible to get the same functionality as the example provides without using RI^R. But that would entail a dedicated aspect and advice for each method resulting in a loss of a dynamic flexibility.

```
class ClassObserver {
  static void A() {
    System.out.println("A Listener");
  }
  static void C() {
    System.out.println("C Listener");
  }
}
```

```
aspect Notifier {
  before(Object targ): within(!Notifier)
    && call(* *(..)) && target(targ) {
    try {
      String className =
        targ.getClass().getName();
      Class.forName("ClassListeners")
        .getDeclaredMethod(className,null)
          .invoke(null,null);
    } catch (Exception e) {}
  }
}
```

4.2 Generative aspects

Some tasks that rely on structural meta-information cannot be done with either RI^R or AI^A[4]. Consider a lexical style guideline for OO program coding, which states that call-sites within a method should only target instance-variable classes of the enclosing class and argument classes of the method. This style rule is a simplified variant of the Class Form of the Law of Demeter [**?**], and detecting violations of this rule is polynomial and requires only static structural metal-information. The reason the style rule cannot be checked with only RI^R is that it does not expose call-sites within method code. In a related AOSD'03 paper [11] we also prove that it is impossible to check such rules in AspectJ. Specifically, we consider the following aspect:

```
abstract aspect Violation {
  abstract pointcut Violation;
  declare warning: Violation: "Violation";
}
```

where `Violation` refers to the style rule violation. Indeed, there is no way to avoid an **`if`** pointcut primitive in the pointcut designator instantiating `Violation` for the above style rule. But the **`if`** pointcut primitive is not statically determinable and therefore inappropriate in for the **`declare warning`** construct.

It is also impossible to perform a complete check dynamically, because at run-time not all class meta-information is available; and, moreover, not all method calls are covered. This holds independently of whether we use RI^R within the aspect or not.

However, reflection can be employed to generate an aspect that *does* implement the `Violation` aspect successfully using the `generateAspect` method shown in Listing 1.[5] Listing 3 shows the generated aspect for a particular class hierarchy: the code listings for class `A` is shown in Listing 2; classes `B`, `C`, `D`, and interface `I` are omitted.

Listing 1: Aspect generator

```
public void generateAspect(StyleRule rule)
    {
  FileWriter writer = null;
  try{
    writer = new FileWriter(fileName);
    writer.write("aspect LoDViolation
        extends Violation {\n");
    List declarations = rule.
        getPointcutDeclarations();
    for (int i=0;i<declarations.size();i++)
      writer.write((String)declarations.get(
          i)+"\n");
    writer.write("pointcut LoD(): "+rule.
        getRulePointcut()+";\n");
    writer.write("pointcut Violation(): !
        LoD();\n");
    writer.write("}");
```

[4] In the context of Java/AspectJ

[5] This does not contradict our statement in [11].

```
  } catch(Exception e) {
    e.printStackTrace();
  } finally {
    try {writer.close();} catch(Exception
        exc) {}
  }
}
```

Listing 2: A.java

```
class A {
 C c=new C();
 B b=new B();
 void foo() {
    foo();
    c.foo();
    b.foo(c);
    D d=new D();
    d.foo(); // style rule violation
 }
}
```

Listing 3: LoDViolation.java

```
aspect LoDViolation extends Violation {
  pointcut Global(): within(*) && call
      (* *.*(..));
  pointcut A(): within(A) && call(* (!(B
       || C)).*(..));
  pointcut A_foo(): withincode(* A.foo())
       && call(* (!(A)).*(..));
  pointcut B(): within(B) && call(* (!(I))
      .*(..));
  pointcut B_foo_C(): withincode(* B.foo(C)
      ) && call(* (!(B || C)).*(..));
  pointcut C(): within(C) && call(* *.*(..)
      );
  pointcut C_foo(): withincode(* C.foo())
       && call(* (!(C)).*(..));
  pointcut D(): within(D) && call(* *.*(..)
      );
  pointcut D_foo(): withincode(* D.foo())
       && call(* (!(D)).*(..));
  pointcut LoD(): (Global() && ((A() && (
      A_foo())) || (B() && (B_foo_C()))
       || (C() && (C_foo())) || (D() && (
      D_foo()))));
  pointcut Violation(): !LoD();
}
```

5 Conclusion

In a nutshell, this paper heightens the awareness that CRM and ARM are alternative providers of computational reflection (Figure 2). Both CRM and ARM exhibit aspectual and reflective features. AOSD approaches differ in the features they use. AspectS combines the aspectual and reflective features of the Smalltalk MOP. AspectJ combines the reflective features of Java with new aspectual features.[6]

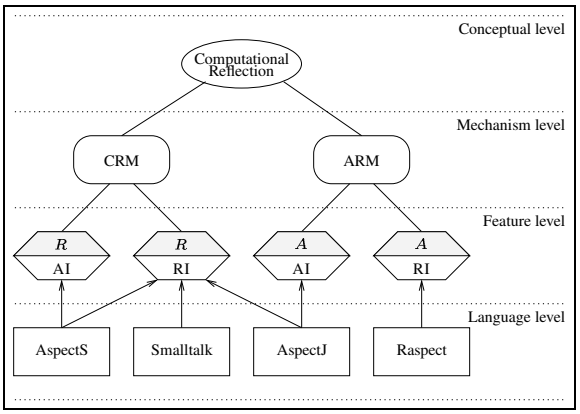

Figure 2: Computational Reflection Space

The leftmost feature-level hexagon in Figure 2 denotes the implementation of an aspectual interface over reflection, namely AI^R, as was demonstrated in Section 2. The rightmost hexagon in Figure 2 denotes the implementation of a reflection API over AOP, namely RI^A, as was demonstrated in Section 3. The two center hexagons, RI^R and AI^A, represent the traditional implementation of reflection and AOP.

Figure 2 also maps the space of AOSD languages. Pure OOP languages, like Smalltalk, utilize just the reflective features of CRM. Analogously, a pure AOP languages would be one that utilizes just aspectual features. In the figure, we propose a novel *Raspect* language, which utilizes only the reflective features of ARM.

We have identified, analyzed, and provided examples of explicit aspectual support in CRM and explicit reflective support in ARM, and illustrated (Sections Section 2 and Section 4) AOP and Reflection interaction in AspectS and AspectJ from a practical software engineering perspective. Understanding the software engineering tradeoffs could also impact future language design and implementation.

References

[1] L. Bak, G. Bracha, S. Grarup, and R. Griesemer. Mixins in strongtalk. In B. Magnusson, editor, *The Inheritance Workshop at ECOOP 2002*, Málaga, Spain, June11 2002. ECOOP 2002.

[6] The mapping of other AOSD languages, e.g., Demeter, DJ, Composition Filters, HyperJ, and Fred, is omitted, but would be an interesting question for discussion in the workshop...

[2] J. Baker and W. Hsieh. Runtime aspect weaving through metaprogramming. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, pages 86–95, Enschede, The Netherlands, Apr. 2002. AOSD 2002, ACM Press.

[3] J. Brant, B. Foote, R. E. Johnson, and D. Roberts. Wrappers to the rescue. In E. Jul, editor, *Proceedings of the 12th European Conference on Object-Oriented Programming*, number 1445 in Lecture Notes in Computer Science, pages 396–417, Brussels, Belguim, July 20-24 1998. ECOOP'98, Springer Verlag.

[4] S. Chiba. Load-time structural reflection in java. In E. Bertino, editor, *Proceedings of the 14th European Conference on Object-Oriented Programming*, number 1850 in Lecture Notes in Computer Science, pages 313–336, Cannes, France, June 12-16 2000. ECOOP 2000, Springer Verlag.

[5] P.-C. David, T. Ledoux, and N. M. N. Bouraqadi-Saâdani. Two-step weaving with reflection using aspectj. In *Proceedings of the 16th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Tampa Bay, Florida, Oct. 14-18 2001. OOPSLA'01, ACM SIGPLAN Notices 36(11) Nov. 2001.

[6] J. d. R. G. Kiczales and D. G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[7] Y.-G. Guéhéneuc. Overall impression on the AOP workshop. `http://www.yann-gael.gueheneuc.net/Work/Publications/Documents/Trip+rep%ort+AOP+Workshop01.doc.pdf`, May 2001.

[8] R. Hirschfeld. AspectS—aspect-oriented programming with Squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Architectures, Services, and Applications for a Networked World*, number 2591 in Lecture Notes in Computer Science. Springer Verlag, 2002.

[9] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of squeak, a practical smalltalk written in itself. In *Proceedings of the 12th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 318–326, Atlanta, Georgia, Oct. 5-9 1997. OOPSLA'97, Addison-Wesley.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 18-22 2001. ECOOP 2001, Springer Verlag.

[11] K. Lieberherr, D. H. Lorenz, and P. Wu. A case for statically executable advice: Checking the Law of Demeter with AspectJ. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 40–49, Boston, Massachusetts, Mar. 17-21 2003. AOSD 2003, ACM Press. To Appear.

[12] K. J. Lieberherr, I. Holland, and A. J. Riel. Object-oriented programming: An objective sense of style. In N. K. Meyrowitz, editor, *Proceedings of the 3rd Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 323–334, San Diego, California, Sept. 1988. OOPSLA'88, ACM SIGPLAN Notices 23(11) Nov. 1988.

[13] C. V. Lopes and G. Kiczales. Recent developments in AspectJ. In S. Demeyer and J. Bosch, editors, *Object-Oriented Technology. ECOOP'98 Workshop Reader*, number 1543 in Lecture Notes in Computer Science, pages 398–401. Workshop Proceedings, Brussels, Belguim, Springer Verlag, July 20-24 1998.

[14] D. H. Lorenz and J. Vlissides. Pluggable reflection: Decoupling meta-interface and implementation. Technical Report NU-CCS-02-10, College of Computer and Information Science, Northeastern University, Boston, MA 02115, Sept. 2002. To appear in International Conference on Software Engineering, 2003.

[15] P. Maes. Concepts and experiments in computational reflection. In N. K. Meyrowitz, editor, *Proceedings of the 2nd Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 147–155, Orlando, Florida, Oct. 4-8 1987. OOPSLA'87, ACM SIGPLAN Notices 22(12) Dec. 1987.

[16] T. Skotiniotis, K. Lieberherr, and D. H. Lorenz. Aspect instances and their interactions. Unpublishded, 2003.

[17] G. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Communications of the ACM*, 44(10):95–97, 2001.