Flexible Foundations and Movable Walls

A Position Paper for the ECOOP/OOPSLA '91 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

13 August 1991

The Three Pigs Revisited

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 and technology. 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.

The foundations upon which contemporary programming environments stand are the operating systems, object-bases, networking facilities, and programming tools that support application development and execution. When these system components are themselves constructed of first-class objects, the full power of our object-oriented tools and techniques can be brought to bear on these components. In a system with such an architecture, immutable, monolithic components can be replaced with malleable object-oriented abstract classes, components, and frameworks. Such components can adapt and evolve as requirements change.

One particularly intriguing consequence of object-oriented framework evolution is that it is often possible to discern new structural relationships among framework components as a framework matures. Reflective languages and systems might allow us to reassess where we place the walls that separate traditional system components such as compilers, programming environments, persistent object-bases, and operating systems. Such a refactored system might allow an object created by one process to move about the system autonomously. It is interesting to ask what sort of metalevel architecture might be required to support such an approach.

When Last We Met

This paper (as some readers may already have noticed) begins where my position statement for last year's Reflection Workshop [Foote 1990][1] left off. Among the points made in that paper were:

• Object-oriented programming systems ought to themselves be build out of first-class, highly dynamic objects that are accessible, via some mechanism, at runtime. Such an approach can, under certain circumstances, permit the system to undergo dynamic *metamorphosis*.

• All significant elements of a languages' programming model ought to be themselves reflected in first-class elements of that language's metalevel architecture.

• An object-oriented reflective metalevel architecture might draw from a rich palette of potential metaobjects, including variables, selectors, messages, interpreters, script sets, handles, environments, continuations, contexts, message queues, stores, and closures, classes, types, prototypes, signatures, methods, code, threads, and instances...

• Reflective metalevel architectures should be designed with message passing at the bottom, malleable self-representations, extensionality, abstract inheritance, first -class representation, abstract dispatching, and abstract scope.

• Reflective metalevel architectures can address a veritable "class menagerie" of different issues.

• Reflection need not be inefficient. The techniques described in [Deutsch 1984], [Johnson 1988b] and [Chambers 1989] ought to work as well with reflective embellishments to an existing system as well as with normal application code.

Frameworks Facilitate Change

Object-oriented languages make it much easier to design software components and systems that can adapt as requirements evolve and change [Foote 1988]. The judicious use of object-oriented techniques can promote the emergence of reusable abstract classes, components, and object-oriented frameworks [Johnson 1988].

An object-oriented framework is composed of a set of abstract classes and components that together comprise an abstract or generic solution to a range of potential application requirements. The framework's abstract classes and components must be customized by the framework's clients to suit the requirements of specific applications. Frameworks, unlike abstract classes, are often characterized by an inversion of control. That is, rather than calling framework elements directly, the client must sometimes instead supply the framework with components that are "called-back" by the framework at an appropriate time.

We believe that durable, reusable object-oriented artifacts emerge most readily from an iterative reapplication of existing abstract classes, components, and frameworks to successive, related requirements. This lifecycle perspective holds that object-oriented entities evolve at every level along a trajectory that takes them through an initial prototype phase, and expansionary exploratory phase, and a consolidation phase, in which structural insight gained during successive reapplications is exploited to increase the generality, structural integrity, and reusability of these entities. As entities evolve, many will progress

[1] A PostScript copy of this paper can be had by those with Internet access using anonymous ftp on p300.cpl.uiuc.edu (128.174.72.1).

from loosely structured, application specific, inheritance-based "white-box" entities to fully encapsulated, fairly general, component-based "black box" entities. This lifecycle perspective is addressed in more detail in [Foote 1990].

Programming with Programmable Objects

The marriage of object-oriented tools and techniques with the notions of reflection and metalevel architecture permits the full power of these approaches to be brought to bear upon object-oriented languages and systems themselves.

In a programming system with a fully reflective object-oriented metalevel architecture, those elements of the system that define and implement the language's programming model are themselves cast as first-class dynamic objects. A language with such an architecture is easy to extend, since existing elements of the system can be specialized and reintroduced into the system. Moreover, the components of such a system can themselves serve as the basis for an object-oriented framework that can support an evolving family of different programming approaches,and paradigms.

Ralph Johnson and I are currently engaged in an exploration of the properties of object-oriented reflective metalevel architectures [Foote 1992]. The method by which we are pursuing this investigation is via the construction of an object-oriented framework.

We are conducting our investigation of object-oriented reflective metalevel architectures by treating these lifecycle observations as a sort of design methodology. That is, we are building a framework through a pro-active process of successive application of a sequence of related but diverging requirements, so as to gain insight into what general components and design principles underlie them. What follows is a brief snapshot of our work to date.

Id: A simple, primitive prototype

Our initial prototype was an interpreter for an object-oriented language called Id, which was, in turn, modeled primarily after a subset of the Self programming language [Ungar 1987][2]. Self was chosen because its particularly simple, elegant design seemed relatively easy to implement, and well suited to metalevel embellishment. Id was written in CLOS [Bobrow 1988][3]. Like Self, Id is a pure object-oriented language, with first-class blocks, methods, integers, symbols, booleans, selectors, arrays, and contexts. Id represents code as scripts, which (in Id) are parse-tree level objects, rather than byte-codes.

Ego: A Dawning of Rudimentary Self-awareness

The second language we implemented, Ego, retained much of the structure of the Id kernel, while shifting much of its implementation from the underlying interpreter into Ego itself. The Ego interpreter is written in old-fashioned (CLtL 1, pre-CLOS) Common Lisp. The system's (considerable) appetite for polymorphism and inheritance is addressed at the Ego level rather than by the kernel. This can be seen as an effort to force ourselves to live in the house we were constructing. Ego represents all user and interpreter level data structures as first class instances, and allows user level code to inspect and modify these data structures in an unrestricted fashion. Hence, Ego provides rudimentary reflective facilities at approximately the level of Smalltalk-80 [LaLonde 1988].

---

[2]Some features inspired by CLOS and ABCL/R are present as well.

[3]May Day PCL, to be precise.

Ego provides no facilities for reflecting into the interpreter-level code itself. The lack of such facilities in Smalltalk-80 prompted our interest in reflection in the first place [Foote 1989]. As a consequence, we placed a priority on providing first-class access to mechanisms like message dispatching, context management, control management, and method look up in our initial efforts to build our framework. Those portions of existing systems where mature object-oriented models are already present (class structure for instance) have been given secondary priority (for the moment). Ego uses Self-styled prototypes as repositories for shared behavior.

SuperEgo: A Reflective Self

The third language in this series, SuperEgo[4], or Super, allows full, dynamic, reflective access to those elements of the system that define the behavior of the language itself.

The definition of an active SuperEgo interpreter, and, hence, the semantics of the Ego language, are defined by objects called Ensembles[5]. An Ensemble is comprised of a collection of Performers, each of which specializes in performing a particular kind of Script. Hence, localized changes to how the system performs certain scripts and other activities can be made by specializing the system so that a specific Ensemble is recruited to perform certain actions.

Note that KSL [Ibrahim 1988] distributed the definitions of code level components across a set of objects that corresponded to the language's grammatical structures. ABCL/R [Watanabe 1988] matched an evaluator to each object in the system. The question of how and when to invoke a customized Ensemble raises interesting issues, which we are currently in the process of investigating.

In SuperEgo, concerns such as code representation, variable namespace, method namespace, code semantics, instance representation, process/processor/context/state, control/continuation, and method look up are all dealt with by distinct objects, which are themselves subject to localized, dynamic manipulation.

The Dimensions of Reflectivity Revisited[6]

In a system in which the architecture is distributed among a constellation of first-class objects, it is more useful to think of a number of spires protruding in multiple directions, as opposed to a single reflective tower. [Ferber 1988] identified three such dimensions (structure, computation, and communication). It seems possible to discern at least one more major dimension: representation. This dimension addresses the potential first-classness of instance objects themselves, and the storage management concerns (such as garbage collection) that accompany it.

Of course, in a system a distributed object-oriented architecture, certain individual metalevel components may be reflectively modified, while others are not. Each of these might itself be thought of as a reflective dimension. (This is the "reflective porcupine" I referred to last year.) This notion of dimension is consistent with the conventional mathematical meaning of the term in many respects. Taken in this sense, a reflective dimension can be thought of as a part of the system that is orthogonal to other dimensions. Orthogonality, in turn would

---

[4]A concurrent languge named "Schizo" has also been contemplated.

[5]The notion of using theatrical metaphors to describe reflective systems was impressed upon me most recently by [Kiczales 1991].

[6]This section's title is borrow from Mamdouh Ibrahim's 1990 Reflection Workshop position paper.

imply that a substitution of a behaviorally identical component along one dimension should have no effect along another.

Some metaarchitectures constructed from the same metalevel framework may exhibit dependencies (coupling) along some dimensions that are independent in other component sets derived from the same framework. A requirement that certain components of a framework co-vary (and hence must be customized together) is characteristic of many conventional frameworks. Mathematical modelers usual strive to represent their modeling domains using as small a number of independent variables as possible. Metaarchitects, on the other hand, should strive to identify as many independent dimensions as they can, since each such dimension can then serve as an orthogonal locus for customization.

Movable Walls and Renegade Objects

A characteristic of the lifecycle of object-oriented entities is the emergence of structural insight into an application domain as the result of successive reapplication. Constructing the elements of programming systems out of dynamic first class objects can lead to a reassessment of where the walls between objects, environments, operating systems, databases, and command languages should be placed.

Traditional programming environments, whether they are based on the compile/link/load model, or the image model seen with languages such as Lisp or Smalltalk, tend to trap objects on reservations circumscribed by runtime support requirements They may influence or exploit objects outside their addressing spaces only indirectly, via I/O. I believe that advances in hardware and software technology make it appropriate to reassess the relationship among object-oriented languages themselves, their runtime environments, and the rest of the world. For instance, shared memory, distributed, and networked systems, concurrent systems, and object-oriented databases all present novel runtime challenges. It is already clear that reflective techniques are having a significant impact on the way in which we think around concurrent systems. Reflective techniques have also been employed to add persistence to objects in existing image-based systems [Paepcke 1990]. The growing popularity of object-oriented databases also attests to the observation that first-class objects have a place outside individual applications.

Voltaire is said to have said that if God did not exist, it would be necessary to invent him. One can make a case that the architects of systems such as X and the Macintosh Operating Systems have discovered that the same is true of object-oriented programming in general, and of dynamic, autonomous objects in particular. In those cases where linguistic support for object-orientation has not been present, object-like entities have been constructed in a home-brew fashion, out of elements like C structs and pointers to functions. Indeed, it is difficult to think of a significant GUI package that does not employ object-oriented facilities. What is striking in packages like the X Toolkit is not just that object-orientation was found to be indispensable, but that some of the object-oriented facilities that were included go beyond not only C++, but Smalltalk and Lisp as well in terms of metalevel sophistication.

For instance, the resource managers seen in both X and the Macintosh Toolbox can be thought of as forerunners of genuine rudimentary object-oriented databases. X Resource IDs are reference or handle-like objects with a scope that may span several active address spaces. Atoms are IDs that play a role similar to that of Lisp Symbols. The resource manager itself contains sophisticated facilities for interpreting and converting strings to resources. Since the C (and C++) namespaces are not accessible at runtime, the resource manager also is recruited to aid in the cumbersome task of mapping runtime name strings on to compile time program level names. Many X widgets are themselves framework-like

structures, with elaborate callback hooks that perform the sorts of functions one might employ CLOS :before or :after methods for. This all suggests that the next generation of object-oriented programming language will have to cope with a need for features like as expression evaluation, smart handles, and persistent, distributed, and shared objects.

Can reflective object-oriented metalevel architecture help to address these needs? And if so, how? Can the relationships among system components like compilers, loaders, interprocess communication, and storage management be refactored in such a way so as to allow objects to flourish outside the processes in which they were spawned? What sort of facilities might be necessary to support such autonomous objects, and where must these facilities be located in order to allow them to work? And, what sorts of metalevel linguistic manipulation might we undertake to simplify all of this?

It is interesting to speculate as to what some of the answers might be. For instance, if the sort of dynamic translation seen in Smalltalk and Self were available as a sort of operating system level service, along with storage and namespace management services, it is possible to imagine how individual objects might flourish beyond the umbilicals that tie them to their originating processes. A key contribution that reflective languages might make to this vision is that the vastly more complex semantics for handles and namespace management need only be mobilized for objects outside their wombs. Object-oriented techniques themselves are essential if one is to even contemplate such a system organization.

Conclusion

Object-oriented frameworks are ideal tools both for constructing and investigating reflective object-oriented metalevel architectures. Such approaches may permit us to construct languages and systems that can evolve as requirements change.

References

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

[Bobrow 1988]
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

[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

[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
Dept. of Computer Science
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 1990]
Brian Foote
A Fractal Model of the Lifecyle of Reusable Object-Oriented Entities
(in preparation)

[Foote 1991]
Brian Foote
Reflective Object-Oriented Metalevel Architectures: Pyrite or Panacea?
Position Paper for the OOPSLA/ECOOP '90 Workshop on Reflection and Metalevel Architectures

[Foote 1992]
Brian Foote
A Framework for Object-Oriented Reflective Metalevel Architectures
Ph. D. Thesis (still in progress)
Dept. of Computer Science
University of Illinois at Urbana-Champaign

[Friedman 1984]
D. P. Friedman and M. Wand
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

[Hoeltze 1990]
Urs Hoeltze, Bay-Wei Chang, Craig Chambers, and David Ungar
The Self Papers, and the Self Manual
(unpublished)

[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 1991]
Gregor Kiczales, Jim Des Rivieres, and Daniel G. Bobrow
The Art of the Metaobject Protocol
MIT Press, 1991

[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

[Paepcke 1990]
Andreas Paepcke
PCLOS: Stress Testing CLOS
OOPSLA/ECOOP '90 Proceedings
Ottawa, Ontario, Canada
October 21-25, 1990
pages 194-221

[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