Objects, Reflection, and Open Languages

Brian Foote

Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield
Urbana, IL 61801 USA
(217) 333-3411
foote@cs.uiuc.edu

Objects + Reflection = Open Languages

The marriage of reflection with object-oriented programming and design techniques holds out the promise of dramatically changing the way that we think about, organize, implement, and use programming languages and systems. The combination of object-oriented programming languages and reflective metalevel architectures allows the full power of the object-oriented approach to be brought to bear upon object-oriented languages themselves Together, they have the potential to permit the long deferred promise of truly open programming languages and system to be realized.

Object-Oriented Object-Oriented Languages

A programming language with an *object-oriented metalevel architecture* is one in which programs are themselves constructed out of first-class objects. Metalevel objects, or *metaobjects* are objects that define, implement, support, or otherwise participate in the execution of application, or *base level* programs.

A *reflective* object-oriented language allows a running program to look at or change the objects out of which it is built. Together, these metaobjects constitute the system's *self-representation*. These objects *reify* otherwise implicit aspects of the underlying system. The ability to inspect, but not alter, the objects that implement a system is referred to as *introspection*.

A reflective object-oriented program can access the very objects that in turn define how it works, and can alter them dynamically if necessary. Changes made to these objects are immediately reflected in the actual dynamic state of the state of the system, and vice versa. The requirement that this *dynamic consistency* be maintained is sometimes refereed to as the *causal connection* requirement.

Programming languages built out of objects are easy to extend. Features may be added to a suitably designed reflective object-oriented language by adding a set of metaobjects to support them to the language. These objects may, of course, be specializations of existing objects. A well designed reflective metalevel architecture can limit the scope of extensions to a single object, class, or computation, or allow them to be in effect system wide. Features that have usually required the definition and implementation of whole new languages, such as backtracking [LaLonde & Van Gulik 1987], futures [Foote & Johnson 1989] and persistence [Paepke 1990] have been added to existing languages using reflective facilities.

Reflection can also allow programmers to create new metalevel objects and selectively substitute them for existing part of the running system. This ability allows programmers to add objects to trace a program's execution, for example, from within the language. Support for debugging in most languages is usually treated in an ad-hoc fashion by individual implementations. An object-oriented reflective metalevel architecture allows support for debugging to be provided at the language level.

A language that supports the dynamic redefinition of existing parts of the system is said to be mutable. A language that supports the addition of new features, but not the redefinition of old ones is said to be extensible [Stroustrup 1991]. The ability to exploit the existing definition of a system to augment its behavior gives the programmer considerable leverage over the rest of the system.

It is easy to dynamically introduce new objects and object types into a running reflective program, since these objects, as well as the objects that define them, are themselves first-class, dynamic objects. By contrast, consider the difficulty associated with dynamically handling a new type of object in a C++ program that has already been compiled.

Because programming systems with object-oriented reflective metalevel architectures have a model of themselves embedded within them, they exhibit a substantial capacity for metamorphosis. Reflection has the potential to extend the runtime reach of a programming system inward into the definition of the language itself, downward into its implementation, and outward into realms currently considered the provinces of operating systems, programming environments and database systems. Reflection has the potential to bring areas as disparate as programming language design, compiler construction, code generation, debugging, tracing, concurrent programming, and semantics together under a single umbrella.

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. As a system evolves, internal structure emerges. Objects are ideal for capturing this structure. As walls crumble, these constituent objects can serve as a structural redoubt one level below . Without objects, one would have to fall all the way back into the implementation to reorganize a system. A comprehensive reexamination of how a system supports running objects could permit autonomous objects to break free of the processes that spawned them and migrate unencumbered among other processes, processors, and persistent object bases.

A good sign that a programming language feature is needed is when a lot of people go to a great deal of effort to build it themselves atop existing languages. There is abundant evidence that dynamic metalevel objects are such a feature. Open systems need open languages.

Frameworks Facilitate Change

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

of reusable abstract classes, components, and object-oriented frameworks [Deutsch 1983] [Johnson & Foote 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.

Durable, reusable object-oriented artifacts emerge most readily from an iterative reapplication of existing abstract classes, components, and frameworks to successive, related requirements. 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. During the consolidation phase, 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 from loosely structured, application specific, inheritance-based "white-box" entities to fully encapsulated, fairly general, component-based "black box" entities.

A Framework for Reflective Metalevel Architectures

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 A particularly intriguing consequence of this approach is that 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. I am currently engaged, with Ralph Johnson, in the construction of such a framework.

This framework, a framework for reflective metalevel architectures, can be thought of as a set of generic building blocks for object-oriented languages. It will allow individual "languages" to be constructed by specializing its abstract classes and components. Because of the lifecycle characteristics of object-oriented frameworks, the construction of this structure is both the means by which this investigation is proceeding, and the end toward which it is striving. This framework was "seeded" using a Super-Ego, a simple prototype object-oriented language based on Self [Ungar & Smith 1987].

Reusable object-oriented frameworks cannot be constructed in a top-down fashion. They are the result of an iterative process that unfolds at all levels of a system as objects are successively reapplied to address changing requirements. I am attempting to exploit this characteristic pattern of framework evolution by successively addressing design

requirements drawn from several increasingly different object-oriented programming languages: Self, Smalltalk, CLOS, and ABCL/R. As this work progresses, the elements of the framework will, I hope, come to embody a collection of common structural insights into this application domain. The objects in the framework that emerges from this process will, I hope, cast a penumbra that will cover a broad range of different programming problems.

The design of the framework is being guided by the following principles and observations:

• 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.

• All significant elements of a languages' programming model ought to be themselves reflected in first-class elements of that language's metalevel architecture. For instance, if the programming model makes extensive use of a notion like "class", then Class objects should be explicit, first-class elements of the metaarchitecture that coexist with ordinary application objects at runtime.

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

What follows is a snapshot of our work to date:

Id

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]. 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]. 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 polymorphic parse-tree level objects, rather than simple byte-codes.

Ego

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 [Steele 1984]. The system's (considerable) appetite for polymorphism and inheritance was 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 represented all user and interpreter level data structures as first class instances, and allowed user level code to inspect and modify these data structures in an unrestricted fashion. Hence, Ego provided rudimentary reflective facilities at approximately the level of Smalltalk-80.

Ego provided 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) were given secondary priority. Ego used Self-styled prototypes as repositories for shared behavior.

SuperEgo

The third language in this series, SuperEgo[1], allows full, dynamic, reflective access to those elements of the system that define the behavior of the language itself. In SuperEgo, the following concerns are (or will be) all dealt with by distinct objects, which are themselves subject to localized, dynamic manipulation.

Structural
Store
Instance
State
Templates
Sharing
Scripts

Computational
Semantics
Dispatching
Communication
Variables
Primitives
Closures
Context
Control
Namespace
Animus
Scheduling

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 (one can think of a sort of "reflective porcupine"). [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. The following sections briefly examine some of these architectural concerns.

Store

The Super-Ego prototype provides a first-class, albeit primitive, object memory similar to Smalltalk-80s. We plan to explore the consequences of reifying more of this layer's functionality to provide additional first-class access to the structural field. Nothing prohibits the use of multiple object memories. Variant stores might be created to facilitate interaction with distributed object bases or servers, or to allow different storage management schemes to be introduced. Access to such stores might be regulated on a per-

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

object basis. Access to these sorts of facilities might also be handled at the level of the state management protocol.

State

The architecture of Super-Ego distinguishes between objects themselves, instances, which are metaobjects that support objects, and handles or references. Objects, given this view, appear to have distinct identities only as a result of the cooperation of all the behind-the-scenes metaobjects that together determine how they behave. Just as the "real" Prince of Denmark can never appear on any stage, objects themselves appear to have a distinct identity only as the result of other objects playing the necessary roles to bring their behavior about.

Like Self, Super-Ego represents both state and behavior using dictionary-like instance objects. Super-Ego allows base level programs to read slots by sending messages to themselves using a slot name as a selector. They may write their variables using an explicit assignment operator.

Objects and protocols that allow the language's representational mechanics to be specialized have a wide-range of potential uses. For instance, the representational protocol can be exploited to add persistent objects to a language. By changing these protocols at the metalevel, existing code can operate correctly with such objects. Distributed or remote objects can be implemented by altering these mechanisms as well.

Smart handles or references have the potential to be of utility for constructing storage management schemes. For instance, reference counting might be implemented using code housed in reference objects themselves. Smart handles might also play a role in persistent object management, and in constructing capability-style protection mechanisms.

A smart handle might also play the role of a Thinglab Path [Borning]object in ensuring that access to a component is implicitly gained only through a container. First-class variables may also be useful loci for dependency and constraint mechanisms.

It would be quite useful were it possible for different users of the same object to look at it using subsets of the object's overall protocol. One user might look at one part of an object, another at another part. However, any update made through any such "view" will naturally be reflected immediately in all others, since they all look at the same underlying object.

There are several approaches that might be taken to implement views. One might be to use multiple objects, tied together using constraints or some other dependency mechanism. Another is a component-based solution. Such an approach would require an outer dispatching class that distributed the messages it received in some fashion to its components. Views are another mechanism through which different levels of access and protection might be provided to different clients of an object.

Specialized metalevel state code can also be used to cache or memoize computational or other results on a per class or per object basis.

Scripts

One of the best known reflective facilities in any programming language is the ability of (most) languages in the Lisp family to treat user 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

program representation is couched an an appropriate level, that is, approximately that of an abstract syntax tree. 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.

Compilation in Super-Ego is a three-stage process. During the first stage, concrete syntax is compiled into program objects, or *scripts*. When a script is encountered, it is incorporated into a closure-like object that retains its static scope. The third stage executes scripts, using interpretation, or dynamic compilation into executable native code objects. This dynamic compilation strategy is taken from Smalltalk-80 [Deutsch 1984] and Self [Chambers 1989].

Sharing

Super-Ego employs Self-style prototype and trait objects, instead of Smalltalk-style Classes, as repositories for shared behavior and state. Smalltalk classes are easy to build atop this abstraction. This organization is conceptually simple, and easy to bootstrap. We expect, however, to be able to construct architectures that more closely model Smalltalk Classes, and CLOS classes and specializers, as well as some unique mechanisms, using variants on the emerging Repertoire protocol.

There are occasions where it is desirable to selectively add behavior to the repertoire of an object on a per instance basis, with little overhead. The ability to add simple, anonymous subclasses to individual instances at runtime would solve this problem. Behavior can also be dynamically changed through additions to a class's method dictionary.

There are also times when an object might need to dynamically change its class. It might add a lightweight class that specializes or extends or augments its default behavior, or it might need to switch to a completely new repertoire of behaviors.

Traditional inheritance is an extremely useful static, per-class mechanisms. There are problems that require dynamic, per-instance solutions. Tracing access to an individual object for a period of time, or constraining it to notify a window of changes to its state are two applications that require dynamic, per-instance mechanisms.

Templates

Object-oriented languages differ considerably in the models they use for creating objects. Self creates object by cloning existing objects. Any object may serve as a prototype for a new object. The object hierarchy is usually arranged so that the parent of the actual template or prototype object is contains the code and data that all siblings of the prototype should share. These objects are called trait objects, and have much in common with traditional classes.

Smalltalk and CLOS use explicit class objects both to create new instances and to house their shared state and behavior. Classes in these languages have become the repositories for a variety of related responsibilities. [Borning 1986] enumerates a number of these.

Semantics

The definition of an active SuperEgo interpreter, and, hence, the semantics of the Ego language, are defined by objects called Ensembles[1]. 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.

Both 3-KRS [Maes 1987a] and 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 still in the process of investigating.

Dispatching

Traditional function and procedure calls can be thought of as consisting of two distinct computations: an overt application level computation, and a behind-the-scenes metalevel computation that sets the stage for the execution of the application level code. In languages like Pascal or C, the metalevel computation is concerned primarily with constructing an appropriate context for the execution of the target routine. Once this relatively simple task has been accomplished, the body of the called routine is executed. Once this computation has completed, the routine's result must be moved to the calling context, and this context must be made the current one once again.

Object-oriented languages like C++, Smalltalk-80, and CLOS raise the level of abstraction of message sends primarily by increasing the complexity of metalevel computation associated with the send. As one moves from sequential to concurrent systems, the metalevel becomes even richer and more complex, as issues like scheduling, synchronization, and load balancing come into play. One can make a case that reflection is coming into its own precisely because the metalevel itself is becoming a more interesting place.

To provide maximal flexibility and power, it is necessary to identify all the objects that might usefully benefit from overriding parts of the dispatching process, so that one can design a mechanism that permits them to do so.

When a message is to be sent, some object, a *dispatcher*, must be located and put in charge of making the computation happen. Differences among dispatching models will be determined by the way the dispatcher works, and by the way in which it is determined. Some of the ways in which method look up itself is done have already been discussed. The question of how the dispatcher is determined is many respects even more fundamental than how it works once it is identified. Instance, Class, Context, and Operator objects may all participate in the process of nominating a dispatcher The participants in the nomination process may nominate themselves, or another object as the current dispatcher.

---

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

The question of how the dispatcher is determined is important because it defines what object is given control when a message is sent. For instance, if objects nominate their dispatchers, then constructing an operator (generic function) based dispatching model requires that each object employ a dispatcher that hands off control to a generic function style dispatcher.

Namespace

First-class environments can be used manipulate what set of temporary or global values is accessible at a given time. This capability can be useful for loading new versions of a system into an existing environment. Namespace manipulation can be applied to conventional objects or environments to allow iteration over all named fields by debuggers or inspectors.

Control

Metalevel code in Super-Ego has a view of all the objects that define the control history of all active processes. These objects can be manipulated to construct control mechanisms such as catch and throw, and exception mechanisms. [LaLonde & Van Gulik 1987] describes a backtracking mechanism constructed using the reflective capabilities of Smalltalk-80 Contexts.

Animus

Like Smalltalk-80, Super-Ego represents the object through which the active thread emerges into the system using a first-class process object. Such objects are useful for exploiting medium-grained multitasking, and are essential for scheduling.

Movable Walls

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 about concurrent systems. Reflective techniques have also been employed to add persistence to objects in existing image-based systems. 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.

There is tremendous potential leverage available at the language level to address many of these issues. It is clear however, that if language designers do not seize the opportunity to provide solutions to these problems, the broader software community is prepared to go ahead without them. The result will be a further proliferation of facilities built on top of second-class substrates, instead of into first-class ones.

The Hamiltonians vs. the Jeffersonians

During the early development of American representative democracy, two schools of thought emerged. The Jeffersonians felt that the people could be trusted with decisions such as the election of the president, while the Hamiltonians believed that an educated elite, and not the masses, should make such choices.

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. They who are

prepared to embrace linguistic licentiousness must be prepared to promote responsibility as well.

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 red herring. Existing programming languages already give clumsy programmers more than ample opportunities to shoot themselves in the foot[1]. 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 use 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 to designing and building, as opposed to using, reflective facilities. Metaprogramming should not be undertaken frivolously. However, a language should not be an obstacle to its users, or to its own evolution.

Open Languages

One of the explanations for the renewed popularity of reflection, particularly in the object-oriented community, is that we[2] have become heirs to a long tradition of trying to make programming languages as <u>open</u> as possible. This section examines some of this tradition.

Nowhere has the open languages tradition flourished more than in the Lisp community. Indeed, Self, Smalltalk, 3-Lisp, 3-KRS, CLOS, ObjVLisp, the Actor languages, and ABCL/R can all trace their ancestry either directly or indirectly to Lisp. A metacircular definition was given for Lisp in [McCarthy et. al. 1962]. This definition gave a blueprint for a program that could take other Lisp programs in the form of Lisp data structures and execute them. This program, in turn, allowed Lisp data structures (s-expressions) to be passed to a function (eval) and executed as code. In [McCarthy 1978] McCarthy recalls the following about the preparation of [McCarthy 1960]:

"...Another way to show that LISP was neater than Turning machines was to write a universal LISP function and show that it was briefer and more comprehensible thatn the description of a universal Turing machine. This was the LISP function ***eval[e,a],*** *which computes the value of a Lisp expression* ***e*** *-- the second argument a being a list of assignments to variables. (****a*** *is required to make the recursion work). Writing* ***eval*** *required inventing a notation representing LISP functions as LISP data, and such a notation*

---

[1]Indeed, the current popularity of C++, which has extensive facilities for overloading and extending familiar C-style code, has done quite a bit to counter some of this "reflection is too dangerous" sentiment.

[2]That is, "we in the reflection community'

was devised for the purposes of the paper with no thought that it would be used to express LISP programs in practice..."

It is clear that McCarthy recognized that a programming language could be at least as effective as alternative formal approaches for defining how a programming language works. However, McCarthy and his group did not at first foresee the practical potential that their approach had for actual metaprogramming.

In Smalltalk-72 [Goldberg & Kay 1976] [Ingalls 1983] code was viewed by the interpreter as simply a stream of tokens. The object returned by any computation could hijack the local message stream, and gobble down as many tokens as it saw fit, Pac-Man style. Smalltalk-74 introduced a programmer accessible object that represented the incoming message stream. Thus, not only could all the message stream operations be examined in Smalltalk, but the user could also define his (sic) own extensions to the message stream semantics.

With Smalltalk-76, the designers of Smalltalk began to retreat from the most extravagant of such dynamic excesses in the name of efficiency. Still, the imperative that elements of the system itself be aggressively promoted to first-class status was retained. Ingalls referred to the following as the Smalltalk Philosophy:

"choose a small number of general principles and apply them uniformly".[1]

On the topic of kernel size and openness, Ingalls stated:

"We have always sought to reduce the size of the Smalltalk kernel. This is not only an aesthetic desideratum; kernel code is inaccessible to the normal user, and we have always tried to minimize the parts of of our system that can not be examined and altered by the curious user."

This requirement that the user be able to examine or alter as much of the system as possible meshes nicely with our our contemporary working definitions for reflection. Indeed, the extensive set of metaobjects in Smalltalk-80 [Goldberg & Robson 1983] is one of the best existing examples of the power of such objects. [Deutsch and Shiffman 1984] describes a strategy for efficiently implementing what we now call Smalltalk's highly reflective Context objects. It is important to note that way before we had a name for it, the Smalltalk community was doing the seminal work on what we are now calling object-oriented reflective metalevel architectures.

The goal of aggressively promoting elements of the language to first-class status, as well a recognition of the perils of regress can be seen in the Actor community in this passage from [Lieberman 1986]:

"Many object-oriented systems supply linguistic mechanisms for creating objects with methods, variables and extensions. An alternative approach, which is advocated in the actor formalism, is to define methods, variables, and extensions as objects in their own right, with their behavior determined by a message passing protocol among them.

---

[1]This, in turn, resembles the notion of orthogonality found in the Revised Report on Algol 68 [van Wijngaarden et. al. 1976]: "The number of independent primitive concepts has been minimized in order that the language be easy to describe, to learn, and to implement. On the other hand, these concepts have been applied*"orthogonally"* in order to maximize the expressive power of the language while trying to aviod deleterious superfluities.

Obviously, and object representing a method cannot itself have a methods, otherwise infinite recursion would result."

This passage from the same paper also recognizes the merits of a small, highly extensible object-oriented kernel:

"The mechanisms for sharing knowledge in object-oriented languages have now grown so complicated that it is impossible to reach universal consensus on the best mechanism. Using object-oriented programming itself to implement the basic building blocks of state and behavior is the best approach for allowing experimentation and coexistence among competing formalisms."

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. Object-oriented languages with reflective metalevel architectures can provide this sort of foundation.

The cobbler's children, it has been said, are often the last to be shod. Just as objects are good for building programs, so too they are good for building programming languages and programming systems.

REFERENCES

[Abelson & Sussman 1985
Harold Abelson and Gerald Jay Sussman
with Julie Susmann
Structure and Interpretation of Computer Programs
MIT Press, Cambridge, Mass.
McGraw-Hill, New York, 1985

[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

[Bennett 1987]
John K. Bennett
The Design and Implementation of Distributed Smalltalk
OOPSLA '87 Proceedings
Orlando, FL, October 4-8 1977 pages 118-330

[Bobrow et. al. 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

[Bobrow & Kiczales 1988]
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

[Borning 1979]
Alan Borning
ThingLab -- A Constraint Oriented Simulation Laboratory
Technical Report No. SSL-79-3
Xerox Palo Alto Research Center
July 1979

[Borning & Ingalls 1982]
A. H. Borning and D. H. H. Ingalls
A Type Declaration and Inference System for Smalltalk
9th POPL, 1982, pages 133-141

[Borning & O'Shea 1986]
A. Borning and T. O'Shea
DeltaTalk: An Empirically and Aesthetically Motivated Simplification of the Smalltalk-80 Language
(unpublished) 1986

[Borning 1986]
Alan Borning
Classes versus Prototypes in Object-Oriented Languages
Proceedings of the ACM/IEEE
Fall Joint Computer Conference
Dallas, TX, November 1986, pages 36-40

[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 et. al. 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

[Cointe 1987]
Pierre Cointe
Metaclasses are First Class:
The ObjVlisp Model
OOPSLA '87 Proceedings
Orlando, FL, October 4-8 1977, pages 156-167

[Danvy & Malmkjaer 1988]
Oliver Danvy and Karoline Malmkjaer
Intensions and Extensions in a Reflective Tower
1988 ACM Conference on Lisp and Functional Programming, Smowbird, UT, July 1988
pages 327-341

[des Rivieres & Smith 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 1988]
Jim des Rivieres
Control-Related Meta-Level Facilities in Lisp
in Meta-Level Architectures and Reflection
P. Maes and D. Nardi, editors
Elsevier Science Publishers
B. V. North-Holland, 1988, pages 101-110

[Deutsch 1983]
L. Peter Deutsch
Reusability in the Smalltalk-80 Programming System
ITT Proceedings of the Workshop on Reusability in Programming, 1983, pages 72-76
(reprinted in Tutorial on Software Reusability, IEEE Computer Society Press, 1987)

[Deutsch & Schiffman 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

[Ellis & Stroustrup 1990]
Margaret A. Ellis and Bjarne Stroustrup
The Annotated C++ Reference Manual
Addison-Wesley, Reading, MA, 1990

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

[Foote & Johnson 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
Object-Oriented Reflective Metalevel Architectures: Pyrite or Panacea?
OOPSLA/ECOOP '90 Workshop on Reflection and Metalevel Architectures
Mamdouh Ibrahim, Brian Foote, Jean-Pierre Briot, Gregor Kiczales, Satoshi Matsuoka, and Takuo Watanabe, organizers

[Foote 1991]
Brian Foote
Flexible Foundations and Movable Walls
OOPSLA '91 Workshop on Reflection and Metalevel Architectures
Phoenix, AZ
Mamdouh Ibrahim, Brian Foote, Pierre Cointe, Gregor Kiczales, Satoshi Matsuoka, and Takuo Watanabe, organizers

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

[Goldberg & Kay 1976]
Adele Goldberg and Alan Kay, editors with the Learning Research Group
Smalltalk-72 Instruction Manual
Xerox Palo Alto Research Center

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

[Goldberg 1984]
Adele Goldberg
Smalltalk-80: The Interactive Programming Environment
Addison-Wesley, Reading, MA, 1984

[Halstead 1985]
Robert H. Halstead, Jr.
MultiLISP: A language for Concurrent Symbolic Computation
ACM Transactions on Programming Languages and Systems
Volume 7,. Number 4
October 1985, pages 501-538

[Hewitt & de Jong 1983]
Carl Hewitt and Peter de Jong
Analyzing the Role of Description and Actions in Open Systems
AAAI '83, pages 162-167

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

[Ibrahim & Cummins 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

[Ingalls 1978]
Daniel H. H. Ingalls
The Smalltalk-76 Programming System Design and Implementation
5th ACM Symposium on POPL, pages 9-15
Tucson, AZ, USA, January 1978

[Ingalls 1983]
Daniel H. H. Ingalls
The Evolution of the Smalltalk-80 Virtual Machine
in Smalltalk 80: Bits of History, Words of Advice
Glenn Krasner, editor
Addison-Wesley, Reading, MA
1983

[Jefferson 1985]
David. R. Jefferson
Virtual Time
ACM Transactions on Programming Langauges and Systems
Volume 7, Number 3, pages 404-425, July 1985

[Johnson & Foote 1988]
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 et. al. 1988]
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

[Kaiser & Garlan 1987]
Gail E. Kaiser and David Garlan
Melding Software Systems f

rom Reusable Building Blocks
IEEE Software, Volume 4 Number 4
July 1987 pages 17-24

[Keene 1989]
Sonya E. Keene
Object-Oriented Programming in Common Lisp
A Programmer's Introduction to CLOS
Addison-Wesley, 1989

[Kiczales & Rodriguez 1990]
Gregor Kiczales and Luis Rodriguez
Efficient Method Dispatch in PCL
Proceedings of the 1990 ACM Conference
on Lisp and Functional Programming
Nice, France, June 1990, pages 99-105
The Art of the Metaobject Protocol
MIT Press, 1991

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

[Kim et. al. 1989]
Won Kim and Frederick H. Lochovsky, editors
Object-Oriented Concepts, Databases,
and Applications
Addison-Wesley, Cambridge, MA, 1989

[Krasner et. al. 1983]
Glenn Krasner, editor
Smalltalk 80: Bits of History,
Words of Advice
Addison-Wesley, Reading, MA
1983

[LaLonde et. al. 1986]
Wilf R. LaLonde, Dave A. Thomas
and John R. Pugh
An Exemplar Based Smalltalk
OOPSLA '86 Proceedings
Portland, OR, October 4-8 1977 pages 322-330

[LaLonde & Van Gulik 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

[Lieberman 1986]
Henry Lieberman
Using Protypical Objects to Implement
Shared Behavior
in Object-Oriented Systems
OOPSLA '86 Proceedings
Portland, OR, October 4-8 1977 pages 214-223

[Lieberman et. al. 1988]
Henry Lieberman, Lynn A. Stein,
and David Ungar
Treaty of Orlando
Special OOPSLA '87 Addendum to
the Proceedings
SIGPLAN Notices, May 1988
Volume 23, Number 5

[Lindsey & van der Muelen 1973]
C. H. Lindsey and S.G van der Muelen
Informal Introduction to ALGOL 68
North-Holland Publishing Co., Amsterdam
American Elsevier Publishing Co., NY, 1973

[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

[Maes et. al. 1988]
Pattie Maes and Daniele Nardi, editors
Meta-Level Architectures and Reflection
Elsevier Science Publishers
B. V. North-Holland, 1988

[McCarthy 1960]
John McCarthy
Recursive Functions of Symbolic Expressions
and their Computation by Machine, part 1
Communiations of the ACM
Volume 3, Number 4, pages 184-185

[McCarthy et. al. 1962]
John McCarthy, Paul W. Abrahams,
Daniel J. Edwards, Timothy P. Hart,
and Michael I. Levin
Lisp 1.5 Programmer's Manual
MIT Press, Cambridge, MA, 1962

[McCarthy 1978]
John McCarthy
History of Lisp
ACM SIGPLAN History of Programming
Languages Conference
Los Angeles, CA June 1-3 1978

pages 217-223

[McCullough 1987]
Paul L. McCullough
Transparent Forwarding: First Steps
OOPSLA '87 Proceedings
Orlando, FL, October 4-8 1987 pages 331-341

[Messick & Beck 1985]
Steven L. Messick and Kent L. Beck
Active Variables in Smalltalk-80
Technical Report CR-85-09
Computer Research Lab, Tektronix, Inc., 1985

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

[Pascoe 1986]
Geoffrey A. Pascoe
Encapsulators: A New Software Paradigm in Smalltalk-80
OOPSLA '86 Proceedings
Portland, OR, September 29-October 2 1986, pages 341-346

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

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

[Smith & des Rivieres 1984]
Brian Cantwell Smith and Jim des Rivieres
Interim 3-LISP Reference Manual
Xerox Intelligent Systems Laboratory ISL-1
Xerox Palo Alto Research Center
June 1984

[Smith 1987]
Randall B. Smith
Experiences with the Alternate Reality Kit: An Example of the Tension Between Literalism and Magic.
CHI+GI 1987 Conference Proceedings

[Steele & Sussmann 1976]
Guy Lewis Steele Jr. and Gerald Jay Sussman
Lambda: The Ultimate Imperative
MIT AI Memo 353
March 10, 1976

[Steele 1976]
Guy Lewis Steele Jr.
Lambda: The Ultimate Declarative
MIT AI Memo 379
November 1976

[Steele 1977]
Guy Lewis Steele Jr.
Debunking the "Expensive Procedure Call" Myth or, Procedure Call Implementations Considered Harmful, or Lambda: The Ultimate GOTO
MIT AI Memo 443
October 1977

[Steele 1984]
Guy L. Steele Jr.
Common Lisp: The Language
Digital Press, 1984

[Steele 1990]
Guy L. Steele Jr.
Common Lisp: The Language Second Edition
Digital Press, 1990

[Stefik & Bobrow 1986]
Mark Stefik and Daniel G. Bobrow
Object-Oriented Programming: Themes and Variations
AI Magazine 6(4): 40-62, Winter, 1986

[Stefik et. al. 1986]
M. Stefik, D. Bobrow and K. Kahn
Integrating Access-Oriented Programming into a Multiprogramming Environment
IEEE Software, 3, 1 (January 1986), pages 10-18

[Stein 1987]
Lynn Andea Stein
Delegation is Inhertance
OOPSLA '87 Proceedings
Orlando, FL, October 4-8 1977 pages 138-146

[Stein et. al. 1989]
Lynn Andea Stein, Henry Lieberman, and David Ungar
A Shared View of Sharing: The Treaty of Orlando
in Object-Oriented Concepts, Databases, and Applications
Won Kim and Frederick H. Lochovsky, editors
Addison-Wesley, Reading, MA, 1989

[Stroustrup 1986]
Bjarne Stroustrup
The C++ Programming Language
Addison-Wesley, Reading, MA, 1986

[Stroustrup 1991]
Bjarne Stroustrup
The C++ Programming Language
Second Edition
Addison-Wesley, Reading, MA, 1991

[Sussmann & Steele 1978]
Guy Lewis Steele Jr. and Gerald Jay Sussman
The Art of the Interpreter, or
The Modularity Complex
(Parts Zero, One, and Two)
MIT AI Memo 453
May 1978

[Tiemann 1988]
Michael D. Tiemann
Solving the RPC problem in GNU C++
1988 USENIX C++ Conference
Denver, CO, October 17-21 1988

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

[van Wijngaarden et. al. 1976]
A. van Wijngaarden, B. J. Mailoux,
J. E. L. Peck, C. H. A. Koster,
M. Sintzoff, C. H. Lindsey,
L. G. L. T. Meertens and R. J. Fisker
Springer-Verlag, Berlin, Heidelberg, New York
1976

[Yokote & Tokoro 1986]
Yasuhiko Yokote and Mario Tokoro
The Design and Implementation of
ConcurrentSmalltalk
OOPSLA '86 Proceedings
Portland, OR, September 29-October 2 1986
pages 331-340

[Wand & Friedman 1986]
Mitchell Wand and Daniel P. Friedman
The Mystery of the Tower Revealed:
A Non-Reflective Description of
the Reflective Tower
ACM Conference on
Lisp and Functional Programming
Boston, MA, August 1986

[Watanabe & Yonezawa 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

[Yonezawa et. al. 1989]
Akinori Yonezawa, editor
ABCL: An Object-Oriented Concurrent System
MIT Press, Cambridge, MA
1989