

# Prototype-Based Languages: Object Lessons from Class-Free Programming (Panel)

Moderator: **Randall B. Smith**

Sun Microsystems Laboratories  
2550 Garcia Ave. MTV 29-116  
Mountain View, CA 94043  
`randall.smith@sun.com`

## Panelists:

**Mark Lentczner**  
Glyphic Technology  
1209 Villa Street  
Mtn. View, CA 94041

`markl@netcom.com`

**Walter R. Smith**  
Apple Computer, Inc.  
1 Infinite Loop,  
MS 305-2B  
Cupertino, CA 95014

`wrs@apple.com`

**Antero Taivalsaari**  
Nokia Research Center  
P.O. Box 45, 00211  
Helsinki, Finland

`antero.taivalsaari@research.nokia.com`

**David Ungar**  
Sun Microsystems  
Laboratories  
2550 Garcia Ave. MTV 29-116  
Mtn. View, CA 94043

`ungar@sun.com`

## Introduction

An alternative to the class-based object-oriented language model has emerged in recent years. In this prototype-based paradigm there are no classes in the traditional sense. In object-oriented programming, a class usually provides a particular set of functionality: an object is created by instantiation of the class, the behavior for instances is held by the class, and the number and name of instance variables is specified by the class. In most prototype-based systems however, a new object is made by copying, behavior can be held directly in an individual object or inherited from others, and what an object implements by instance variables or by methods is not proscribed.

The purpose of the panel is to create a public forum in which the designers of four fairly mature and robust prototype-based languages discuss the benefits and problems of the prototype approach, and argue the merits of their particular design decisions.

**Brief history.** Prototypes are arguably as old as classes: germs of the prototype ideas can be seen as early as 1963 in Sketchpad [9]. The following decades saw this approach taken in the actor-based

languages and derivatives, and languages with a strong graphical component (e.g.: [1], [7], [8], [10]). The 80's brought further discussion and elaboration, and a few experimental languages or prototype-based systems ([2], [3], [5] - [8], [10]; for a review see [4]).

Today there are quite a few prototype-based languages including several systems that are mature enough to enable us to ask about real-world experience. This panel was limited to fairly pure object-oriented languages, in an attempt to gain clarity on the issues. (By "fairly pure" we mean that almost all data are objects, and most computation is triggered by message sending). This panel features designers of four such prototype-based programming language systems: Glyphic Script\*, Kevo, Self, and NewtonScript†. Each system is robust and efficient enough to support real-world applications.

**Why prototypes?** Proponents of prototypes usually argue that the resulting language is more *con-*

\* Glyphic and Codeworks are trademarks of Glyphic Technology

† NewtonScript and Newton Toolkit are trademarks and Newton is a registered trademark of Apple Computer, Inc.

crete (since copying and modification of an example object is more direct), conceptually simpler, since there is only one kind of object, and more flexible since any object can hold behavior or, by being copied, serve as a source for new objects.

Lentzner and Ungar both point out that it is natural in prototype systems to unify state and behavior. Under this unification, it is impossible to distinguish between code that directly reads or writes state ("instance variables") from code that invokes methods, making it much easier to reuse code. This unification can also be achieved in class-based systems, but traditionally code within the methods of a class represents a store into an instance variable as different from a message send.

**Problems and issues.** One of the more interesting problems panelists mention might be called the "family resemblance" problem. In a prototype-based object model, any object holding behavior can have that behavior applied to itself. Therefore, with inheritance (or delegation), an object tends to behave at least partially like its inheritance children. Yet parents and children often play extremely different roles in the system, so this family resemblance is not always appropriate. In the following sections we see this problem show up in various ways for three of the languages - Kevo manages to escape the problem by avoiding delegation altogether.

A class gives each instance a set of instance variables for every class in the inheritance path. How does the corresponding thing happen in prototype-based languages? The four systems have different approaches to what is sometimes called the "copy down" question.

The reader might also note how structure arises in the various languages, which do not have class hierarchies to impose order.

**Format.** In the following sections, each panelist gives a brief overview of his language model, discusses the benefits and drawbacks he has seen in the prototype approach, and illustrates his language by working the same simple problem. The panel session should be an interesting forum for discussion and debate.

---

**Randall B. Smith**, panel moderator, is co-leader of the *Self* project at Sun Microsystems Laboratories. He previously worked at Xerox PARC where he built the *Alternate Reality Kit*, the *Shared Alternate Reality Kit*, and with David Ungar designed the *Self* language. He received his Ph.D. in theoretical physics from the University of California at San Diego in 1981.

---

## References:

- [1] Borning, A.H., *Thinglab - a Constraint-Oriented Simulation Laboratory*. Ph. D. dissertation, Stanford University (1979).
- [2] Borning, A.H., The Programming Language Aspects of Thinglab, A Constraint-Oriented Simulation Laboratory. In *ACM Transactions on Programming Languages and Systems*, 3, 4 (1981) pp. 353-387.
- [3] Borning, A. H., Classes Versus Prototypes in Object-Oriented Languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference* (1986) pp 36-40.
- [4] Dony, C., Malefant, J., Cointe, P., Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and their Validation. In *n OOPSLA '92 Conference Proceedings*, Published as *Sigplan Notices*, 27, 10, (1992). pp. 201-217.
- [5] LaLonde, W.R., Thomas, D.A., An Exemplar based Smalltalk. In *OOPSLA '86 Conference Proceedings*, Published as *Sigplan Notices*, 21, 11, (1986) pp 30-37.
- [6] Lieberman, H. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *OOPSLA '86 Conference Proceedings*, Published as *Sigplan Notices*, 21, 11, (1986). pp. 214-223.
- [7] Myers, B.A., Giuse, D., Dannenberg, R.B., Vander Zanden, B., Kosbie, D. S., Pervin, E., Mickish, A, Marchal, P., "Garnet: Comprehensive Support for Graphical Highly Interactive User Interfaces." *IEEE Computer*, 23, 11, (Nov. 1990), pp 71-85.
- [8] Smith, R. B., Experiences with the Alternate Reality Kit: An Example of the Tension Between Lit-

eralism and Magic, in Proceedings of the CHI+GI '87 Conference (1987) pp. 61-67.

[9] Sutherland, I.E., *Sketchpad: a man-machine graphical communication system*. MIT Lincoln Laboratory Tech. Rept. No. 296, 1963.

[10] Yonezawa, A., Briot, J., and Shibayama, E, Object-Oriented Concurrent Programming in ABCL/1, In OOPSLA '86 Conference Proceedings, Published as Sigplan Notices, 21, 11, (1986), pp 258-268.

---

## Glyphic Script

Mark Lentzner

Glyphic Script ([1], [2]) is the result of two years of effort to create a small portable, and practical development environment and language. Our primary goals were directness of the programming process and power of the programming language. Early on, we felt that prototypes would enable us to reach these goals more quickly and easily.

In Glyphic Script, an object consists of a number of properties which may be variables or methods. Any object can serve as the parent (or "class" or "prototype") of other objects. This is a single inheritance model where properties of a parent may be inherited by its children. Inheritance is seen as an organizing tool for the programmer, not as a pure type abstraction mechanism. The object model has two types of copying: the "new" operation to create an instance and the "copy" operation to create a peer. Through property "scopes" the object has control over what is copied, what is shared and what is inherited by the objects created with these operations.

### How Well Prototypes Supported Our Goals

Directness is a subjective property of programming environments. We say that an environment more direct if the programming process has fewer levels of indirection to achieve an end result. Directness was one of the primary arguments for choosing a prototype-based language: We reasoned that classes, even when implemented as objects (for example in Smalltalk), still represent a level of abstraction away from

the elements of the program.

Classes describe a program entity, whereas prototypes are an example of a program entity. It should be more direct for a programmer to edit an example of a program entity (a prototype) than to edit a description of a program entity (a class) to make a change. In practice with our system, we observe that programmers build and debug prototypes, and then generalize by creating instances. Programmers clearly make use of the directness of prototypes.

The power of a language is also a subjective property. We can draw on our experience with the system to estimate it: Like many other interactive systems, much of the Codeworks development environment for Glyphic Script is written in Glyphic Script. A sizable amount of code has been written in Glyphic Script including a data collection library, an application framework, a viewing system, an import/export library, and a debugger. The whole system comprises nine thousand lines of code. Our ability to create a full applications development environment attests to the general power of the language.

### Problems and Surprises of Prototypes

**Property Scopes.** The part of the language that gave us the most trouble and took the longest to settle was property scopes. While it is clearly possible to create a prototype language without the notions of property scope, we found that it was difficult to get by without them for long. Initially we started with the kinds of scopes that one finds in C++: The emphasis was on controlling visibility of properties to other objects. In its first year of evolution, the language ended up with a set of scopes with a very different function: Now the role of scopes is to control inheritance and copying to descendants. Property scopes were required to make prototypes practical.

**Keeping Classes.** We never found the need to remove the concept of class from the system: The concept of class works well as an organizing principle in libraries and programs. While the development environment and libraries talk of classes, these are really prototype objects and the language semantics do not treat them any differently. Instead of eliminat-

ing classes, Glyphic Script demystifies them and makes them truly regular object.

**Prototype Failure.** Occasionally prototypes don't work as prototypical objects: In the case of numbers, there is an object that is the parent of all numbers and contains all their shared behavior. This object isn't a good prototype at all: if you send it an arithmetic message, it will fail! This type of behavior can also be observed to a degree in user created prototypes where the instance variables aren't set to valid values. While this problem is minor (we did not consider it important enough to complicate the object model to get around it), the programmer has to be aware of this new type of failure and why it happens.

### Variables and Methods

There is a property that many prototype-based languages share that really has nothing to do with prototypes: In three of the four languages represented at this panel, access to methods and access to object variables are treated the same. A method accessing an object's variable (including its own) cannot distinguish this from a no argument message send. This feature is independent of prototypes. However, it seems to go part and parcel with the territory, and we are not sure why! We choose this unification because we felt it simplified the object model and allowed the programmer to later control access to a variable via a method without substantial recoding, and it reduced the amount of syntax in the language. We feel that more attention needs to be paid to this choice and its relative merits in the future.

### Conclusion

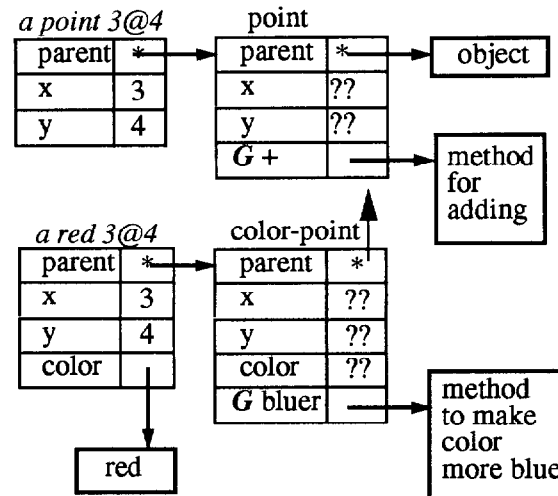
Glyphic Script has shown itself to meet its goals as a direct and powerful language. It owes much of that success to the choice of prototypes for its object model. However, it did not abandon the concept of class in the process. We hope that Glyphic Script helps show that prototypes are not just conceptual curiosities, but a powerful concept for the next generation of practical languages.

### The Example

This section explains how the example of Cartesian points is handled in Glyphic Script:

Cartesian points are objects that inherit from a common prototype called "point" in the standard system library. In the figure, point contains properties for x and y (whose initial values are "??", a special value in Glyphic Script meaning "unknown"), and a "+" property for adding points.

Point also inherits a host of standard properties from "object", a common grand-parent in the system. A new point object is created by sending the message "new" to point and then setting the x and y properties to useful values.



**Figure:** Points and colored points in Glyphic Script.

To create a "color-point", a programmer sends the message "new" to point. This creates a new object with only x and y properties, and a parent of point. Then the programmer adds a "color" property to the new object and a "bluer" property with a method. New objects created by sending "new" to this new prototype would have three properties to assign values to: x, y, and color.

In this example, all properties have the "copied" scope, except the two labeled "G" for "global" scope ("+" in point and "bluer" in color-point). Copied properties are copied during a "new" operation. Global properties are not copied during "new", but are inherited and thus represent shared properties.

---

*Mark Lentzner is a founder of Glyphic Technology where he has been a principal designer of the language Glyphic Script and its development*

*environment Codeworks. He received a B.A. in Applied Mathematics from Harvard University. Previously he worked at Apple Computer (originally in the Smalltalk group, then manager of the Sound and Music effort) and GO Corporation (on application frameworks). When he is not staring at a computer screen, he prefers to be rock climbing.*

---

## References

- [1] Glyphic Technology, "Glyphic Codeworks(tm) Scripting". Unpublished manual (1994).
- [2] Schwartz, B., Lentzner, M., "Direct Programming Using a Unified Object Model". In "OOPSLA '92 Addendum to the Proceedings". Published as "OOPS Messenger", 4, 2, (1993) 237.

---

## Self

David Ungar

The language Self was originally designed with to be a successor to Smalltalk. Self was designed by removal; we removed classes and variables from Smalltalk and tried to see if we could still program in it. Peter Deutsch inspired us to unify Smalltalk's seven kinds of variables into one kind of access, which we based on inheritance. [1]

In Self, prototypes are prototypical; a prototype has exactly the same set of slots as any copy. In fact, there is no linguistic way to distinguish them. For example, the prototypical panel object would have slots named "members" "topic" "name" and "parent", and each copy would have exactly the same slots. Any shared information would reside in its ancestors.

Each prototype implements a "copy" message, which "clones" (shallow-copies) the prototype and then performs any initialization required.

Self unifies variables and methods in a fashion that makes code more reusable. Every identifier in Self code is interpreted as a message-send, and if no receiver is given, the message is sent to "self." Either data or code can be found in a slot as the result of sending a message. Data is just returned, and code is run. Assignment is accomplished when sending a message with one argument finds a slot containing a

special assignment primitive operation.

In Self, objects inherit from objects, and inheritance performs exactly one function; that of allowing the same information to be in two places (objects) at the same time. For example, consider a method that is inherited. You could just copy that method down to every object that inherited the method, and the system would behave the same, *until* you needed to change the method. Then you would have change every copy, or replace every copy somehow. Inheriting one copy of a method makes it easier to change, as well as grouping methods into class-like clumps (called traits objects in Self) for easy comprehension. And, since Self lets you freely intermix methods and data, data can be inherited just as methods. Such inherited data slots enable objects to share state (somewhat like class variables) more flexibly.

Since in Self, objects inherit from objects, and since there are no classes to enforce structural conformance, objects can easily alter their parents at runtime. This dynamic inheritance capability turns out to be an excellent vehicle for implementing objects whose behavior changes completely as they move among a small set of states. For example, a window that can either be expanded or iconified can be easily implemented in Self by switching a parent pointer between an object holding state and inheriting behavior for the icon, and another object representing the expanded version. Data common to the states can be conveniently stored in the child. I know of no other programming construct as well-suited to this sort of example.

However, three interesting issues have arisen in Self's design.

**1. Browsing vs. Programming:** Not unique to Self, this problem arises for every OOPL with inheritance. Although the conceptualization of the system as an inheritance hierarchy is well suited to understanding the effect of changing a method, it does not let you see what set of messages any particular object, or kind of objects, responds to. The environment must provide an alternate view that subordinates inheritance by simply showing each object as the union of inherited attributes. This need suggests that inheritance may not be fundamental to using or browsing objects, but that rather it arises as a consequence of programming them.

**2. Traits are not concrete:** Although Self uses objects to hold bundles of shared behavior (traits), such objects cannot respond to messages as well as you might expect. For instance, if you send “print” to the traits object for points, it will try to send itself “x”, but, since it holds only shared information, it has no “x” slot. This problem seems to be a consequence of Self manifesting a programmer-level phenomenon at the base level of runtime objects.

**3. Corrupted prototypes:** Since a Self prototype is just like any other object, it will understand messages that change its state. For example, you could add an element to the prototypical list. Then every method that copied the prototype list would be surprised by the presence of an extra element. Concreteness can be a two-edged sword.

**4. Fewer structural guarantees:** In a class-based language, you can guarantee that any object inheriting a method like “countSnorts” will possess a required instance variable like “snortCount”, just by defining both in the same class. In Self (though not in all prototype languages), you cannot.

**Benefits:**

- 1. Unique objects:** Objects such as nil, true, false can be constructed without building a class.
- 2. Simplicity:** Prototype-based languages remove the class-instance relation and so can be drawn with one less color of mental chalk. Also, the infinite-meta-class-regress does not arise when there are no classes. Copying is easier to explain than instantiating.
- 3. Structural diversity & code reuse:** Most class-based languages require that all of a class's instances, and all of its subclasses' instances include all of its instance variables. This implementation convenience forces the programmer to cleave classes into abstract superclasses and concrete subclasses. Prototypes allow the programmer to change his or her mind about which attributes are stored vs. computed without refactoring parents, or rewriting methods.
- 4. Concreteness:** A prototypical circle can be graphically depicted on the screen, but how do you show the class of all circles? Similarly, a programmer can understand what a circle object is by inspecting the contents of slots in the prototype. But, for the Circle class, he or she would have to read the instantiation

code. In Self, even methods are stored as prototype activation records, so that locals can be set by the programmer with values to help other programmers understand.

**Example**

In order to make a cartesian point object in Self, you would take an empty object and add slots to hold and assign to x and y. Then you would create another object to hold shared information for all points, and refer to that one by a parent slot in your point object.

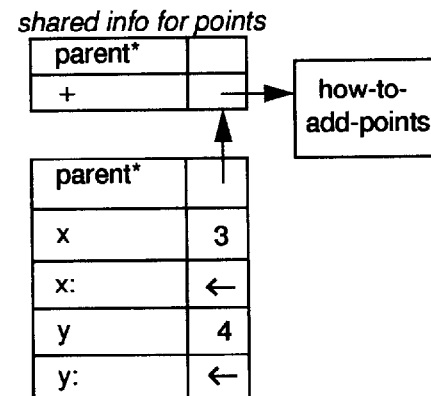


Figure 1. A point in Self.

New points could be obtained by cloning the original point. Now suppose you want to extend your point with color information. First, you would take a point and add slots to it to hold the color information for that point. Then, you would interpose a new parent object to hold information shared by all colored point

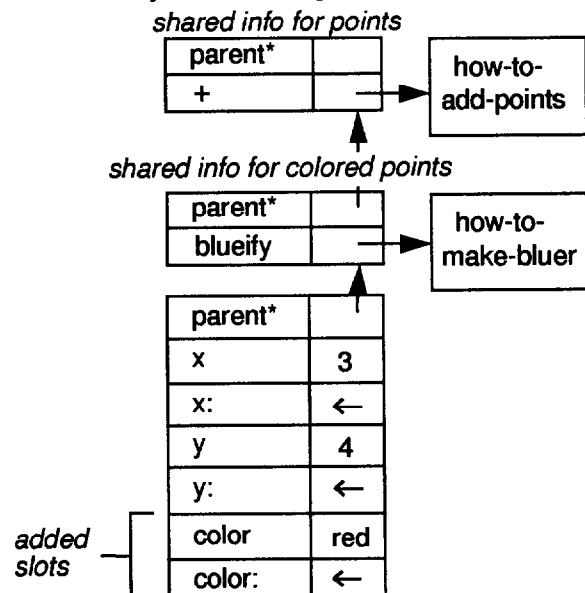


Figure 2. A colored point in Self.

---

*David Ungar has long been fascinated by programming paradigms that change the way people think, novel implementation techniques that make new languages feasible, and user interfaces that vanish. He co-leads the Self project at Sun Microsystems Laboratories. Before that, Dave taught at Stanford, consulted for Apple's Newton group and for ParcPlace Systems, and obtained a doctorate at U.C. Berkeley.*

---

## References

[1] David Ungar and Randall B. Smith, "SELF: The Power of Simplicity," *Proceedings of the 1987 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Orlando, FL, October, 1987, pp. 227-242. A revised version appeared in the *Journal of Lisp and Symbolic Computation*, 4(3), Kluwer Academic Publishers, June, 1991.

---

# NewtonScript

Walter R. Smith

## Goals and audience

NewtonScript was created specifically for developing applications for the Newton platform [Smith94a]. It is a general-purpose language (in fact, NewtonScript compilers have been written in NewtonScript), but particularly well suited to Newton programming. Newton's language, view system, and persistent object store were designed in parallel to work well together.

The language and its development environment, the Newton Toolkit, are being used by several thousand Newton developers. Many of them have no previous experience with dynamic object-oriented languages. Newton Toolkit has generated surprisingly positive feedback; the adjustment to prototype-based programming appears to be relatively easy and pleasant for most of our developers. (Of course, this group is self-selected by the decision to adopt a brand-new platform in the first place!)

Thus, NewtonScript's most important contribution to the field may simply be a demonstration that a prototype-based language can be practical—currently, it

is the *only* language publicly available for Newton—and that it can be acceptable to "real" programmers.

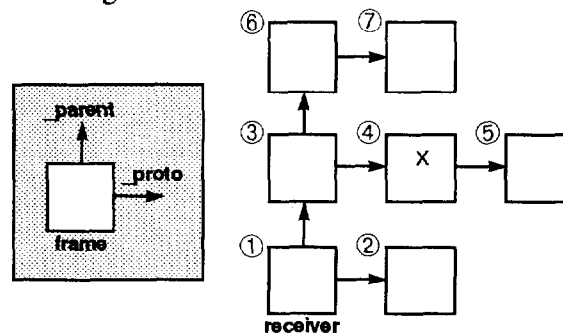
## Object model and inheritance scheme

NewtonScript is based on an "impure" object model. Some data objects, such as integers, strings, and arrays, are manipulated through primitive language constructs or function calls. Only one kind of data object, called a *frame*, can respond to messages.

A frame is a collection of named slots, each slot containing a reference to another object. Frames may be treated as "dumb" records when appropriate, including slot access operators and an in-line constructor syntax. However, they also form the basis of the object-oriented features of NewtonScript.

When a message is sent to a frame (the *receiver*), the system searches that frame for a slot with the same name as the message. The slot must contain a function object. That function is then run using the receiver as a context—that is, the function's free variables are looked up as slots in the receiver. Assignment uses a special "copy-on-write" rule (see [Smith94a]).

A frame may inherit properties by specifying up to two other frames that will be searched for slots it does not contain. These inheritance paths are specified by frame slots called `_proto` and `_parent`. First, the chain of `_proto` frames is searched; if the slot is not found, the search moves to the `_parent` and tries again.



**Figure 1. NewtonScript inheritance.** Lookup proceeds from the receiver in the order shown. Assignment to the variable X will create a slot X in frame ③.

This system evolved symbiotically with the Newton view system. The user interface of a Newton

application is built using frames that may inherit properties through their `_proto` slots. When a view is opened, a small frame is created that contains a `_proto` slot pointing to the corresponding application object and a `_parent` slot pointing to the enclosing view's frame. Thus, the `_proto` chain is used for refinement, while the `_parent` chain is used for containment. Of course, this inheritance structure is useful for more than the view system. For example, it can be used to make class-like objects for shared behavior.

Unlike SELF [Ungar87], NewtonScript separates message sending from slot access. A variable reference is always satisfied by getting the value out of a slot, never by executing a method. This is due mostly to lack of time, not for any technical reason, and may be changed in the future.

### Issues and benefits of prototypes

The main benefit we perceive from using prototypes is that programming is much more direct. Rather than having to write a new class to make a single control act differently, the programmer can just change the control directly. Once one object is working, it's easy to get the effect of a class by using it as the `_proto` of others.

It's easier to explain and use a user interface toolkit without the distinction between classes and instances. There is only one kind of relationship, inheritance, instead of two, instantiation and subclassing. Making a new control and making a new kind of control involve the same operations.

We use the inheritance system to minimize memory usage. Much of the information stays in ROM or compressed storage. If a slot is not changed, no RAM needs to be allocated to store it.

The term "prototype-based" is in some ways a misnomer, since many of the objects a programmer creates in NewtonScript are not prototypes. The prototype for a radio button, for example, has no name slot, and thus cannot display itself. Users of the prototype are required to supply enough information to make it work. In practice, this does not appear to be a significant barrier to understanding.

NewtonScript lacks scoping, although it would be useful to avoid accidental encapsulation violations

through name conflicts. The uniformity of the object model makes it difficult to find a consistent place to specify and enforce scoping (the class is such a place).

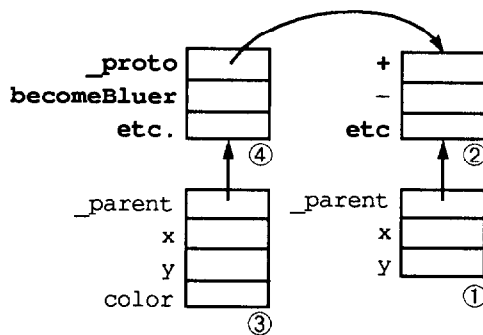
We use a relatively simple bytecode interpreter, having few machine or human resources to expend on more complex techniques, so NewtonScript does not currently approach the speed of C or even an industrial-quality Smalltalk interpreter. This is not entirely due to the use of prototypes, of course, but more simple optimizations become available when classes are used.

### Cartesian point example

The example can be done in NewtonScript using class-like objects [Smith94b]. A point is a frame containing `x` and `y` slots for the values and a `_parent` slot pointing to a frame with the common features of points, such as arithmetic operations.

When a message is sent to a point object, a slot with the corresponding name will be found by following the `_parent` link to the "class" object. The function in that slot will execute in the context of the point object; for example, a reference to the variable `x` will get the value of the `x` slot in that object.

Colored points have an extra `color` slot, and refer to a class object containing methods specific to colored points. This object inherits the original points behaviors through a `_proto` link to the other class object.



**Figure 2. Cartesian point example.** Frame ① is a point "instance" object. Frame ② is the "class" object for points. Frame ③ is a colored point. Frame ④ is the "class" object for colored points.

The variable/method lookup for a colored point searches first in the point itself, then in the colored



point class, and finally in the original point class.

---

*Walter R. Smith is a Senior Software Engineer at Apple Computer, Inc., and one of the principal designers of the Newton platform. His main contributions to Newton were the persistent object store and NewtonScript, the application development language. He received a B.S. in Applied Mathematics from Carnegie Mellon University in 1988, and would probably have a Ph.D. by now if he had ever returned from that summer job in the Newton Group*

---

## References

- [Smith94a] Walter R. Smith. The Newton application architecture. In *Proceedings of the 39th IEEE Computer Society International Conference*, pp. 156-161, San Francisco, 1994.  
Also available as `ftp://ftp.apple.com/pie/newton/articles/COMPCON-Arch.ps`
- [Smith94b] Walter Smith. Class-based NewtonScript programming. *PIE Developers*, January 1994.  
Also available as `ftp://ftp.apple.com/pub/wrs/class-based-NS.ps`
- [Ungar87] David Ungar and Randall B. Smith. Self: the power of simplicity. In *OOPSLA '87 Conference Proceedings*, pp. 227-241, Orlando, Florida, 1987. Published as *SIGPLAN Notices* 22, 12, December 1987.

---

## Kevo

### A delegation-free prototype-based object-oriented language

Antero Taivalsaari

One of the key features of object-oriented programming languages is incremental modification, i.e., the ability to build and refine programs incrementally, rather than editing existing code. In class-based languages the basic incremental modification mechanism is inheritance, while in prototype-based languages alternative mechanisms exist. Self, for instance, uses delegation as the elementary incremental modification mechanism, allowing objects to flexibly share each others' behavior and state.

In the literature delegation is often used as a synonym for prototype-based programming. This is misleading, however, since it is possible to build prototype-based languages which do not support delegation at all, but which nevertheless provide similar support for incremental modification. This was shown by Borning already in 1986, but the ideas were not taken any further. Kevo is an attempt to prove that delegation-free prototype-based languages are indeed feasible and practical.

## Introduction to Kevo

Kevo is a prototype-based language designed around self-sufficient and concrete objects. By self-sufficient, it is meant that each Kevo object contains all the properties needed for implementing a certain abstraction, and that these properties are logically independent of the other objects' properties. This is different from Self which relies heavily on sharing, and requires the programmer to divide objects into separate traits and prototype structures. By concrete, it is meant that Kevo allows the programmer to manipulate each object directly, either on a per-object or per-group basis.

Kevo does not support inheritance in the traditional sense. And unlike many prototype-based object-oriented systems, Kevo does not support delegation. Instead of inheritance and delegation, the essence of inheritance - incremental modification - is captured using concatenation, i.e., by duplicating existing objects and by allowing flexible editing and combination of objects. In practice, concatenation is supported by providing two user-redefinable copying operations, new and clone, and a set of module operations that allow different kinds of modifications to be performed on objects. For instance, to create a ColoredPoint object, one would simply copy an existing Point object, and then add the desired new properties to the copy using the module operation ADDS. Late binding of methods and variables ensures that earlier defined ("inherited") properties can adapt to later modifications in an incremental fashion.

As a result of the above mentioned arrangements, Kevo objects look quite different from Self objects.

Unlike in Self, in which objects are composed of two separate parts - traits and prototypes - Kevo objects are viewed as logical wholes that do not share properties with each other, unless the programmer really wishes so. From the logical point of view this means that even the methods of objects are independent, and can be modified without affecting the other objects in the system. (Note that at the implementation level virtual copying techniques are used extensively to conserve memory, and therefore physically methods may be shared although logically they are not; the programmer is not aware of such sharing in any way other than checking the megabytes-used-meter).

One apparent shortcoming of a naive object model supporting fully independent objects is the inability to manipulate larger groups of objects at once. In practice, having to, e.g., fix the same bug in possibly thousands of copies of the same object manually does not sound very appealing, and therefore a mechanism for groupwise manipulation is needed. To overcome this problem, Kevo internally maintains information about clone families: groups of objects with similar behavior and structure. These families allow objects to be rapidly compared against other objects, serving as a guideline for change propagation when larger groups of objects need to be modified. For instance, to add a new method or a variable to all ColoredPoint objects, one would apply a special groupwise module operation ADDS\* to one of the existing ColoredPoint objects.

Due to the absence of inheritance and delegation, the Kevo system is organized quite differently from most other object-oriented systems. Instead of inheritance- or delegation-based structuring, Kevo relies heavily on part-whole based structuring, or composition; the system is divided into increasingly smaller subsystems, modules or applications, according to the needs of the programmer, using objects as "directories" to other objects. As a result, objects belonging to one subsystem can be kept in one place, without having to introduce additional structuring facilities such as class categories or modules. The resulting structure resembles conventional file systems, making the system intuitive to learn and use.

Our current implementation of Kevo is built around a

straightforward, multitasking threaded code interpreter. Besides basic optimizations such as inline caching, no advanced compilation techniques have been used, and therefore the system cannot compete with Self in performance. Otherwise the implementation is fairly complete, featuring an iconic user interface that allows the programmer to maneuver in the object hierarchies, inspect and alter the variables of objects, add, remove and rename variables and operations, redefine and invoke operations, and define new objects on the fly. Additionally, tools are provided for visually analyzing the commonalities and differences between objects, as well as controlling the execution of a virtually unlimited number of quasi-concurrent tasks. In general, our experiences with Kevo have been very positive, and we are currently working on a next-generation implementation to make the system even more practical.

### **Benefits/Issues of Prototypes vs. Classes**

Prototypes have many benefits especially when exploratory programming is concerned. They are cognitively more lightweight than classes, support direct manipulation more naturally, avoid many modularity problems of class-based systems, and result in simpler and conceptually more elegant language implementations. Considering these benefits, it is appropriate to ask why haven't prototype-based systems received more widespread acceptance thus far. Besides the easy explanation - lack of maturity - the best answer to this question seems to be that there has been an over-emphasis on technical issues. The advocates of prototype-based systems have failed to advertise the conceptual benefits of prototypes, and emphasized technical curiosities instead. While features such as unanticipated sharing of data slots, dynamic inheritance, and prioritized parents are often useful, they are questionable and even dangerous in serious, large-scale software development. Compared to other issues, these features have received considerable attention in the literature, diverting the focus away from the real benefits of prototypes, and may have increased resistance against adopting prototype-based techniques more widely.

In summary, prototypes are beneficial, but a more

concept-oriented view of prototypes is still lacking, and this is exactly what we should be aiming at in the future. Delegation-free languages serve as a promising alternative in this respect.

### Illustrations.

Note: due to the use of virtual copying techniques, Kevo objects look very different depending on whether they are examined from the logical or implementation point of view. The figures below illustrate Kevo objects strictly from the logical viewpoint, and any optimizations performed by the actual Kevo system are ignored.

(1) *How does Kevo represent Cartesian points?*

x	3
y	4
+	how to add points

Figure 1: A Cartesian point in Kevo.

(2) *How would a programmer make a refinement of the Cartesian point abstraction to create a kind of colored point with x, y, and color attributes?*

Simply by making a copy of an existing Cartesian point object, and by adding the desired new variables and methods to the copy.

x	3
y	4
+	how to add points
color	"red"
becomeBluer	how to be more blue

Figure 2: A colored point in Kevo.

(3) *How would these colored points share colored point specific behavior (such as a method for the message "becomeBluer")?*

From the logical viewpoint, Kevo objects do not share any behavior. Rather, each object maintains its own copy of each method. In order to, e.g., redefine a certain method in all the ColoredPoint objects,

groupwise module operations are used.

x	3
y	4
+	how to add points
color	"red"
becomeBluer	how to be more blue

x	3
y	4
+	how to add points
color	"red"
becomeBluer	how to be more blue

Figure 3: Two colored points in Kevo.

---

*Antero Taivalsaari is a research engineer at Nokia Research Center, Helsinki, Finland. He finished his doctoral thesis in 1993, focusing on prototype-based languages and problems of traditional object-oriented systems. He is the designer of Kevo, a prototype-based language that he implemented during his stay as a visiting researcher in Canada in 1991-1992.*

---

### References.

- Taivalsaari, A., Kevo - a prototype-based object-oriented language based on concatenation and module operations. University of Victoria Technical Report DCS-197-1R, Victoria, B.C., Canada, June 1992
- Taivalsaari, A., Concatenation-based object-oriented programming in Kevo. Actes de la 2eme Conference sur la Representations Par Objets RPO'93 (La Grande Motte, France, June 17-18, 1993), Published by EC2, France, June 1993, pp.117-130
- Taivalsaari, A., A critical view of inheritance and reusability in object-oriented programming. Ph.D. thesis, Jyvaskyla Studies in Computer Science, Economics and Statistics 23, University of Jyvaskyla, Finland, December 1993, 276 pages (ISBN 951-34-0161-8).