Class Warfare: Classes vs. Prototypes

Brian Foote
Dept. of Computer Science
University of Illinois at Urbana-Champaign
4 August 1989

An examination of recent work on prototype-based architectures for object-oriented programming raises a number of interesting questions, the most obvious being: *Are prototype-based object-oriented architectures superior to class-based architectures?* Another way of addressing the same issues might be to ask: *Are class-based architectures over-centralized, and excessively rigid?* This work also raises the related question of whether dynamic, implicitly typed object-oriented architectures are preferable to static, explicitly typed architectures. To some extent, this question boils down to that of asking: *Should the* <u>*primary*</u> *goal driving the design of an object-oriented architecture be maximal flexibility or maximal efficiency?* Finally, by putting the general issue of what object-oriented architectures are the best on the table, this work would seem to make an examination of what role metalevel architectures and reflection might play in the design of object-oriented systems appropriate as well. Put simply, this question becomes: *Are prototype-based architectures flexible enough?*

It is customary for functions such as this one to attempt to take bold, provocative positions on this issues at hand, in the hope that this will lead to spirited, enlightening exchanges among the workshop participants. Of course, the questions raised above are complex, and can defy attempts to find simple, general answers. That said, my positions on the questions above are:

- *Prototype-based architectures are better*
- *Classes are too rigid*
- *Flexibility über alles*
- *No*

I will attempt to justify these positions below. Given the workshop organizer, embracing prototype-based architectures can hardly be construed as an act of courage. However, by advocating the addition of mechanisms for adding more flexibility for such architectures, I may have succeeded in staking out a position to the left of Ungar's.

Recent work on prototype-based architectures for object-oriented programming [Ungar 1987] [LaLonde 1986] [Borning 1986] has shown that these architectures can parsimoniously subsume most of the strengths of class-based object-oriented architectures. Prototype-based architectures are simpler, and (hence) easier to comprehend than class-based schemes. Perhaps more importantly, prototype-based architectures give the programmer a range of organizational alternatives, including the construction of sets of objects that behave like classes.

It may (or may not) surprise the authors of [Ungar 1987] to learn that their work is cited by some as evidence for the contention that if classes did not exist, programmers would find it necessary to invent them. Do class-like objects emerge in prototype-based systems? The answer would seem to be yes. Do prototype-like objects emerge in class-based systems?

Again, the answer is yes. The more important questions would appear to be: If classes (or prototypes for that matter) did not exists could programmers invent them?

It is relatively easy to build class-like objects in prototype-based systems. The need for prototype-like objects arises frequently in class-based systems as well. For instance, graphical objects such as character fonts and icons that are constructed interactively rather than created using some initialization protocol are better dealt with using prototypes. The experiences of the authors of Thinglab and the Alternate Reality Kit underscore this point.

However, the full power of prototype objects in languages such as SELF can be difficult to implement in class-based languages such as Smalltalk-80. Some of this difficulty results from the fact Smalltalk requires that the object that specifies how any given instance may behave (i.e. its class) be distinct from the instance itself. This dualistic philosophy is in contrast to the monistic view taken in SELF, which allows object to embody descriptions of their own behavior. As a result, SELF-like objects that specify behavior (heavyweight instances) must be simulated in Smalltalk using lightweight, perhaps anonymous, classes. Smalltalk does not supply explicit support for anonymous classes, however, they can be constructed (with some difficulty) by the user.

Dynamically changing the class of an object is complicated in Smalltalk- as well. To do so, a new class must be created, then an instance of that class with the state of the original object must be set up. Finally, the old object must be asked to **become:** the new object.

We [Foote 1989] have extended the Smalltalk- Virtual Machine to circumvent this problem. Our interpreter includes a primitive that permits the class of an object to be changed dynamically. This in turn makes it relatively easy to introduce dynamic behavior changes. This feature also facilitates the construction of *metaobjects.* Metaobjects are lightweight classes that have but a single instance. This instance is called the metaobject's *referent.* A metaobject contains an explicit reference to its referent. An object, working in tandem with a metaobject, is functionally similar to a SELF object that describes its own behavior.

Borning [1986] identifies a number of different roles that class objects play in Smalltalk-80:

1) *generators of objects*
2) *descriptions of the representations of their instances (templates)*
3) *descriptions of the message **protocol** of their instances*
4) *elements in the description of the object taxonomy*
5) *a means of implementing differential programming (by difference)*
6) *repositories for methods for receiving messages*
7) *devices for dynamically updating many objects when a method is changed*
8) *sets of all instances of those classes*

They who would dispense with classes must address the issue of how to distribute these responsibilities in an alternative architecture. Indeed, different schemes for distributing these functionalities account for many of the differences among different object-oriented architectures. For instance, the use of prototype cloning vs. metalevel object "factories" distinguishes the prototype-based architectures from the class-based ones. A desire to separate the description of the object taxonomy from the inheritance mechanism distinguishes the Exemplar proposal of [LaLonde 1986]. Should some of these functions be embodied in specific objects at all? Or should they be implicitly distributed throughout the system, and calculated when needed?

Class-based architectures can impose a degree of structural rigidity on a system that can stifle its evolution. This is because they do not allow the kind of dynamic system reorganization that prototype-based architectures permit. This rigidity can be particularly harmful in mature, successful systems that must then evolve further to meet a host of new requirements. It is essential that the structure of a system be able to evolve in such a way that it matches that of the problem itself. (Form must continue to follow function.)

Class-based object oriented systems are far superior to conventional programming systems in meeting such demands [Foote 1988]. They can themselves become ossified, however. It is very difficult to predict the demands that will be placed on a system as it evolves, and hence it is essential that the programming system used to build it provide as much flexibility as possible so that the system can be adapted to accommodate new requirements as they arise.

Three phases can be identified in the lifecycle of object-oriented systems and components. An initial prototyping phase, an exploratory, expansionary phase, in which a successful design must incorporate a range of new requirements, and a consolidation phase, during which a mature system is reorganized to cleanly incorporate the successful additions that were made during the second phase. Tools to aid this-reorganization process are clearly needed. It is more likely to be successful if the underlying programming system itself does not inhibit the gradual metamorphosis of evolving objects. Object-oriented architectures that support the emergence of new architectural approaches will have a distinct advantage over those that do not during during this phase of a system's evolution. (Evolve or die...)

Some problems demand more flexibility than either Smalltalk or SELF-like architectures can currently offer. Such problems (such as the construction of monitors, futures, and actor-like objects) require that certain objects be able to control how messages sent to them are dispatched. The addition of a handful of reflective facilities [Foote 1989], such as an ability to selectively redefine an object's message dispatching mechanism, can allow such problems to be addressed.

I believe that the conceptual simplicity of prototype-based systems makes them particularly suitable for meta-architectural embellishment. It is possible to envision a user modifiable meta-level for a language like SELF (EGO: A Reflective SELF?) that would provide explicit access to concrete definitions of objects such a slots, as well as to methods for evaluating numbers, methods, blocks, and the like. Such an architecture would permit the selective redefinition of, for example, slots (to provide active variables) or of the evaluate method for a given family of objects (to provide explicit control over the message dispatch process).

Reflection, of course, is not by any means a panacea. Is metalevel hocus-pocus a backhanded way of not addressing a system's serious structural problems? That is to say, is it employed in situations that are analogous to those that force people to run hardware emulators to keep sclerotic programs running? Certainly, a mechanism that allows the programmer to construct localized deviations from the default semantics of his or her programming system has a high abuse potential. Like any other powerful tool, reflection can be used for good or ill. None-the-less, I believe that the complexity of the systems that currently confront today's software engineers demands that the programming systems that they use provide a maximal degree of flexibility. Prototype-based architectures, because of their simplicity, may provide an ideal foundation for building systems that achieve such flexibility.

References

[Borning 1986]
A. H. Borning
Classes vs. Prototypes in Object-Oriented Systems
ACM/IEEE Fall Joint Computer Conference, November 1986

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

[Foote 1989]
Brian Foote and Ralph E. Johnson
Reflective Facilities in Smalltalk-
To appear in: OOPSLA '89 Proceedings

[Johnson 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

[LaLonde 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

[Maes 1987]
Pattie Maes
Concepts and Experiments in
Computational Reflection
OOPSLA '87 Proceedings
Orlando, FL, October 4-8 1977 pages 147-I 55

[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