Object-Oriented Reflective Metalevel Architectures: Pyrite or Panacea?

A Position Paper for the ECOOP/OOPSLA '90 Workshop
on Reflection and Metalevel Architectures

Brian Foote

Dept. of Computer Science
1304 W. Springfield
Urbana, IL 61801
(217) 333-3411
foote@cs.uiuc.edu

31 July 1990

Introduction

Recent research on reflective object-oriented languages and metalevel architectures has generated a great deal of excitement, as well as a certain degree of skepticism among members of the object-oriented community. Reflective approaches appear to hold out the promise of dramatically changing the way that we *think about*, *implement*, and *use* programming languages and systems. There are those who fear, however, that by opening the door to unrestricted language level access to the unwashed masses, we are opening up a Pandora's Box.

It is customary at functions such as this one to attempt to stake out an extreme position on the matter at hand so as to properly highlight the issues and provoke discussion. In this instance, I intend to align myself squarely with the zealots.

Programming with programmable objects

Object-oriented programming languages and systems are having a profound impact on the way that we organize, design, implement, and maintain software. These techniques allow us to treat software components and systems alike as durable, yet malleable artifacts, which evolve along with their requirements [Foote 1988] [Johnson 1988a]. It is, I trust, unnecessary to further elaborate upon the general benefits of object-oriented approaches given the nature of this group.

The current surge of interested in object-oriented reflection and metalevel architectures is, I believe, based on the observation that object-oriented languages and programs are as much themselves an appropriate domain for object-oriented techniques as are windowing systems, operating systems, or accounting systems. The vision underlying this observation is one of a programming system in which the language definition itself is distributed across a constellation of objects which are themselves subject to dynamic scrutiny and modification.

Such a system would allow users to construct new language level objects which would stand on an equal footing with previously existing features. A language built of such programmable objects would be arbitrarily extensible, and would permit language as well as application level objects to be utilized to help the system adapt and evolve as requirements change. This is far from the only potential consequence of such an organization, however. Reflective approaches offer the possibility of bringing areas as disparate as programming language design, compiler construction and code generation, programming environments, debugging, tracing, parallel programming, database systems, operating systems, computation theory, and programming language semantics together under a single umbrella.

A particularly intriguing consequence of this approach is that the components of such a system can themselves serve as the basis for a uniform (pure) object-oriented framework that can support an evolving family of different programming approaches and paradigms. The exploration of the properties of such a framework is a central focus of my own current research [Foote 199?].

Existing programming languages limit extensibility

Modern programming languages owe much of their power to their ability to allow programmers to compose new abstractions using a handful of built -in facilities. This is particularly true of object-oriented languages. However, in most languages, there is a point beneath which the programmer cannot reach [Foote 1989]. The fundamental structures that define how the language itself is implemented are usually either implicit, immutable, or entirely hidden from view. It is useful to think of these objects as comprising the metaarchitecture of the language. Those structures that are explicitly manipulated by a typical program can be thought of as comprising a languages manifest architecture (or simply, architecture).

An *object-oriented* language with a *reflective metalevel architecture* can allow this goal to be achieved. A collection of such objects can server as a *framework* for a family of related metaarchitectures, and hence can facilitate experimentation with such architectures, as well as their orderly evolution.

Reflection permits dynamic metalevel access

A *reflective* computational system [Smith 1983] is one that is able to inspect, manipulate, and alter a full, explicit, active, *causally connected* representation of its own internal structures. The components of this representation are called *reifications* of this underlying structure [Friedman 1984]. "Causally connected" means that any changes made to part of a process's self-representation are immediately *reflected* in its actual state and behavior. The ability to inspect (but not alter) the objects that implement a system is sometimes referred to as *introspection*.

A system's *metalevel* is comprised of those entities that pertain to, represent, or support other computational objects. In systems constructed using metacircular interpreters [McCarthy 1965] [Sussman 1978] these interpreters are programs, written in the system's underlying tongue, that define the meaning of programs running beneath them. However, programs running at a given level were usually incapable of inspecting or affecting the structures of that defined their interpreters.

Smith's 3-LISP language [Smith 1982] [Smith 1983] demonstrated how reflection could be incorporated into a Lisp-based language. In 3-LISP, user level code may specify code that is run at the level of its interpreter. This code may gain explicit access to aspects of its state at this level that are implicit at the user level.

Object-oriented reflection permits a distributed metalevel architecture

Maes's 3-KRS [Maes 1987a] [Maes 1987b] was the first language to incorporate reflection into an object-oriented framework (or vice versa). The combination of reflection with an object-oriented metalevel architecture dramatically increases the power and practicality of reflection. The metalevel of an object-oriented system can be distributed across a constellation of objects, each of which reifies certain aspects of the overall structure. By contrast, Lisp-based metacircular interpreters must often funnel the interpretation process through a single monolithic case statement in their basic eval functions.

A distributed architecture allows local customization

A metalevel architecture that distributes its structure across a family of interacting objects thereby permits individual constituents of this structure to be specialized or pre-empted in the same way as user level objects. That is to say, the full power of the object-oriented approach can be brought to bear on the metalevel. Such an architecture can permit modifications to a language's default behavior to be made on a localized, or even per-object basis.

A useful way to think about this distinction is this: Dynamically modifying a component of a distributed representation is a little like removing your own appendix. Constructing and invoking a new metacircular interpreter is like swallowing yourself whole.

Language Design: the Hamiltonians vs. the Jeffersonians

Allowing the programmer to make local modifications to the existing components of a programming language reflects a distinctly Jeffersonian philosophy towards language design. It is undeniable that such power has a high abuse potential. The price of freedom, after all, is responsibility.

One can make the case that the Hamiltonian view that the definition of the language itself should beyond the reach of the programmer has unnecessarily hamstrung language as well as application evolution. The traditional view of language design has been that a programming language emerges, fully formed and carved in stone by a single hand, from some castle in the Alps. Mistakes made by the designers of such languages may persist for a generation, until such languages are supplanted altogether. Consider Pascal's inflexible array dimensions. The programs written in these languages must then be translated somehow, or die ignominious deaths. This approach makes it impossible for a language to attempt to cope with new challenges, such as the need to support parallelism, or a persistent object store, or to support application specific extensions.

I believe the argument that reflection is dangerous because it gives programmers too much power is a specious one. Existing programming languages already give clumsy programmers more than ample opportunities to shoot themselves in the foot. I'll concede that reflective systems allow the clumsy programmer to fire several rounds per second into his foot, without reloading. Still, I'm confident that, as is the case with features such as pointers, competent programmers will make appropriate use of the power of reflection with care and skill. Potential abuse by inept programmers should not justify depriving the programming community of a potentially vital set of tools.

Certainly, metaprogramming is not for everyone. Most users will not need to resort do designing and building, as opposed to using, reflective facilities. Metaprogramming should not be undertaken frivolously. However, a language show not be an obstacle to its users, or to its own evolution. The age of Software Stalinism is past.

Metaobjects and Metainstances

A characteristic of any expanding line of research is a certain amount of terminological inconsistency. So it is in this area. For instance, the term *metaobject* is use by some researchers [Maes 1987a,b] [Watanabe 1988] [Ferber 1989] to refer to a metalevel object that represents and implements its referent. Researchers in the CLOS community [Bobrow 1988] [Kiczales 1990] [Des Rivieres 1990] use the term metaobject as a generic term to refer to a whose range of different metalevel entities. I've found myself subscribing to the latter usage of late, since an ability to loosely use a term to refer to generic metalevel entities seems to be a useful one. I've been using the term *metainstance* to refer to the former notion. This notion retains the meaning of the original term, and may be slightly more precise, and liberates the term metaobject so that we are free to use it as terminological vulgate. This name change is by no means intended to disparage the importance of metainstances. This notion, as exemplified in ABCL/R, for instance, is central to the construction of meaningful metaarchitectures.

A related issue surrounds the persistent confusion that results from the different ways in which the term *metaclass* is used. Is metaclass a relation between two objects, i.e. class of the class of x? Is a metaclass any instance of the class Metaclass? Does the :metaclass option in CLOS define the class or metaclass of the class with which it is used? My current feeling is that the term is so inherently subject to confusion that it should be banned in all contexts except the first one above.

Another interesting definitional issue is the precise boundary between the application and metalevels. As an extreme example, one can ask the question: Is assignment itself an operation that allows a program to modify a causally connected representation of some aspect of its own state?

Manifest Architecture vs. Metaarchitecture

It is useful to think about programming languages as being implemented in terms of a set of metaobjects which are manipulated by programmers to compose programs. Such an analysis is instructive even with those languages that were defined in terms of the traditional declarative/linguistic metaphor. For instance, once can talk about the properties of a procedure object in Pascal, even though the only meaningful operations on such an object might be declaration (or instantiation)

The *manifest architecture* (or application architecture, or just plain architecture) of a programming language is comprised of those elements of a programming language that are employed by a programmer to address a problem in the application domain at hand.

The *metaarchitecture* defines the nature of the palette of objects which a programmer defines to construct an application program. The metaarchitecture defines how elements of the application architecture are implemented.

Most all languages have both a manifest architecture and a metaarchitecture of some sort. In many traditional languages, the only metaoperations supported for most metaobjects are definition and redefinition.

In some languages, the metaarchitecture is *implicit*, and in others it is *explicit*. A language with an implicit metaarchitecture is one in which the programmer is encouraged to think about the program as being implemented in terms of notions with a certain sort of mechanics, even though these objects are not part of the language specification itself, and are nowhere to be seen. A language with an explicit metaarchitecture overtly acknowledges the existence of such elements in its metaarchitecture.

Metaobjects may be *accessible* or *inaccessible*. Inaccessible metaobjects are those for which no operations (with the possible exception of instantiation) are defined. Accessible metaobjects are those for which a set of explicit operations are available to the programmer. Access to such objects may be introspective or reflective.

A malleable self-representation is important

One of the best known reflective facilities in any programming language is the ability of (most) languages in the Lisp family to treat users composed data structures as code. Lisp programmers are far less reluctant to devise solutions to problems that entail computing new code than are programmers in other languages. One explanation for this is that Lisp's *self-representation* is couched an an appropriate level, that is, approximately that of an abstract syntax tree. (Of course, given the syntax of Lisp, this is a relatively easy). Lisp S-expressions are malleable aggregates that can be composed, indexed, decomposed, and inspected.

In contrast, Smalltalk-80 programs can, in principle, compute code as well. However, to do so, they must manipulate either source strings, or byte codes. The former representation is at too high a level, and the latter at too low a level to be convenient. Smalltalk-80 employs parse trees, but they are not set up to play the role necessary to be useful as a reflective self-representation. Such a representation, with a two-way mapping between it and source code (Smith's O and O**-1), as well as between it and whatever machine level representations are used, seems desirable. An example of a language that explicitly represents abstract syntax level objects is KSL [Ibrahim 1988].

Metaobjects should mirror the metaarchitecture

A key requirement in the design of a set of metaobjects is that elements in this architecture mirror the structure of the language model (or programming model, or metaarchitecture) itself to as great a degree as is possible. The language model can be thought informally as the story that is told by the language's defining documents to programmers about how the structures and operations that comprise the language fit together (i.e. its semantics). When the metalevel architecture's structures parallel the language model, they provide a natural locus for information pertaining to the language's semantics, and hence its translation. This model can serve as the basis for self-analysis tools that would otherwise have to be constructed in a more cumbersome fashion as part of an external environment or collection of tools. Furthermore, such self-analysis capabilities might be mobilized by a system itself to evaluate and optimize its own structure.

Indeed, since the the metaobjects can be thought of as constituting the language's implementation, the degree to which the language model can be expressed in the metalevel architecture can serve as a test of the model's comprehensiveness. Questions of both completeness and granularity1 arise in this assessment. The decisions the language architect makes regarding what structures can be made explicit (reified) and modified (reflected upon) from the user level, and what features are implicitly *absorbed* [Smith 1983] by the language naturally dictate limits (or lack thereof) on the power of user-level extensions.   The story you tell, and the nature of the architecture that corroborates it, is of premier importance.]

A metaobject palette

A reflective object-oriented metalevel architect might elect to reify a variety of different notions. A decision to include or exclude an element of the list below may have surprising and far reaching consequences with regard to the architecture's ultimate scope. A discussion of these consequences is beyond the scope of this paper. Here, in no particular order, is a palette of possible metaobjects:

Variables/Slots
Selectors/Generic Functions/Keys/Operators
Messages
Evaluators/Processors/Interpreters
Method Dictionaries/Script/Scripts Sets/Role//Containers/Behaviors
Records/Repertoire/Relation/Table/Index
A-List/Association/Pair/Tuple/Entry
Handles/Pointers/Names/Addresses/Locations/References/OOPs
Environments
Continuations
Contexts (Method, Block)
Mailboxes/Message Queues
State Memory/Stores/Storage
Blocks/Closures

1[Smith 1983] referred to a related notion of "vantage point".

Classes
Types
Prototypes/Traits
Signatures/Protocols
Methods (Primitive and otherwise)
Code/Expressions/Parse Trees/Byte Codes/Forms
(Denotations?)
Processes/Threads/Engines

...and last but not least:
Objects
Integers/Floats/Reals/Strings
Arrays/Aggregates/Structures/Collections/Lists
(other primitive objects)

Some design principles

The following design principles seem to be of utility in designing a maximally flexible uniform reflective object-oriented metalevel architecture:

Message Passing at the Bottom
All computation shall be conducted via message passing. This is particularly important when designing the kernel objects. This design philosophy is reminiscent of the Self language [Ungar 1987].

Object-Oriented Uniformity
Everything shall be an object including variables, "code", contexts, oops? A successful architecture should be cleanly layered, with a simple object-oriented kernel at the bottom.

Malleable Self-Representation
Computations shall be represented by first class objects at a semantically interesting level

Extensionality
Behavior only shall determine an object's identity (The Duck Test)

Inheritance is signature composition
at least from an extensional standpoint

State and Behavior
Shall be externally indistinguishable (state is, of course, architecturally significant at certain levels) Hiding state behind accessors is an effective way of maintaining an appropriate level of external abstraction in many cases. However, state emerges as an appropriate local focus as an end in itself for applications such as constructing inspectors, debuggers, and as a locus for allowing objects to migrate to distributed systems or secondary storage.

Active Dispatching vs. Passive Lookup

Shall be externally indistinguishable This permits users to customize the behavior of objects by intercepting messages sent to them on a per object basis.

Dynamic vs. Static Scope
(Operational vs. Denotational Scope) Static scope shall be dynamically implemented (as it usually is, anyway)

Reflection subsumes inheritance

Reflection subsumes inheritance (and a variety of other sharing mechanisms). A reflective architecture can serve as the basis for the construction and evaluation of sharing mechanisms. Indeed, the distributed nature of an object-oriented metalevel architecture can allow several such mechanisms to coexist in the same system. New sharing mechanisms are among a host of different artifacts that can be constructed using a well designed reflective language.

An object, during the course of its lifetime, may enter into a number of different relationships with other objects in its environment. Some of these relationships will be permanent, or at least relatively static, and others might be quite ephemeral. Some of these relationships might be quite complex. In conventional object-oriented systems, one such relationship, inheritance, is well supported. However, the facilities for allowing the programmer to construct similarly powerful mechanisms of this sort of his or her own are limited. Appropriate metalevel architectures can permit the construction of sharing mechanisms that support , for instance, dynamic, domain-level object composition as well as static code sharing.

Reflective applications: The Class Menagerie

Reflective object-oriented metalevel architectures would be of little interest if they did not permit programmers to address problems they could not address in any other way. There is a extremely broad range of important problems that reflective techniques can address.

Reflection may prove particularly valuable in environments that must confront rapidly changing, highly volatile requirements, and in applications that make use of concurrency. Object-oriented techniques allow reflective embellishments to affect only localized parts of a system, or to remain in effect for only a short period of time (as might be required for debugging, for example).

Among the areas that can be addressed are: inheritance and sharing schemes, including the object composition and the construction of dynamic aggregates, maintenance, including debugging and tracing, constraint satisfaction and maintenance, the construction of intelligent agents, persistent object bases distributed systems, and concurrent systems.

Indeed, it is probably in the realm of concurrency, where metalevel issues such as processor assignment, load balancing scheduling, performance monitoring, and synchronization must be dealt with along side of object-level computation, that reflection may come into its own.

Reflection must be efficient to be practical

It is imperative that reflective extensions to a language be implemented efficiently if they are to win practical acceptance. For the sake of efficiency, reflective extensions to an object-oriented language should be built into the existing language, not built on top of it.

One strategy for achieving this end is to make sure that representations of existing low-level language structures are made concrete (reified) only when they are explicitly needed. Des Rivieres and Smith, and Maes have discussed how this strategy can be used to retain efficiency in reflective systems.

Another is to use a technique like *customization* [Chambers 1989] to dynamically provide efficient expansions of reflective code. Some of the implementation techniques used in the Self language to achieve efficiency seem applicable to reflective extensions as well. Our group at the University of Illinois [Johnson 1988b] has been investigating methods for efficiently compiling Smalltalk-80 code. There are strategies for using memory management units that can play a role in implementing reflective facilities as well.

Other promising strategies include inline caching and multiple representations [Deutsch 1984] [Foote 1989] discusses how inline caching might be used to implement alternative message dispatching schemes. The work by Deutsch and Shiffman emphasizes the extremely valuable notion of a *dynamic change of representation* to implement Smalltalk-80's highly reflective Context objects.

Ultimately, thought, the key to providing for the efficient implementation of reflective facilities may be to exploit the distributed nature of the language's self-representation. A well designed self-representation will provide appropriate loci for information that can facilitate the efficient implementation of user supplied features. The elements of this representation that are specialized to represent and implement user extensions can serve as repositories for information that can be used by the system to generate good code.

Conclusion: We must build atop a flexible foundation

I believe that we in the object-oriented programming community are discovering a principle that Pacific Rim architects already know. That is: To stay the seismic upheaval that confronts a system during the course of its lifetime, it must be built atop a flexible foundation. We can no longer think of our programming languages and systems as rigid structures that will withstand generations of buffeting in this face of changing requirements. We must instead construct systems that not only build on past experience, but are able to adapt and evolve as well. We must build our houses of brick rather than straw, but make sure that there is shock absorbing material in place in their foundations as well.

References

[Agha 1986]
Gul Agha
ACTORS: A Model of Concurrent Computation in Distributed Systems
MIT Press, 1986

[Bawden 1988]
Alan Bawden
Reification without Evaluation
Proc. Symposium on Lisp and Functional Programming, 1988
pages 342-248

[Bobrow 1988a]
D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon
Common Lisp Object System
Specification X3J13
Document 88-002R
SIGPLAN Notices, Volume 23,
Special Issue, September 1988

[Bobrow 1988b]
Daniel G. Bobrow and Gregor Kiczales
The Common Lisp Object System
Metaobject Kernel -- A Status Report
Proceedings of the 1988 Conference on Lisp and Functional Programming

[Briot 1989]
Jean-Pierre Briot
Actalk: A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment
LITP 89-33 RXF, Rank Xerox
ECOOP '89

[Chambers 1989]
Craig Chambers, David Ungar, Elgin Lee
An Efficient Implementation of SELF
a Dynamically-Typed Object-Oriented Language Based on Prototypes
OOPSLA '89 Proceedings
New Orleans, LA
October 1-6 1989, pages 49-70

[Des Rivieres 1984]
Jim des Rivieres and Brian Cantwell Smith
The Implementation of Procedurally Reflective Languages
Proc. of the 1984 ACM Symposium on Lisp and Functional Programming
August, 1984, pages 331-347

[Des Rivieres 1990]
Jim Des Rivieres and Gregor Kiczales
The Art of the Metaobject Protocol
(in preparation)

[Deutsch 1984]
L. Peter Deutsch and Allan M. Schiffman
Efficient Implementation of the Smalltalk-80 System
Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages,
1983, pages 297-302

[Ferber 1989]
Jacques Ferber
Computational Reflection in Class-Based Object-Oriented Languages
OOPSLA '89 Proceedings
New Orleans, LA
October 1-6 1989, pages 317-326

[Foote 1988]
Brian Foote
Designing to Facilitate Change with Object-Oriented Frameworks
Masters Thesis, 1988
University of Illinois at Urbana-Champaign

[Foote 1989]
Brian Foote and Ralph E. Johnson
Reflective Facilities in Smalltalk-80
OOPSLA '89 Proceedings
New Orleans, LA
October 1-6 1989, pages 327-335

[Foote 199?]
Brian Foote
A Framework for Object-Oriented Reflective Metalevel Architectures
Ph. D. Thesis (in progress)
University of Illinois at Urbana-Champaign

[Friedman 1984]
D. P. Friedman and M. Want
Reflection without Metaphysics
Proc. Symposium on Lisp and Functional Programming, pages 348-355, August 1984

[Goldberg 1983]
Adele Goldberg and David Robson
Smalltalk-80: The Language and its Implementation
Addison-Wesley, Reading, MA, 1983

[Ibrahim 1988]
Mamdouh H. Ibrahim and Fred A. Cummins
KSL: A Reflective Object-Oriented Programming Language
Proceedings of the International Conference on Computer Languages
Miami, FL, October 9-13 1988

[Johnson 1988a]
Ralph E. Johnson and Brian Foote
Designing Reusable Classes
Journal of Object-Oriented Programming
Volume 1, Number 2, June/July 1988
pages 22-35

[Johnson 1988b]
Ralph E. Johnson, Justin O. Graver, and

Laurance W. Zurawski
TS: An Optimizing Compiler for Smalltalk
OOPSLA '88 Proceedings
San Diego, CA, September 25-30, 1988
pages 18-26

[Kiczales 1990]
Gregor Kiczales et. al.
CLOS Metaprotocol Specification
(in preparation)

[LaLonde 1988]
Wilf R. LaLonde and Mark Van Gulik
Building a Backtracking Facility in
Smalltalk Without Kernel Support
OOPSLA '88 Proceedings
San Diego, CA, September 25-30, 1988
pages 105-122

[Maes 1987a]
Pattie Maes
Computational Reflection
Artificial Intelligence Laboratory
Vrije Universiteit Brussel
Technical Report 87-2

[Maes 1987b]
Pattie Maes
Concepts and Experiments in
Computational Reflection
OOPSLA '87 Proceedings
Orlando, FL, October 4-8 1977 pages 147-155

[Smith 1982]
Brian Cantwell Smith
Reflection and Semantics in a
Procedural Programming Language
Ph. D. Thesis, MIT
MIT/LCS/TR-272

[Smith 1983]
Brian Cantwell Smith
Reflection and Semantics in Lisp
Proceedings of the 1984 ACM
Principles of Programming Languages
Conference
pages 23-35

[Ungar 1987]
David Ungar and Randall B. Smith
Self: The Power of Simplicity
OOPSLA '87 Proceedings
Orlando, FL, October 4-8 1977 pages 227-242

[Watanabe 1988]
Takuo Watanabe and Akinori Yonezawa
Reflection in an Object-Oriented Concurrent
Language
OOPSLA '88 Proceedings
San Diego, CA, September 25-30, 1988
pages 306-315