Aspect-Oriented Programming using Reflection and Metaobject Protocols

Gregory T. Sullivan
Artificial Intelligence Laboratory
Massachusetts Institute of Technology
gregs@ai.mit.edu
http://www.ai.mit.edu/projects/dynlangs/

April 27, 2001

1 Introduction

Some of the original inspiration for Aspect-Oriented Programming (AOP) [KLM+97] draws from the research in dynamic, reflective object-oriented languages and metaobject protocols (MOPs) [KdB91][1]. A MOP lets the programmer delve "under the covers" and programmatically affect basic language mechanisms such as dynamic method dispatch and class instantiation. These powerful facilities enable the sort of crosscutting metaprogramming that AOP strives to deliver. We believe that there are two reasons why research on providing aspect-oriented programming features to the programmer has strayed from its metaobject protocol roots:

Too much rope: Metaobject protocols, while elegant, are complicated. Providing access to the implementation of a language's runtime may be placing too much power and complexity in the hands of the programmer.

Too much overhead: It is generally assumed that the presence of a runtime MOP has too negative an impact on performance to be worthwhile.

[1] The astute reader will notice that many of the same personalities are involved in MOP's and AOP.

Research groups at MIT and elsewhere are revisiting the fundamental ideas of reflection and metaobject protocols, applying them to aspect-oriented programming, and inventing techniques to minimize the potential drawbacks mentioned above.

In the remainder of this article, we give a brief overview of reflection and metaobject protocols, show how they are excellent tools for aspect-oriented programming, and give an overview of techniques that address both language design and language implementation concerns.

2 Reflection and Metaobject Protocols

Computational reflection [Smi82, Mae87] enables a program to access to its internal structure and behavior and also to programmatically manipulate that structure, thereby modifying its behavior. The Java programming language provides some reflection capability. For example, a Java program can ask for the class of a given object, find the methods on that class, and then invoke one of those methods. Some research groups, such as the DJ project at Northeastern University [OL01], take advantage of Java reflection to implement aspect-oriented features.

A *metaobject protocol* defines execution of an application in terms of behaviors implemented by metaclasses (e.g. `Class` or `VirtualFunction`). For example, dynamic method dispatch may involve a method named `dispatch` on virtual functions that takes as arguments the values for a given call. The `dispatch` method would determine the most applicable method given the arguments, and then chain to that method implementation. A programmer could override the default behavior of the `dispatch` method in order to affect what happens when a virtual function is called.

Java's built-in reflection capabilities fall short of a full metaobject protocol in two important respects:

1. Java's reflection is "read-only". For example, a program can query the methods of a class, but a program cannot dynamically *change* the methods of a class. Full reflection allows modification of any meta-information that can be reified.
2. Java does not allow subclassing of metaclasses such as `Class` and `Method`. With a full MOP, subclassing metaclasses is a way to incrementally change the default behavior of a language.

In the terminology of [KLM+97], Java provides *introspection* but not *intercession*. Java does provide some dynamism with the fairly heavyweight mechanism of dynamic class loading. Other mainstream programming languages, such as C++, provide even less in the way of computational reflection.

A research group at E.M.N. in France [DS01] is working on providing metaobject protocols using reflection in support of aspect-oriented programming.

3 Aspect-Oriented Programming using Metaobject Protocols

A metaobject protocol allows the programmer to incrementally modify the default behavior of a programming language. For example, to effect the *before*, *after*, and *around* advice of AspectJ (see related article), we may choose one of the following strategies:

1. *Specialize the default behavior.* If we want to add behavior to every call of a set of virtual methods, we can specialize the metaobject protocol's `dispatch` method to each such virtual method. The version of `dispatch` for a specific virtual method would perform the aspect-specific behavior and then chain to the default `dispatch` method.
2. *Dynamically replace methods.* A full reflection protocol allows runtime method redefinition. If we can identify which application methods are affected by an aspect definition, we can replace the default implementations of the methods with "woven" versions.

In [Böl99], Böllert uses reflection in Smalltalk to dynamically add aspect behavior via inheritance and dynamic method definition.

4 Aspect Languages using Metaobject protocols

Aspect-oriented programming is all about enabling the programmer to concisely address functionality that may crosscut the actual implementation of their application. This may be accomplished by using a "general purpose" aspect-oriented language such as AspectJ, or more "concern-specific" aspect languages such as in [LK97, Sei99]. Either way, the existence of a robust metaobject protocol provided by the host programming language makes implementing aspect languages much more straightforward.

Thus it is through aspect-oriented languages that we tame the complexity and power of metaobject protocols. An aspect language is the interface to the functionality of a metaobject protocol.

Jonathan Bachrach at MIT has developed a *The Java Syntactic Extender*, a procedural macro system for Java [Bac01]. A powerful macro system, combined

with a metaobject protocol, allows the programmer to design *domain-specific aspect languages* for manipulating specific crosscutting concerns.

For example, if the runtime exposes facilities for monitoring system load and distributing processes, it is straightforward to write macros that facilitate programmer control over dynamic process distribution.

5 Optimistic Optimization

It has been observed that, when a metaobject protocol is available, uses of the more powerful and dynamic features of the metaobject protocol are relatively rare; that is, most individual methods do not use advanced features of the metaobject protocol. Also, for any given application, use of the metaobject protocol will tend to be fairly constrained and predictable. We take advantage of these observations by using *optimistic optimization*. The idea is that, after an application starts running, we produce, using *partial evaluation* [JGS93, Sul01] techniques, specialized versions of the application's methods that are optimized assuming that mutable parts of the metaobject protocol will not change. All such optimistic optimizations are *guarded*, so that if the assumptions upon which the optimizations are based are ever violated, the optimizations are undone.

For example, we apply standard optimization techniques to call sites, but if at runtime there are changes to the dispatch mechanisms exposed by the metaobject protocol, we may have to undo those call site optimizations.

6 Summary

We advocate providing reflection (both introspection and intercession) as part of a language implementation, and we think that can be done without a large performance penalty. A metaobject protocol then exposes, in a principled way, crosscutting aspects of a running application. Finally, aspect languages, built on top of a metaobject protocol, give the programmer a structured tool for manipulating the exposed concerns.

References

[Bac01] Jonathan R. Bachrach. The Java Syntactic Extender. See `http://www.ai.mit.edu/~jrb/jse`, April 2001.

[Böl99] Kai Böllert. On weaving aspects. In *Proceedings of Aspect-Oriented Programming Workshop at ECOOP'99, Lisbon, Portugal*, June 1999.

[DS01] R. Douence and M. Südholt. A generic reification technique for object-oriented reflective languages. In *Higher-Order and Symbolic Computation*, volume 14(1). Kluwer, 2001. http://www.emn.fr/cs/research/teams/object/Welcon

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, International Series in Computer Science, June 1993. ISBN number 0-13-020249-5 (pbk).

[KdB91] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge (MA), USA, 1991.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, New York, NY, June 1997.

[LK97] Cristina Videira Lopes and Gregor Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010, P9710047, Palo Alto, CA, USA, February 1997.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In Norman Meyrowitz, editor, *Proceedings of the 2nd Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, pages 147–155, Orlando, FL, USA, October 1987. ACM Press.

[OL01] Doug Orleans and Karl Lieberherr. DJ: Dynamic Adaptive Programming in Java. Technical Report NU-CCS-2001-02, College of Computer Science, Northeastern University, Boston, MA, March 2001.

[Sei99] Lionel Seinturier. Jst: An object synchronization aspect for java. In *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP99*, 1999.

[Smi82] B. Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, Massachusetts Institute of Technology, 1982. Laboratory of Computer Science TR 272.

[Sul01] Gregory T. Sullivan. Dynamic partial evaluation. In Olivier Danvy and Andrzej Filinski, editors, *Programs as Data Objects 2*, volume ? of *LNCS*, page ? Springer-Verlag, May 2001.