Understanding Aspects
(Extended Abstract)

Mitchell Wand*
College of Computer and Information Science
Northeastern University
Boston, MA 02115, USA
wand@ccs.neu.edu

1 Background

Since the mid-90's, Kiczales and others have identified new limitations of traditional layered architectures for complex systems. The most serious of these new limitations is the problem of *scattering*. A typical large system requires many services, such as logging, locking, authentication, etc. While the functionality to implement a particular service may be located in a single module, the code to invoke the service is typically scattered across many modules.

Kiczales's solution was to use a special-purpose language to give a declarative specification of where or when each service should be invoked. A service typically takes the form of a class called an *aspect*, possibly containing several methods. The potential points at which a method of an aspect can be invoked are called *join points*, and each method of the aspect is associated with a *point cut*, which is the set of join points at which it should act. Kiczales called this *aspect-oriented programming* (AOP). This idea has attracted widespread attention in the software-engineering community.

Different aspect-oriented programming languages use different join-point models and different ways to invoke services. Join points may be temporal entities or spatial ones. A typical join point in AspectJ [4] is a temporal event: the execution of a method call along with an abstraction of the call stack at that call. The invocation of a service takes the form of advice [2] on the execution of the method call. In Hyper/J [7], the join points are spatial: they are class definitions and method definitions within those classes, and actions typically take the form of merging methods from different modules into a single composite class.

The difficulty with existing AOP languages is that they defeat modular reasoning as we know it. Modular reasoning usually means that we can reason about a module independent of the context in which it is installed, and that we can deduce the behavior of a composite module from the behavior of its components. Typical AOP languages, most notably AspectJ, violate both of these principles. Wand et al [9] and Jagadeesan et al [3] have given examples of program fragments that are contextually equivalent in ordinary contexts, but that may be distinguished by aspect-oriented contexts. Among the contextual equivalences that fail are most of the transformations associated with refactoring, such as moving a method from one class to another.

*Work supported by the National Science Foundation under grant number CCR-0097740.

2 Reconceptualizing AOP

To the extent that modular reasoning forms the conceptual foundation for good software engineering principles, the failure of modular reasoning in current AOP systems is cause for concern. Our goal in this talk is to discuss a reconceptualization of AOP that we hope will allow an eventual reconciliation between AOP and modular reasoning.

A number of researchers [3, 8, 9] have provided semantics for advice; our goal is to abstract away from the mechanism of advice to consider aspect-orientation in general, along the lines of Masuhara and Kiczales [6].

Our reconceptualization takes the form of three theses:

Scattering is inevitable. A specification, such as

```
pop(push(x,s)) = s
```

specifies a conspiracy between the implementation of `push` and the implementation of `pop`. These two procedures must conspire to produce the specified behavior. When the conspirators are in the same module, as they are in a typical ADT, this leads to locality of knowledge: if we wish to change the representation of stacks, we need only change the implementations of `push` and `pop`.

But in the kinds of applications for which AOP is useful, the specification involves conspirators in different modules. We may wish to specify, for example, a policy for when an authentication service should be invoked. Such a policy specification necessarily involves entities both from the underlying application and from the invoked service.[1] If we wish to change the policy specification or the representation of some entity mentioned in the policy, we will most likely need to change the implementations of all the actors in the conspiracy, which may be scattered across many modules.

[1]We have phrased this asymmetrically, in terms of the "underlying application" or "base program," but the analysis would be unchanged if the system consisted entirely of communicating aspects.

How, then, do we restore some notion of modularity?

Aspects are modular units of specification. Modularity can be restored by thinking not about programs, but about specifications. Each aspect of a complex system, be it authentication, logging, or display, is probably specified separately, using its own names and its own logic or specification language. The great innovation of AOP is the introduction of declarative *policy languages* (in AspectJ, the language of point-cut descriptors) for specifying the interaction of the aspects.

The idea of aspects as units of specification was more evident in early AOP languages like COOL and RIDL [5]. They used special-purpose declarative languages for the specification of particular aspects. Current AOP languages instead use full-fledged programming languages as surrogates for the specification of the individual aspects, using declarative policy languages only to specify the way in which the aspects should interact.

A join-point model is a joint ontology. In a modular specification, each module or aspect is likely to define its own entities; these entities constitute an *ontology* for that module. A policy specification, by its very nature, constrains the behavior of named entities in a number of different modules. For this to be sensible, these modules must agree on the meanings of these names. There must be a *joint ontology* on which all the participants in the policy must agree. The participants must agree on what entity each name refers to in the real world, and on how that entity is represented; otherwise, they cannot communicate. For example, a client that expects services to be provided by procedure calls cannot directly communicate with a server that provides services by method calls, because the client and server do not have a shared ontology.

Of course, each participant may bring additional knowledge about these entities. The joint ontology refers to that part of the participants' knowledge that is held in common. The join points are distinguished entities in that joint ontology.

For example, in AspectJ, the entities in the joint ontology are events like the execution of a method call; the ontology determines what data is associated with each such event (typically, the signature of the method, the values of the parameters, and similar data for each method call on the stack). The aspects and the base program share this common model of the operation of the underlying machine. In Hyper/J, the joint ontology represents a repository containing code. The join point entities in its ontology are class definitions and method definitions within those classes. Other AOP languages can be analyzed similarly.

3 Implications

In the talk, we will consider the implications of this reconceptualization of AOP. We will see how existing AOP languages fit this model, and how other widely-used systems, not usually considered in this way, draw their power from aspect orientation.

This perspective suggests some research directions. Here some examples:

Domain- and application-specific ontologies Existing aspect languages like AspectJ and Hyper/J are, by design, general-purpose languages, and therefore have general-purpose joint-point models. These ontologies may be either too coarse or too fine for a particular application. Can we develop languages and tools that will let a system designer design an ontology and join-point model that is appropriate to a specific application or domain? UML and OCL [1, 10] provide an approximation to this kind of facility; can we do better?

Domain- and application-specific aspect languages In the talk, we will show an example of an extremely widely-used aspect-oriented language, and see how its power comes from being domain-specific. Can we work with domain experts to provide other, better examples? What kinds of tools can we build to help develop and support such languages?

Ontology-respecting equivalence When we say "equivalent in all contexts", we do not usually really mean *all* contexts. We usually mean something like all *type-respecting* contexts. Can we develop a similar notion of equivalence in all *ontology-respecting* contexts? Can such a notion be used to restore a notion of modular reasoning for aspects that respect ontologies?

4 References

[1] M. Fowler and K. Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, New York, 1997.

[2] R. P. Gabriel, J. L. White, and D. G. Bobrow. CLOS: Integrating object-oriented and functional programming. *Communications of the ACM*, 34(9):28–38, Sept. 1991.

[3] R. Jagadeesan, A. Jeffrey, and J. Riely. A calculus of untyped aspect-oriented programs. In *Proceedings European Conference on Object-Oriented Programming*, 2003. To appear.

[4] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Berlin, Heidelberg, and New York, 2001. Springer-Verlag.

[5] C. V. Lopes and G. Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010, Xerox PARC, Palo Alto, CA, USA, February 1997.

[6] H. Masuhara and G. Kiczales. Crosscutting in aspect-oriented mechanisms. In *Proceedings European Conference on Object-Oriented Programming*, 2003. To appear.

[7] H. Ossher and P. Tarr. Hyper/J: multi-dimensional separation of concerns for Java. In *Proceedings of the 22nd International Conference on Software Engineering, June 4-11, 2000, Limerick, Ireland*, pages 734–737, 2000.

[8] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, 2003. This proceedings.

[9] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *TOPLAS*, 2003. To appear.

[10] J. Warmer and A. Kleppe. OCL: The constraint language of the UML. *Journal of Object-Oriented Programming*, May 1999.