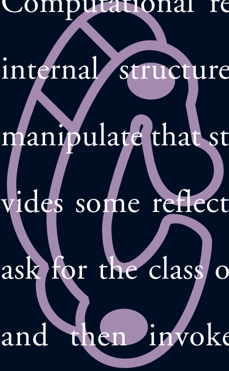
ASPECT-ORIENTED PROGRAMMING USING REFLECTION AND METAOBJECT PROTOCOLS

Providing programmers with the capability to modify the default behavior of a programming language.

Gregory T. Sullivan



Computational reflection [5, 7] enables a program to access its internal structure and behavior and also to programmatically manipulate that structure, thereby modifying its behavior. Java 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 [6], take advantage of Java reflection to implement aspect-oriented features. A metaobject protocol (MOP) defines execution of an application in terms of behaviors implemented by metaclasses (examples include `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 MOP:

- 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 metainformation that can be reified.
- 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.

Java provides 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. Various research projects, including one at E.M.N. in France [3] have been working on providing MOPs using reflection in support of AOP. A MOP allows the programmer to incrementally modify the default behavior of a programming language. For the before, after, and around advice of AspectJ (see the article by Kiczales in this issue), one of the following strategies might be utilized:

- Specialize the default behavior. For example, if we want to add behavior to every call of a set of virtual methods, we can specialize the MOP's dispatch method to each 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.
- 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 [2], Böllert uses reflection in Smalltalk to dynamically add aspect behavior via inheritance and dynamic method definition.

Advantages of Using a Runtime MOP

A MOP allows one to implement aspect code directly, without a static compilation phase. This is beneficial in several ways. Aspect behavior is more robust in the face of dynamic events such as class loading. Aspect languages based on static weaving are fragile with respect to dynamic class loading, because the newly loaded class code may not have been processed by the weaver. Additionally, the aspect logic has the entire runtime state and control at its disposal. The points at which instrumentation can take place are not limited by those points identified and modified by a static compilation process.

Another benefit is that the aspect code can make decisions based on actual runtime values. Thus, the aspect code can dynamically decide how to instrument the running application. Furthermore, the aspect code can evolve over time, based on runtime data. The AO behavior can be both added and removed at runtime.

In short, all conceivable AO features that can be implemented by a compilation (also known as weaving) implementation can also be implemented by a runtime MOP. Furthermore, the AO behavior can be controlled and monitored more flexibly and with finer granularity when implemented with a runtime MOP. This is all well and good, but we need to answer the following questions: How can we provide the powerful functionality of MOPs without overwhelming the programmer? Can we provide the functionality of a runtime MOP with performance comparable to more static approaches? We are making positive progress addressing both of these questions.

It has been observed that, when a MOP is available, uses of the more powerful and dynamic features of the MOP are relatively rare; that is, most individual methods do not use advanced features of the MOP. Also, for any given application, use of the MOP 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 [4, 8] specialized versions of the application's methods that are optimized assuming mutable parts of the MOP 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 are allowed to apply standard optimization techniques to call sites, but if at runtime there are changes to the dispatch mechanisms exposed by the MOP, we may have to undo call site optimizations.

Assuming we have a full-featured MOP available at runtime, how do we expose some of its features to the programmer? As argued here, it is straightforward to implement the features of a general-purpose AO language such as AspectJ using a MOP. It should be noted that we have nothing against using static techniques to "precompile" aspect behavior and take some of the burden off the MOP. Jonathan Bachrach at MIT has developed the Java Syntactic Extender, a procedural macro system for Java [1]. A powerful macro system, combined with a runtime MOP, allows the programmer to design domain-specific aspect languages for manipulating specific crosscutting concerns. For example, if facilities for monitoring system load and distributing processes are exposed at runtime, it is straightforward to write macros that facilitate programmer control over dynamic process distribution. **C**

References

1. Bachrach, J.R. *The Java Syntactic Extender*; www.ai.mit.edu/~jrb/jse, April 2001.
2. Böllert, K. On weaving aspects. In *Proceedings of Aspect-Oriented Programming Workshop at ECOOP'99*, (Lisbon, Portugal, June 1999).
3. Douence, R. and Südholt, M. A generic reification technique for object-oriented reflective languages. In *Higher-Order and Symbolic Computation 14*, 1. Kluwer, 2001.
4. Jones, N.D. Gomard, C.K. and Sestoft, P. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International International Series in Computer Science, June 1993.
5. Maes, P. Concepts and experiments in computational reflection. In *Proceedings of the 2nd Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, (Orlando, FL, Oct. 1987).
6. Orleans, D. and Lieberherr, K. *DJ: Dynamic adaptive programming in Java.* Tech. Rep. College of Computer Science, Northeastern University, Boston, MA, March 2001.
7. Smith, B. Reflection and semantics in lisp. In *Conference Record of POPL '84: The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1984.
8. Sullivan, G.T. Dynamic partial evaluation. In O. Danvy and A. Filinski, Eds., *Programs as Data Objects 2*, Springer-Verlag, May 2001.

Gregory T. Sullivan (gregs@ai.mit.edu) is a research scientist with the Artificial Intelligence Laboratory at MIT in Cambridge, MA.