Architectural Balkanization in the Post-Linguistic Era

Brian Foote[1]
University of Illinois at Urbana-Champaign
foote@cs.uiuc.edu
20 August 1993

A POSITION PAPER
for the
OOPSLA '93 Workshop on
Object-Oriented Reflection and Metalevel Architectures

An Architectural Vacuum

All around us, traditional edifices are crumbling. As change accelerates, new forces rush to fill the voids. Such change, though inevitable, is not always for the better. In the absence of traditional restraints, problems that arise may be dealt with in a haphazard, piecemeal fashion. The lack of an overriding vision to supplant the old order is a recipe for confusion, and even conflict.

In the software world, the architectural landmarks that have guided the organization of traditional systems for at least a generation are proving to be of increasingly limited value in coping with emerging challenges. The familiar division of labor among languages, compilers, linkers, loaders, applications, operating systems, and file systems now seems strained. In particular, the programming language notion itself seems frayed.

It is increasingly evident that the software world is looking to the object-oriented community for a fresh architectural vision Objects themselves are being asked to bear architectural burdens now borne by more traditional system components. However, for such a vision to be realized, designers must have a better understanding of exactly what it means to build object-oriented programming systems, languages, and applications themselves out of *first-class objects*. And this, after all, is precisely the research agenda of the object-oriented reflection community

Living Languages

The synergy between reflection and 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-

[1]Address: 1304 W. Springfield, Urbana, IL 61801, USA +(217) 333-3411, FAX +(217) 244-5876

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.

Modern programming languages are arguably the single most significant achievement of two generations of software-related research. However, success often breeds a sense of disciplinary complacency. Maybe the question that should be asked is not what should this-or-that ++ look like. Perhaps we should focus instead on what kind of architectural ideas will succeed programming languages themselves.

The challenges we are calling upon the next generation of programming languages to address are quite different from those that drove the design of the current crop of languages. Concurrent computing, distributed computing, persistent object bases, and graphical user interfaces all present challenges that are inadequately addressed by mainstream programming languages. Together, they emphasize the need to provide a way for languages to adapt as changing requirements are confronted.

Traditional programming languages were in essence carved in stone when their defining reports (and revised reports, and revised revised reports) were issued. Building languages themselves out of objects can allow programming languages be specialized and to adapt as individual programmers address new problems.

The process of building domain specific reusable artifacts can be thought of as building a high-level, domain specific programming language. The developers of such code should not be impeded by their languages or tools. If such developers are indeed engaged in language design (and I believe they are) we should strive to make sure that our programming systems get out of the way and allow them to do the best job that they can.

A living language is one that is able to adapt and evolve as the needs of the people who use it change. A language which is not monolithic, but that is instead built of dynamic, first-class objects .can provide this kind of flexibility. Indeed, as this architectural approach becomes better understood, these objects themselves, and their protocols, rather than the langauges they comprise, will become the primary architectural focus.

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 referred 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 1991] 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.

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.

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. The following are design principles that might characterize a linguistic framework.

- 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 language's programming model ought to be themselves reified as 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.

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 around

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.

Autonomous First-Class Objects

As walls crumble, the objects out of which the parts of a system are built 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 should permit autonomous objects to break free of the processes that spawned them and migrate unencumbered among other processes, processors, and persistent object bases.

Languages without significant metalevel architectures such as C++ trap objects in binary object files. Once C++ programs are compiled, information pertaining to layout, object and class identity and the like is usually completely discarded. Some more modern systems have found it useful to retain some of this information in vendor-specific browsing and debugging structures. The growing movement to standardize runtime type information (RTTI) in the C++ community is evidence of a genuine need for metainformation. Some of these proposals go so far as to all but establish what are in effect first-class class objects in C++. This movement is driven not by linguistic purists, but by the requirements of real programmers in the field. There is frustration that often programmers must construct mechanisms themselves to reestablish information that the compiler knew in the first place, but threw away.

Languages such as Smalltalk-80 and CLOS have what is in some ways the opposite problem. These languages provide fairly complete metalevel architectures, but usually imprison objects in snapshots or images.

For truly autonomous objects to break the umbilicals that tie them to single processes and images, the traditional division of responsibilities among system components must be refactored.

Autonomous objects must have access to global namespace services, so that they can find the other objects to which they are tied. Autonomous objects that interact with object-bases would benefit from the knowledge that truly first-class objects can glean about their own layouts.

For autonomous objects to function on platforms other than their home platforms, code must be bound to them at runtime. Dynamic translation of the sort found in Smalltalk-80 [Deutsch & Schiffman 1984] and Self [Chambers, Ungar & Lee 1989] could be factored

into the operating system so that native code might be available to any object on demand.[2]

A vital factor in realizing autonomous objects is genuine first-classness. They will require a system-wide object model with a fully realized metalevel architecture, and not one based exclusively on v-tables.

Architectural Balkanization

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 or beside existing languages. There is abundant evidence that first-class, dynamic, metalevel objects are such a feature.

Most programming systems that support graphical user interfaces now support mapped, dynamic data structures called *resources*. These are usually cast at about the level of C structs. However, since they often are be created and manipulated using resource editing tools, they must usually employ their own conventions for manipulating what is, in effect, metainformation. Some realizations add unique symbolic objects that resemble Lisp **atoms**, and powerful, dynamic evaluators to allow runtime resource expressions to be processed. Often, elaborate schemes must be devised to set up runtime correspondences between names for routines in the resource namespace, and the same routines as they where know to the compiler and linker. The irony here is that all the facilities that have to be created in an ad-hoc, implementation specific fashion by the architects of these systems are essentially duplicating things that the original programming system knew how to do as well. In fact, the information that the programmer must redundantly recreate for these systems is often information that the compiler knew in the first place, but compiled away.

For similar reasons, many applications are adding, simple interpreted macro languages to their applications, while building these applications in a more powerful object-oriented language that by virtue of the system's architecture cannot be reused, and is hence out of reach.

Consider the difficulty one encounters in trying to construct a query for an object in an object-oriented database for an object one has not encountered before using C++. The only way to address this issue is to once again construct a dynamic language, with its own metalevel data structures, atop of C++.

The potential for Balkanization can be seen most acutely in current efforts to define object brokering services and object models. In many respects, these seem to be architectural end-runs around the linguistic community. This evasiveness is justified, given the degree to which mainstream language designers have avoided the issues these efforts are trying to address. In the end, its will be objects themselves, and not languages that will be the central focus of system design efforts.

---

[2]Less powerful approaches using dynamic linking and shared libraries to collect together code for dynamic coalitions of objects could be contemplated, with some loss of power and flexibility.

However the more pressing danger in the long run is that the next generation operating systems that are now being designed may be built using second-class, C++-style objects, rather than the real things. Such architectural corner-cutting could seriously hamper their abilities to evolve.

I believe that there is a decent chance that traffic on the Information Superhighway, when it is finally built, will be first-class reflective objects. By building objects out of objects, seemly disparate issues like the ones discussed above can be addressed in a principled fashion using a common architectural approach. Object-oriented reflection, then, can be see as a school of architecture, with true first-classness as its driving imperative.

REFERENCES

[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

[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

[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 1991]
Brian Foote
Flexible Foundations and Movable Walls
OOPSLA '91 Workshop on Reflection and Metalevel Architectures
Phoenix, AZ

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

[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

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

[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

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

Metalevel Architectures can
liberate
programmers from Metaprogramming

A Living Language
is one that is able to adapt and evolve
as the needs of the people who use it change

Innovation occurs in a seemingly piecemeal
fashion, as specific needs arise

Objects are resilient enough
to bear the architectural burden of
replacing languages, systems, and applications

Symptoms

OLE2 CORBA
MIP/RTTI
Object-Oriented Databases
Persistent Objects
Distributed Objects
Choices (Taligent?, Cairo?)
Resource Managers (X, Mac OS, Windows OS)
Shared Libraries with Dynamic Linking
Non-Numerical Concurrent Applications

The Vision
Languages themselves built out of First-Class Objects
Highly Dynamic
Highly Autonomous
Highly Mobile

Threads:
Hewitt's Open Systems
Gregor's Open Implementations

The traffic that will ply the information superhighway
will be reflective objects