Self

David Ungar
IBM Corporation
ungar@mac.com

Randall B. Smith
Sun Microsystems Laboratories
randall.smith@sun.com

Abstract

The years 1985 through 1995 saw the birth and development of the language Self, starting from its design by the authors at Xerox PARC, through first implementations by Ungar and his graduate students at Stanford University, and then with a larger team formed when the authors joined Sun Microsystems Laboratories in 1991. Self was designed to help programmers become more productive and creative by giving them a simple, pure, and powerful language, an implementation that combined ease of use with high performance, a user interface that off-loaded cognitive burden, and a programming environment that captured the malleability of a physical world of live objects. Accomplishing these goals required innovation in several areas: a simple yet powerful prototype-based object model for mainstream programming, many compilation techniques including customization, splitting, type prediction, polymorphic inline caches, adaptive optimization, and dynamic deoptimization, the application of cartoon animation to enhance the legibility of a dynamic graphical interface, an object-centered programming environment, and a user-interface construction framework that embodied a uniform use-mention distinction. Over the years, the project has published many papers and released four major versions of Self.

Although the Self project ended in 1995, its implementation, animation, user interface toolkit architecture, and even its prototype object model impact computer science today (2006). Java virtual machines for desktop and laptop computers have adopted Self's implementation techniques, many user interfaces incorporate cartoon animation, several popular systems have adopted similar interface frameworks, and the prototype object model can be found in some of today's languages, including JavaScript. Nevertheless, the vision we tried to capture in the unified whole has yet to be achieved.

Categories and Subject Descriptors: K.2 **[History of Computing]** Software – programming language design, programming environments, virtual machines; D.3.2 **[Programming Languages]** Object-Oriented Languages; D.3.3 **[Programming Languages]** Language Constructs and Features – data types and structures, polymorphism, inheritance; D.1.5 **[Object-oriented Programming];** D.1.7 **[Visual Programming];** D.2.6 [**Programming Environments]** Graphical environments, Integrated environments, Interactive environments; D2.2 **[Design Tools and Techniques]** User Interfaces, Evolutionary prototyping; D2.3 **[Coding Tools and Techniques]** Object-oriented programming; I.3.6 **[Computing Methodologies]** Computer Graphics – Interaction techniques

General Terms. Performance, Human Factors, Languages

©2007 ACM 978-1-59593-766-7/2007/06-ART9 $5.00
DOI 10.1145/1238844.1238853
http://doi.acm.org/10.1145/1238844.1238853

Keywords: dynamic language; object-oriented language; Self; Morphic; dynamic optimization; virtual machine; adaptive optimization; cartoon animation; programming environment; exploratory programming; history of programming languages; prototype-based programming language

1. Introduction

In 1986, Randall Smith and David Ungar at Xerox PARC began to design a pure object-oriented, dynamic programming language based on prototypes called Self [US87, SU95]. Inspired by Smith's Alternate Reality Kit [Smi87] and their years of working with Smalltalk [GR83], they wanted to improve upon Smalltalk by increasing both expressive power and simplicity, while obtaining a more concrete feel. A Self implementation team was formed, first by the addition of Ungar's graduate students at Stanford, and then by the addition of research staff when the group moved to Sun Labs in 1991. By 1995, Self had been through four major system releases.

Self's simplicity and uniformity, particularly in its use of message passing for all computation, meant that a new approach to virtual machine design would be required for reasonable performance. The Self group made several advances in VM technology that could be applied to many if not most object-oriented languages. The group also created an innovative programming environment that could host multiple distributed users, and pioneered novel graphical user interface techniques, many of which are only now seeing commercial adoption.

Although the present paper has just two authors, the Self project was a group effort. The other members' dedication, hard work and brilliance made Self what it is. Those people are: Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, Elgin Lee, John Maloney, and Mario Wolczko. In addition, our experience was deeply enriched by Ole Lehrmann Madsen, who spent a year with us as a visiting professor. We also appreciate the efforts of Jecel Assumpcao who, over the years, has maintained a web site and discussion list for Self. We are indebted to the institutions that supported and hosted the Self project: Sun Microsystems, Stanford University, and Xerox PARC. While at Stanford, the Self project was generously supported by the National Science Foundation Presidential Young Investigator Grant #CCR-8657631, and by IBM, Texas Instruments, NCR, Tandem Computers, and Apple Computer.

Work on the project officially ceased in June 1995, although the language can still be downloaded and used by anyone with the requisite computing environment. But the ideas in Self can readily be found elsewhere: ironically, the implementation techniques developed for Self thrive today in almost every desktop virtual machine for Java™, a language much more conservative in design. We feel deeply rewarded that some researchers have understood and even cherished the Self vision, and we dedicate this paper to them.

This paper has four general parts: history, a description of Self and its evolution, a summary of its impact, and a retrospective. We begin with our personal and professional histories before we met in 1986, and summarize the state of object-oriented lan-

guages at that time with special emphasis on Smalltalk, as it was an enormous influence. We also discuss the context at Xerox PARC during the period leading up to the design of Self, and describe Smith's Alternate Reality Kit, which served as inspiration for some of Self's key ideas. This is followed by a description of Self that emphasizes our thoughts at the time. Moving our viewpoint into the present, we assess the impact of the system and reflect upon what we might have done differently. Finally, we sum up our thoughts on Self and examine what has become of each of the participants (as of 2006).

2. Before Self

Nothing comes from nothing; to understand Self's roots, it helps to look at what PARC and Smith and Ungar were doing earlier.

2.1. Smith Before Self

Perhaps because his father was a liberal-minded minister, or perhaps because he was also the son of a teacher, Smith has always been fascinated by questions at the boundaries of human knowledge, such as "What is going on to make the universe like this?" Thus it was perhaps natural for him to enter the University of California at Davis as a physics major. But along the way, he discovered computers: he so looked forward to his first programming course that on the opening day he gave his instructor a completed program and begged him to enable student accounts so that this excited student could submit his deck of cards (which calculated the friction in yo-yo strings). Computing felt more open and creative than physics: unlike the physical universe, which is a particular way, the computer is a blank canvas upon which programmers write their *own* laws of physics. There really was no major in computing in those days; the computer was perceived as a big, expensive tool, and Smith happily stuck with his physics curriculum, getting his PhD at UCSD in 1981. He then returned to his undergraduate alma mater as a lecturer in the UC Davis Physics Department.

One of the mysteries of physics is that a few simple laws can explain a wide range of phenomena. Smith enjoyed teaching, and was always impressed that he could derive six months of basic physics lectures from F=ma. Much progress in physics seems to be about finding theories with increasing explanatory power that at the same time simplify the underlying model. The notion that simplicity equated to explanatory power would later manifest itself in his work designing Self.

The draw of computing inevitably won him over, and Smith stopped chasing tenure in Physics, taking his young family to Silicon Valley in 1983 so he could work at Atari Research Labs, then directed by Alan Kay. During that year a rather spectacular financial implosion took out much of Atari. Smith was one of only a few remaining research staff members when the company was sold in 1984 to interests who felt no need for research. Atari Labs were closed and Smith joined the Smalltalk group at Xerox PARC.

2.2. Ungar Before Self

When Ungar was about six and struggling to tighten a horse's saddle girth, his father would say "Think about the physics of it." What stuck was the significance of how one chose how to think about a problem. Sometime in his early teens, Ungar was inspired by the simultaneously paradoxical and logical power of Special Relativity. Still later, experience with APL in high school and college kindled his enthusiasm for dynamic languages. Then, as an undergraduate at Washington University, St. Louis, he designed a simple programming language.

In 1980, Ungar went to Berkeley to pursue a Ph.D in VLSI design. Eventually, he got a research assistantship working on VLSI design tools for Prof. John Ousterhout, and was also taking a class on the same topic. At that time, the only way for a Berkeley student to use Smalltalk was to make the hour-plus drive down to Xerox PARC. Dan Halbert, also in the VLSI class, was making that trip regularly (in Butler Lampson's car) to use Smalltalk for his doctoral research on programming by demonstration. Halbert gave a talk in the VLSI class on how well Smalltalk would support VLSI design by facilitating mixed-mode simulation. In a mixed-mode system, some blocks would be simulated at a high level, others at a low level, and Smalltalk's dynamic type system and message-passing semantics would make it easy to mix and match. This chain of events kindled Ungar's interest in Smalltalk.

Dan Halbert took Ungar down to PARC several times in late 1980 and demonstrated Smalltalk. After seeing Smalltalk's reactive graphical environment and powerful, dynamic language, Ungar was hooked. He yearned to solve real problems in Smalltalk without the long drive. He obtained an experimental Smalltalk interpreter, written at HP, but it ran too slowly on Berkeley's VAX 11/780. This frustration would completely change the focus of Ungar's dissertation work, redirecting him from VLSI to virtual machines (see section 2.4.4). In the summer of 1985, Ungar left Berkeley and began teaching at Stanford as an assistant professor. He completed his dissertation that academic year, and received his PhD in the spring of 1986.

2.3. Object-Oriented (and Other) Programming Languages Before Self

The design of Self was strongly influenced by what we knew of existing languages and systems. Here are a few languages that were in Ungar's mind as a result of his lectures at Stanford.

Simula was the first object-oriented language per se. In its first published description, Dahl and Nygaard stated that its most important new concept was quasi-parallel processing [DN66]. Its designers were trying to use computers to simulate systems with discrete events. A key insight was the realization that the same description could be used both for modeling and for simulation. They extended Algol 60 by adding "processes" (what would now be called coroutines) and an ordered set feature. A Simula process grouped related data and code together, and this grouping came to be thought of as object-oriented programming. Multiple instances of a process could be created, and "elements" were references to processes. Simula's designers felt it was important to keep the number of constructs small by unifying related concepts. Although Simula's influence on Self was profound, it was indirect: Simula famously inspired Alan Kay, who in the 1970s led the Smalltalk group at the Learning Research Laboratory in Xerox PARC.

Parnas [Parn72] explained key principles of object-oriented programming without ever using the work "object." He convincingly showed that invariants could be better isolated by grouping related code and data together, than by a pure subroutine-based factoring.

Hoare argued convincingly for simplicity in language design [Hoar73]. This paper was one of Ungar's favorites and influenced him to keep the Self language small. It is interesting in view of Self's lack of widespread adoption that this aesthetic can also be found in APL, LISP, and Smalltalk, but not in the very popular object-oriented programming languages C++ and Java.

C++ [Strou86] was created by Bjarne Stroustroup, who had studied with the Simula group in Scandinavia but had then joined the Unix group at Bell Laboratories. Stroustroup wanted to bring the benefits of object-orientation and data abstraction to a community accustomed to programming in C, a language frequently considered a high-level assembler. Consequently, C++ was designed as a superset of C, adding (among other things) classes, inheritance, and methods. (C++ nomenclature uses "derived class" for subclass and "virtual function" for method.) To avoid incompatibility with C at the source or linker levels, and to avoid adding overhead to programs that did not use the new features, C++ initially omitted garbage collection, generic types, exceptions, multiple inheritance, support for concurrency, and an integrated programming environment (some of these features made it into later versions of the language). As we designed and built Self in 1987, C++'s complicated and non-interpretive nature prevented us from being influenced by its language design. However, its efficiency and support for some object orientation led Ungar and the students to adopt it later as an implementation language and performance benchmark for Self; we built the Self virtual machine in C++, and aimed to have applications written in Self match the performance of those written in (optimized) C++.

APL [Iver79] was an interactive time-shared system that let its users write programs very quickly. Although not object-oriented, it exerted a strong influence on both Smalltalk and Self. Ingalls has reported its influence on Smalltalk [Inga81], and APL profoundly affected Ungar's experience of computing. In 1969, Ungar had entered the Albert Einstein Senior High School in Kensington, Maryland, one of only three in the country with an experimental IBM/1130 time-sharing system. Every Friday afternoon, students were allowed to program it in APL, and this was Ungar's first programming experience. Though Ungar didn't know it at the time, APL differed from most of its contemporaries: it was dynamically typed in that any variable could hold a scalar, vector, or matrix of numbers or characters. APL's built-in (and user-defined) functions were polymorphic over this range of types. It even had operators: higher-order functions that were parameterized by functions. The APL user experienced a live workspace of data and program and could try things out and get immediate feedback. Ungar sorely missed this combination of dynamic typing, polymorphism, and interpretive feel when he went on to learn such mainstream languages as FORTRAN and PL/I.

Ungar's affection for APL led to a college experience that had a profound impact. As a freshman at Washington University, St. Louis, in 1972, Ungar was given an assignment to write an assembler and emulator for a simple, zero-address computer. The input was to consist of instructions such as:

```
push 1
push 2
add
```

The output was to be the state of the simulated machine after running the given assembly program. His classmates went upstairs and, in the keypunch room (which Ungar recalls as always baking in the St. Louis heat) began punching what eventually became thick card decks containing PL/I programs to be run on the school's IBM System/360. His classmates built lexers, parsers, assemblers, and emulators in programs about 1000 lines long; many of his classmates could not complete their work in the time allowed.

Meanwhile, Ungar's fascination with APL had led to an arrangement permitting him to use the Scientific Time Sharing Corporation's APL system gratis after hours. He realized that with a few syntactic transformations (such as inserting a colon after every label), the assembler program to be executed became a valid APL program. Reveling in APL's expressiveness, he wrote each transformation as a single, concise line of code. Then he wrote one-line APL functions for each opcode to be simulated, such as:

```
∇ADD X
PUSH POP + X
∇
```

Finally came the line of APL that told the system to run the transformed input program as an APL program. The whole program only took 23 lines of APL! This seemed too easy, but Ungar was unwilling to put in the hours of painstaking work in the keypunch sweatbox, so he turned in his page of APL and hoped he would not flunk. When the professor rewarded this unorthodox approach with an A, Ungar learned a lesson about the power of dynamic languages that stayed with him forever.

In retrospect, any student could have done something similar in PL/I by using JCL (IBM System/360 Job Control Language) to transform the program to PL/I and then running it through the compiler. But none did, perhaps because PL/I's non-interpretive nature blinkered its users. Ungar always missed the productivity of APL and was drawn to Smalltalk not only for its conceptual elegance, but also because it was the only other language he knew that let him build working programs as quickly as in the good old days of APL. The design of Self was also influenced by APL; after all, APL had no such thing as classes: arrays were created either ab initio or by copying other arrays, just as objects are in Self.

2.4. Smalltalk

Smalltalk [Inga81] was the most immediate linguistic influence on Self. Smalltalk's synthesis of language design, implementation technology, user interface innovation and programming environment produced a highly productive system for exploratory programming. Unlike some programming systems, Smalltalk had a principled design. Ingalls enumerated the principles in [Inga81], and many of them had made a strong impression on Ungar at UC Berkeley. We embraced these values as we worked on Self. Table 2 on page 39 enumerates these principles and compares their realizations in Smalltalk, the Alternate Reality Kit (described in section 2.6), and Self.

2.4.1. Smalltalk Language

It is truly humbling to read in HOPL II about Alan Kay's approach to the invention of Smalltalk [Kay93]. Starting from notions of computation that were miles away from objects, Kay tells of years of work that produced a pure object-oriented environment including an interactive, reactive user interface and programming environment. Smalltalk introduced the concept (and reality) of a world of interacting objects, and we sometimes feel that Self merely distilled Smalltalk to its essentials (although we hope that Self made contributions of its own).

Smalltalk-76 introduced the concept of a purely dynamically typed object-oriented language. A Smalltalk computation consists solely of objects sending messages to other objects. To use an object, one sends a *message* containing the name of the desired operation and zero or more arguments, which are also objects. The object finds a method whose name matches the message, runs the method's code, and returns an object as a result. Thus, the process that the reader may know as "method

invocation" in Java is called "message sending" in Smalltalk. The "class" is central to this story: every object is an "instance" of some class and must have been created by that class. A window on the screen is an instance of class `Window`, `17` is an instance of class `Integer`, and so on. Classes are themselves objects, and classes are special in that they hold the methods with which possessed by each of its instances (the *instance* variables). Also, a class typically specifies a superclass, and objects created from the class also possess any variables and methods defined in the superclass, the super-superclass, etc. Thus, all objects belonging to a given class possess the same set of variables and methods. Variables are dynamically typed, in that any variable can refer to any object, but that object had better respond to all the messages sent to it, or there will be a runtime error. Methods are selected at run time based on the class of the receiver.

In addition to the two pseudo-variables "self," denoting the current receiver, and "super," denoting the current receiver but bypassing method lookup in its class, there are six kinds of genuine variables: global variables, pool variables which pertain to every instance of a class in a set of classes, class variables which pertain to every instance of and to a given class, instance variables which pertain to a single instance, temporary variables of a method, and arguments. An instance variable can be accessed only by a method invocation on its holder, while temporaries and arguments pertain only to the current method invocation. (Arguments differ from the other kinds of variables in being read-only.)

By the time we had started working with Smalltalk, it had evolved from Smalltalk-76 to Smalltalk-80. This new version cleaned up several aspects of the language but also introduced a complicating generalization that would later motivate us to eliminate classes entirely. In Smalltalk-72, classes were not objects, but, according to Dan Ingalls, as the Smalltalk group "experienced the liveliness" of that system, they realized it would be better to make classes be objects. So, in Smalltalk-76, all classes were objects and each class was an instance of class Class, including class Class itself. That meant each class had the same behavior, because class Class held the common behavior for all classes. In Smalltalk-80, each class was free to have its *own* behavior, a design decision that brought a certain utility and also seemed in keeping with the first-class representation of classes as objects. However, it also meant that a class had to be an instance of some unique class to hold that behavior. The class of the class was called the metaclass. Of course, if the metaclass were to have its own behavior, it would require a meta-metaclass to hold it, and thus Smalltalk-80 presented the programmer with a somewhat complex and potentially infinite world of objects that resulted from elaborating the "instance of" dimension in the language. Smalltalk-80 makes this meta-regress finite by using a loop structure at the top of the meta-hierarchy, but many users had a lot of trouble understanding this. Although this could be seen as a poor design decision in going from Smalltalk-76 to Smalltalk-80, it might be argued that this is a problem one is forced to confront whenever classes are fully promoted to object status. Either way, this conceptually infinite meta-regress and the bafflement it caused new Smalltalk-80 programmers gave us a strong push to eliminate classes when we designed Self. As we look back at Smalltalk-80 in 2006, it seems to us that, given the desire for a live and uniform system, the instance-class separation sprouted into a tangle of conceptually infinite metaclasses that would seem inevitable if an entity cannot contain its own description.

2.4.2. Smalltalk Programming Environment

In addition to learning the Smalltalk language, the user also had to master a programming environment that came with its own organizational concepts. The Smalltalk programming environment was astounding for its time—it introduced overlapping windows and pop-up menus, for example—and exerted a strong influence on the Self project.

The programming environment used by Smalltalk programmers centers on the browser, inspector, and debugger. There are a few other tools (e.g., a method-by-method change management tool), but these three deliver much of what the programmer needs, and even these three share common sub-components. Hence, even in the Smalltalk programming environment, there was a sense of simplicity. Ironically, even though simple, the environment delivered features we miss when using some modern IDEs for languages such as Java. For example, one Java IDE in common use contains several times the number of menu items available in the Smalltalk tools, yet there is no way to browse a complete class hierarchy.

The "learnability" aspect of the Smalltalk programming environment was a key concern of the Smalltalk group when Smith joined it in 1984. The PARC Smalltalk group had descended from the Learning Research Group, which focused on the educational value of programming systems. Many in the group were aware that the Smalltalk-80 system was somewhat more difficult to pick up than they had hoped in the earlier days, and saw that the programming environment, being what the user sees, must have been largely responsible. Alan Kay had envisioned the Dynabook as a medium in which children could explore and create, and had conceived of Smalltalk as the language of the Dynabook. Hence one sensed a kind of subtext floating in the halls like a plaintive, small voice: "What about the children?" Although Smalltalk had started off as part of this vision, that vision had somehow become supplanted by another: creating the ultimate programmer's toolkit.

The browser, the central tool for the Smalltalk programmer, was the result of years of enhancement and redesign. It is fair to say it does an excellent job of enabling users to write their Smalltalk code, and it has served as a model for many of today's IDEs (though some bear a closer resemblance than others). The browsers feature small titled panes for selecting classes from within a category and methods within a class, plus a larger, central text pane for editing code. However, by the time it was released in Smalltalk-80, the browser had come to present a system view significantly removed from the underlying execution story of objects with references to one another, sending messages to each other. The standard Smalltalk-80 browser presents the user with notions such as categories (groups of classes), and protocols (bundles of methods), neither of which has a direct, first-class role in the Smalltalk runtime semantics of the program. For example, before a programmer can try creating even the simplest class, she must not only give the class a name, which may seem logical, but also decide on a System Category for the class, even though that category has nothing to do with the class's behavior. Furthermore, the standard Smalltalk-80 browser features a prominent and important "instance/class" switch that selects either methods in the selected class or methods in the selected class's class (the metaclass). Recall that a class, since it is an object, is itself an instance of some class, which would hold methods for how the class behaves, such as instantiation, access to variables shared amongst all instances, and the like. But what about the class's class's class? And the class's class's class's class, and so on? One finds no extra switch positions for presenting those methods. Furthermore, if the pur-

pose of the browser is to show the methods in any class, why is the switch even needed?

At a deeper level, it was obvious to us that the use of tools, as great as they were, tended to pull one away from the sense of object. That is, the inspector on a Smalltalk hash table was clearly not itself the hash table. This was a natural outgrowth of the now famous Model View Controller (MVC) paradigm, invented by the PARC Smalltalk group as the framework for user interfaces. Under the MVC scheme, the view and controller were explicitly separate objects that let the user see and interact with the model. This was an elegant framework, but we questioned it. If the metaphor was direct manipulation of objects, then we thought s that the UI and programming environment should give a sense that what one saw on looking at a hashtable actually *was* the hashtable. In section 2.6 on the Alternate Reality Kit and in sections 4.3 and 5.3 on user interface designs for Self, we discuss our approaches to providing a greater sense of direct object experience.

2.4.3. Smalltalk User Interface

To set the stage for the Smalltalk user interface, we first describe the state of user interface work when we met Smalltalk. Overlapping windows were first used in Smalltalk, and the early Smalltalk screens would look familiar even today. In those days at PARC and Stanford it was not uncommon to argue over a tiled-windows versus messy-desktop paradigm, though the latter ultimately came to dominate. HCI classes would discuss direct manipulation as though it were a somewhat novel concept, and everyday computer users were not clear whether the mouse-pointer window paradigm had real staying power, as it seemed to pander to the novice. In fact, the acronym Window Icon Mouse Pointer (WIMP) was often used derisively by those who preferred the glass teletype. Smith recalls that in some of his user studies it would take subjects roughly 30 minutes to get used to the mouse.

At the time Smalltalk was being designed, each application had its user interface hard-wired so that its implementation was inaccessible to the user; the interface could neither be dissected nor modified. Smalltalk was a breed apart: its user interface was itself just another Smalltalk program that ran in the same virtual machine as the programmer's own applications. Thus, by pointing the mouse at window W and hitting "control-C" to invoke the Smalltalk debugger, one could find oneself browsing the stack of an interrupted thread that handled UI tasks related to window W. One could then use this debugger to modify the code and resume execution to see the effects of the changes. Most of us hoped that something like that would eventually take over the world of desktop computing, but today that dream seems all but dead. There is no way to get into your word processor and modify it as it runs, though in those days, that would have been routine for the curious Smalltalk user.

At PARC in the early 1980s, researchers could sense how user interface innovations created down the hall were sweeping through the entire world. Silicon Valley researchers just assumed that the computer desktop UI was still fertile ground for innovation, feeling that the basic notions of direct manipulation would probably stick, so that invention would most fruitfully occur within that broad paradigm. We were smitten with direct manipulation and wanted to push it to an extreme. In particular, we were fascinated by the notion that the computer presents the user with a synthetic world of objects. It felt to us that the screens we saw in those days hosted flat, 2D, static pictures of objects. We wanted to feel that those were real objects, not pictures of them. This desire for "really direct manipulation" consciously motivated much of our work and would show up first in the Alternate Reality Kit, as described in section 2.6, and ultimately in Self.

2.4.4. Implementation Technology for Smalltalk and Other Interpreted Languages

Although much work had been done to optimize LISP systems that ran on stock hardware, Ungar was not very aware of that work when the Self system was built. The contexts are so different and the problems differ enough that it is hard to say what would have been changed had he known more about LISP implementations. Ungar was familiar with the LISP machine [SS79], but as it was a special-purpose CISC machine for LISP, he felt it would not be relevant to efficient implementation of Self on a RISC.

In contrast, it is quite likely that Ungar, although not consciously aware of it at the time, was inspired by APL when he came up with the technique of customization for Self (section 4.1.1). As mentioned above, any variable in APL can hold a scalar, a vector or a matrix at any time, and the APL operations (such as addition) perform computation that is determined upon each invocation. For example, the APL expression **A + B** executed three times in a loop could: add two scalars on its first evaluation, add a scalar to each element of a matrix on its second evaluation, and add two matrices element-by-element on its third. Although the computation done for a given operation could vary, the designers of the APL\3000 system [John79] observed that it was often the same as before. They exploited this constancy by using the runtime information to compile specialized code for expressions that would be reused if possible, thus saving execution time. If the data changed and invalidated code, it was thrown away and regenerated. Ungar had read about this technique years before implementing Self, and it probably inspired the idea that the system could use different compiled versions of the same source code, as long as the tricks remained invisible to the user.

When Smalltalk was developed in the early to mid 1970s, commercially available personal computers lacked the horsepower to run it. Smalltalk relied on microcode interpreters running on expensive, custom-built research machines. Developed in house at Xerox PARC, these machines (called Altos [Tha86], later supplanted by Dolphins, and then Dorados) were the precursors of 1990s personal computers. The Dorado was the gold standard: it was fast for its time (70ns cycle time), but had to be housed in a separate air-conditioned room; a long cable supplied video to the user's office. These expensive and exotic machines allowed the PARC researchers to live in a world of relatively abundant cycles, personal computers, and bitmapped displays years before the rest of us.

Even with this exotic hardware, Smalltalk's implementers at PARC had to resort to compromises that increased performance at the cost of flexibility. For example: arithmetic, identity comparison, and some control structures were compiled to dedicated bytecodes whose semantics were hard-wired into the virtual machine. Thus, the source-level definitions of these messages were ignored. A programmer, seeing the definitions, might think that these operations were malleable, edit the definition and accept it, yet nothing would change. For example, Smith once changed the definition of the if-then-else message to accept "maybe" as the result of comparisons involving infinity. He was surprised when, though the system displayed his new definition, it kept behaving in accordance with the old one. And Mario Wolczko, who taught Smalltalk before joining the Self group, once had a student create a subclass of Boolean, only to discover

that it did not work. The Self system was built later and enjoyed the luxury of more powerful hardware. Thus, it could exploit dynamic compilation to get performance without sacrificing this type of generality (section 5.1).

In 1981, Ungar built his own Smalltalk system, Berkeley Smalltalk (BS). Its first incarnation followed the "blue book" [GR83], which used 16-bit object pointers, an object table, and reference counting.[1] This kind of object pointer is known as an indirect pointer, because instead of pointing directly to the referenced object, it points to an object table entry that in turn points to the referenced object. This indirection doubles the number of memory accesses required to follow the pointer and therefore slows the system. L. Peter Deutsch, an expert on dynamic language implementations who had worked on the Dorado Smalltalk virtual machine [Deu83] at Xerox PARC, began a series of weekly tutoring sessions on Smalltalk virtual machines with Ungar. Deutsch had just returned from a visit to MIT, where he was probably inspired by David Moon to suggest that Ungar build a system that handled new objects differently from old ones. After obtaining promising results from trace-driven simulations, Ungar rewrote Berkeley Smalltalk to use a simple, two-generation collection algorithm that he called Generation Scavenging [Ung84]. Ungar realized that, in addition to directly increasing performance by reducing the time spent on reclamation, this collector would indirectly increase performance by making it possible to eliminate the object table. This optimization was possible because Generation Scavenging moved all the new objects in the same pass that found all pointers to new objects and could thus use forwarding pointers. In addition, since most objects were reclaimed when new, old objects were allocated so rarely that it was reasonable to stop the mutator for an old-space reclamation and compaction and thus again use forwarding pointers. The resulting system was the first Smalltalk virtual machine with 32-bit pointers and the first with generational garbage collection. Ungar told Deutsch about his excitement at removing the overhead of pointer indirection involved with the object table. Deutsch didn't share the excitement; he estimated the speedup would be less than 1.7. Ungar disagreed, and talked Deutsch into betting a dinner on it. So, when the new algorithm was running, Ungar tuned and tuned till it was 1.73 times faster than the previous tuned version of Berkeley Smalltalk: Deutsch treated Ungar to a very fine dinner in a Berkeley restaurant. As of this writing (2006), almost all desktop- and server-based object-oriented virtual machines use direct pointers, thanks perhaps in part to Deutsch's willingness to make a bet and graduate student Ungar's desire to prove himself to Deutsch and claim a free meal.

At Berkeley, Deutsch and Ungar continued their discussions. When the Sun-1 came out, Deutsch decided to build a system based on dynamic compilation to native code and inline caching that would let him run Smalltalk at home [DS84]. Deutsch and Schiffman's PS ("Peter's Smalltalk") system was in many ways the precursor of all dynamically compiling object-oriented virtual machines today. After Ungar spent a few months trying to optimize his interpreter and receiving only diminishing returns, he realized that only a compilation-based virtual machine (such as PS) could yield good performance. It was this experience that led Ungar to rely on compilation techniques for the Self system.

1. The "blue book" (our affectionate name for the first book on Smalltalk) was the authoritative guide (since it was the only one) and contained the code (in Smalltalk) for a reference implementation. Ungar recalls that Dave Robson used to call this Smalltalk-in-Smalltalk as "the slowest Smalltalk virtual machine in the world."

Meanwhile, during the 1980-1981 academic year, Berkeley professor David Patterson was finishing up his Berkeley RISC project, demonstrating that a simple instruction-set architecture with register windows could run C programs very effectively. By eliminating the time spent to save and restore registers on most subroutine calls, the RISC architecture could execute the calls very quickly. Ungar and others at Berkeley saw a match between RISC's strengths and the demands of a Smalltalk implementation. Patterson saw this too, and in collaboration with Prof. David Hodges started the Smalltalk on a RISC (SOAR) [PKB86, Ung87] project. Based on a simple RISC machine, SOAR added some features to support Smalltalk and relied on a simple ahead-of-time compiler [BSUH87] to attain 70ns-Dorado-level performance on a (simulated) 330ns microprocessor. The rack-sized Dorado ran at a clock speed of 14 MHz, while the (simulated) chip-sized SOAR microprocessor ran Smalltalk just as fast with a mere 3 MHz clock. (Today, in 2006, commercial microprocessors run at clock speeds about a thousand times faster than SOAR's, and have no trouble at all with interpreted Smalltalk.) This system was another proof that compilation could hold the key to dynamic object-oriented performance.

For his doctoral research, Ungar helped design the instruction set, wrote the runtime system (in SOAR assembler) and ran benchmarks. Then he removed one architectural feature at a time and substituted a software solution so as to isolate the contribution of each feature. One of the most important lessons Ungar learned from the project was that almost all the system's "clever" ideas had negligible benefit. In fact, the vast bulk of speed improvements accrued from only a few ideas, such as compilation and register windows. In his dissertation, he called the temptation to add ineffective complexity "The Architect's Trap." A few years later, in 1988 and 1989, Ungar had to relearn this lesson in the evolution of the Self language, as described in section 4.2.

In 1988, after PS had been completed and Ungar had graduated, the Smalltalk group at Xerox PARC spun off a startup company called ParcPlace Systems to commercialize Smalltalk. For their ObjectWorks product, they built a Smalltalk virtual machine called HPS. Extending the ideas in PS, HPS used Deutsch's dynamic translation technique and a clever multiple-representation scheme for activation records. Unlike PS, it was written in C, not assembler, and employed a hybrid system for automatic storage reclamation. The latter, on which Ungar consulted, comprised a generation scavenger for new objects and an incremental, interruptible mark-sweep collector for the old objects. An object table permitted incremental compaction of the old objects. When it was built, around 1988, it was probably the fastest Smalltalk virtual machine, and its success with dynamic translation served as an inspiration.

2.5. Xerox PARC in the 1980s

By the early 1980s Xerox PARC had established itself as the inventor of much of the modern desktop computer. At a time when most of us in the outside world were just becoming comfortable with time-shared screen editors running on character-mapped displays that showed 25 lines of 80 fixed-width characters, each PARC engineer had his own personal networked computer with keyboard, mouse, and a bit-mapped display showing multiple windows with menus and icons. They authored WYSIWYG documents, sent them to laser printers, e-mailed them to each other, and stored them on file servers. All this has now, of course, become commonplace.

But another part of the vision held by many at PARC was slower to materialize in the outside world, and in many ways never did. That dream depicted users as masters of their own computers, able to modify applications in arbitrary ways and even to evolve them into entirely new software. This vision of "everyman as programmer" was part of Alan Kay's story that motivated the work of the PARC Smalltalk group [Kay93]. This group looked to the idea of dynamic, object-oriented programming as the underlying mechanism that would make the elements of the computer most sensibly manifest. Kay's group had a tradition of attending to the user interface (the Smalltalk group had introduced the idea of overlapping windows) and of focusing on education. Kay and his group felt that students should be creators in a powerful and flexible medium and that a dynamic object-oriented language was the key enabler. Kay's group had developed several versions of the Smalltalk language: Smalltalk-72, Smalltalk-76, and finally Smalltalk-80.

Alan Kay left PARC in 1982 but his group carried on under the leadership of Adele Goldberg. It had hosted a few researchers who created more visual programming environments such as Pygmalion [FG93], Rehearsal World [Smi93], and ThingLab [BD81]. These environments were written in Smalltalk but were themselves essentially visual programming languages, with somewhat different semantics from Smalltalk itself. For example, ThingLab took the user's graphically specified constraints to generate code that maintained those constraints. Perhaps because it was a graphical environment, or perhaps because of SketchPad's inspiration [Suth63], ThingLab's designer, Alan Borning, presented users with a set of "prototype" objects to copy and modify in their work. The copying of a prototype became recognized as being a little deeper than it might seem at first glance; it offered an alternative to instantiating a class that felt much more concrete. This distinction may not seem very compelling in a compile-first/run-later environment such as Java or C, but in Smalltalk, where one is always immersed in a sea of running objects even while writing new code, the advantages of working with concrete instances was more apparent. Perhaps from similar intuitions, others had been exploring the idea of adding "exemplars" to Smalltalk [LTP86], instances that accompany the class hierarchy and serve as tangible representatives of the classes.

When Smith joined PARC in 1984, he would add to this list of visual programming systems written in Smalltalk by creating the Alternate Reality Kit, or ARK. Like ThingLab and SketchPad, ARK would be a construction environment based on prototypes.

2.6. ARK, The Alternate Reality Kit

Smith had always loved teaching physics. When he was lecturing in the UC Davis Physics Department, he felt the students became somewhat disconnected from the material when he covered topics such as relativity and quantum mechanics, because few if any demonstrations were available to provide a tangible connection to relevant physical experience. When he left academia for Silicon Valley research life at Atari Corporate Research in 1983, Smith started to investigate how a simulation environment might provide a tangible experience for learning relativity by letting students see what the world would be like if the speed of light were, say, 5 mph. He hoped that someday students playing in such a simulated world would obtain such an automatic and intuitive understanding of relativity that they would laugh off mental puzzlers such as the twin paradox as a trivial misunderstanding. When he joined PARC, Smith began to think about generalizing on his previous work. Smith began to realize that changing the speed of light to 5 mph was just an instance of a more powerful idea: a simulation can provide a way for students to experience how the world is not, as well as how it is. In the real world, we are stuck with the laws of physics we have been given. In a simulation, we can see what role a law plays by watching what happens when we change it. Smith set to work to create an environment making it possible to create such simulations; because of this emphasis on changing the nature of reality, Smith called the system the Alternate Reality Kit. Smalltalk's ability to change a program as it ran was the key to granting the ARK user the power to change physical law in an active universe.

The Alternate Reality Kit, implemented in Smalltalk-80, emerged as an open-feeling kit of parts, featuring lots of motion and subtly animated icons (see Figure 1). A user could grab objects, throw them around, and modify them in arbitrary ways through messages sent by buttons. For its time, the system had unusual, "realistically" rendered objects. The lighting model implied a third dimension, and most objects were intentionally drawn without an outline to remind the viewer of real-world objects, which also do not generally have outlines. A drop shadow for objects lifted "out of the plane" also provided a sense of a third dimension. Having only one-bit-deep displays meant all this had to be achieved with stipple patterns, requiring careful rendering and a little more screen real estate than might otherwise be required. This look would later be carried into the Self user interface. Today, drop shadows and pseudo-3D user interface elements with highlights and beveled edges are commonplace, and we are seeing more animation as well. ARK may have been the first system to include many of these ideas.

ARK also foreshadowed Self's elimination of the class concept by sweeping Smalltalk's classes under the rug. For example, it featured a "message menu" that the user could "pop up" directly on any display object and contained a list of every Smalltalk message to which the display object could respond. Selecting from the menu created a button that was attached to the object that could be pressed to send the message, then discarded if not needed, dropped onto other objects for use there, set aside, or simply left in place for future use. If the message required parameters, the button had retractable plugs that could be drawn out and dropped on the parameter objects. If the message returned a result, that object was made into a display object and popped up onto the screen. To create the menu of available messages, the underlying Smalltalk system started with the class of the display object and simply scanned up the class hierarchy, collecting the methods from each class as it went. As a result, the presence of a class was effectively hidden from the ARK user, even though classes were of course being used under the covers.

Furthermore, in ARK, any object could be modified and new kinds of state and behavior introduced within the simulation while everything was running. Unlike Smalltalk, ARK enabled the user to add an instance variable directly to an object, simultaneously specifying the name of the variable and its value. Because ARK was a Smalltalk program, making a new kind of object was implemented at the Smalltalk level as three steps: 1] make a new subclass specifying the new instance variable, 2] instantiate that class to make a new object O, and 3] replace the on-screen instance with O. In other words, the role of the Smalltalk class was again being hidden. The class was implementing something that in ARK felt not only more tangible but more to the point: working directly with instances.

Thus, even though ARK users worked directly with instances, they had full access to sending Smalltalk messages and making new kinds of objects. The notion of making a new kind of object simply by modifying an existing instance foreshadowed the pro-

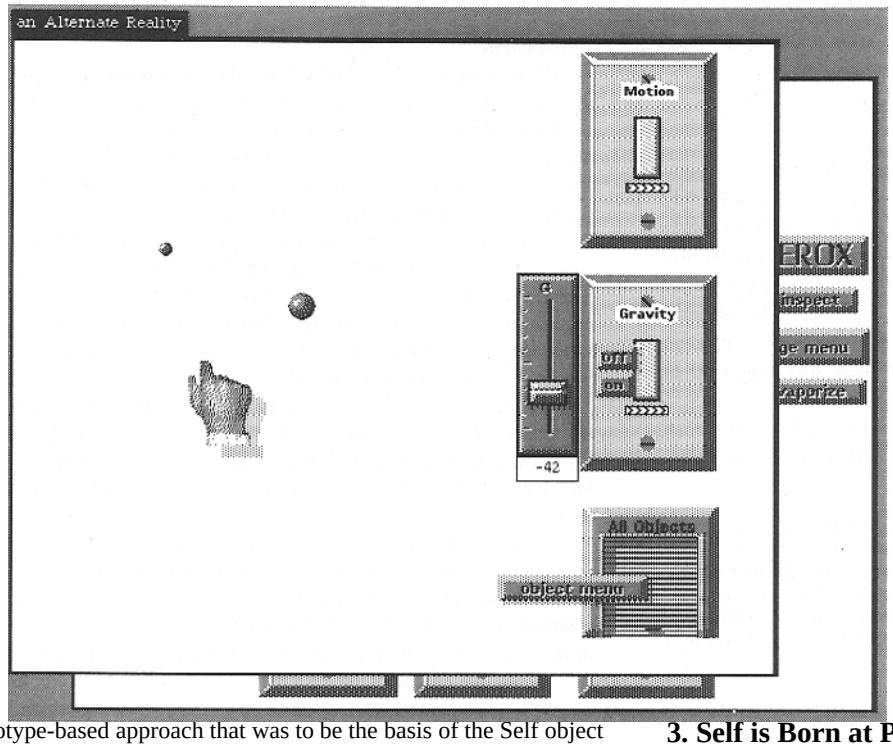


Figure 1. The Alternate Reality Kit (ARK), an interactive simulation environment that was also a visual programming system. Some ideas in ARK influenced the design of Self. The screen shows several buttons, some attached to objects. New objects could be made by a copy-and-modify process, and any new state in the new object was accessed through new buttons. This foreshadowed Self's use of prototypes and the way Self entirely encapsulates state behind a message-passing mechanism. ARK also had a feel of being a live world of moving, active objects that was unusual for its time and influenced the programming environment, as well as in some sense the deeper semantics and overall goals, of Self.

totype-based approach that was to be the basis of the Self object model. In ARK, it seemed unnecessary to even think about a class, and Self would have none.

A final aspect of ARK influenced the later design of Self (section 3). When a new instance variable was created for an object in ARK, it seemed natural to have the system automatically create "setter" and "getter" methods that would then show up in the message menu used for creating buttons. Thus the message menu presented a story based on the object's behavior, hiding the underlying state. It was clear that with setter and getter methods, the full semantics of an object was available through message passing alone: any notion of state was hidden at a deeper implementation level.

A downside to automatically exposing instance variables through getters and setters is that it broadens the public interface to an object, and so might make it more difficult to change an object since other parts of a system might come to rely on the existence of these methods. Note that automatic getters and setters do not really violate the design principle of encapsulation, as the sender of a set or get message has no idea what kind of internal state (if any) is employed.

ARK also brought together some of the personalities who would later create Self. It was a demonstration of Smith's ARK in late 1985 or early 1986 that made Ungar realize that he wanted to collaborate with Smith: Smith showed Ungar an ARK graphical simulation of load-balancing in which processes could migrate from CPU to CPU. Ungar suggested attaching a CPU to one process so that when the process migrated, it would take the (simulated) CPU with it. When Smith was able to do this by just sticking a CPU widget to a process, Ungar realized that there was something special here; ARK was the kind of system that appeared simple but let its users easily do the unanticipated. Ungar was so taken with ARK that he later used a video of it for the final exam in his Stanford graduate course on programming language design. When Bay-Wei Chang took this exam, he was inspired to join the Self project. The spirit that shone through ARK illuminated the path for Self.

3. Self is Born at PARC

In 1985, as Smith was working on the Alternate Reality Kit, Ungar joined the faculty at Stanford. Stanford was just "down the hill" from PARC, and the Smalltalk group decided to bring Ungar in to collaborate with the group a few days per week. In 1981 the Smalltalk group had released Smalltalk-80 [GR83], the latest and perhaps most complete and commercially viable in the string of Smalltalk releases. The group considered it their natural charter to invent Smalltalk-next, and a follow-on to Smalltalk-80 was perhaps overdue. To tackle this design problem, the Smalltalk group decided to break into teams, each of which would propose a next language. Smith and Ungar paired up to create their own proposal for a language that would eventually become Self.

At the time we felt that Smalltalk was striving to realize a Platonic ideal, an apotheosis, of object-oriented programming. Smalltalk seemed to be heading toward a model in which computation proceeds by sending messages (containing objects as arguments) to objects and receiving objects in return. That's all. There is nothing about bits. Once in a while, one of these messages might turn on a pixel on a display. But, really, the notion of computation rests on a higher plane than bits in memory and is more abstract. Ungar likened this model of computation to Rutherford's experiments to learn about the atomic nucleus. Rutherford could not look inside an atom; he had to shoot subatomic particles at atoms and record how they bounced off. The pattern led him to deduce the existence of the nucleus. Similarly, we felt that there should be no way to look inside of an object; an object should be known only by its behavior, and that behavior could be measured only by the measurements on the behavior of objects returned in response to messages.

3.1. The Basic Ideas

When we started to design Self, we were partly inspired by ARK: we wanted the programming environment's graphical display of an object to *be* the object for the programmer. We

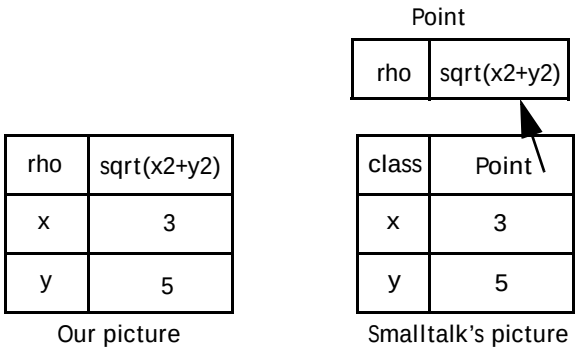


Figure 2. When we pictured a simple point object, we imagined it differently from in Smalltalk. In particular, the state and behavior of the object itself drew our attention, but the class did not. Since we wanted the language and environment level to mimic a hypothetical physical embodiment, we left classes out of Self. A Self object contains slots, such as rho and x in the figure, and a slot may function either as a holder of state (such as x) or as a holder of behavior (such as rho). (For simplicity of illustration, assume that the computed object is returned by the message send.)

noticed that whenever we drew an object on the whiteboard, our pictures were different from Smalltalk's; our objects always looked like small tables (see Figure 2) with no classes in sight.

As mentioned above, we employed a minimalist strategy in designing Self, striving to distill the essence of object and message. A computation in Self consists solely of objects which in turn consists of slots. A slot has a name and a value. A slot name is always a string, but a slot value can be any Self object. A slot can be asterisked to show that it designates a *parent*.

Figure 3 illustrates a Self object representing a two-dimensional point with `x` and `y` slots, a parent slot called `myParent`, and two special assignment slots, `x:` and `y:`, that are used to assign to the `x` and `y` slots. The object's parent has a single slot called `print` (containing a method object to print the point).

We found the resulting instance-oriented feel of the environment appealing because it lent more clarity and concreteness to a program design, with no loss of generality from Smalltalk-80. Additionally, Self's design eliminated metaclasses, which were one of the hardest parts of Smalltalk for novices to understand,

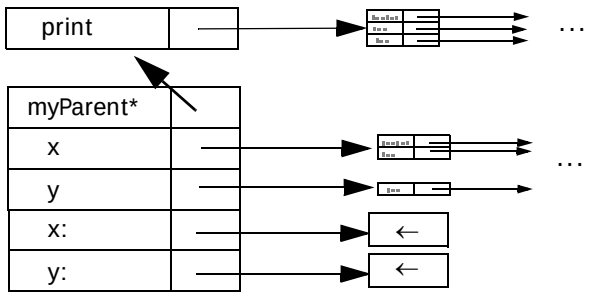


Figure 3. A Self point has x and y slots, with x: and y: slots containing the assignment primitive for changing x and y. The slot myParent carries a "parent" marker (shown as an asterisk). Parent slots are an inheritance link, indicating how message lookup continues beyond the object's slots. For example, this point object will respond to a print message because it inherits a print slot from the parent.

and avoided Smalltalk's long-standing schism between instance attributes and class attributes.[2] (The latter are also called "static methods and variables" in Java and C++.)

Recall that the novice Smalltalk-80 programmer had to learn about the scoping rules for each of Smalltalk's six classes of variables (see Figure 4). Smith recalls thinking this was somehow an odd story for an object-oriented language, in which getting and setting state could be done with message passing to objects. Sometime soon after joining the Smalltalk group at PARC (possibly in 1982 or 1983), Smith mentioned this variable-vs.-message dichotomy to Dave Robson, a senior member of the Smalltalk group and co-author of the Smalltalk "blue book" [GR83]. Smith recalls Robson replying in a somewhat resigned tone, "Yeah, once you're inside an object, it's pretty much like Pascal."

Ungar independently stumbled on the same question during a lunch in which Deutsch offhandedly suggested that these six types of variable accesses could be unified. We started to think of trying to use message sending as the only way to access storing and retrieval of state, and came up with a design that could merge all variable accesses with message passing (see Figure 5). We presented the design in an informal talk to the Smalltalk group in 1986, and in 1987 wrote the paper "Self: The Power of Simplicity" [US87].

We implemented inheritance with a variation on what Henry Lieberman called a "delegation" model [Lieb86]: when sending a message, if no slot name was matched within the receiving object, its parent's slots were searched for an object with a matching slot, then slots in the parent's parent, and so on. Thus our point object could respond to the messages `x`, `y`, `x:`, `y:`, and `myParent`, plus the message `rho`, because it *inherited* the `rho` slot from its parent. In Self, any object could potentially be a parent for any number of children and could be a child of any object. This uniform ability of any object to participate in any role of inheritance contributes to the consistency and malleability of Self and, we hope, to the programmer's comfort, confidence, and satisfaction.

To accomplish this unification, we decided to represent computation by allowing a Self object optionally to include code in addition to slots. An object with code is called a *method*, since it does what methods in other languages do. For example, the object in the `rho` slot above includes code and thus serves as a method. However, in Self, any object can be seen as a method; we regard a "data" object (such as `17`) as containing code that merely returns itself. This viewpoint unifies computation with data access: when an object is found in a slot as a result of a message send it gets *run*; a datum returns itself, while a method invokes its code. Thus, when the `rho` message is sent to our point object, the code in the object in the `rho` slot is found and that object's method runs. This unification reinforces the interpretation that it is the experience of the client that matters, not the inner details of the object used to create that experience.

Self's unification of variable access and message passing relied on the fact that a method would run whenever it was referenced.

2. Later, to support a programming environment, mirrors were added to Self. A mirror on an object contains information about that object, and may seem somewhat like a class that contains information about its instances. However, as discussed in section 4.4, an object may exist with no mirrors, unlike instances, classes, and metaclasses. Furthermore, had we been willing to guarantee that every object would transitively inherit from a root, we could have put reflective functionality in that root with no need for mirrors.

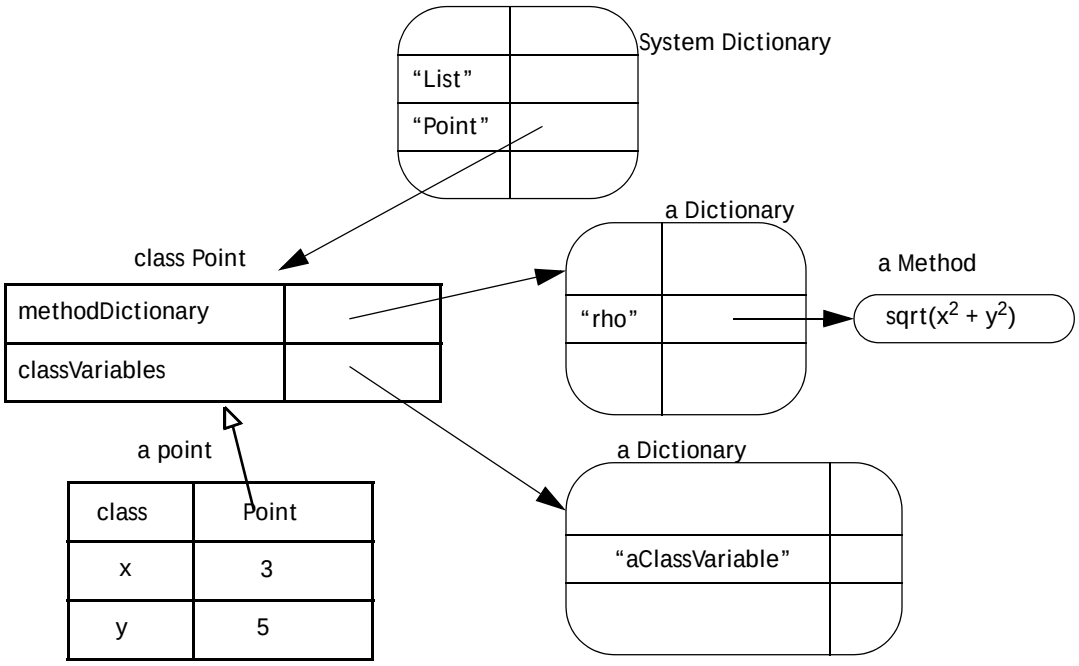


Figure 4. Smalltalk uses dictionary objects to hold the variables accessible from different scopes, though instance variables such as x and y for a point are directly available within the object. Class variables and global variables such as Point and List are held in such special dictionary objects with string objects (in quotes) as keys. Methods are also held in a special dictionary. All these dictionaries must exist in this way, as the entire language semantics relies on their existence. Not shown here are "Pool" variables, temporaries and arguments within a method context, or temporaries within a block closure object. In Figure 5 we show how Self achieves this scoping using objects and inheritance.

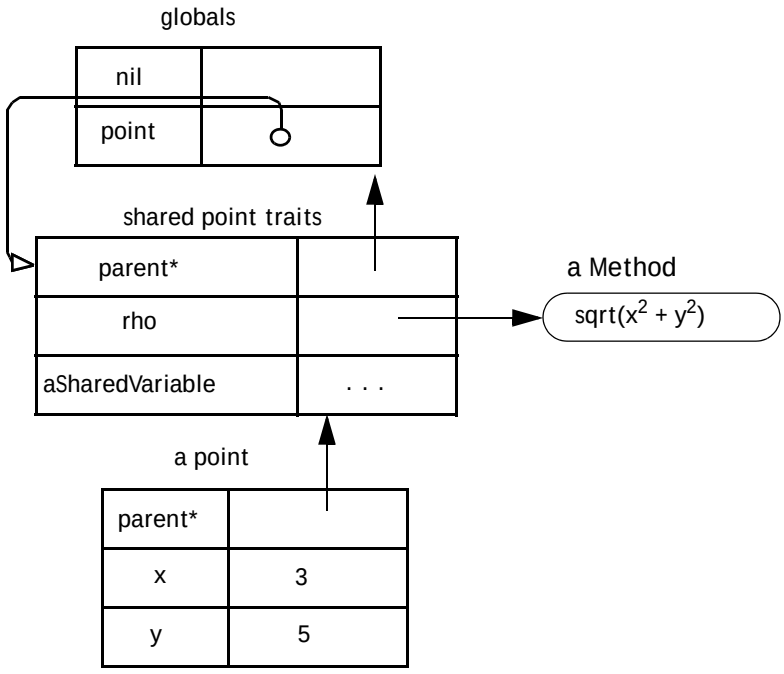


Figure 5. Self's object design gets many different scopes of variables for free. In Self, shared variables can be realized as slots in an ancestor object. Here, aSharedVariable is shared by all points, and the global variables point and nil are shared by all objects. This contrasts with Smalltalk, which needs a different linguistic mechanism for class variables and globals.

Consequently, there was no way (until the later development of reflection in Self; see section 4.4) to refer to a method. Many languages exploit references to functions, but Ungar felt that such a facility weakened the object orientation of a language. He felt that since a function always behaves in the same way, unlike an object which can "choose" how to respond to a message, functions-as-first-class entities would be too concrete. In other words, a function called by a function always runs the same code, whereas a method called by a method runs code that is chosen by the receiver. Smith understood Ungar's reservations about functions but was bothered by the complexity of introducing new fundamental language-level constructs (such as a new kind of slot with special rules for holding methods, or new kind of reference for pointing to a method without firing it).

The reader may wonder how one could ever get a method into a slot in the first place. In the first implementation of Self, the programmer just used a textual syntax to create objects with slots containing data and code. Later, we had to have some way for a program to make new objects and manipulate old ones. The invention of mirrors (section 4.4) added more elegant primitive operations to manipulate slots.

In contrast to many other object-oriented languages including C++ and Java, a number is an object in Self, just as in Smalltalk, and arithmetic is performed by sending messages, which can have arguments in addition to the receiver. For example, `3 + 4` sends the message `+` to the object `3`, with `4` as argument. This realization of numbers and arithmetic makes it easy for a programmer to add a new numeric data type that can inherit and reuse all the existing numeric code. However, this model of arithmetic can also bring a huge performance penalty, so implementation tricks became especially critical. Self's juxtaposition of a simple and uniform language (objects for numbers and messages for arithmetic in this case) with a sophisticated implementation let the programmer to live in a more consistent and malleable computational universe.

3.2. Syntax

In settling on a syntax for Self, we automatically borrowed from Smalltalk, as the two languages already had so much in common already. But Self's use of message sending to replace Smalltalk's variable access mechanisms would force some differences. Where Smalltalk referenced the class Point by having a global variable by that name, Self would reference the prototypical point with a slot named "point" and one would have to send a message, presumably to "self," to get a reference. So the Self programmer would write

```
self point.
```

which was verbose, but seemed acceptable. It raised the uncomfortable issue of what the token "self" meant. Could an object send "self" to itself to get a reference to itself? Smith recalls proposing that every object have a slot called "self" that pointed to itself. But Ungar pointed out that Smith's proposal only put off the problem one level, as even with the slot named "self," one would have to send the message "self" to something to get that reference! Smith counterproposed that perhaps there could be an implied infinity of self's in front of every expression, just as in spoken language, one can say "X" or one can say "*I* say: 'X'," or even "*I* say *'I* say "X"'," and so on. In spoken language we don't bother with this addition of "*I* say..." as it goes without saying. One could imagine an infinite number of them in front of any spoken utterance, and that they are just dropped to make spoken language tractable. However, Smith could never formalize this into a working scheme. So, as in Smalltalk, "self" would be a built-in token, providing the self-reference reference ex nihilo.

But that wasn't the end of it. In a method to double a point,[3] the Smalltalk programmer would assign to the two instance variables

```
x ← x * 2.
y ← y * 2.
```

whereas the Self programmer would perhaps write:

```
self x: (self x * 2).
self y: (self y * 2).
```
[4]

This was getting a bit verbose. One day at PARC, in one of the early syntax discussions, Ungar suggested to Smith that the term "self" be elided. Smith remembers this because he was embarrassed that he had to ask Ungar for the definition of the word "elide." Ungar explained that it meant the programmer could simply leave out "self." Under the new proposal, our example became:

```
x: (x * 2).
y: (y * 2).
```
[5]

At first hesitant, Smith came to like this as dropping the "self" was like dropping the utterance "I say" in natural language. Furthermore, eliding "self" neatly solved the infinite recursion problem of an object's having to send "self" to self to create a self-reference. In retrospect we feel that was a brilliant solution to a deep problem; at that time, it just seemed weirdly cool.

At this point, readers familiar with C++ will be wondering what the fuss was all about. It is true that C++ unifies the syntax for calling a member function of the receiver with that of calling a global function. Moreover, it unifies the syntax for reading a variable in the receiver with that of reading a global variable, and it unifies the syntax for assigning to a variable in the receiver with that of assigning to a global variable. In summary, C++ has six separate operations that mean six separate things but are boiled down to three syntactic forms: `aFunction(arg1, arg2-----..)`, `aVariable`, and `aVariable =`. What we had in Self after eliding "self" was just *a single* syntax and unified semantics for all six.

3.3. More Semantics

Ungar realized that, having removed variables, he and Smith had stumbled into enshrining message sending as the conceptual foundation of computation. Rather than each expression starting with a variable to serve as some reference, in Self the "programming atom" became a message send. Ungar in particular felt that the syntax could shift people's thinking about programs so that they would—unconsciously—tend to write better encapsulated and more reusable code. Smith was less interested in syntax, as he felt that whatever reasonable syntax was provided, the underlying semantics would shine through. So, any syntactic realization of the Self computational model would suffice for shifting people's thinking. Smith therefore felt that since we could choose any reasonable syntax, we should stick with the familiar and thus choose Smalltalk, as it was gaining popularity at the time. Looking back from 2006, Self might have become more popular had we devised a C-style syntax instead.

At this point we still had no good way to deal with temporary variables and arguments, whose scope was limited to a method context. (A method context in Smalltalk or Self is essentially a stack frame, a.k.a. an activation record.) Smith came up with the

3. One has to wonder how the language would have turned out without Cartesian point objects as fodder for our examples.
4. Parentheses added for clarity.
5. Parentheses added for clarity.

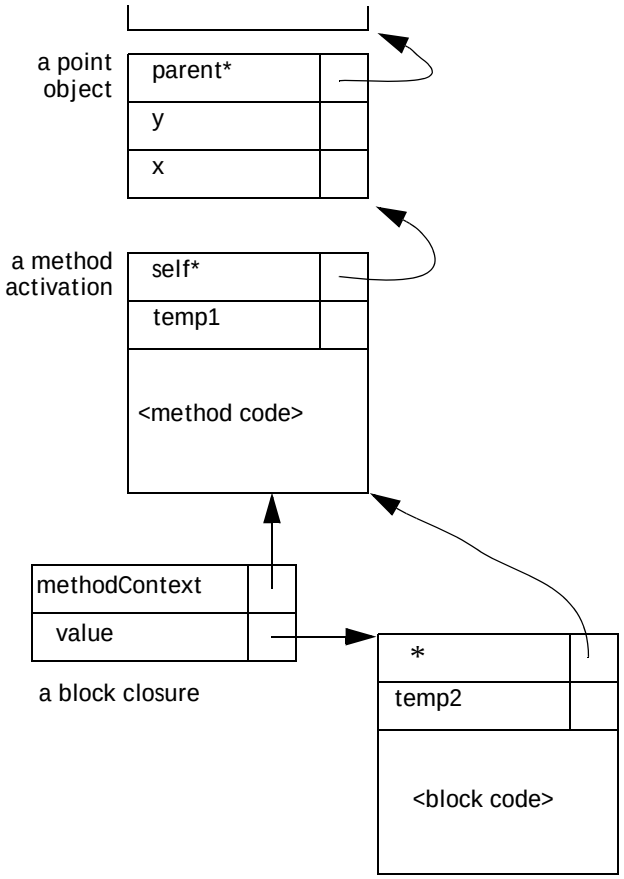


Figure 6. Lexical scoping of method activations and block closures via inheritance. We were pleased that the lexical scoping rules of methods and block closures could be explained through inheritance. But doing so made us realize there is a fundamental distinction between self (which is essentially a parent of the current method activation) and the point at which method lookup starts (which is the activation itself, so that temporaries and arguments in that activation are accessed). In this example, the method code can mention temp1 as well as x and y, as message sends start with the current activation and follow up the inheritance chain. But new method sends to self will have their self parent slot set to the point object.

As detailed in the text, when a block closure is invoked, the closure's activation is cloned, and the implicit parent is set to the enclosing method activation. This link is broken when the enclosing method activation returns.

Thus in the case illustrated here, the code in the block can access temp2, temp1, self, x, y, parent, and any other slots further up the inheritance chain.

idea that rather than retaining Smalltalk's temporaries and method arguments as variables, they too should be slots in an object whose parent was the message receiver. This formulation implied that slot name lookup would start in the local method context and then pass on to the message receiver, and so on up the inheritance hierarchy. Consequently, lookup would not really start at "self," but rather at something like Smalltalk's "thisContext," a pseudo-variable that serves as a reference to the local method activation. Smith explained this to Ungar in Smith's office at Xerox PARC and sensed that though Ungar felt this was a wild idea, he also felt it was somehow right. (Figure 6 illustrates this point.)

Although this tap dance removed the last vestige of variables from the execution story, it left a complexity that bothers us to this day. In Self, everything is a message send that starts looking for matching slots in the current method context, then continues up through the receiver (self) and on up from there (see Figure 6). But any method found in the lookup process creates a new method context inheriting from self, not from the current context. It's as though the virtual machine has to keep track of two special objects to do its job: the current context to start the lookup, and "self," to be the inheritance parent of new activations. Smith wondered how bad it would be to install new activations as children of the current activation, so "self" would no longer be such a special object, but Ungar convinced him that the resulting interactions between activations would amount to dynamic scoping and would be likely to create accidental overrides, with confusing and destructive side effects.

Block closures within a method can be represented as objects as well, as also illustrated in Figure 6. When invoked, a block closure is lexically able to refer to temporaries and arguments in its enclosing method, but is itself an object that can be passed around without evaluation if desired. In Self or Smalltalk, a block closure can be sent the `value` message to run its code. The `value` method in a block context differs from other methods: when such a method runs its parent slot is set, not to the current receiver, but rather to the enclosing context in which the block originates.

Although the Self model enabled inheritance and slot lookup to explain what many other languages didn't even bother to explain with the language's fundamental semantics, the appearance of special cases (such as the value method in a block) bothered us. We had several discussions at the whiteboards at PARC, trying to figure out a unifying scheme, but none was satisfactory.

As we strove for more and more simplicity and purity, we came up against other limits we simply could not wrestle into a pristine framework. We wanted every expression in Self to be composed of message sends. In particular, we wanted every expression to start off by sending one or more messages to the current context and on up through self. Literals, though, fail to conform: a literal is an object (usually one of just a few kinds, such as numbers and strings) that is created in place in the code just where it is mentioned. For example, the Self expression

```
x sqrt
```

sends the message `x` to `self`, then sends `sqrt` to the result. For a few weeks during our design phase we puzzled over how to support the expression

```
3 sqrt
```

within a pure message-sending framework. Most languages would treat the 3 as a "literal" (something that is not the result of

computation but rather "literally" interpreted directly in place). As an example, in Smalltalk this code fragment would be compiled so as to place the object 3 directly in the expression, followed by the send of `sqrt` to that object. We wondered if Self could treat the textual token 3 not as a literal, but rather as a message send to `self`. That way, 3 would be on the same footing as other widely referenced objects, such as `list`, the prototypical list. The idea was that somewhere in the stratosphere of the inheritance hierarchy would be an object with a slot whose name would be "3" and that would contain a reference to the actual object 3. Send "3" to an object, and the result of the lookup mechanism would be a reference to the object that, in the receiver's context, meant 3. This result would normally be the regular 3, but it might, for example, be a 1 in the context of some object that could only understand mod 2 arithmetic. In the end we gave up on this, as it seemed to hold too much potential for mischief and obfuscation. For instance, 2 + 2 could evaluate to 5! It seemed to both of us like more expressive freedom than was really needed, and supporting those objects with such a conceptually infinite number of slots seemed a heavy burden to place on the virtual machine. We decided to give up on pushing uniformity this far.

Our design for the unification of assignment with message sending also troubled us a bit. An object containing a slot named "x" that is to be assignable must also contain a slot named "x:" containing a special object known as the assignment primitive. This slot is called an assignment slot, and it uses the *name of the slot* (very odd) to find a correspondingly named data slot *in the same object* (also odd). This treatment leads to all sorts of special rules; for instance, it is illegal to have an object contain an assignment slot without a corresponding data slot, so consequently the code that removes slots is riddled with extra checks. Also, we were troubled that it took a pair of slots to implement a single container. Other prototype-based languages addressed this issue by making a slot-pair an entity in the language and casting an assignable slot as such a slot pair. Another alternative might have been to make the assignment object have a slot identifying its target, so that in principle any slot could have served as an assignment slot for any other.

Both authors strove for simplicity, but each had his own focus. Smith's pure vision grounded in the uniformity of the physical world led him to advocate such interesting features as parent slots for methods and message-passing for local variable access. In contrast, Ungar couldn't wait to actually use the language, and so he was thinking about the interaction between language features and possible implementation techniques. For example, unlike Lieberman's prototypes [Lieb86, SLU88], a Self object does not add an instance variable on first assignment, but rather must already contain a data- and assignment-slot pair if the assignment is to be local. Otherwise, it delegates the assignment (which is just a one-argument message send) to its parent(s) (if any). Ungar also was thinking about customization (section 4.1) at that point; to make instance variable access and assignment efficient when a sibling might implement them as methods, Ungar realized that one could compile multiple versions of the same inherited method for each "clone-family." The requirement that an assignable slot be accompanied by a corresponding assignment slot created a clear distinction at object creation time between a constant slot and a mutable slot that was intended from the start to aid the implementer. Ungar knew that an efficient implementation would have to put information shared across all clones of the same prototype in a separate structure, which was eventually called a *map* [CUL89]. (See section 4.1 for details.)

When Ungar came up to PARC as a consultant, he had to sign in by writing his name on an adhesive name tag and wearing it while on the premises, yet no one ever paid any attention to it. So Ungar took to writing more and more absurd names on his tag, such as "nil," "super," and even "name tag." One day, he came into the common area outside Smith's office at PARC, and upon seeing Smith immediately exclaimed, "I have a name for the language! Self!" He had moments earlier signed his name tag "self" when inspiration had struck. Smith commented that all those selfs missing from the syntax could maybe be inherited from the title of the language. The name appealed to us immediately, and from that day forward we had no doubt that the language would be called "Self."

4. Self Takes Hold at Stanford and Evolves

In June 1986, Ungar (at Stanford) asked Sun for some equipment: an upgrade to 4Mb of main memory for 14 machines ($28K), a Sun 3/160S-4 workstation with 4MB for ($15K)—this was a diskless machine—a 400MB disc drive ($14K), a tape drive ($3K), and an Ethernet transceiver ($500). When we started the effort to build a Self system, hardware was primitive and expensive!

Ungar recalls spending his first year at Stanford (1985-1986) casting about for a research topic. His first PhD student, Joseph Pallas, was working on a multiprocessor implementation of Berkeley Smalltalk [Pal90] for the Digital Equipment Corporation Systems Research Laboratory Firefly [TSS88], an early coherent-memory multiprocessor. As far as we know, this system was the first multiprocessor implementation of Smalltalk.

In Ungar's June 1986 summary of his first year's research at Stanford, Self was not mentioned at all. But nine months later, he had found his topic: in a March 1987 funding proposal, Ungar wrote: "Self promises to be both simpler and more expressive than conventional object-oriented languages." He also wrote about "developing programming environments that harness the power of fast and simple computers to help a person create software," of "transforming computing power into problem-solving power," of "shortening the debug, edit, and test cycle," and how "dynamic typing eases the task of writing and changing programs." He explained the potential advantages of Self: its unification of variable access and message passing, that any Self object could include code and function as a closure, its better program-structuring mechanisms, including prototypes. Finally, he noted that obtaining performance for Self would pose a challenge.

In 1988, Smith went to England for a year, and Ungar's consulting assignment at PARC changed from designing languages to implementing automatic storage reclamation for what was to become the HPS Smalltalk system. This was the end of Self at PARC.

Ungar had decided that Self's replacement of variable access by message passing made it so impractical that devising an efficient implementation of Self would make a good research topic. He was also eager to see if the language design would hold up for nontrivial programs. Ungar's May 1988 report, "SELF: Turning Hardware Power into Programming Power," proposed a complete, efficient Self virtual machine, a Self programming environment with a graphical interface based upon artificial reality, and a high-bandwidth, low-fatigue total immersion workstation. (We never got around to the last one.) He discussed the benefits of the language and the special implementation challenges it posed. To tackle the implementation issues, we proposed custom compilation and inlining of primitive operations and messages sent to constants. (These were the first optimizations we tried.)

We proposed to investigate dynamic inheritance and to explore mirror-based reflection, the latter as a means to inspect a method as well as objects intended only to provide methods for inheritance by other objects. Ungar christened the latter sort of object a "traits" object. Years later, others would formulate a framework for combining bundles of methods in a class-based framework, reusing the word "traits" with a slightly different meaning [SDNB03].[6]

The report went on to outline proposed work on a graphical programming environment designed to present objects as physical hunks of matter—objects, not windows. Ungar proposed to use well-defined lighting for reality and substance, and to manage screen updates so as to avoid distraction. (This seems to foreshadow our work in cartoon animation.) He also proposed to implement a graphical debugger for Self.

In January 1989, Ungar asked Sun for more equipment: SPARC machines, a server, a workstation for his home, and three diskless machines, each with 24Mb of memory. He also asked for a fast 68030 machine. As the Stanford Self project progressed in 1989 and 1990, it was able to start work on the programming environment and user interface, later known as UI1 or Seity. The goal (far from realized) was eventually to replace most uses of C and C++ (and, of course, Smalltalk) for general-purpose programming. We wanted to harness the ever-increasing raw computational power of contemporary workstations to help programmers become more productive.

4.1. Implementation Innovations

Faced with designing an interpreter or compiler to implement a language, one often takes a mathematical, mechanistic view and focuses on getting correct programs to execute correctly. Inspired by Smith's Alternate Reality Kit [Smi87], Ungar took a different tack: he concentrated on getting the user to buy into the reality of the language. Even though Self objects had no physical existence and no machine was capable of executing Self methods, the implementation's job was to present a convincing illusion that these things did exist. That is why, despite all the convoluted optimizations we finally implemented, the programmer could still debug at the source level, seeing all variables while single- stepping through methods, and could always change any method, even an inlined one, with no interference from the implementation.

To structure complexity and provide the freest environment possible, we layered the design so that the Self language proper included only the information needed to execute the program, leaving the declarative information to the environment. In other words, in Smalltalk and Java, classes served as both structural templates (i.e., concrete types) and were visible to the programmer, but in Self the structural templates (embodied by *maps*) were hidden inside the virtual machine and thus invisible to the programmer. Abstract types were regarded as programmer-visible declarative information, and Self left those to the environment. For example, one language-level notion of abstract type, the *clone family*, was used in the work of Agesen et al. [Age96] in their Self type-inference work. There is no clone family object in the Self *language*, but such objects could be created and used by the programming environment. This design kept the language small, simplified the pedagogy, and allowed users to extend the domain of discourse.

The Stanford Self team believed that performance would be critical for acceptance, yet our design philosophy placed a large burden on the compiler. Deutsch and Schiffman's PS system had simply translated Smalltalk's bytecodes to machine code, with only peephole optimization [DS84]. But unlike Smalltalk, the Self bytecodes contained no information about what is a variable access or assignment vs. a message send, and there are no special bytecodes for simple arithmetic, nor special bytecodes for commonly used control structures. Simple translation would not suffice. Obtaining performance without sacrificing the programming experience would be our challenge.

The problem in this area was a magnification of one faced by a Smalltalk implementation: a style of programming in which methods are short, typically one to five lines of code, resulting in frequent message sends. Frequent sends hurt performance because each method invocation in Smalltalk (and Self) is dynamically dispatched. In other words, every few operations a Smalltalk program called a subroutine that depended on the runtime type of the value of the first argument (a.k.a. the receiver). In Self, the situation was even worse, because every variable access or assignment also required a message send. Other, more static languages, such as C++ (and later Java), incorporated static type-checking, and this added information facilitated use of dispatch tables (a.k.a. vtables) to optimize virtual calls. This technique was not suited for Self or Smalltalk because without static types every dispatch table needs an entry for every method name. This requirement would result in prohibitive time and space requirements to update and maintain dispatch tables. Thus, to make Self work well, we would not only have to implement prototypes effectively, but would also have to find new techniques to eliminate the overhead of virtual function calls by inline expansion of methods whose bodies could not be known before the program runs.

4.1.1. The First Self Virtual Machine, a.k.a. Self-89

Back when the language had been designed, Ungar had deviated from Lieberman's prototype model for implementation considerations. In Lieberman's system, an object initially inherited all of its attributes and gained private attributes whenever an assignment occurred. From the beginning, Ungar tried to keep an object's layout constant to reduce run-time overhead. He therefore incorporated the distinction between variable and constant slots into Self. Assignment could only change a variable slot, not create a new slot. Furthermore, to represent an object, space would be required for only its variable slots and one pointer to shared information about its constant slots and its layout. This shared information was called a map (Figure 7). During 1986-87, graduate students Elgin Lee and Craig Chambers joined the project. Lee wrote the first memory system and implemented maps to achieve space usage that was competitive with Smalltalk [Lee88]. Later, Chambers reimplemented the memory system [CUL89]. We achieved our goal: the per-object space overhead of Self was only two words.

In 1988, Chambers wrote the first Self compiler [CU89, CUL89]. This compiler represented Self programs using expression trees and introduced three techniques: customization, type prediction, and message splitting. Each of these ideas was inspired by our desire to run no more slowly than Smalltalk. Wherever we thought that Self's object model would hinder its performance, we tried to devise a technique to recoup the loss, at least in the common cases. To maintain the interactive feel of an interpreter, we also introduced dependency lists (described below).

Customization. A Smalltalk object belongs to a specific class, and its instance variables occur at fixed offsets specified by the class. Even an inherited instance variable has the same offset as

6. According to an email exchange with Black and Schärli, Self's traits played into their thinking but were not the primary inspiration.

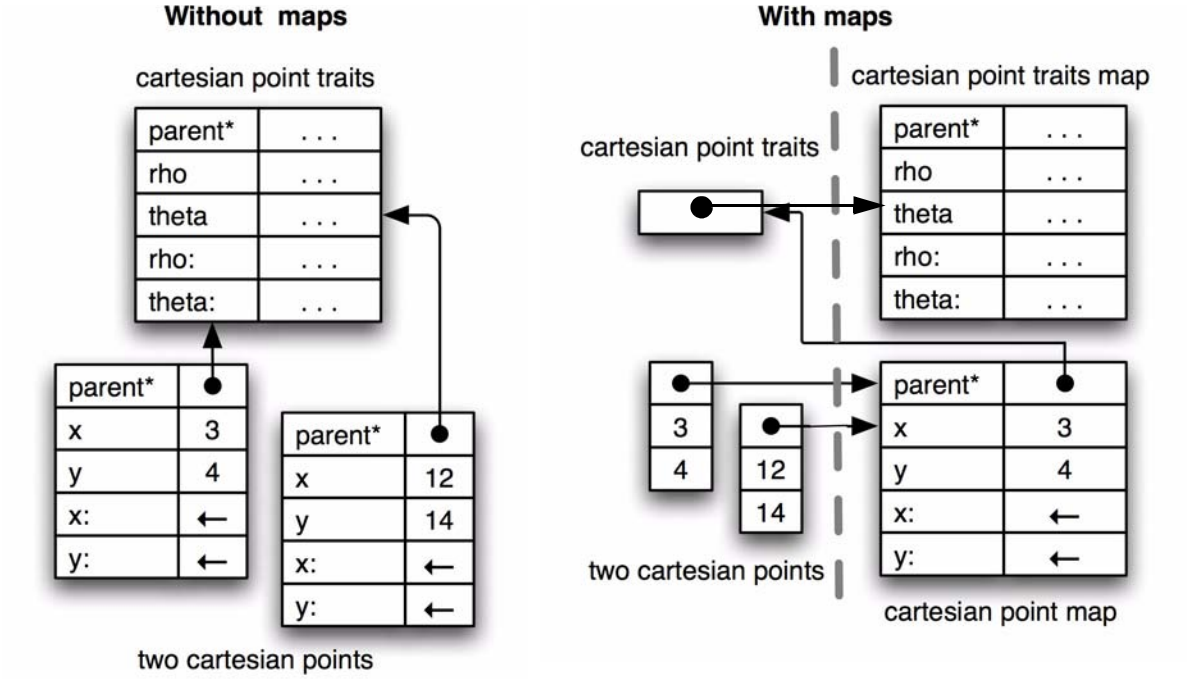


Figure 7. An example of the representations for two Cartesian points and their parent, also known as their "traits" object. Without maps, each slot would require at least two words: one for its name and another for its contents. This means that each point would occupy at least 10 words. With maps, each point object needs to store only the contents of its assignable slots, plus one more word to point to the map. All constant slots and all format information are factored out into the map. Maps reduce the 10 words per point to 3 words. (A Self object also has an additional word per object containing a hash code and other miscellany.) Since the Cartesian point traits object has no assignable slots, all of its data are kept in its map.

it would in an instance of the class from which it is inherited. As a result, the Smalltalk bytecodes for instance variable access and assignment can refer to instance variables by their offsets, and these bytecodes can be executed quite efficiently. In Self, there is no language-level inheritance of instance variables, and so the same inherited method might contain an access to an instance variable occurring at different offsets in objects cloned from different prototypes. (All objects cloned from the same prototype have the same offsets, and are said to comprise a clone family.) In some invocations of the inherited method, the bytecode might even result in a method invocation. To compile accesses as efficient, fixed-offset load operations, Ungar had realized—back at PARC—that the virtual machine could compile multiple copies of the same inherited method, one per clone family. This trick would not compromise the semantics of the language because it could be done completely transparently. This technique, known as customization, was implemented in Chambers' first Self compiler (see Figure 8).

Type Prediction. In Smalltalk and Self, even the simplest arithmetic operations and control structures were written as messages. In the case of control structures, blocks are used to denote code whose execution is to be deferred. Thus, even frequently occurring operations that need not take much time must be expressed in terms of general and relatively time-consuming operations. For example, the code `a = b ifTrue: [...]` sends a message called "=" to `a`, then creates a block, and finally sends "ifTrue:" to the result of "=" with a block argument. The Smalltalk system uses special bytecodes for arithmetic and simple control structures to reduce this overhead. For Self, we kept the bytecode set uniform, but built heuristics into the compiler to expect that, for instance, the receiver of "=" would probably be a (small) integer and that the receiver for "ifTrue:" would likely be a Boolean. This information allowed the compiler to generate code to test for the common case and optimize it as described below, without loss of generality. For example, in Self but not Smalltalk, the programmer can redefine what is meant by integer addition. This idea was called "type prediction."

Message Splitting. As mentioned above, Smalltalk implemented if-then constructs such as `ifTrue:` with specialized bytecodes, including branch bytecodes. Since in Smalltalk (and Self) the "true" and "false" objects belong to different classes (clone-families in Self), the branch bytecodes conceptually test the class of the receiver to decide whether to branch or not. To achieve similar performance in compiled Self code without special bytecodes, we had allowed the compiler to predict that the receiver of such a message was likely to have the same map as either the "true" or the "false" object, but could be anything. The Self compiler was built around inlining as its basic optimization, so to optimize `ifTrue:` for the common case without losing the ability for the user to change the definition of `ifTrue:`, the compiler had to insert a type-case (a sequence of tests that try to match the type (represented in Self by the map) of the receiver against a number of alternatives) and then inline different versions of the called method in each arm of the type-case construct. In the "true" arm, it could inline the evaluation of the "then" block, in the "false" arm, it could inline "nil," and in the uncommon case, it could not inline at all but just compile a message-send. In other words, one message-send of "ifTrue:" was split into three sends of "ifTrue:" to three different types of receiver (true, false, and unknown). We dubbed this technique "message splitting."

Dependency Lists. For this compiler Chambers also created Self's dependency system, a network of linked lists that allowed the virtual machine to quickly invalidate inline caches and com-

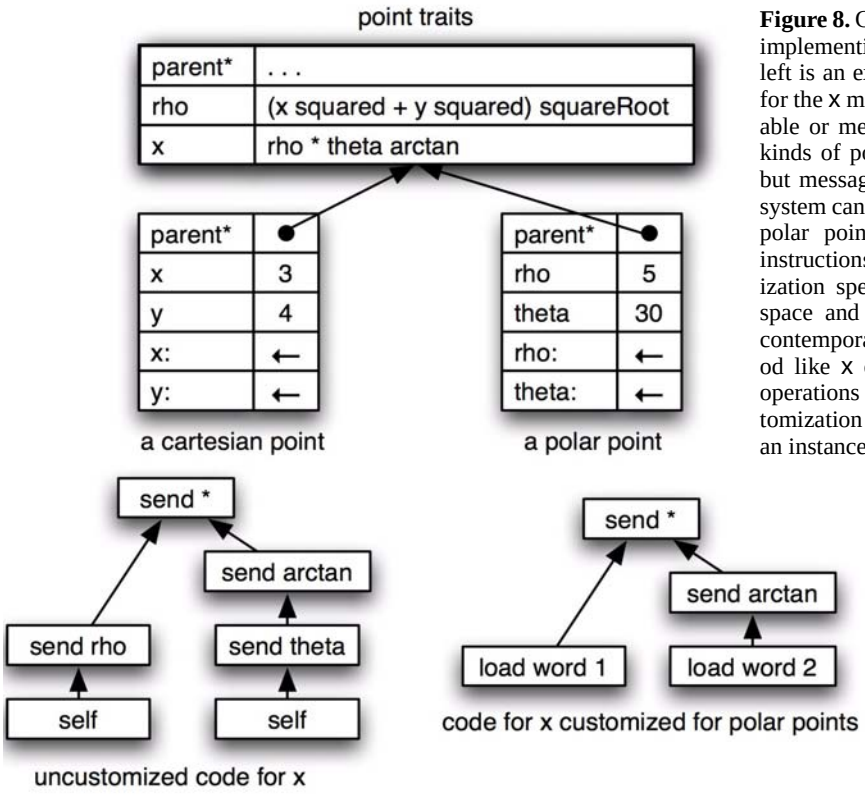


Figure 8. Customization: At left are three objects implementing Cartesian and polar points. Below left is an expression tree for uncustomized code for the x method. Since rho may be either a variable or method, to use the same code for both kinds of points, nothing more can be compiled but message sends for rho and theta. But if the system can compile a specialized version of x for polar points, it can replace these with load instructions, as in the code below right. Customization speeds sends to self at the expense of space and complexity. In Smalltalk and other contemporary object-oriented languages, a method like x could include instance variable load operations at its source level. Ungar devised customization to regain the speed lost by expressing an instance variable access as a send to self.

piled methods when the programmer made changes to objects [Cham92]. In addition to these techniques, the compiler also supported source-level debugging so that the system could be as easy to understand as an interpreter.

When this compiler was completed in 1988, the Self virtual machine comprised 33,000 lines of C++ and 1,000 lines of assembly code. It ran on both a Motorola 68020-based Sun-3 workstation and a SPARC-based Sun-4 workstation. The latter was a RISC microprocessor with a 60ns cycle time, and an average of 1.6 cycles per instruction. We had written about 9,000 lines of Self code, including data structures, a simple parser, and the beginnings of a graphical user interface. The largest benchmark we used at that time was the Richards operating system simulation [Deu88], and the compiler produced Self code that ran about three times faster than Smalltalk, but about four times slower than optimized C++ [CU89]. There was an issue with compilation speed: on a Sun 4-260 workstation, the compiler took seven seconds to compile the 900-line Stanford integer benchmarks, and three seconds to compile the 400-line Richards benchmark. This was deemed too slow for interactive use; we wanted compile times to be imperceptible.

Ungar recalls that Chambers articulated an important lesson about types: the information a human needs to understand a program, or to reason about its correctness, is not necessarily the same information a compiler needs to make a program run efficiently. Thereafter, we spoke of *abstract types* as those that help the programmer to understand a program and of *concrete types* as those that help an implementation work well. In many languages, the same type declaration (e.g., 32-bit integer) specifies both semantics and implementation. As we implemented Self, we came to believe that this common conflation inevitably compromised a language's effectiveness.[7] Accordingly, we hoped to show that type declarations for the sake of performance were a bad idea, and we made the point that Self's performance—without explicit declarations—had already pulled even with Johnson's Typed Smalltalk system [John88, CU89].

4.1.2. The Second-Generation Self Virtual Machine, a.k.a. Self-90

We weren't satisfied with the performance of our first Self compiler and in 1989 proceeded to improve the Self system. In early 1989, Chambers rewrote the memory system and then implemented a far more ambitious compiler [CU90]. This compiler was based on a control flow graph and included many optimization techniques that had been invented for static languages, such as more extensive inlining, interprocedural flow-sensitive type analysis, common subexpression elimination, code motion, global register allocation, instruction scheduling, and a new technique called extended splitting.

Extended Splitting. Recall that the first Self compiler had been based on expression trees. As a consequence, the only message sends that it could split were those whose receivers were the results of the immediately preceding sends. With the addition of flow-sensitive type analysis, the new compiler could split a message based on the type of a value previously stored in a local variable. We observed that it was common for the same local to be the receiver for several message sends, although the sends might not be contiguous, so Chambers extended the new compiler to split paths rather than individual sends. This technique was called "extended splitting" and the ultimate goal was to split off entire loops, so that, for example, an iterative calculation

7. The creators of the Emerald system had the same insight [BHJ86] and had probably discussed it with Ungar during a visit to the University of Washington.

involving (small) integers could be completely inlined with only overflow tests required. Many of the benchmarks we were using then consisted of iterative integer calculations because we were trying to dethrone performance champion C, and those sorts of programs catered to C's strengths.

The new compiler yielded decidedly mixed results. The performance of the generated code was reasonable: the Richards benchmark shrank to three-fourths of its previous size and slowed by just a bit, small benchmarks sped up by 25% - 30%, and tiny benchmarks doubled in speed. The problem was that compiler ran an order of magnitude slower. For example, it took a majestic 35 seconds to compile the Richards benchmark, a wait completely unsuitable for an interactive system. Perhaps we had fallen prey to the second-system syndrome [Broo]. In day-to-day use, we stuck with the original compiler.

This second-generation system did introduce other improvements, including faster primitive failure, faster cloning, faster indirect message passing (for messages whose selectors are not static), blocks that would not crash the system when entered after the enclosing scope had returned, and a dynamic graphical view of the virtual machine (the "spy"), written by Urs Hölzle, who had joined the Self project in the 1987-1988 academic year.

4.1.3. The Third-Generation Self Virtual Machine, a.k.a. Self-91

By mid-1990, Hölzle had made many small but significant improvements to the Self virtual machine: he had improved the performance of its garbage collector by implementing a card-marking store barrier; he had redone the system for managing compiled machine code by breaking it up into separate areas for code, dependencies, and debugging information; he had added an LRU (least-recently used) machine-code cache replacement discipline; he had started on a profiler; and he had improved the method lookup cache. The Self virtual machine comprised approximately 50K lines of C++.

Hölzle had devised a new technique that would turn out to be crucial: polymorphic inline caches (PICs) [HCU91]. Self was already using inline caching [DS84], a technique that optimized virtual calls by backpatching the call instruction. Deutsch and Schiffman had noticed that most virtual calls dispatched to the same function as before, so rather than spending time on a lookup each time, if the call went to the same method as before, the method could just verify the receiver's map in the prologue and continue. This technique worked, but we discovered that some fairly frequent calls didn't follow this pattern. Instead, they would dispatch to a small number of alternatives. To optimize this case, when a method prologue detected an incorrect receiver type, Hölzle's new system to create a new code stub containing a type-case and redirect the call instruction to this stub. This type-case stub, called a polymorphic inline cache (PIC), would be extended with new cases as required. This optimization sped up the Richards benchmark, which relied heavily on one call that followed this pattern, by 37%. We realized that, after a program had run for a while, the PICs could be viewed as a call-site-specific type database. If a call site was bound to the lookup routine, it had never been executed; if it was bound to a method, it had been executed with only one type; and if it was bound to a PIC, that PIC contained the types that had been used at that site. Hölzle modified Chambers' compiler to exploit the information recorded in the PICs after a prior run and sped up Richards by an additional 11%.

In 1990, Chambers worked to improve the compilation speed without sacrificing run-time performance [CU91]. We had learned that much published compiler literature neglected the compilation speed issue that was so critical to the interactive feel we wanted for Self. Striving for the best of both worlds, Chambers devised a more efficient implementation of splitting and enhanced the compiler to defer the compilation of uncommon cases. The latter idea was suggested to us by then-student John Maloney at an OOPSLA conference (Maloney would later join the Self project). Deferred compilation avoided spending time on the cases that were expected to be rare, such as integer overflow and out-of-bounds array accesses. The compiler still generated a test for the condition, but instead of compiling the rarely executed code, would compile a trap to make the system go back and transparently recompile a version with the code included for the uncommon case. The new version would be carefully crafted to use the same stack frame as the old, and execution would resume in the new version. The whole process was (naturally, given our proclivities) transparent to the user.

In addition to hastening compilation, this optimization sped up execution because the generated methods were smaller and could use registers more effectively. However, in subsequent years, it turned out to be a source of complexity and bugs. As of this writing (2006), Ungar, who has only his spare time available to maintain Self, has disabled deferred compilation. Back in 1990, though, we were excited: the system compiled the Richards benchmark 7 times faster than previously, the compiled code was about three-fourths the size, and it ran 1.5 times faster. This brought Richards performance to one third that of optimized C++. We released this system as Self 1.1 in January 1991.

With all the improvements, compilation speed on our Sun 4-260 was still too slow; compiling Richards took 5.5 seconds. In addition, this third compiler suffered from brittle performance; because it used heuristics to throttle its inlining, it was sensitive to the program's exact form, and small changes to a program could result in large changes to its performance, as method sizes crossed inlining thresholds. However, after three compilers, it was time for Chambers to stop programming and start his doctoral dissertation. He did so and graduated in 1992.

It was then Hölzle's turn to take on the challenge of combining interactivity with performance. Building on Chambers' compilers and his own work with polymorphic inline caches, he started to experiment with "progressive compilation" and would eventually achieve the best of both worlds (section 5.1).

4.2. Language Elaborations

In 1988 and 1989, the students and Ungar writing Self code at Stanford ran into situations that seemed to need better support for multiple inheritance and encapsulation than were covered by the language outlines as sketched out at Xerox PARC. Self's simple object model was a good base for exploring these topics since there were few interactions with other language features. Smith was following other research interests at this point, and so Chambers, Ungar, Chang, and Hölzle set about enhancing the language with some clever ideas: prioritized multiple inheritance, the sender-path tiebreaker rule, and parents-as-shared-parts privacy [CUCH91].

Prioritized Multiple Inheritance. Back in the late 1980s, multiple inheritance was a popular research area, especially rules for dealing with collisions arising from inheriting two attributes with the same name [Card88]. Class-based languages suffered from the need to deal with structural collisions arising from inheriting different instance variables with the same name, as well as behavioral collisions arising from inheriting different methods with the same name, and we thought that this area would be easier in classless Self. There were two popular search

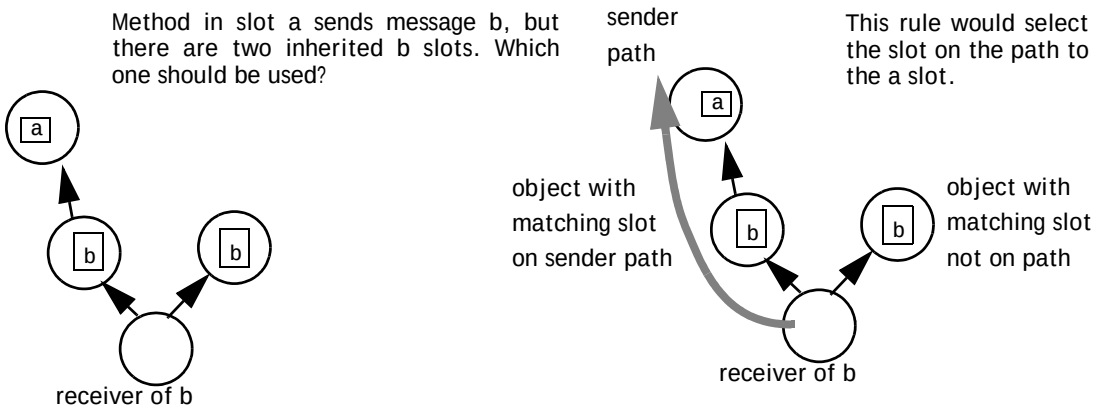


Figure 9. Sender path tiebreaker rule. For a while, multiple inheritance conflicts were resolved according to the inheritance path of the sending method. In this situation there is a "tie" with two inherited 'b' slots. The 'b' slot on the left is selected because it is on the path to the slot whose code sent the 'b' message.

strategies for multiple inheritance: unordered for safety, and ordered for expressiveness. In the unordered case, all parents were equal and any clashes were errors. In the ordered case, parents were searched in order, and the first match won. Seeking the best of both worlds, the Stanford students and Ungar devised a priority scheme: each parent was assigned an integer priority by the programmer, and the lookup algorithm searched parents in numeric order. Equal-numbered parents were searched simultaneously, and multiple matches in equal-numbered parents generated an "ambiguous message" run-time error.

Sender-path Tiebreaker. Having devised and implemented a powerful multiple inheritance scheme, we set about using multiple inheritance wherever we could. As a result, we struggled with many "ambiguous message" errors in our code. Since most of these errors seemed unjustified, we came up with a new rule that we thought would automatically resolve many of the conflicts. This rule stemmed from our belief that parent objects in Self were best considered to be shared parts of their children. When combined with the typical case of a method residing in an ancestor of its receiver, we believed that a matching slot found on the same inheritance path as the object holding the calling method ought to have precedence. This was called the "sender-path tiebreaker rule" (see Figure 9).

Shared-part Privacy. Smalltalk provides encapsulation for variables but not methods; in Smalltalk, instance variables are private to the enclosing object, but all methods are public. Since we believed that a Self variable should be thought of as just a particular implementation of two methods, the original design for Self omitted encapsulation for all variables (as well as methods). Influenced by Smalltalk and, to a lesser extent, by C++, the Self group (then at Stanford) sought to fix this by adding privacy to the language. At this time we had yet to build a graphical user interface, and so we started with a discussion of syntax in Ungar's office that lasted for hours. Eventually, Chambers facetiously proposed an underscore prefix ("_") for private slots and a circumflex prefix ("^") for public slots. When Ungar agreed, Chambers tried to unpropose them but failed, and so those prefixes became Self's privacy syntax. After agreeing on syntax, we then had to devise a semantics for privacy. Consider a slot a containing a method that sends the message b. If b is private, how should we decide whether to allow the attempted access to b found in slot a's code? Reasoning that in Self, parents are shared parts of their children, we decided that slot b should be accessible to a given message from code in a if both the object holding the a slot and the object holding the b slot were either the *same as* or *ancestors of* the receiver of the message. This concept was called "shared-part privacy" (see Figure 10).

Chambers deftly made these changes to the virtual machine. He did it so easily that back then, Ungar felt that there was no language feature too intricate for Chambers to put into the system in a day or so. Of course, having invented a powerful new privacy scheme, we set about writing programs that put it to work whenever possible.

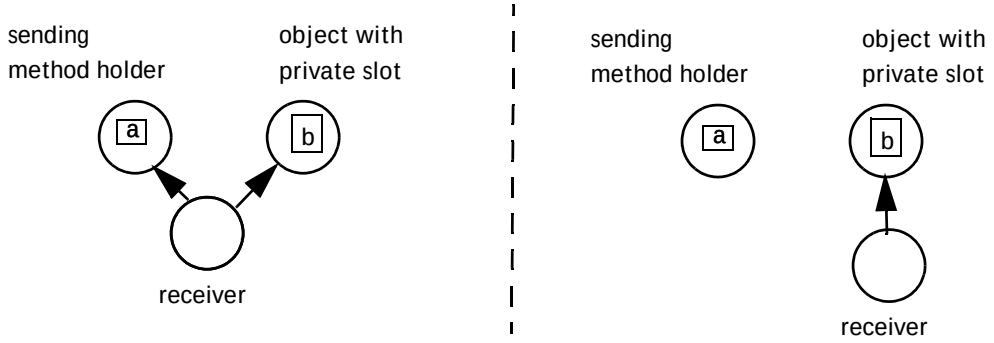


Figure 10. Privacy based on parents-as-shared-parts. Inherited method a sends message b to self, labeled as the receiver. In each case b is a private slot. Since both the sending method holder and the private slot holder are parents of the receiver on the left, that access would be allowed. On the right, the access would be denied.

In a July 1989 report, "SELF: Turning Hardware Power into Programming Power," the multiple inheritance with send-path tiebreaking idea appeared. Ungar was enthusiastic about this idea at the time: "SELF's multiple inheritance innovations have improved the level of factoring and reuse of code in our SELF programs." The first version of the privacy (a.k.a. encapsulation) idea also appeared as a proposal. By the end of March 1990, we had added our new privacy semantics, had changed "super" to "resend" to accommodate "directed resends," and had changed the associativity rules for binary messages to require parentheses.

Something unexpected occurred after we started using our multiple inheritance and privacy schemes. Over the following year, we spent many months chasing down compiler bugs, only to discover that Chambers' compiler was correct and it was our understanding of the effects of the rules that was flawed. For example, a "resend" could invoke a method far away from the call site, running up a completely different branch of the inheritance graph from what the programmer had anticipated. In addition, the interactions with dynamic inheritance turned out to be mind-boggling. Eventually, Ungar realized that he had goofed. Prioritized multiple inheritance, the sender-path tiebreaker rule, and shared-part privacy were removed from Self by June 1992. We found that we could once again understand our programs. Self's syntax still permitted programmer to specify whether a slot was public, private, or unspecified, but there was no effect on the program's execution. This structured comment on a slot's visibility proved to be useful documentation.

In the process of revisiting Self's semantics for multiple inheritance, Chambers suggested that we adopt an "unordered up to join" conflict resolution rule (see Figure 11). Although it might have worked well, we never tried this idea; once bitten twice shy.

To this day, many object-oriented language designers shy away from multiple inheritance as a tar pit, and others are still trying to slay this dragon by finding the "right" concepts. Our final design for Self implemented simple, unordered multiple inheritance and has proven quite workable. Although many language designers (including Ungar) have used examples to motivate the addition of facilities, at least for prioritized multiple inheritance, the sender path tiebreaker rule, and shared-part privacy, it would have been better to let the example be more difficult to express and keep the language simpler. Ironically, in his dissertation,

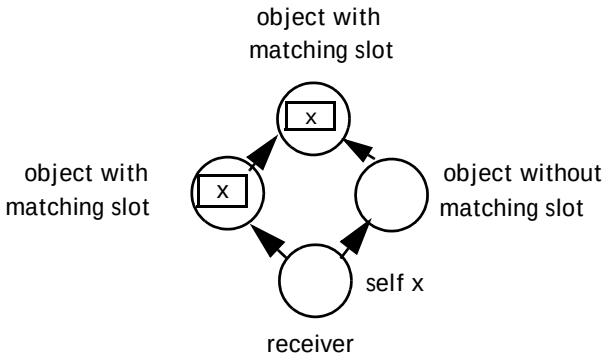


Figure 11. Unordered up to join: Under Chambers' proposed scheme, there would be no conflict in this case, since the first match precedes a join looking up the parent links. Under our old sender-path tiebreaker, there still could be a conflict if the sending method were held by any but the leftmost object. There is also a conflict under the current rules for Self.

Ungar had written about this danger for CPU designers, christening it "The Architect's Trap" (section 2.4). On the one hand, some lessons seem to require repetition. On the other hand, maybe we just gave up too soon.

4.3. UI1: Manifesting Objects on the Screen

In the spring of 1988, Bay-Wei Chang, then a graduate student at Stanford, took Ungar's programming languages class. He became interested in Self and was impressed when, during the final exam for the class, a video tape on the Alternate Reality Kit was shown. This was, in Chang's own words, "a cruel trick to play, as after the video I sat with my mouth agape for precious minutes." In the fall of 1988 Chang undertook an independent project working on version 1 of the Self UI, and officially joined the Self project in early 1989. Inspired by the Alternate Reality Kit, Ungar encouraged Chang to craft a user experience that would be more like the consistent illusion of a Disneyland ride than the formal system of a programming language. We wanted to construct the illusion that objects were real (see Figure 12). In May 1989, with the incorporation of Interviews/X and Pixrect primitives into Self, Chang was able to write a mock-up of a direct-manipulation Self user interface. By the end of 1989, this environment was further improved with fast arrowheads, better object labeling, and optimizations that included our own low-level routines to copy data and draw lines. As a result performance improved from 10 to 30 frames/sec. on a monochrome SPARCstation-1.

The original version of UI1 (written by Chang at Stanford in 1988-1989) had run on machines with monochrome frame buffers. By 1990, we had eight-bit frame buffers, although (as Ungar recalls) we had grayscale monitors and there was no hardware acceleration. Ungar realized that, by reducing our palette of colors (actually, grays), we could use colormap tricks to get smooth, double-buffered animation on the screen. We achieved 30 frames/sec. on a color SPARCstation with a graphics accelerator. By the end of one year (May 1990) this version of UI1 was working (see Figure 13).

As of 2006, colormaps have disappeared from most computers, so the reader may not know this term. A "colormap" is simply an array of colors. An image composed of pixels that use a colormap doesn't store the color information directly in the pixel, but rather stores the colormap array index for that color in the pixel. The key advantage of the colormap for animation effects arises from the simple reality that a window on the screen typically has about a million pixels, whereas a colormap has only 256 entries. Thus, a computer can run through this very short color map, changing the colors stored in various indices, to obtain a nearly instantaneous visual effect on millions of pixels. Consider a colormap with the color white stored at both index 0 and index 128. A screen image with pixel data that is all 0 except for a region with 128 appears to the user as entirely white. But when a program stores the color black at colormap index 128, a black region suddenly appears on the white background.

Suppose that a 256-color colormap is split into four identical parts, so that every entry from 0 through 63 is replicated three more times through the colormap indices. This limits the range of available colors to 64, but it frees up two "bit planes" for drawing: setting bit 7 in a pixel's value (effectively adding 128 to the data for that pixel) has no visual effect. Nor does setting bit 6. Suppose the colormap is suddenly modified so that all indices with bit 7 set to 1 are black. Black regions will instantaneously appear on the screen wherever pixels have values with

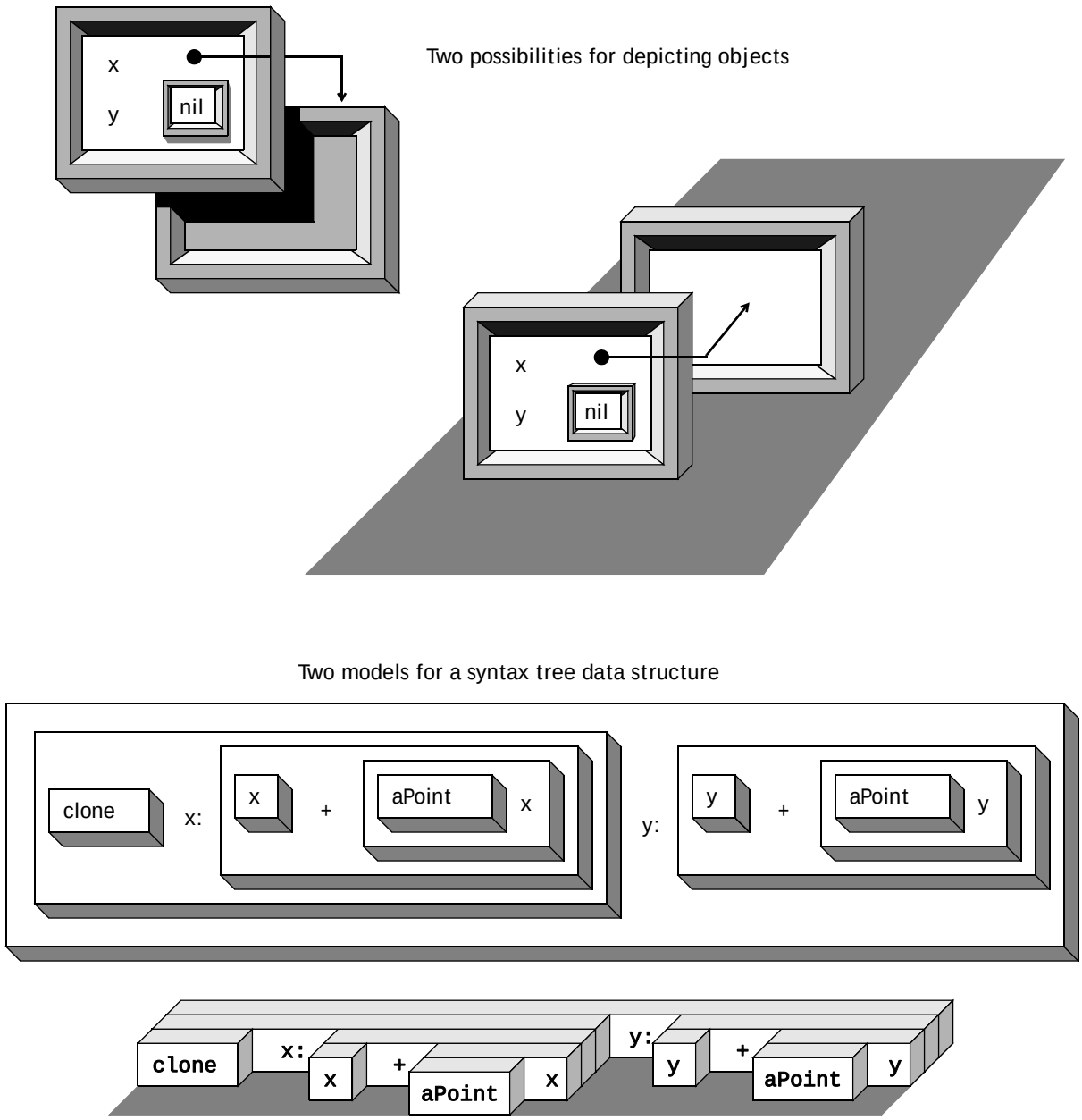


Figure 12. Visions of a Self user interface taken from a May 1988 grant proposal. Above, two possibilities for objects; below, two possibilities for a syntax tree. From the proposal: "We are interested in pursuing a style of interaction that can exploit what the user already knows about physical objects in the real world. For this reason, we call this paradigm *artificial reality.* For example, instead of windows that overlay without any depth or substance, we will represent objects as material objects, with depth, lighting, mass, and perhaps even gravity."

that bit set. Suppose then that the color map is restored except that now bit 6 is set to black. The first set of black regions disappears but a new set of black regions appear. Clearing bit 7 in the image data, drawing with bit 7, then changing the colormap appropriately makes yet another change appear on the screen. Alternately clearing and drawing with first bit 7, then bit 6, animated images can be made to appear over the background. The two bit planes are being used to achieve an animation drawn with one color. Because one plane remains visible while another, invisible plane is used for drawing, this scheme is an instance of what is termed a "double buffering" animation technique.

In UI1, Chang and Ungar carried this technique further by dividing up the frame buffer into two sets of one-bit planes and two sets of three-bit planes. The one-bit planes double-buffered the arrows, and the three-bit planes double-buffered the boxes. (The boxes needed three bits so they could have highlight and shadow colors.) The arrows were separated from the boxes to make it easier to depict the arrows as being in front of the boxes. At any given time, the colormap would be set so that one arrow and one box plane was visible. UI1 would then compute the next frame of arrows and boxes into the invisible planes, then switch the colormaps. Later at Sun, when we added dissolves[8], we would put each key frame into a separate plane and update the color-

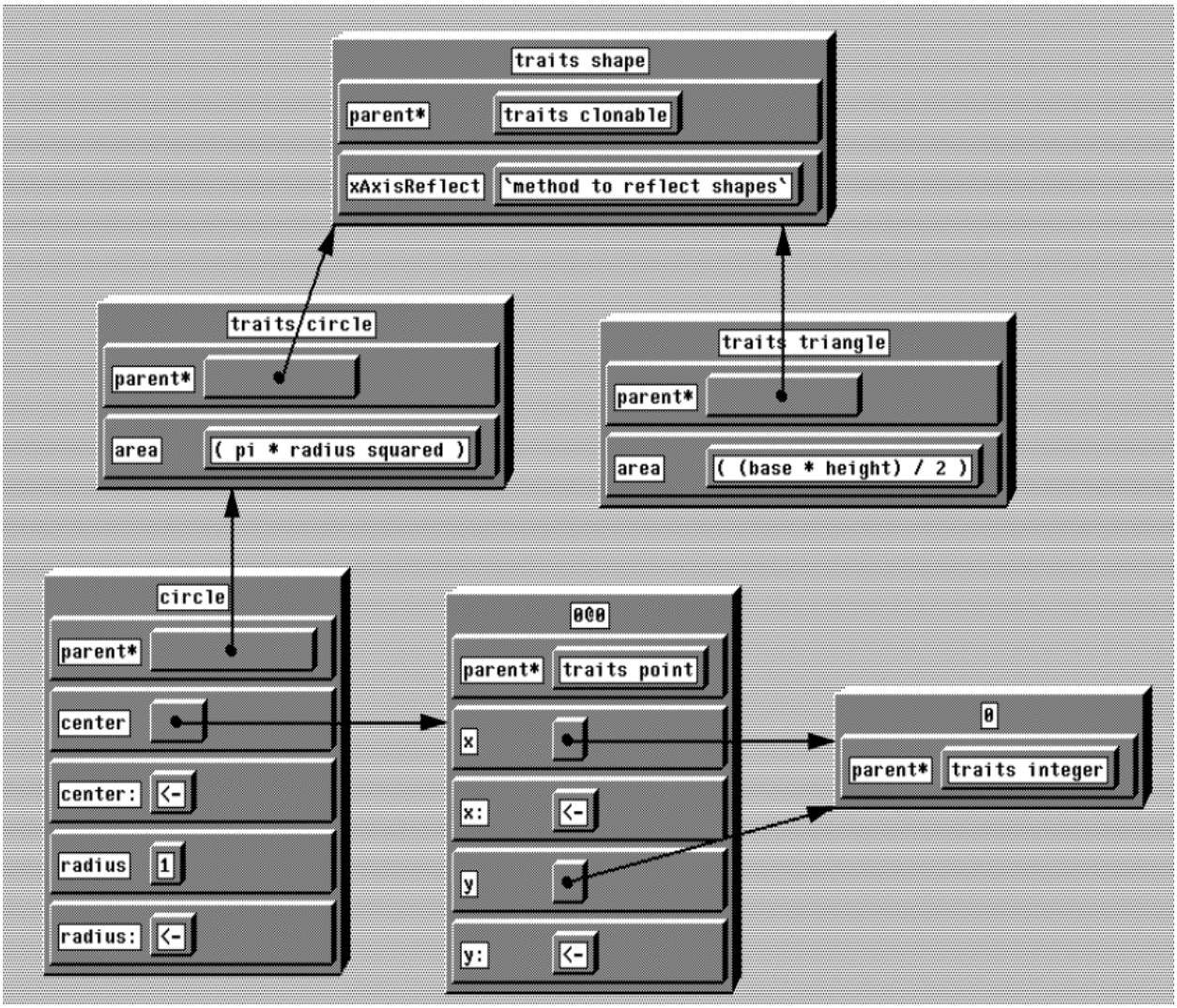


Figure 13. The original Self programming environment, the first version of UI1, was designed to be object-centered. Each box represented a Self object, and a pseudo-3D style attempted to convey a sense of physical reality. (Picture copied from [CU90a].)

map for each frame. This trickery enabled UI1 to display 20 to 30 frames/sec. smooth-looking animation on the hardware of 1991.

At this time, we were writing Self code with a text editor, and feeding the files to a read-eval-print loop. We even built a text-based source-level debugger with commands resembling those of the Gnu debugger, *gdb*. But we knew that eventually we wanted to live in a world of live objects—after all, we were inspired by Smalltalk! However, it was not until UI2 (section 5.3) and the Self transporter (section 5.5) that we could make the change, and even then at least one of the team members, Ole Agesen, still sticks to text editing. At this writing, we asked Agesen to recall why he kept using the older approach: he responded that he was in a rush to complete his thesis, partly in fear that the project would be canceled, so he didn't want to take the time to learn how to transition, nor take the risk of relying on as yet unproven technologies for his thesis work.

For UI1, we pushed hard on being object-centered; there would be nothing on the screen (except for pop-up menus) that was not an object. No browsers, no inspectors, just objects. It was the Self *language* that made this a reasonable approach. For example, to understand a Smalltalk program, one must understand the behavior as manifested by the inheritance hierarchy, as well as the state of all the variables in the current scope. The Smalltalk browser could show the inheritance story, but the variable values were held in several objects scattered at conceptually remote places in the system, and viewing them required other tools, such as the "inspector," unrelated to the Smalltalk inheritance hierarchy. But Self's use of message passing for variable access meant that the inheritance hierarchy of actual objects was all the programmer needed to see both behavior and state. And, as previously mentioned, to use the Smalltalk-80 browser, one had to learn the role of categories, method protocols, and the instance/class switch as well. But for a Self environment, Chang and Ungar needed only to build a good representation of a Self object, and that would serve most of the programmer's needs.

Unlike Smalltalk, in which one could have multiple inspectors on the same object, Self's UI1 allowed only one representation of the object on the screen. We were trying to preserve the illu-

8. A "dissolve" is a transition in which one frame smoothly changes into the next. Each pixel slowly changes from its value in the first frame to its value in the new frame.

sion that the picture on the screen *was* the object. An object was rendered using a pseudo-3D representation and had a context-dependent pop-up menu. Clicking on a slot would sprout an arrow to its contents. If the referenced object was not already on the screen it would be summoned. If it was already there, the arrow would just point to it. UI1's object-centrism was successful in helping us ignore the artifice behind the objects. About ten years later, we finally decided to compromise this principle because it was so handy to have a collection of slots from disparate objects. For example, one might want to look at all implementers of "display." We called such things *slices*, and included them in our later environment, UI2.

4.4. Reflecting with Mirrors

By unifying state and behavior and eliminating classes, the design of Self encapsulated the structure of an object. No other object could tell what slots some object possessed, or whether a slot stored or computed a result. This behavior is well-suited for *running* programs, but not for *writing* programs, which requires working with how an object is implemented. To build a programming environment, we needed some programmatic way to "look inside" of an object so that its slots could be displayed.

Ungar's first thought was to follow Smalltalk's practice and add special messages to an object to reveal this information. This would also have fit with Smith's idea of emulating physical reality, in which every object is fully self-contained. This architecture proved to be unworkable, since one could not even utter the name of a method object without running it. Ungar reasoned that Self needed a kind of ten-foot pole that would look at a method without setting it off. He used the word "mirror" for the ten-foot pole, both to connote smoke-and-mirrors magic and also to pun on the optical meaning of "reflection."

The mirror behaves like a dictionary whose keys are the names of the object's slots and whose contents are objects representing the slots. To display an object, the environment first asks the virtual machine for a mirror on the object. Following an object's slot through a mirror yields another mirror that reflects the object contained in the slot. In this fashion, once a mirror has been obtained, all of the information encapsulated in the mirror's *reflectee* can be obtained from the mirror. A method is always examined via its mirror, and is thus prevented from firing. By May 1990, we had read-only reflection (a.k.a. introspection) via mirrors.

Once we started thinking about mirrors, other advantages of this architecture became apparent. For example, since only one operation in the system creates a mirror, and since, to the virtual machine, a mirror looks slightly different than an ordinary object, introspection can be disabled by shutting off the mirror creation operation and ensuring there are no existing mirrors. Much later (ca. 2004), we exploited the mirror architecture to implement remote reflection for the Klein project, by implementing an object that behaved like a mirror but described a remote object [USA05].

Also, mirrors were a natural place to support the kinds of changes to objects that a programmer would effect with a programming environment. To minimize the extra complexity in the virtual machine, Ungar borrowed a page from functional programming. With the sole exception of the side-effecting `define` operation, all of the primitive-level reflective mutation operations created altered copies instead of modifying existing objects. For example, when the user changed a method on the screen, the programming environment would have to alter the contents of a constant slot, and this was a reflective operation. However there was no reflective operation that altered a constant slot in place; instead there was a functional operation that produced a new object with an altered slot. After obtaining this new object, the environment would invoke `define`, which would redirect all references from the original object to (a copy of) the new one. This design ensured that only the `define` operation[9] needed to invalidate any compiled code, since none of the others altered existing objects. The "copy of" was part of the define operation's semantics to optimize this operation when the old and new objects were the same physical size in memory. (In that case, the system could just overwrite the old object with the contents of the new.)

At the time, mirrors seemed merely a good design but not significant enough to publish. This system was working by the end of June 1991, and was used by UI1 to allow the user to change the objects on the screen. The VM was even able to update code that had been inlined, thanks to trapping returns and lazily recompilation. This was a milestone: the graphical programming environment was finally usable for real programming. Years later, Gilad Bracha, who was working at Sun and knew of the Self project work, thought it would be important to generalize and explain this design, and he and Ungar published a paper about the architectural benefits of mirrors [BU04].

After the project moved to Sun (in 1991) and Smith rejoined us, he pointed out that the benefits of mirrors came at the cost of uniformity. In thinking about models for systems, Smith always turned to the physical world, which does not support a distinction between direct and reflective operations. There is no difference between a physical object used directly or reflectively: it is the same physical object either way. Furthermore, Smith noted that there are many different types of reflective operations, and any attempt to distinguish between reflective and non-reflective operations was therefore certain to get it wrong in some cases or from some points of view. In fact, we sometimes do find it unclear whether a method should take a mirror as argument or the object itself.

Smalltalk, in contrast, placed many reflective operations in the root of the inheritance hierarchy, to provide reflection for every object. Something similar might have been done for Self, to avoid the dichotomy that was bothering Smith. However, by this point we were reveling in Self's support for lightweight objects that needed no place in the inheritance hierarchy, and it would have been impossible to reflect upon such objects without mirrors. Ungar still feels that reflective operations are of a different breed, while Smith still wishes they could be unified with ordinary operations.

Another approach to unifying invocation and access would have been to add a bit to a slot to record whether the slot was a method or data slot. Then, a special operation could have extracted a method from a method slot and put it into a data slot. This approach would have had its own problems: what would it mean to put 17 into a method slot? In our opinion, we never fully resolved whether a method should fire because it is a method or because it is in a special kind of slot. We also continue to wrestle with writing code in which it is unclear whether we should pass around objects or mirrors on them. There was also the efficiency loss in creating an extra object to do reflection. The performance penalty went unnoticed when we were using mirrors for a programming environment, but became problematic in the Klein system [USA05], which mirrors to convert a hundred thousand objects to a different representation. In this application the efficiency loss was so critical that a switch

9. This operation was inspired by, but not quite the same as, Smalltalk's "become:" operation.

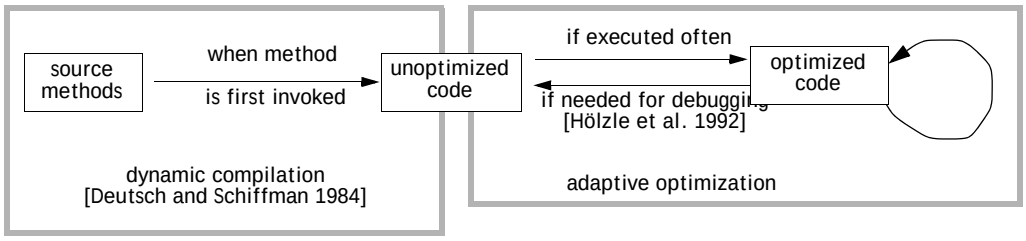


Figure 14. Compilation in the Self-93 system.

was added to the virtual machine to support a model in which a method could be stored and retrieved from a variable slot. Although Ungar feels that the preponderance of evidence weighs on the mirror and methods-firing-by-themselves side of the debate, Smith acknowledges clear advantages for mirrors but still harbors doubts, and this issue is not completely resolved.

Recall that, when we first saw Self objects on the screen at Stanford in 1988-1989, we realized that some objects had too many slots to view all at once: we needed some way to subdivide them, much like Smalltalk's method categories. To support this non-semantic grouping, *annotations* were added to the language. Via reflection, the system could annotate any object or any slot with a reference to an object. The virtual machine supported the annotation facility but ignored the annotation contents. It optimized the space required when all objects cloned from the same prototype contained the same annotations by actually storing the annotations in the maps. The programming environment (written in Self) used the annotations to organize an object's slots in (potentially nested) categories.

5. Self Moves to Sun and Becomes a Complete Environment

As Ungar and his students worked on mirrors in California, Smith was in Cambridge, England at Rank-Xerox EuroPARC where he had been working on a multi-user version of the Alternate Reality Kit. But his interest in Self and prototype-based languages persisted, and while in Cambridge he was able to work with Alan Borning and Tim O'Shea, who had connections with the PARC Smalltalk group. With these two plus Thomas Green and Moira Minoughan, also working at EuroPARC, he explored a few other language ideas [GBO99].

On his return to Xerox PARC at the end of 1989, Smith was amazed to find Self running so well, though a little concerned that it had acquired complexities such as multiple inheritance, mirrors, and annotations (which he felt were too much like Smalltalk's method protocols, having no runtime semantics). He decided to join Ungar and carry Self forward into a larger implementation effort. Smith had been thinking about subjectivity in programming languages, but further language work was becoming a harder sell to PARC management. The authors decided to take the Self ideas to other research labs. (We later returned to subjectivity in [SU96].) By of the end of June 1990, the Self team had given talks on the developing Self system at U.C. Berkeley, PLDI'90, and IBM Hawthorne Laboratories. In the fall of 1990, we considered moving to the Apple Advanced Technology Group, but—encouraged by Emil Sarpa, Bill Joy, and Wayne Rosing—decided to join Sun Microsystems' research labs. Self already ran on the SPARC processor and thus there was a chance to get a leg up in adoption. The labs were just being formed, and the Self project would be one of Sun Labs' first groups. In January 1991, the Self project joined Sun Microsystems Laboratories. Ungar's students (Craig Chambers, Bay-Wei Chang, Urs Hölzle, and Ole Agesen, the last graduate student to join the project) became consultants and over the years more researchers were hired to work on the project: John Maloney, Lars Bak, and Mario Wolczko.

5.1. More Implementation Work

As 1991 ended, the virtual machine encompassed 75,000 lines of code; in 1992, our first Apple laptop computers arrived and we started work on our first Macintosh port. By the end of 1992, Lars Bak had obtained a 5x speedup on the Sun computers for Self's browsing primitives (implementers, etc.) by rewriting the low-level heap-scanning code. He had also trimmed Self's memory footprint by 18%. The Macintosh port went slowly at first; it was not until January 1996 that Self ran (with an interpreter) on a PowerPC Macintosh.

Recall from section 4.1 that the third Self compiler ran the benchmarks pretty well but still compiled too slowly, and suffered from brittle performance. Hölzle took up the challenge. He had built polymorphic inline caches (PICs) [HCU91], and then proposed a new direction: laziness. He suggested that we build two compilers: a fast-and-dumb compiler that would also include instrumentation (such as counters in PICs), and a slow-and-smart compiler that would reoptimize time-consuming methods based on the instrumentation results (see Figure 14). The first time a method was run, the system used a fast-but-unsophisticated compiler that inserted an invocation counter in the method's prologue. As the method ran, its count increased and its call sites became populated with inline caches. Periodically, another thread zeroed out the counters. If a method was called frequently, its counter would overflow and the virtual machine would recompile and optimize it. However, because the method with the overflowing counter might have been called from a loop, the system would walk up the stack to find the root method for recompilation. After selecting the root, that method would be compiled with a slow-but-clever optimizing compiler that would exploit the information in the inline caches to inline callees. Finally, the set of stack frames for the recompiled methods would be replaced by a stack frame for the optimized methods (called "on-stack replacement"), and execution would resume in the middle of the optimized method.[10]

By the time Hölzle was through, his system, Self-93, ran well indeed, with almost no pauses for compilation [Höl94, HU94, HU94a, HU95, HU96]: in a 50-minute interactive graphical session, using a new metric that lumped together successive pauses, we found that, on a 28.5 MIPs, 40 MHz SPARCstation-2, two-thirds of the lumped pauses were less than 100ms, and 97% were less than a second [HU94a]. This system reduced the time to start the graphical user interface from 92 to 26 seconds. Not only were pauses reduced, but benchmarks sped up. This system ran a suite of six large and three medium-sized programs 1.5 times faster than the third-generation Self system.

10. In later years, when Ungar had to maintain and port the virtual machine single-handed, he would disable on-stack replacement to simplify the system and eliminate hard-to-reproduce bugs.

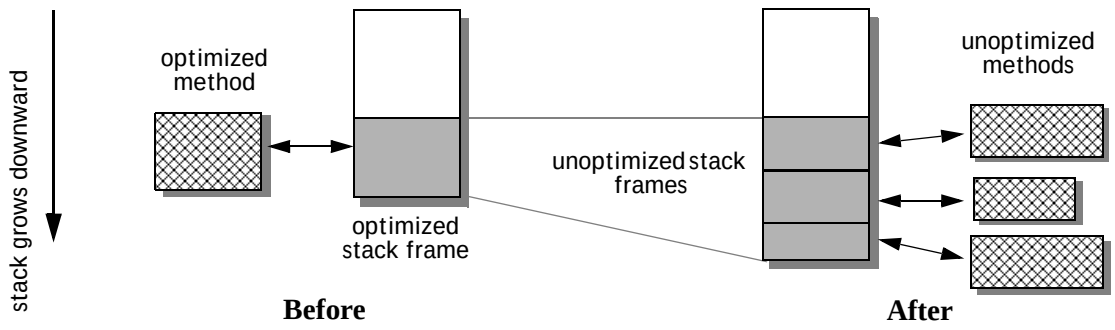


Figure 15. Transforming an optimized stack frame into unoptimized form.

As Hölzle was performing this feat of legerdemain, Ungar was worried about preserving the system's transparency. He wanted the system to feel like a fast interpreter, and that meant that if the user changed a method, any subsequent call to the changed method had to reflect the change. However, if the method had been inlined and if execution were suspended in the caller, when execution resumed the calling code would proceed to run the obsolete, inlined version of the callee. To remedy this problem, Ungar suggested on-stack replacement in reverse: replacing the one, optimized stack frame for many methods with multiple stack frames for unoptimized methods (see Figure 15). Hölzle brought this idea to life [HCU92], and Self's sophisticated optimizations became invisible to the programmer. We had finally realized our vision for the virtual machine: high performance for Self, a pure and dynamic language, combined with high responsiveness and full source-level debugging. However, Self's virtual machine required many more lines of C++ code (approximately 100,000), more complexity, and more memory than contemporary (but slower) Smalltalk virtual machines.

In August 1993, Mario Wolczko left the University of Manchester where he had been working on a Smalltalk multiprocessor, and joined our project and performed some space engineering, cleaned up the representation of debugging information, refactored the implementation of maps, and fixed a large number of bugs. He also implemented a feedback-mediated control system that managed old-space collection and heap expansion. The policy was implemented in Self, with only the bare bones mechanism in the virtual machine. This work was ahead of its time, and we are unaware of its match in current systems.

Ole Agesen was the last PhD student in the Self project, graduating in 1996. He worked on many portions of the Self system, including an interface to the dynamic linker for calling library routines, and for his dissertation built a system that could infer the types of variables in Self programs, despite Self's lack of type declarations. Agesen's work showed how to prune unused methods and data slots from a Self application [APS93, AU94, Age95, AH95, Age96].

5.2. Cartoon Animation for UI1

With the Alternate Reality Kit, Smith wanted to deliver a feeling of being in a separate world by having lots of independent things happening in a physically realistic and often subtle way. He tried for realistic graphics, including shadows and avoiding outlines, but never even thought about cartoon animation techniques, in which fidelity to physics is less important than emphasizing certain motions through physically implausible accelerations and deformations. In building the UI1, however, the Stanford group believed that a physical feel would be a help to the programmer, and after seeing ARK, were convinced that animation should feature heavily in any Self user interface.

When he moved Sun in 1991, Ungar had been watching a lot of Road-Runner and Popeye cartoons with his five-year-old son, Leo. It occurred to Ungar that the animation techniques he saw in the cartoons could be applied to dynamic user interfaces. Since he also had a VCR with an exceptionally agile jog-shuttle feature, he was able to review many scenes one frame at a time. Smith and neuroscientist Chuck Clanton (who was then consulting at Sun) were also fascinated by animation.

Coincidentally, in 1990 Steven Spielberg and Warner Brothers put Tiny Toon Adventures on the air, a show that strove to recreate the style and quality of the classic Warner Brothers cartoons in the late 1930s through early 1950s. In our first year at Sun, Smith and Ungar would stop work every day at 4:30 to watch these cartoons and then dissect them. The cartoons inspired us to read Thomas and Johnson's book *Disney Animation: The Illusion of Life* [TJ84] and *Road-Runner*, director Chuck Jones' autobiography [Jone89]. We would stare in fascination at each frame of the Road-Runner zooming across the screen. We were struck by the clarity with which Jones could show a scrawny bird and an emaciated coyote crossing the entire width of a movie screen in only a handful of frames by using motion blur and slowly dissipating clouds of dust. This combination of speed and legibility stood in stark contrast to the leisurely pace of many animated computer interfaces of the time in which each small change of position was painstakingly redrawn. Even some of the best interface research at the time used uniform, unblurred motion [RMC91]. We starting thinking about the role motion blur could play in graphical computer interfaces. Smith asked the key question: "If you could update the screen a thousand or a million times a second, would you still need motion blur?" These explorations led us to an understanding of how to bridge the gap between cartoons and interfaces, and how to make changes more legible *without* slowing things down.

There is an interesting difference between Smith's use of animation in the Alternate Reality Kit and cartoon animation. In any animation, the incoming light creates patterns on the viewer's retina that trickle up the nervous system and reach the higher levels of cognition after considerable processing. In ARK, Smith's goal was to create a sense of realism by replicating the retinal patterns caused by real-world objects. In contrast, cartoon animation is more concerned with getting the viewer's higher cognitive levels to perceive objects and motion. In depicting a bouncing billiard ball, a cartoonist might use a "squash and stretch" around the moment of impact, thereby making the bounce clearly legible to viewers. A literal moment by moment capture of the human retina watching an actual billiard ball bounce would reveal a blur that only somewhat resembles cartoon-style stretch. Although both kinds of animation

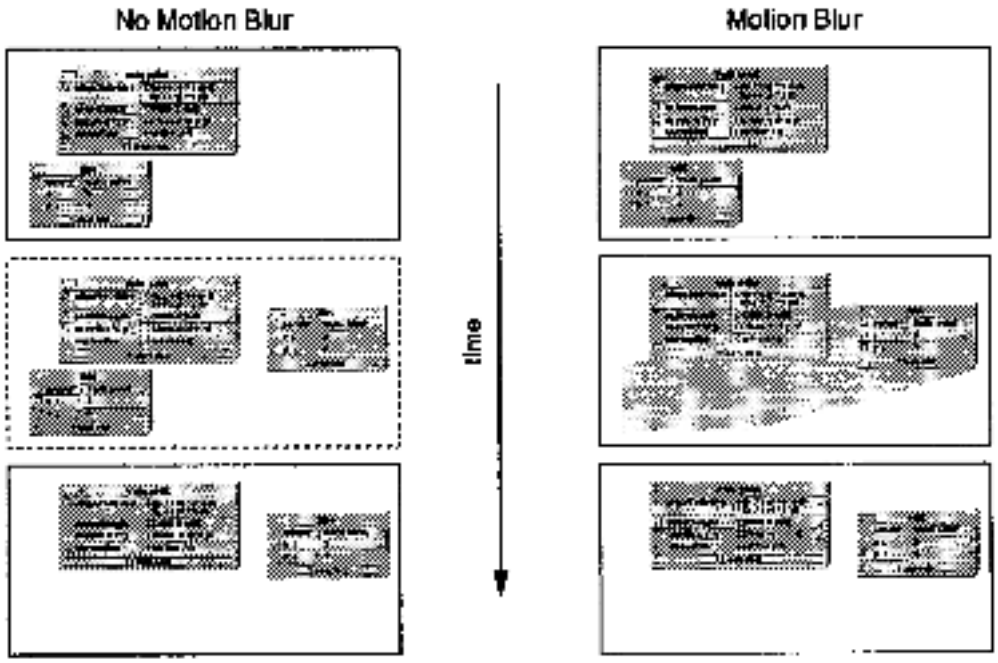


Figure 16. When objects are moved suddenly from one position to another, it can seem as if there are two instances of it on the screen at the same time. The eye sees something like the middle frame of the "no-motion-blur" figure, even though such a frame doesn't actually ever appear on the screen. Motion blur reduces this effect and gives a visual indication of the object's travel, so that it is easy to see which object moved where.

have the same goal, cartooning trades the literal replication of sensory inputs for better legibility at higher levels of cognition.

Recall that colormap trickery enabled UI1 to display 20 to 30 frame-per-second smooth-looking animation on 1991 hardware (section 4.3). Once the basic techniques were implemented, we set about using cartoon animation everywhere we could to improve UI1's feel and legibility. (A popular phrase at the time was: "moving the cognitive burden from the user to the computer.") In those days, windows, menus, and dialog boxes would just appear in a single frame and vanish just as abruptly. (As of 2006, they still do in many window systems.) But, we believed that every abrupt change startled the user, and forced him or her to involuntarily shift his or her gaze. So, we strove to avoid bombarding the user with abruptly changing pixels. Just as the Road Runner would enter the frame from some edge, every new Self object appearing on the screen would drop in from the top, slowing down as it did, and wiggling for an instant as it stopped. Every pop-up menu would smoothly zoom out, then the text would fade in. We became excited about the user experience that was emerging. Ungar came in every day over one Christmas break (probably 1991) to get good-looking motion blur into the system.

Figure 16, taken from [CU93], illustrates motion blur. Chang realized that objects should move in arcs, not straight lines, and also suggested that an object wiggle when hit by a sprouted arrow (see Figures 17 through 19). Ungar played with the algorithm and its parameters until he got the wiggle to look just right. It would have been much more difficult to break this new ground with any other system: he needed both Self's instant turnaround time to try ideas freely, and also its dynamic optimizations so that the animation code would run fast enough.

Table 1 summarizes UI1's cartoon animation techniques. By June 1992, we had implemented all of our cartoon animation, including motion blur, menu animation, and contrast-enhancing highlighting of menu selection. Chang had also started video taping users to evaluate UI1, taping eight subjects before completing his dissertation. His work on cartoon animation and its effect on users' productivity became a part of his dissertation and was presented in several conferences [CU93, CUS95, Chan95]. Although Ungar also wanted to measure the effect of animation on the number of smiles on users' faces, we never did. Now, in 2006, many of these techniques can be seen in commercial systems and web sites.

5.3. UI2 and the Morphic Framework

When Smith rejoined the group on the move to Sun, he was thinking of ways to push the UI1 framework in additional directions. He felt the analogy to physical objects was not taken far enough in most user interfaces. For programming purposes, he felt that the analogy meant that every object should be able to be taken apart, even as it is running. This physicality was after all the goal of the language-level objects, and with a tight correspondence between on-screen object and language-level object, the deconstruction of live on-screen objects seemed to complete the paradigm.

In working with physical objects, one is free to take them apart and rearrange parts even while the universe continues to run: there is no need to jump to a special set of tools in a different universe. Physical objects do not support a use/mention distinction: the hammer in use is the same as the hammer examined for improvement, repair, or other modifications. So Smith wanted to be able to pick up a scroll bar from a running word processor and reattach it to some other application.

Figure 17. On a click, a menu button transforms itself from a button into the full menu. After a selection has been made, it shrinks back down to a button. (In this and other figures, only a few frames of the actual animation are shown. These figures taken from [CU93].)

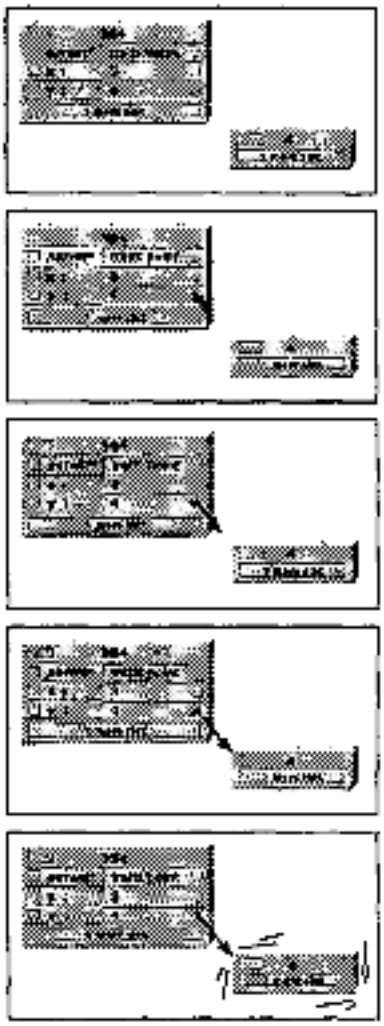


Figure 18. Arrows grow from their tail to hit their target. The target reacts to the contact with a small wiggling jolt (here suggested by a few lines). Arrows also shrink back down into their tail.

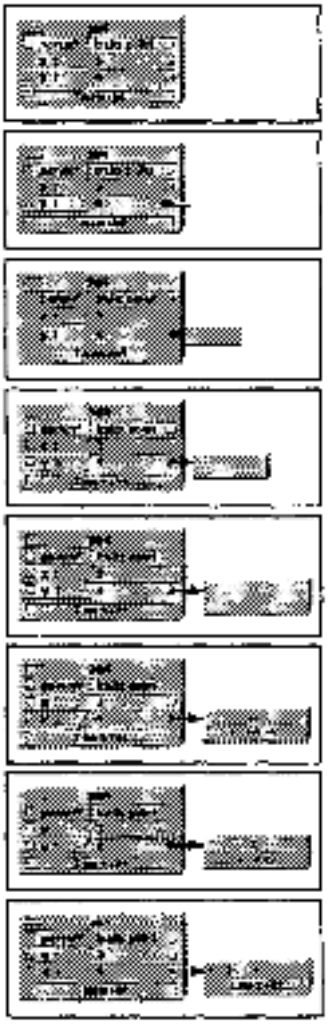


Figure 19. Objects grow from a point to the full-size object; any connecting arrow grows smoothly along with the object. Currently, text does not grow along with the object, instead fading in smoothly on the fully grown object.

The UI1 interface contained only representations of objects: there was no special support for building conventional GUIs with elements such as scroll bars, text fields, buttons and sliders. Mainly for this reason, Smith started an effort within the group to build UI2, a new framework that would retain the basic object representation idea and animation techniques of UI1, but add the ability to create general GUIs [MS95, SMU95]. In keeping with the language concepts of malleability and concreteness, within UI2 it would be possible to take objects apart directly: thus a direct copy-deconstruct-reconstruct method would be an important part of building new GUIs based on recognizable GUI widgets.

John Maloney, hired at this time, began creating much of the UI2 framework, which came to be called the Morphic framework. The Morphic framework enhanced the sense of direct manipulation by employing two principles we called *structural reification* and *live editing*.

Structural Reification. We decided to call the fundamental kind of display object in UI2 a "morph," a Greek root meaning essentially "physical form." Self provides a hierarchy of morphs. The root of the hierarchy is embodied in the prototypical morph, a kind of golden-colored rectangle. Other object systems might choose to make the root of the graphical hierarchy an abstract class with no instances, but prototype systems usually provide generic examples of abstractions. This is an important part of the structural reification principle: there are no invisible display objects. The root morph and its descendants are guaranteed to be fully functional graphical entities. Any morph inherits methods for displaying and responding to input events that enable it to be directly manipulated.

Table 1: Summary of UI1 Cartoon Animation Techniques (from [CU93])

Technique	Principle	Examples from Cartoons	Examples from the Self Interface
Solidity	solid drawing	• Parts of Snow White's dwarves may squash and stretch, but they always maintain their connectedness and weight	• Objects move solidly • Objects enter screen by traveling from off screen or growing from a point • Menus transform smoothly from a button to an open menu • Arrows grow and shrink smoothly • Transfer of momentum as objects respond to being hit by an arrow
	motion blur	• Road Runner is a blue and red streak	• Stippled region connects old and new locations of a moving object
	dissolves	• n/a	• Objects dissolve through one another when changing layering
Exaggeration	anticipation	• Coyote rears back onto back leg before chasing after Road Runner	• Objects preface forward movement with small, quick contrary movement
	follow through	• Road Runner vibrates for an instant after a quick stop	• Objects come to a stop and vibrate into place • Objects wiggle when hit by an arrow
Reinforcement	slow in and slow out	• Coyote springs up from ground, with fastest movement at center of the arc	• Objects move with slow in and slow out • Objects and arrows grow and shrink with slow in and slow out • Objects dissolve through other object with slow in and slow out • Text fades in onto an object with slow in and slow out
	arcs		• Objects travel along gentle curves when they are moving non-interactively
	follow through		• Objects do not come to a sudden standstill, but vibrate at end of motion

In keeping with the principle of structural reification, any morph can have "submorphs" attached to it. A submorph acts as if it were glued to the surface of its hosting morph. Composite graphical structure typical of direct-manipulation interfaces arises through the morph-submorph hierarchy. Many systems implement composition using special "group" objects, which are normally invisible. But because we wanted things to feel very solid and direct, we chose to follow a simple metaphor of sticking morphs together as if with glue.

A final part of structural reification arose from our approach to laying out submorphs. Graphical user interfaces often require subparts to be lined up in a column or row: Self's graphical elements are organized in space by "layout" morphs that force their submorphs to line up as rows or columns. John Maloney was able to create efficient "row and column morphs" as children of the generic morph that were first-class, tangible elements in the interface. A row morph holding four buttons aligned in a row is at the bottom of the ideal gas simulation frame in Figure 22. Row or column morphs embody their layout policy as visible parts of the submorph hierarchy, so the user need only access the submorphs in a structure to inspect or change the layout in some way. The user who did not care about the layout mechanism paid a price for this uniformity, and was confronted with it anyway while diving into the visual on-screen structures.

Live Editing. Live editing simply means that at any time the user can change any object by manipulating it directly. Any interactive system that allows arbitrary runtime changes to its objects has some support for live editing, but we wanted to push that to apply to the user interface objects directly. The key to live editing is UI2's "meta menu," a menu that pops up when the user holds down the third mouse button while pointing to a morph. The meta menu contains items such as "resize," dismiss," and "change color" that let the user edit the object directly. Other menu elements enable the user to "embed" the morph into the submorph structure of a morph behind it, and give access to the submorph hierarchy at any point on the screen.

Lars Bak did a lot of work to create the central tool for programming within UI2, the object "outliner," analogous to the Smalltalk object inspector. (We were in part inspired by "MORE," an outlining program we had recently started using.) The outliner shows all of the slots in an object and provides a full set of editing facilities. With an outliner you can add or remove slots, rename them, or edit their contents. Code for a method in a slot can be edited. Access to the outliner through the meta menu makes it possible to investigate the language-level object behind any graphical object on the screen. The outliner supports the live-editing principle by letting the user manipulate and edit

slots, even while an object is “in use.” Figure 20 shows an outliner being fetched onto the screen for an ideal gas simulation.

Recall that popping up the meta menu is the prototypical morph’s response to clicking the third mouse button. All morphs inherited this behavior, even elements of the interface like outliners and pop-up menus themselves. But our Self pop-up menus were impossible to click on: one found them under the mouse only when a mouse button was already down. Releasing the button in preparation for the third button click caused a frustrating disappearance of the pop-up. Consequently, we provided a “pin down” button that, when selected, caused the menu to become a normal, more permanent display object. The mechanism was not new, but providing it in Self enabled the menu to be interactively pulled apart or otherwise modified by the user or programmer. It is interesting to compare this instance of the use-mention problem and solution with the analogous use-mention issue that faced us at the language level: that of accessing a method object in a slot without running the method. There, the solution was to introduce a special mentioner object, the mirror (see section 4.4).

Live editing is partly a result of having an interactive system, but it is enhanced by user interface features that reinforce the feel that the programmer is working directly with concrete objects. The example running through the rest of this section will clarify how this principle and the structural reification principle help give the programmer a feeling of working in a uniform world of accessible, tangible objects.

Suppose the programmer (or motivated user) wishes to improve an ideal gas simulation by extending the functionality and adding user interface controls. The simulation starts simply as a box containing “atoms” that bounce around inside. Using the third mouse button, the user invokes the meta menu to select “outliner” to get the Self-level representation of the object (Figure 20). The outliner makes possible arbitrary language-level changes to the ideal gas simulation.

Now the user can start to create some controls right away. The outliner has slots labeled “start” and “stop” that can be converted into user interface buttons by selecting from the middle-mouse-button pop-up menu on the slot. Pressing these buttons starts and stops the bouncing atoms in the simulation. In just a few gestures the user has gone through the outliner to create interface elements while the simulation continues to run.

The uniformity of having “morphs all the way down” further reinforces the feel of working with concrete objects. For example, the user might wish to replace a textual label with an icon. The user begins by pointing to the label and invoking the meta menu. The menu item labeled “submorphs” lets the user select which morph in the collection under the mouse to denote (see Figure 21). The user the can remove the label directly from the button’s surface. In a similar way, the user can select one of the atoms in the gas tank and duplicate it; the new atom can serve as the icon replacing the textual label. Structural reification is at play here, making display objects accessible for direct and immediate modification.

As mentioned above, all the elements of the interface such as pop-up menus and dialog boxes are available for reuse. Say the user wants the gas tank in the simulation to be “resizable” by the simulation user. The user can create a resize button for the gas tank simply by “pinning down” the meta menu and removing the resize button from it. This button could then be embedded into the row of controls along with the other buttons (see Figure 22).

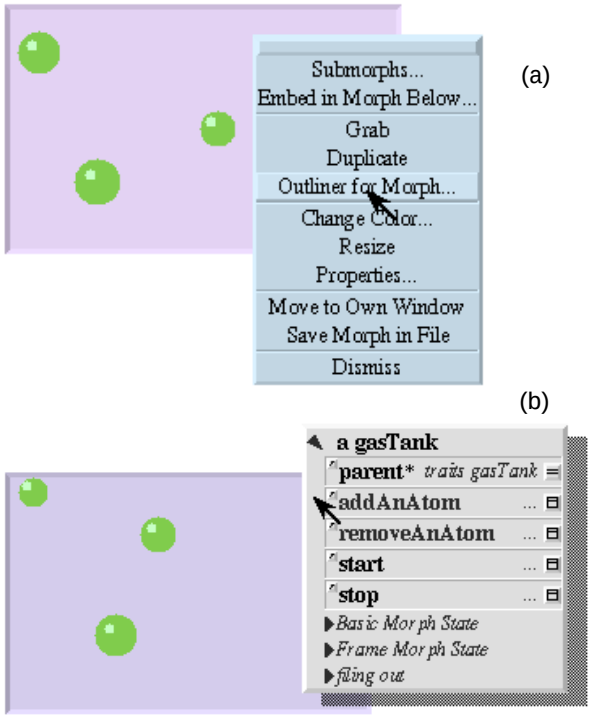


Figure 20. In UI2 the user pops up the meta menu on the ideal gas simulation (a). Selecting “outliner” displays the Self-level representation, which can be carried and placed as needed (b). (The italic items at the bottom of the outliner are slot categories that may be expanded to view the slots. Unlike slots, categories have no language level semantics and are essentially a user interface convenience.)

During this whole process, the simulation can be left running: there is no need to enter an “edit” mode or even to stop the atoms from bouncing around. The live editing principle makes the system feel responsive, and is reminiscent of the physical world’s concrete presence.

5.4. From UI2 to Kansas

Smith was fascinated by shared spaces (we might now call them “shared virtual realities”) and had explored with a shared version of his Alternate Reality Kit during his year at Rank-Xerox EuroPARC (1988-1989) [GSO91, SOSL97, Setal93, Setal90, Smi91]. After he and Ungar joined Sun and the UI2 framework was underway, Smith decided to make UI2 into a shared world in which the team could work together simultaneously. The transformation took only a day or two of diligent work, thanks in part to the fact that the system was built on top of the X windowing system, and of course in part to the fact that Self was intended to be a flexible system allowing deep, system-wide changes. The idea was to transform a single Self world with a single display into a single Self world with potentially many displays that could be anywhere on the network. Thus, whenever a morph displayed itself, rather than merely use the local X display window, the code would iterate over a list of several display windows, some of them remote. Moreover, the entire list of windows was queried for events to be dispatched to appropriate objects in the central Self world.

In addition, each remote window had an associated offset so that, although several users could be in the space at once, they could be shifted to individual locations. Because the resulting

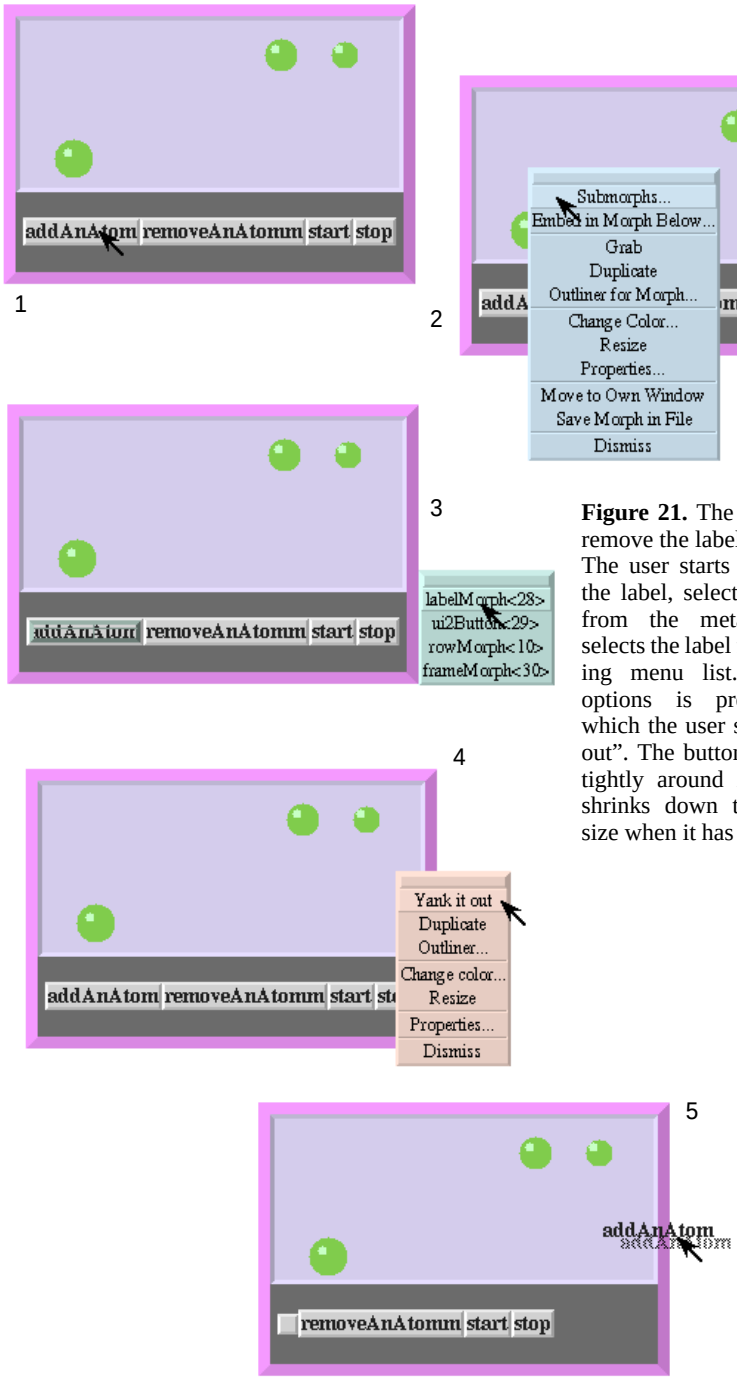


Figure 21. The user wishes to remove the label from a button. The user starts by pointing to the label, selects "submorphs" from the meta menu, and selects the label from the resulting menu list. A menu of options is presented, from which the user selects "yank it out". The button, which wraps tightly around its submorphs, shrinks down to a minimum size when it has no submorphs.

effect was to put many people in a single, vast flat space, we named the system "Kansas."

When Kansas was running, an error in the display code could cause all the windows to hang: the other threads of the underlying Self virtual machine would continue to run, though the display thread was suspended. We decided the right thing to do here was to open up a new shared space, "Oz," that would be created by the same virtual machine that created Kansas, but would be a new (mainly empty) world containing only a debugger on the suspended thread. The users, finding themselves suddenly "sucked up" into the new overlaying world of Oz, could collaboratively debug and fix the problem in Kansas, then, as a final act in Oz, resume the suspended thread so that normal Kansas life might resume. Much of the work for this was done by Mario Wolczko; Smith, Wolczko, and Ungar wrote a description for a special issue of the *Communications of the ACM* on debugging [SWU97].

UI1 was beautiful; its use of cartoon animation techniques gave a smooth and legible appearance. However, it gave no help to the user who wanted to either dissect or create a graphical object. We wanted to remedy this shortcoming in Morphic, and replicating UI1's beauty took a back seat to architectural innova-

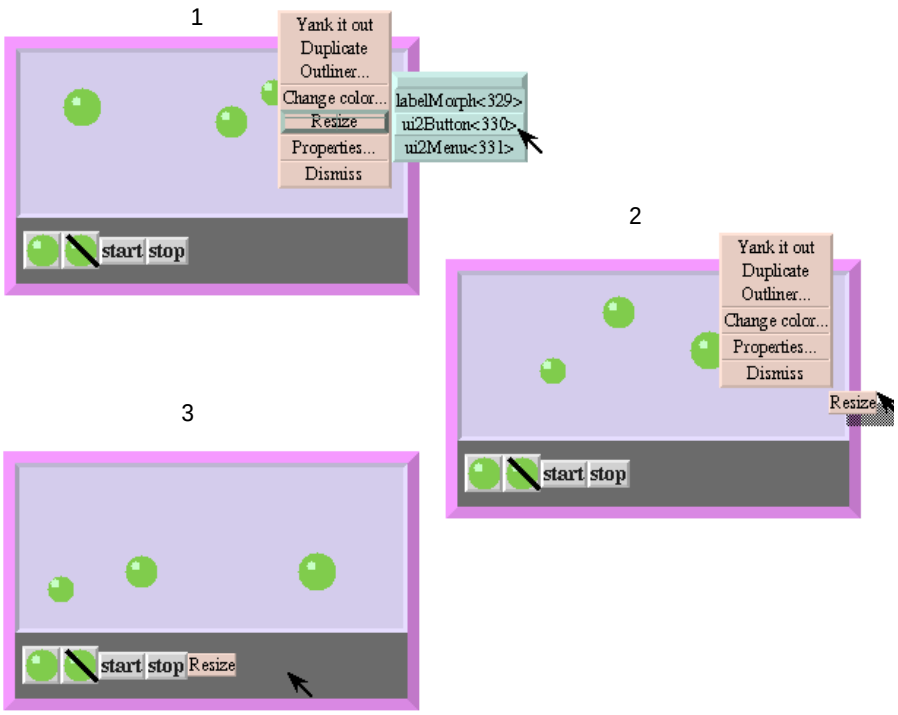


Figure 22. The environment itself is available for reuse. Here the user has created the menu of operations for the gas tank, which is now a submorph of the surrounding frame. The user has "pinned down" this menu by pressing the button at the top of the menu, and can then take the menu apart into constituent buttons: here the user chooses the resize button for incorporation into the simulation.

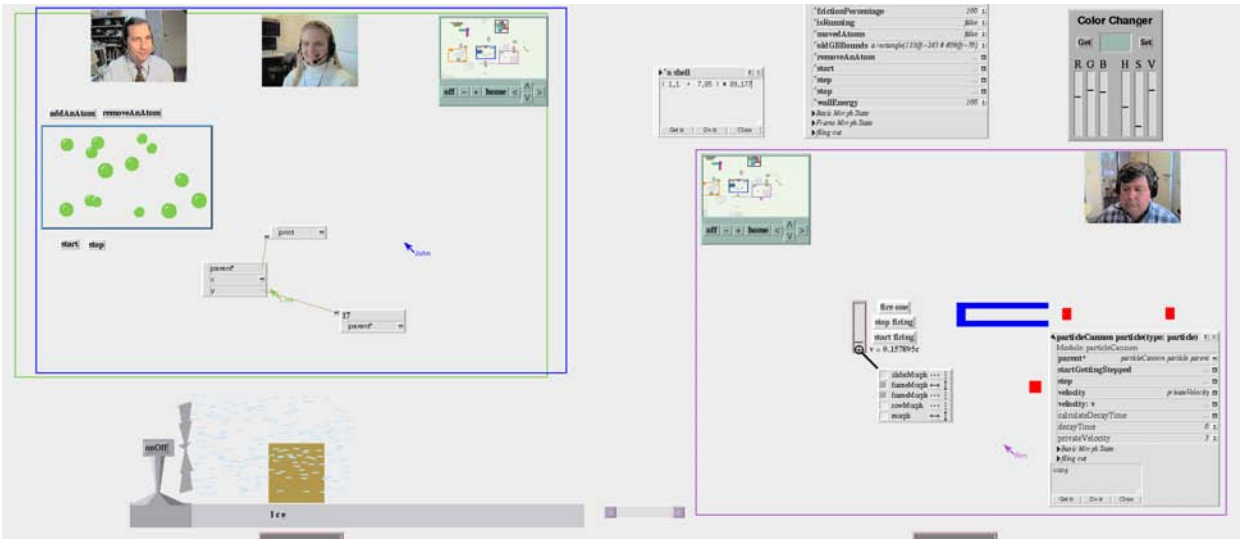


Figure 23. The Kansas shared space version of UI2. Here three users are shown, two of whose screens largely overlap so they can see each other and work on a common set of objects; the third user to the right is by himself. Video images from desktop cameras are sent over the network to appear on special objects near the top of each user's screen boundary. Users can be aware that other users are nearby thanks to audio from each user that diminishes in volume with distance, and to the miniature "radar view" tools that give an overview of the nearby extended space (a radar view can be seen in the upper left corner of the rightmost user's screen). The radar view can be used to navigate through the larger space as well.

tion. With the exception of slow-in and slow-out, cartoon animation techniques such as smooth dissolves, motion blur, anticipation, follow-through, and "squash and stretch" were never incorporated into the Morphic framework.

After the Self project ended in 1995, Smith became interested in distance learning. By 1998, he had added desktop video and desktop audio to the system, as depicted in Figure 23. In 1998 and 1999, Smith used this audio- and video-link-enabled Kansas in a series of experiments comparing small co-present study groups with small study groups in Kansas who communicated using the audio and video links. In these studies, groups of five to seven students in the same course would not attend lectures at all, but rather gather (either in the same room or in Kansas) to discuss a video tape of the classroom lectures. The experiments were carried out at California State University at Chico, approximately 200 miles north of the Sun Labs campus. A special Kansas world of Self objects was installed in a network at Chico, and the Sun researchers could "drop in" remotely, and even reprogram the system while a study session was in progress. The results of these experiments were that there were no significant difference between student grades, although there were some behavioral differences [SSP99, Setal99].

The audio- and video-link technologies were specific to Sun workstations; since we wanted Self to run on a wider set of platforms, they were never part of the mainstream Self system. But users of Self today can share their screens to work together. In normal use, Self programmers seem to prefer having their own private world of Self objects, which discourages routine use of the Kansas features, though they are still often useful for remote demos and collaborative debugging or development sessions. (In fact, Ungar has been recently using it to work with a collaborator 3,000 miles away.)

5.5. The Transporter

Inspired by Smalltalk and then ARK, we wanted to submerge the Self programmer in a world of live objects as opposed to some text editor. In fact, from the start of Self in 1986, we hoped that Self's prototypes would feel even more alive than Smalltalk's classes. Once we had a decent virtual machine and UI1, we could experiment with this idea: we could create objects, interactively add slots and methods, try them out, and instantly change them.

Although the idea of programming in a sea of live objects was inspired by Smalltalk, Self was not Smalltalk, and the differences caused problems. In Smalltalk, the programmer creates classes, instantiates them, changes methods, and inspects objects. A Smalltalk method is always part of a class, a class is a special kind of object, and every class (by convention) resides in the same spot: the System Dictionary. So, a Smalltalk program can usually be considered a collection of class definitions, including any methods. Since a Smalltalk object is created by instantiation, all initialization had to be done programmatically, and initialization is not much of a special problem. But in Self, there are no "class" objects. There is no one System Dictionary. Any object may serve as a namespace, and any object may hold methods. Objects are typically created by copying prototypes, and initialization code is frowned upon, as the prototype is already supposed to be initialized, functional, and prototypical. In fact, the prototype ideal (as Smith used to say) is to always have everything initialized so that every prototype can be functional as is. But this view means that the state of objects is an integral part of a "program." Consider the following example: Suppose Alice, using her own object heap, writes a program that she wishes to give to Bob. Bob will typically be using his own object heap, and so needs to incorporate Alice's additions and changes. In Smalltalk, the additions and changes that were typically a part of a program were restricted, but in Self, the problem amounted to recreating arbitrary changes to objects.

Up to around 1992, whenever we wanted to "get serious" so as to share our work with the group, we wrote Self programs in a text editor, then read them in and debugged them. As we debugged, we had to either change the file and reread it, or change both the file and the running environment. This was painful. We needed a system that would turn arbitrary sets of objects and slots into a text file that could then be read in to another world of objects. To meet this need, Ungar started his last major effort in the Self project, the transporter [Ung95].

Ungar realized that Self "programs" involved adding slots to or modifying slots in existing objects and thus the transporter would have to operate at the level of individual slots. Slot a might be part of one "program" and slot b another, even though both were in the same object. Since extra information was needed that was not part of the execution model, that information was added to the system around 1994 by extending Self's existing annotations. Each slot was annotated with the name of the source file to which it would be written. At first, Ungar tried to write a system that would infer other data, such as the proper initialization for a slot when it was subsequently read in. After many unsuccessful attempts, it became clear that inference would not work, and more information would be needed in the annotations. For example, each slot had to be annotated with initialization instructions: should it be initialized to whatever it originally contained or to the results of some expression? At the end, Ungar came to a fundamental realization: what was later dubbed "orthogonal persistence" [AM95] was, at least in this context, a flawed concept. Simply making a set of slots persistent is easy. But installing those slots into another arbitrary world of objects so that they function as they did in their original home is difficult, and probably even impossible in the general case. The task at least requires more information than what is needed for the slots merely to function in a live image. The programmer has to provide information that will enable the slots to function as the programmer intends (see Figure 24). To keep the burden of specifying the extra information for the transporter as light as possible, Ungar integrated it into the programming environment in such a way that it would be easy for a programmer to "push a button" and save a program as a source file that could then be read in to another user's heap of objects.

In June of 1994 the transporter was finished and the programming environment was augmented with affordances for the extra information. The Self team had moved its 40K lines of Self code, comprising data and control structures, the user interface, and the programming environment, to the Self transporter; in other words, we were all (but Agesen) doing our Self programming inside the graphical programming environment. The Self team made a leap from programming in text editors to programming in a live world, and then transporting the results to text files for sharing with others.

5.6. Significant Events While at Sun

The first Self release, 1.0, had occurred at the end of September 1990 and went to over 100 sites. The next release, 1.1, came at the end of June 1991 and went to over 150 sites. It featured a choice of compilers, and support for lightweight processes. Released in August 1992, Self 2.0 featured the sender-path tie-breaker and shared-part privacy rules. It also introduced full source-level debugging of optimized code, adaptive optimization to shorten compile pauses, lightweight threads within Self,

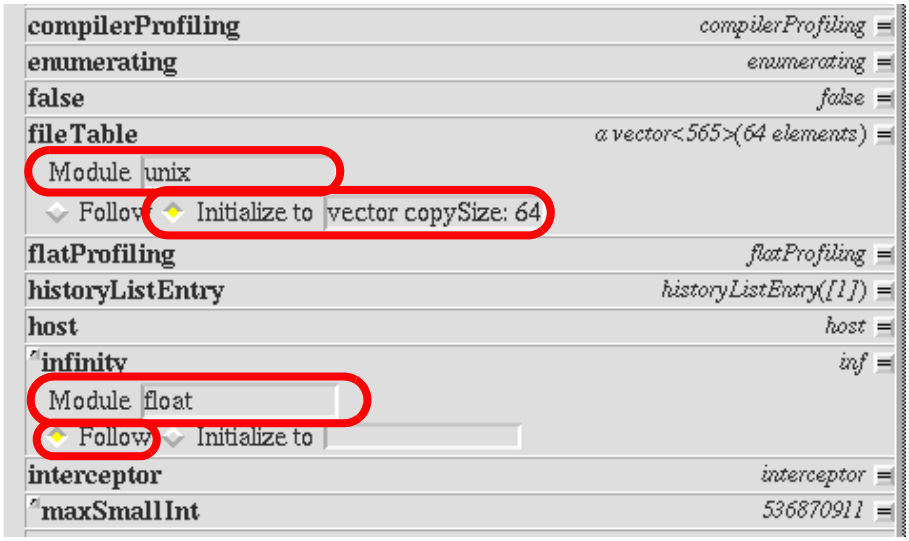


Figure 24. Annotations for transporting slots.The ovoid outlines have been added to show the module and initialization information for two slots (named fileTable and infinity) in a Self object.

support for dynamically linking foreign functions, support for changing programs within Self, and the ability to run the experimental Self graphical browser under OpenWindows. Self Release 2.0 ran on Sun-3s and Sun-4s, but no longer had an optimizing compiler for the Motorola 68000-based Sun-3 (and therefore ran slower on the Sun-3 than previous releases). Self 3.0, featuring Hölzle's adaptive optimization and the simplified multiple inheritance and privacy rules, was released January 1, 1994. In 1995, we felt we had achieved a fully self-contained system: we had an elegant language with a clever implementation unified with a novel user interface construction / programming world. Having made the final determination of which features to implement for the release by voting with hard candy (see Figure 25), we released Self 4.0 into the larger world as a beta in February 1995, and in final form July 1995. (See appendix 11 for the actual release announcements.)

During this 1991-1995 period at Sun, we felt that Self was gaining a following, especially within the academic community. Craig Chambers and Ungar gave a tutorial at the 1992 OOPSLA conference describing the various Self implementation techniques that sold out. At the 1993 OOPSLA conference, a demo of the Morphic framework and Self programming environment proved so popular that we had to schedule a second, then a third showing due to overflowing crowds. That same year, we presented the project to Sun CEO Scott McNealy, who enthused about Self being "a spreadsheet for objects" but cautioned us about "fighting a two-front war."[11]

At one of the OOPSLA conferences in the late 1980's, Ungar had met Ole Lehrmann Madsen. Madsen, a former student of Kristen Nygaard, was one of the designers of the Beta language [MMN93], a professor at Århus University and the advisor of Ole Agesen, Lars Bak, and Erik Ernst. Ungar recalls Madsen proudly explaining how Beta supported virtual classes, and Ungar pointing out that the same idiom just fell out of Self's semantics with no special support required. This may have been the moment that kindled Madsen's interest in Self. He later sent us Agesen, Bak, and (intern) Ernst, and also spent a sabbatical year with the project in 1994-1995. During his stay, he built a structured editor for Self, and we all enjoyed many rewarding discussions about the various approaches to object-oriented programming. In 1995, Madsen invited the authors to present a paper at the ECOOP conference, giving an overview of the system, emphasizing the common motivational design threads running through Self's language semantics, virtual machine, and user interface [SU95].

We felt that Self might make a good medium for teaching and learning about object-oriented programming, and in 1994-1995 we sponsored work with Brown University and the University of Manchester to develop courses based on Self. In addition to the paper publications, we felt that a videotape might be a good way to present the Self story and in October of 1996 released *Self: the Video*, a 21-minute tape describing the language semantics and shows the user interface, including its shared space aspect [StV96].

Also in 1996, Self alumnus Ole Agesen, who was working at Sun Laboratories, built a system called Pep that ran Java atop the Self virtual machine. It seemed, for a time, to be the world's fastest Java system, demonstrating the potential of Self's implementation techniques for Java programs with a high frequency of message sends [Age97].

When the Self project had joined Sun back in January of 1991, we told the company that we expected to build a fully functional programming environment in three to five years, with six to eight people. Our manager, Jim Mitchell, had us draw up a detailed project schedule. Three years later, we had delivered a fully functional programming environment, user interface, graphical construction kit, and virtual machine. Then, in September 1994, the project was cancelled, possibly in part as a consequence of the company's decision to go with Java. Self was officially wrapped up by July 1995.

Ungar remained at Sun through the summer of 2006 and, aided by Michael Abd-El-Malek and Adam Spitz, kept the Self system alive, including ports to first the PowerPC and then the Intel x86

11. McNealy never explained what he meant about the two fronts. We suspect he was thinking about asking users to learn both a new language and a new user interface.

Figure 25. The Self group decides which features to implement in Self 4.0 by voting with candy (late 1994). Each member distributed a pound of candy among various containers according to which features he desired most. We weighed the results and ate the winners (and the losers, too). Left to right: Randall Smith, Robert Duvall (student intern), Bay-Wei Chang, Lars Bak, John Maloney (seated), Ole Lehrmann Madsen (visiting professor), Urs Hölzle, Mario Wolczko, and David Ungar. Not shown: Craig Chambers (who had graduated) and Ole Agesen.

Macintoshes. It remains his vehicle of choice for condensing ideas into artifacts. As of this writing, Self 4.3 is available from http://research.sun.com/self.

6. Impact of the Self Project

6.1. The Language

We have always been enthusiastic about the cognitive benefits of unifying state and behavior and of working with prototypes instead of classes. At the 2006 OOPSLA conference, the original Self paper [US87] received an award for being among the three most influential papers published in the conference's first 11 years (from 1985 through 1996). But, for most of the twenty years since Self's design, it was discouraging to see the lack of adoption of our ideas. However, with help from our reviewers (Kathleen Fisher in particular) and from Google (which owes some of its success to Hölzle, a Self alumnus), we delightedly discovered that some researchers and engineers working on portable digital assistants (PDAs), programming language research, scripting languages, programming language theory, and automatic program refactoring had been inspired by the linguistic aspects of Self.

Before continuing, we must express our gratitude for all who have put up informative web sites about prototype-based languages, including Ranier Blome [Blom] and Group F [GroF].

6.1.1. Programming Language Research

After he graduated and left the Self project in 1991, Craig Chambers took a faculty position at the University of Washington, Seattle, where he created **Cecil**, a purely object-oriented language intended to support rapid construction of high-quality, extensible software [Cham92a, Cham]. Cecil combined multi-methods with a classless object model and optional static type checking. As in Self, Cecil's instance variables were accessed solely through messages, allowing instance variables to be replaced or overridden by methods and vice versa. Cecil's predicate objects mechanism allowed an object to be classified automatically based on its run-time (mutable) state. Cecil's static type system relied on types that specify the operations that an object must support, while its dynamic dispatch system was based on runtime inheritance links. For example, any object copied from a "Set" prototype would inherit implementations of "Set" operations such as union and intersection. But there might also be a "Set" type, which promises that any object known at compile time to be that type will implement union and intersection. In private conversations, Chambers has reported to Ungar that his students often struggled with the distinction: when to say "Set (type)" vs. "Set (prototype)." In Ungar's opinion, this illustration of the too-often-overlooked tension between a type system's expressiveness and its comprehensibility is an important result.

The designer of **Omega** [Blas91] wanted to have an object model similar to Self's but did not want to lose the benefits of static type checking. This language managed to unite the two, a surprising feat at the time.

The Self language even influenced a researcher who was deeply embedded in another object-oriented culture, the Scandinavian **Beta** language. Beta is a lineal descendant of Simula, the very first object-oriented language, that features simple, unified semantics based on a generalization of classes and methods called patterns. Like classes, patterns function as templates and must be instantiated before use. Beta allows a designer to model a physical system and then just execute the model, a Beta program [MMM90, MMN93]. In Ungar's opinion, it is one of the cleanest and most unfairly overlooked object-oriented program-

ming languages. Erik Ernst, then a graduate student in the Beta group, spent part of a year interning with the Self project at Sun Labs. After his return to Denmark, he invented gbeta, a superset of Beta that, while still very much in the Beta spirit, added features inspired by Self such as object metamorphosis and dynamic inheritance that straddle the gap between compile- and run-time [Ernst]:

> In gbeta, object metamorphosis coexists with strict, static type-checking: It is possible to take an existing object and modify its structure until it is an instance of a given class, which is possibly only known or even constructed at run-time. Still, the static analysis ensures that message-not-understood errors can never occur at run-time...Like BETA, gbeta supports inheritance hierarchies not only for classes but also for methods. This can be used together with dynamic inheritance to build dynamic control structures... [Erns99].

The authors designed but never built **US,** a language that incorporated subjectivity into Self's computational model [SU94, SU96]. An US message-send consisted of several implicit receivers and dispatched on the Cartesian product, using rules similar to Self's existing multiple-inheritance resolution ones. The extra implicit receivers could be used to represent versions, preferences, or human agents on whose behalf an operation was being performed. Ungar and Smith posited that subjectivity was to object-oriented programming as object-oriented programming was to procedural programming. In procedural programming, the same function call always runs the same code; in object-oriented programming there is one object's worth of context (the receiver) and this object determines which code will run in response to a given message. In the US style of subjectivity, there are many objects' worth of context that determine what happens. Later on, others would coin the term "subject-oriented programming" to describe systems that were somewhat similar [HKKH96].

Slate combined prototypes with multiple dispatch [RS, SA05]. It strove to support a more dynamic object system than Cecil, and thus could support subjectivity as in US without compromising Self's dynamism. Slate put different ideas together in search of expressive power.

In contrast, Self inspired Antero Taivalsaari to simplify things even further. His **Kevo** language eschewed Self-style inheritance. Instead of sharing common state and behavior via special parent links, each Kevo object (at the linguistic level) contained all the state and behavior it could possibly exhibit [Tai93a, Tai92, Tai93]. Kevo's simplification of the runtime semantics of inheritance (i.e., no inheritance!) cast the dual problem of programming-time inheritance into sharp relief: Suppose a programmer needs to add a "union" method to every Set object. In Self, one can add it to a common parent. In Kevo, there were special operations to affect every object cloned from the same prototype. But these seemed to be too sensitive to the past; the operations relied on the cloning history, rather than whether an object was supposed to be a Set. For us, the Kevo language clarified the difference between the essential behavior needed to compute with an object and the cognitive structures needed to program (i.e., reflect upon) an object.

Getting back to something closer to Self, Jecel Assumpcao Jr.'s **Self/R** (a.k.a. **Merlin**) combined a Self-style language with a facility for low-level reflection in an effort to push the high-level language down into the operating system [Assu].

Moostrap was another language that adopted a Self-style object model as part of research into minimal languages based on reflection. Its name stood for Mini Object-Oriented System Towards Reflective Architectures for Programming [MC93, Mule95].

Lisaac combined operating system research with programming language research [SC02]. The authors designed a language resembling Self with prototypes and dynamic inheritance and added some ideas from Eiffel. This language used static compilation and implemented an operating system; that is, it ran on bare metal. Dynamic inheritance was used in the video drivers and the file system.

SelfSync exploited the malleability of Self's object model to provide an interactive, bidirectional connection between an graphical diagram editor and a world of live, running, objects [PMH05]. It can be thought of as a visual programming language perched atop the Self system:

> SelfSync is a Round-Trip Engineering (RTE) environment built on top of the object-oriented prototype-based language Self that integrates a graphical drawing editor for EER[12] diagrams. SelfSync realizes co-evolution between 'entities' in an EER diagram and Self 'implementation objects.' This is achieved by adding an extra EER 'view' to the default view on implementation objects in the model-view-controller [sic] architecture of Self's user interface. Both views are connected and synchronized onto the level of attributes and operations. [D'Hont]

Moving even further from the language design center of Self's creators, Self—though having no static type system whatsoever—inspired work on type systems for object-oriented languages. According to Stanford University professor John Mitchell: "The paper [FHM94] develops a calculus of objects and a type system for it. The paper uses a delegation-based approach and refers to your work on Self. This paper first appeared at the 1993 IEEE Symposium on Logic in Computer Science, and came before many other papers on type systems and object calculi. Abadi and Cardelli and many others also got involved in the topic at various times."

6.1.2. Distributed Programming Research

When Self was designed in 1986, computers were far less interconnected than they are today, and consequently the challenges of getting separate computers to work together have become far more important than they were in the '80s. When researchers tried to combine class-based objects and distributed programming, they discovered a problem: if two Point objects are to reside on separate computers, on which one should the Point class reside? On the one hand, since an object relies on its class to supply its behavior and interpretation, an object separated from its class is going to run very slowly. On the other hand, if the class data are replicated, then great effort must be expended to reconcile the conceptual chasm between a single, malleable class, and the reality of widespread replication of the class's contents. Along with the classless distributed system Emerald [BHJ86], Self's classless object model helped inspire researchers to consider such a model for a distributed system.

The closest such model to Self is probably **dSelf** [TK02], which adopted Self's syntax and object model, but let clones to reside on different machines and allowed an object to delegate to (i.e., be a child of) a parent object on a different machine.

12. EER: extended entity-relationship.

AmbientTalk was a classless distributed programming language designed for ad-hoc networks [DB04, Deid] that does not appear to have been directly influenced by Self.

Obliq, based on a prototype object model and dynamic typing, exploited the key concept of lexical scoping to provide secure, distributed, mobile computation [Card95].

6.1.3. Prototype Object Models for User Interface Languages and Toolkits

Self's legacy from ARK included the desire to bridge the gap between programming-level and graphical-level objects. In particular, prototypes seemed to be more concrete and easier to picture than classes. Perhaps that is why there have been some notable efforts to use a prototype-based object model for GUI languages and toolkits, including **Amulet,** which placed a prototype-based object model atop C++ to make it easier to build graphical interfaces [MMM97]:

> Amulet includes many features specifically designed to make the creation of highly interactive, graphical, direct manipulation user interfaces significantly easier, including a prototype-instance object model, constraints, high-level input handling including automatic undo, built-in support for animation and gesture-recognition, and a full set of widgets. [Myer97]

Amulet was a follow-on to **Garnet**, which also used a prototype-instance object system [MGD90]. Although the authors do not tell us, this system may have been influenced by Self in that there is no distinction between instances and classes, data and methods are stored in "slots," and slots that are not overridden by a particular instance inherit their values.

6.1.4. Other Impacts of the Self Language

In the late 1980s, a team at Apple Computer created one of the first commercial PDAs, the Apple Newton. Although the first products were marred by unreliable handwriting recognition—yes, the Newton aimed to recognize words the way one *naturally* wrote them—soon the Newton became an amazing device. It featured a simple intuitive interface with functionality that could be easily extended. What made it easy to build new Newton applications was its programming language, **NewtonScript**, a pure, object-oriented, dynamically typed language based on prototypes [Smi95] whose designer, Walter Smith, has cited Self as "one of the primary influences" [Smi94]. Like Self, NewtonScript created objects by cloning and had prioritized, object-based multiple inheritance. Unlike Self, slots were added to an object when assigned to, and each object had exactly two parents. Although the Newton was supplanted by the much smaller and lighter (but less flexible) Palm Pilot, it seems likely that the Newton was a key inspiration, so that Self was at least an indirect inspiration in the rise of PDAs.

Scripting Languages. Back when we designed Self, computers seemed to offer limitless power to those who could program them; we wanted to make this power available to the largest number of people, and thus we strove to lower programming's cognitive barrier. Since then, computers have become ever faster and more widely used, trends that have created a niche for scripting languages. These notations were designed to be easy to learn and easy to use to customize systems such as web pages and browsers, but were not intended for large tasks in which performance was critical. In retrospect, it is not too surprising that many scripting languages were devised with object models like Self's. The most popular of these by far seems to be **JavaScript** [FS02], which from the start was built into a popular Web browser and has since become a standard for adding behavior to a Web page. JavaScript was based on a prototype model with object-based inheritance. Unlike Self, slots were added to an object upon assignment, reflective operations were not separated, and many more facilities were built into the language.

In addition to JavaScript, other prototype-based scripting languages have sprung up:

- **OScheme** is a small embeddable Scheme-like interpreter "that provides a prototype-based object model à la Self" [Bair].
- **Io** is a small, prototype-based programming language [Dek06]. More like Actors [Lieb86] than Self, its clones start out empty and gain slots upon assignment.
- **Glyphic Script** was a small, portable, and practical development environment and language that used both classes and prototypes [SL93, SLST94]. An object could be created by either instantiating a class or cloning an instance.
- After GlypicScript, Lentczner developed **Wheat**, a prototype-based programming system for creating of internet programs [Len05]: "Wheat strives to make programming dynamic web sites easy. It makes writing programs that span machines on the internet easy. It makes collaborative programming easy." Wheat uses a tree object system instead of a heap, and each object has a URL. Its programming environment is a collaborative web site. Wheat's design imaginatively melds object-oriented programming with distributed web-based objects.

Refactoring. Self's simplicity can be a boon to automatic program manipulation tools. This simplicity may have encouraged Ivan Moore, a University of Manchester student working for Trevor Hopkins, to create Guru, a system that may well have been the first to automatically reorganize inheritance hierarchies to refactor programs while preserving their behavior [Moor, Moo95, Moo96, Moo96a, MC96]. Subsequent refactoring tools included the Refactoring Browser [RBJ97] for Smalltalk. Although a refactoring tool for Self would be even easier than for Smalltalk or Java, by the time refactoring tools became popular, the Self project was over.

6.1.5. Summary: Impact of Self Language

Self is still used by the authors, and Ungar has based his recent research on metacircular virtual machines on it. In addition, it is also in use by curious students from around the world and by a few other dedicated souls. Jecel Assumpcao in Brazil maintains an e-mail discussion list and a web site (there is one at Sun as well) from which the latest release can be downloaded. Volunteers have ported it to various platforms, and several language variants of Self have been designed. Still, the language cannot be said to be in widespread use; as of 2006 we estimate perhaps a dozen users on this planet.

Several pragmatic issues interfered with Self's adoption in the early 1990s: the system was perceived as being too memory-hungry for its time, and too few people could afford the memory. Perhaps the worst problem was the challenge of delivering a small application instead of a large snapshot. The Self group was working on this when the project was cancelled in 1994: Ole Agesen's work on type inferencing [AU94] showed promise in this area. Wolczko produced a standalone diff viewer that was half the speed of C, started in 1 second, and was correct (as opposed to the C version, which, according to Wolczko, was not). Finally, Self did not run on the most popular personal operating system of the time, Windows, and the complexity of the

virtual machine made a port seem like a daunting task for an outsider.

Self demonstrated that an object-oriented programming language need not rely on classes, that large programs could be built in such a fashion, and that it was possible to achieve high performance. These results helped free researchers to consider prototype-based linguistic structures [Blas94, DMC92, SLST94]. Of course, the languages in this section vary in their treatment of semantic issues like privacy, copying, and the role of inheritance. Yet all these languages have a model in which an object is in an important sense self-contained.

6.2. Implementation Techniques

The optimization techniques introduced by the Self virtual machine have served as a starting point for just about every desktop- and server-based object-oriented virtual machine today; for a nice survey, see [AFGH04]. The authors note that "the industry has invested heavily in adaptive optimization technology" and state that the Self implementation's "technical highlights include polymorphic inline caches, on-stack-replacement, dynamic deoptimization, selective compilation with multiple compilers, type prediction and splitting, and profile-directed inlining integrated with adaptive recompilation." Many subsequent virtual machines rely on these techniques. The survey's authors also mention Self's invocation count mechanism for triggering recompilation, and mention that the HotSpot Server VM, the initial IBM mixed-mode interpreter system, and the Intel Microprocessor Research Labs VM all used similar techniques. They point out that Self's technique of deferring compilation of uncommon code has been adopted by the HotSpot server VM and the Jikes RVM, and that Self's dynamic deoptimization technique that automatically reverts to deoptimized code for debugging "has been adopted by today's leading production Java virtual machines."

In a slightly more exuberant tone, Doederlein comments about the effect of (among other ideas) Self-style optimizations on Java performance: "The advents of Java2 (JDK1.2.0+), Sun HotSpot and IBM JDK, raised Java to previously undreamed-of performance, and has caught many hackers by surprise...Profile-based and Speculative JITs like HotSpot and IBM JDK are often seen as the Holy Grail of Java performance. [Höl94] (*Hölzle's dissertation on the Self VM*) is the root of dynamic optimization" [Doe03]. (Italicized text added by present authors.)

The most direct influence of Self's VM technology was on Sun's HotSpot JVM, which is Sun's Java desktop and server virtual machine and is used by other computer manufacturers including Apple Computer and Hewlett-Packard. It is an ironic story: In the fall of 1994, when the Self project was cancelled, two of Self's people, Urs Hölzle and Lars Bak, left Sun to join a startup, Animorphic Systems. (Hölzle took a faculty position at UCSB and consulted at the startup; Bak was there full time.) The startup built Strongtalk, an impressive Smalltalk system that eventually included a virtual machine based on the Self virtual machine code base (with many improvements) and featuring an optional type system already designed by Animorphic's Gilad Bracha and David Griswold [BG93]. Meanwhile, another Self alumnus, Ole Agesen at Sun Labs East, rewrote portions of Sun's original JVM to support exact garbage collection.[13] On the West Coast this project was nurtured by Mario Wolczko, another Self alumnus, who had written the clever feedback-mediated code to manage the Self garbage collector (see section 5.1). For a while, the Exact VM, as it was called, was Sun's official JVM for Solaris. As Java became popular, Animorphic also retargeted its Smalltalk virtual machine to run Java. Around this time, Bak and Hölzle's startup was acquired by Sun for its Java implementation and their Strongtalk system was left to languish. After the acquisition, Ungar (who had stayed at Sun all this time) loaned himself to the newly acquired group where he contributed the portability framework and the SPARC interpreter for Java. This virtual machine became HotSpot; HotSpot improved on Self by using an interpreter instead of a simple compiler for initial code execution, but retained the key techniques of adaptive optimization and dynamic deoptimization. HotSpot eventually became Sun's primary virtual machine, supplanting the Exact VM. So the Self virtual machine essentially left the company, mutated somewhat, got reacquired, and now runs Java.

A bit later, there was talk within Sun of pushing on the debugging framework for Java. Smith, Wolczko, and others had the thought that surely the underlying code that allowed runtime method redefinition in the face of all those optimizations was laying there dormant in Sun's HotSpot VM. (Recall that the HotSpot VM was originally created by modifying the Self VM.) Wolczko put Mikhail Dmitriev, then a Sun Laboratories intern (and later an employee) to work implementing method redefinition. With a working prototype in hand, Smith and Wolczko convinced the Sun Lab's management to start the HotSwap project to allow fix-and-continue debugging changes. This facility is now part of the standard Sun Java VM, where it is used extensively for interactive profiling. According to Wolczko, this feature remains one key advantage of the NetBeans environment over its competition in 2006.

Other Java virtual machines have been inspired by the adaptive optimization and on-stack replacement in Self, including IBM's Jalapeno (a.k.a. Jikes RVM) [BCFG99], [FQ03]. The JOEQ JVM has also been inspired by some techniques from Self, including what we called "deferred compilation of uncommon cases" [Wha01]. Adaptive optimization has even been combined with off-line profile information for Java [Krin03]. Although we have not been able to find any published literature confirming this, many believe that implementations of the .NET Common Language Runtime exploit some of these techniques.

Dynamic optimization and deoptimization also found applications removed from language implementation: Dynamo exploited adaptive optimization to automatically improve the performance of a native instruction stream [BDB00], and Transmeta used dynamic code translation and optimization to host x86 ISA programs on a lower-power microprocessor with a different architecture. Their code-morphing software may have been partially inspired by HotSpot [DGBJ03]. In addition to the Transmeta system, Apple computer's Rosetta technology uses similar techniques to run PowerPC programs on Intel x86-based Macintosh computers [RRS99]. Moreover, as Ungar types these very letters, he is running a PowerPC word processor, FrameMaker, on an Intel-based MacBook Pro by using SheepShaver, a PowerPC emulator that exploits dynamic optimization [Beau06].

6.3. Cartoon Animation in the User Interface

Eleven years after their key paper on cartoon animation for user interfaces [CU93], Chang and Ungar won the second annual

13. Sun's original "classic" JVM was not especially efficient, and relied on a garbage collection scheme that could not collect all garbage; it could be fooled into retaining vast amounts of space that were actually free. Such a scheme is called "conservative garbage collection" and was developed as a compromise for C-like systems that lack full runtime type information. This compromise was never essential for Java.

"Most Influential Paper" award for this work from the 2004 ACM Symposium on User Interface Software and Technology. Some of the influenced work includes the following:

- When researchers started working on immersive, 3D user interfaces, they built rapid prototyping environments. One such was Alice [CPGB94], whose creators found that the "same kind of 'slow in/slow out' animation techniques demonstrated in the Self programming system... (were) extremely useful in providing smooth state transitions of all kinds (position, color, opacity)."
- InterViews was a user interface toolkit for X-11 in C++. As part of the Prosodic Interfaces project, Thomas and Calder [TC95] took the notion of cartoon animation of graphical objects further, imbuing InterViews objects with elasticity and inertia. They stressed that such techniques meshed naturally with the goal of direct manipulation. In a subsequent paper [TC01], they went further and actually measured the effects of their animation techniques, showing them to be "effective and enjoyable for users." Thomas and Demczuk used some of the same techniques to improve indirect manipulation, showing that animation could help users do alignment operations but that color and other effects were even better. Thomas has even applied cartoon animation techniques to a 3D collaborative virtual environment.
- Amulet [MMM97] incorporated an animation constraint solver that automatically animated the effects of changes to variables that denoted such things as positions and visibilities.
- Microsoft has studied the benefits of motion blur on the legibility of fast-moving cursors [BCR03]. They settled on temporal over-sampling. (We had seen this used in cartoons as well; Smith had christened it "stutter motion blur" as opposed to "streak motion blur.")

Nowadays, although many aspects of cartoon animation can be found on commercial desktops—just click on the yellow button on your Macintosh OS X window to see squash and stretch—other aspects such as anticipation and followthrough remain to be exploited. OS X, though, does seem to have embraced the idea of smooth transitions, and some Microsoft systems also incorporate menus and text that fades in and out.

6.4. User Interface Frameworks

The principles of the Morphic UI have also been carried on into other interface frameworks, including one for Ruby [Ling04]. After Self ended, Maloney carried the Morphic GUI system into the Squeak version of Smalltalk [Mal01]. He followed the layout-as-morph approach with the AlignmentMorph class and its dozens of subclasses. Squeak's current (2006) version of Morphic has diverged from Smith's original architecture in that each morph includes a particular layout policy that is not a morph. However, because the policy is associated with a visible object rather than an often invisible AlignmentMorph, the newer design might be considered closer to Morphic principles. The AlignmentMorph class and its subclasses are used in the latest version of Squeak, and informal discussions with Squeak users give us a sense that the proper way to treat the GUI visual structuring problem is still debated.

7. Looking Back

Now that the world has seen Self and we have received the benefit of hindsight, we can comment on lessons learned and interesting issues.

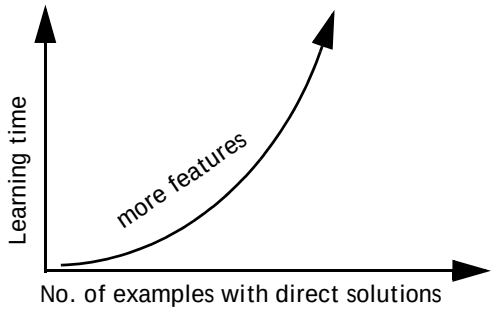


Figure 26. As more features are embedded in the language, the programmer gets to do more things immediately. But complexity grows with each feature: how the fundamental language elements interact with one another must be defined, so complexity growth can be combinatorial. Such complexity makes the basic language harder to learn, and can make it harder to use by forcing the programmer to make a choice among implementation options that may have to be revisited later.

7.1. Language

Minimalism. Ungar confesses, with some feelings of guilt, that the pure vision of Self suffered at his own hands, as he yielded to temptation and tried adding a few extra features here and there. But how could the temptation to feature creep seduce members of the Self team, who so vocally extol the principles of uniformity and simplicity? Looking back, we think it arose from the siren song of the well-stated example. Ungar had to learn the hard way that smaller was better and that examples could be deceptive. Early in the evolution of Self he made three such mistakes: prioritized multiple inheritance, the sender-path tiebreaker rule, and method-holder-based privacy semantics.[14] Each was motivated by a compelling example [CUCH91]. We prioritized multiple parent slots to support a mix-in style of programming. The sender-path tiebreaker rule allowed two disjoint objects to be used as parents, for example a rectangle parent and a tree-node parent for a VLSI cell object. The method-holder-based privacy semantics allowed objects with the same parents to be part of the same encapsulation domain, thereby supporting binary operations in a way that Smalltalk could not [CUCH91].

But each feature also caused no end of confusion. The prioritization of multiple parents implied that Self's "resend" (call-next-method) lookup had to be prepared to back up along parent links to follow lower-priority paths. The resultant semantics took five pages to write down, but we persevered. As mentioned in section 4.2, after a year's experience with the features, we found that each of the members of the Self group had wasted no small amount of time chasing "compiler bugs" that were merely their unforeseen consequences. It became clear that the language had strayed from its original path. Ironically, Ungar, who had once coined the term "architect's trap" for something similar in computer architecture, fell right into what might be called "the language designer's trap." He is waiting for the next one. At least in computer architecture and language design, when features, rules, or elaborations are motivated by particular examples, it is a good bet that their addition will be a mistake.

Prototypes and Classes. Prototypes are often presented as an alternative to class-based language designs, so the subject of

14. In all fairness, recall that Smith was across the Atlantic at the time and so, on the one hand, had nothing to do with these mistakes. On the other hand, Ungar chides him that if he had not wandered off, maybe such mistakes could have been avoided.

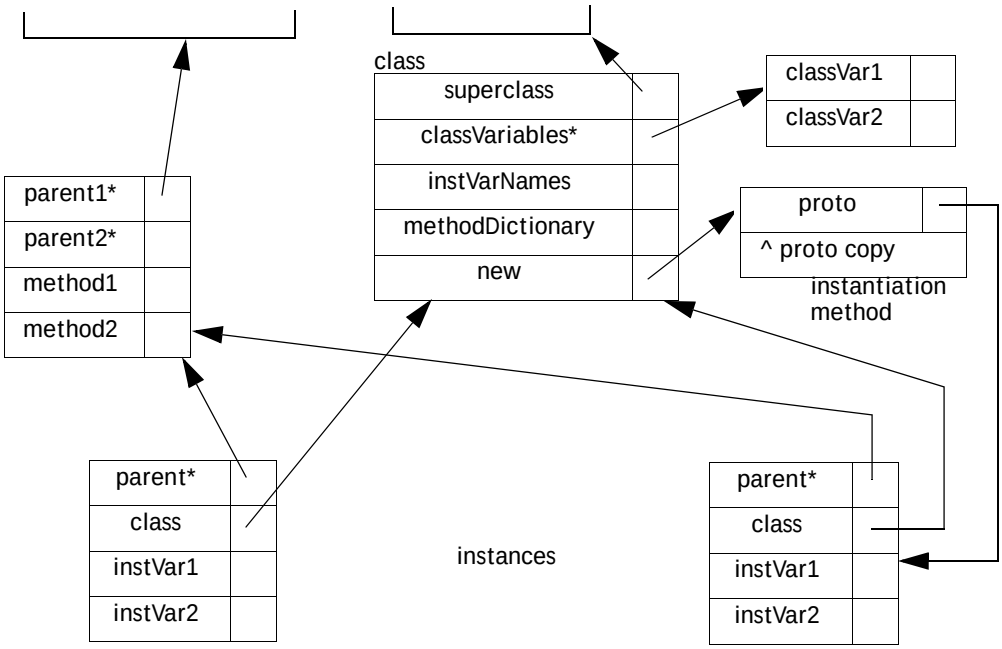


Figure 27. This figure suggests how Self objects might be composed to form Smalltalk-like class structures (demonstrated more completely by Wolczko [Wol96]). He shows that, with some caveats, Smalltalk code can be read into a Self system, parsed into Self objects, then executed with significant performance benefits, thanks to Self's dynamically optimizing virtual machine.

prototypes vs. classes can serve as point of (usually good natured) debate.

In a class-based system, any change (such as a new instance variable) to a class affects new instances of a subclass. In Self, a change to a prototype (such as a new slot) affects nothing other than the prototype itself (and its subsequent direct copies).[15] So we implemented a "copy-down" mechanism in the environment to share implementation information. It allowed the programmer to add and remove slots to and from an entire hierarchy of prototypes in a single operation. Functionality provided at the language level in class-based systems rose to the programming environment level in Self. In general, the simple object model in Self meant that some functionality omitted from the language went into the environment. Because the environment is built out of Self objects, the copy-down policy can be changed by the programmer. But such flexibility incurred a cost: there were two interfaces for adding slots to objects, the simple language level and the copying-down Self-object level. This loss of uniformity could be confusing when writing a program that needs to add slots to objects. Although we managed fine in Self, if Ungar were to design a new language, he might be tempted to include inheritance of structure in the language, although it would still be based on prototypes. Smith remains unconvinced.

A brief examination of the emulation of classes in Self illuminates both the nature of a prototype-based object model and the tradeoff between implementing concepts in the language and in the environment. To make a Self shared parent look more like a class, one could create a "new" method in the shared parent. This method could make a copy of some internal reference to a prototype, and so would appear to be an instantiation device. Figure 27 suggests how to make a Smalltalk class out of Self objects. Mario Wolczko built a more complete implementation of this, and showed [WAU96, Wol96] that it worked quite well: he could read in Smalltalk source code and execute it as a Self program. There are certain restrictions on the Smalltalk source but, thanks to Self's implementation technology, once the code adaptively optimizes, the Self version of Smalltalk code generally ran faster than the Smalltalk version! General meta-object issues in prototype-based languages were tackled by the Moostrap system [Mule95].

The world of Self objects and how they inherit from one another results in a roughly hierarchal organization, with objects in the middle of the hierarchy tending to act as repositories of shared behavior. Such behavior repositories came to be called "traits" objects.[16] The use of traits is perhaps only one of many ways of organizing the system, and may in fact have been a carryover from the Self group's Smalltalk experience. Interestingly, it is likely that our old habits may not have done Self justice (as observed in [DMC92]). Some alternative organizational schemes might have avoided a problem with the traits: a traits object cannot respond to many of the messages it defines in its own slots! For example, the point traits object lacked `x` and `y` slots and so could not respond to `printString`, since its `printString` slot contained a method that in turn sent `x` and `y` messages. We probably would have done better to put more effort into exploring other organizations. When investigating a new language, one's old habits can lead one astray.

Another problem plaguing many prototype-based systems is that of the corrupted prototype. Imagine copying the prototypical

15. Self prototypes are not really special objects, but are distinguished only by the fact that, by convention, they are copied. Any copy of the prototype would serve as a prototype equally well. Some other prototype-based systems took a different approach.

16. Do not confuse these traits objects with the construction written about in the past few years [SDNB03]. These had nothing to do with and predated by many years the more recent use of the word "traits" in object-oriented language design.

list, asking it to print, then finding it not empty as expected, but already containing 18 objects left there by some previous program! Self's syntax makes the previous program's mistake somewhat easy: the difference between

```
list copy add: someObject.
```

and

```
list add: someObject.
```

is that the latter puts some object in the system's prototypical list.

Addressing this problem led to some spirited debate within the Self group. Should the programmer have the right to assume anything about the prototypical list? It is, after all, just another object. To the VM, yes, but to the programmer, it is quite a distinguished object. Our solution, though disturbing at some level, was to introduce a `copyRemoveAll` method for our collections. Use of this method guaranteed an empty list, yet it was clearly only a partial solution. What if some program, now long gone, accidentally uttered the expression:

```
Date currentYear: 1850
```

This would create trouble for any program that subsequently assumed the current year was properly initialized in copies of the Date object (unless it was running inside a time machine).

As we have said many times by now, when designing Self we sought to unify assignment and computation. This desire for access/assignment symmetry could be interpreted as arising from the sensory-motor level of experience. Lakoff and Johnson put it very well [LJ87], although we had not read their work at the time we designed Self: from the time we are children, experience and manipulation are inextricably intertwined; we best experience an object when we can touch it, pick it up, turn it over, push its buttons, even taste it. We believe that the notion of a container is a fundamental intuition that humans share and that by unifying assignment and computation in the same way as access and computation, Self allows abstraction over containerhood. Since all containers are inspected or filled by sending messages, any object can pretend to be a container while employing a different implementation.

Retrospective Thoughts on the Influence of Smalltalk. In writing this paper and looking over the principles of Smalltalk enumerated by Ingalls [Inga81], we realize that in most cases we tried to take them even further than Smalltalk did. Table 2 shows, for each of Ingalls' principles, the progression from Smalltalk through ARK to Self.

Table 2: Ingalls' Principles of Programming System Design

Principle	Smalltalk	ARK	Self
Personal Mastery	If a system is to serve the creative spirit, it must be entirely comprehensible to a single individual.	A primary goal of ARK was to make possible personal construction of alternate realities, increasing comprehension by tangibly manifesting objects in the UI.	Concepts such as classes were removed to get a simpler language.
Good Design	A system should be built with a minimum set of unchangeable parts; those parts should be as general as possible; and all parts of the system should be held in a uniform framework.	ARK contains the world of Smalltalk objects, any of which could appear as a simulated tangible object with mass and velocity (e.g., it was possible to grab the number 17 and throw it into orbit around a simulated planet).	Even the few operations that were hard-wired in Smalltalk, such as integer addition, identity comparison, and basic control structures such as "ifTrue:" are user-definable in Self.
Objects	A computer language should support the concept of "object" and provide a uniform means for referring to the objects in its universe.	In ARK, the uniform means for referring to the objects mentioned in this principle included object references used in Smalltalk, but ARK added something with its ability to represent any object inside an alternate reality. In other words, uniformity of object access was passed up into the UI as well.	A Self object is self-sufficient; no class is needed to specify an object's structure or behavior.
Storage Management	To be truly object-oriented, a computer system must provide automatic storage management.		Generational, nondisruptive garbage collection for young objects and feedback-mediated mark-sweep for old objects.

Table 2: Ingalls' Principles of Programming System Design (Continued)

Principle	Smalltalk	ARK	Self
Messages	Computing should be viewed as an intrinsic capability of objects that can be uniformly invoked by sending messages.	ARK, as Self would later, replaced the notion of variable access with message sends. Hence it might be seen as taking this principle even further.	Smalltalk includes both message passing and variable access/ assignment in its expressions. Self expressions includes *only* message passing; "variables" are realized by pairs of accessor/assignment methods.
Uniform Metaphor	A language should be designed around a powerful metaphor that can be uniformly applied in all areas.	ARK took the metaphor of object and message inherited from the underlying Smalltalk level and pushed it up into the UI, in that every object could be manipulated as a tangible object on the screen with physical attributes such as mass and velocity, suitable for a simulated world.	Self includes no separate scoping rules, and reuses objects and inheritance instead of Smalltalk's special-purpose system and method dictionaries.
Modularity	No component in a complex system should depend on the internal details of any other component.		Self followed Smalltalk in restricting base-level access to other objects to only message-passing. However, Smalltalk includes messages, inherited by every class, that allow one object to inspect the internals of another (e.g., "instanceVarableAt:"). Self improves on Smalltalk's modularity by separating this facility into a separate reflection protocol, implemented by mirror objects. This facility can be disabled by turning off the one virtual machine primitive that creates mirror objects.
Classification	A language must provide a means for classifying similar objects, and for adding new classes of objects on equal footing with the system's kernel classes.	ARK did not pay much attention to classification issues. New kinds of objects could be made by adding new state to some existing instance, but they were anonymous, so the user did not even have a name to go on. The Smalltalk categories used in the browser were also used in the Alternate Reality Kit's "warehouse" icon, which strove to make an instance of any class selected from the warehouse's pop-up hierarchical menu.	Self has no classes. We did not find them essential, opting to supply such structure at higher levels in the system.
Polymorphism	A program should specify only the behavior of objects, not their representation.		In Self, even the code "within" an object is isolated from the object's representation.
Factoring	Each independent component in a system should appear in only one place.		Self's prototype model, which did not build inheritance of structure into the language, simplifies the specification of multiple inheritance.

Table 2: Ingalls' Principles of Programming System Design (Continued)

Principle	Smalltalk	ARK	Self
Virtual Machine	A virtual machine specification establishes a framework for the application of technology.		
Reactive Principle	Every component accessible to the user should be able to present itself in a meaningful way for observation and manipulation.	One of the main goals of ARK was to make objects feel more real, more directly present. This can be seen as an attempt to (as in the original articulation of this principle) "show the object in a more meaningful way for observation and manipulation."	The Morphic User Interface improved upon the Smalltalk-80 UI. In Smalltalk, scroll bars and menus could not be graphically selected, only used. In Self's Morphic they can. (Of course, Self has the luxury of a more powerful platform.)

7.2. Implementation techniques

The efficacy of the Self VM in obtaining good performance for a dynamic, purely object-oriented language came at a high price in complexity and maintainability. One issue that has arisen since the original optimization work has been the difficulty of finding intermittent bugs in a system that adaptively optimizes and replaces stack frames behind the user's back. Since 1995, the Self virtual machine has been maintained primarily by Ungar in his spare time, so the priorities have shifted from probing the limits of performance to reducing maintenance time. Consequently, when we run Self today, we disable on-stack replacement.

Looking back, it's clear that the optimizations devised for Self were both the hardest part of the project, spanning many years and several researchers, and also—despite their complexity—its most widely adopted part of the project. This experience argues for stubborn persistence on the part of researchers and a large dose of patience on the part of research sponsors.

7.3. UI2 and Morphic

On the whole we were satisfied with much of UI2. While the principles of live editing and structural reification helped create the sense of working within a world of tangible yet malleable objects, we could imagine going further. Several things interfered with full realization of those goals.

Multiple views. The very existence of the outliner as a separate view of a morph object weakened the sense of directness we were after. After all, when one wanted to add a slot to an object, one had to work on a different display object, the object's outliner. We never had the courage or time to go after some of the wild ideas that would have made possible the unification of any morph with its outliner. Ironically, Self's first interface, UI1, probably did better in this respect because it limited itself to presenting only outliners.

Text and object. There is a fundamental clash between the use of text and the use of direct manipulation. A word inherently denotes something, an object does not necessarily denote anything. That is, when you see the word "cow," an image comes to mind, an image totally different from the word "cow" itself. It is in fact difficult to avoid the image: that is the way that words are supposed to work. Words stand for things, but a physical object does not necessarily stand for anything. Textual notation and object manipulation are fundamentally from two different realities.

Text is used quite a bit in Self, and its denotational character weakens the sense of direct encounter with objects. For example, many tools in the user interface employed a "printString" to denote an object. The programmer working with one of these tools might encounter the text "list (3, 7, 9)." The programmer might know that this denoted an object which could be viewed "directly" with an outliner. But why bother? The textual string often says all one needs to know. The programmer moves on, satisfied perhaps, yet not particularly feeling as if they encountered the list itself. The mind-set in a denotational world is different from that in a direct object world, and use of text created a different kind of experience. Issues of use and mention in direct manipulation interfaces were discussed further [SUC92].

8. Conclusion

Shall machines serve humanity, or shall humanity serve machines? People create artifacts that then turn around and reshape their creators. These artifacts include thought systems that can have profound effects, such as quantum mechanics, calculus, and the scientific method. In our own field thought systems with somewhat less profound effects might include FORTRAN and Excel. Some thought systems are themselves meta-thought systems; that is, they are ways of thinking followed when building other thought systems. Since they guide the construction of other thought systems, their impact can be especially great, and one must be especially careful when designing such meta-thought systems.

We viewed Self as a meta-thought system that represented our best effort to create a system for computer programming. The story of its creation reveals our own ways of thinking and how other meta-thought systems shaped us [US87, SU95]. We kept the language simple, built a complicated virtual machine that would run programs efficiently even if they were well-factored, and built a user interface that harnessed people's experience in dealing with the real word to off load conscious tasks to precognitive mental facilities. We did all of this in the hope that the experience of building software with the Self system would help people to unleash their own creative powers.

However, we found ourselves trying to do this in a commercial environment. Free markets tend to favor giving customers what they want, and few customers could then (or even now) understand that they might want the sort of experience we were creating.

Years later, the Self project remains the authors' proudest professional accomplishment. We feel that Self brought new ideas to language, implementation, programming environment, and

graphical interface design. The original paper [US87] has been cited over 500 times, and (as previously mentioned) received an award at the OOSPLA 2006 conference for among the three most influential conference papers published during OOPSLA's first 11 years (1985 through 1996). Self shows how related principles can be combined to create a pure, productive, and fun experience for programmers and users.

So, what happened? Why isn't your word processor written in Self? While we have discussed the struggle of ideas that gave birth to Self, we have not addressed the complex of forces that lead to adoption (or not) of new technology. The implementation techniques were readily adopted, whereas semantic notions such as using prototypes, and many of the user interface ideas behind Morphic, were not so widely adopted. We believe that, despite the pragmatic reasons mentioned in section 6.1.5, this discrepancy can better be explained by the relative invisibility of the virtual machine. If there are dynamic compilation techniques going on underneath their program, most users are unlikely to know or care. But the user interface framework and the language semantics demand that our users think a certain way, and we failed to convince the world that our way to think might be better. Did our fault lie in trying to enable a creative spirit that we mistakenly thought lay nascent within everybody? Or are there economic implications to the use of dynamic languages that make them unrealistic? Many of us in the programming language research community secretly wonder if language research has become irrelevant to most of the world's programmers, despite the obvious truth that in many ways, computers remain painful, opaque black boxes that at times seem intent on spreading new kinds of digital pestilence.

Almost two decades after the conception of Self, the imbalance of power between man and machine seems little better. We are still waiting for computers to begin to live up to their full promise of being a truly malleable and creative medium. We earnestly hope that Self may inspire those who still seek to simplify programming and to bring it into coherence with the way most people think about the real world.

9. Epilogue: Where Are They Now?

After the Self project, the people involved followed disparate paths. Smith, Ungar, and Wolczko stayed at Sun Laboratories. Randy Smith used Morphic's shared space aspects to start a project studying distance learning. He also worked on realtime collaboration support for Java, then researched user interfaces techniques for information visualization. Randy now works on trying to make it easier to understand and use sensor networks. He continues to use Self for an occasional quick prototype, especially when a live shared-space demo would be useful.

David Ungar has used Self in much of his research. With help from Michael Abd-El Malek, the complex Self virtual machine was ported to the Macintosh computer system. David also worked on Sun's HotSpot Java virtual machine, and until recently was researching Klein, a meta-circular VM architecture in Self for Self [USA05].

After a brief flirtation with binary translation, Mario Wolczko worked on Sun's ExactVM (a.k.a. Solaris Production Release of Java 1.2 JVM), then managed the group that developed the research prototype for Sun's KVM, a Java VM for small devices. Since then he has been working on architecture support for Java, automatic storage reclamation, and objects, as well as performance monitoring hardware for various SPARC microprocessors at Sun Microsystems Laboratories.

Elgin Lee went to ParcPlace Systems, and now does legal consulting.

Lars Bak left Sun to build a high performance virtual machine for Smalltalk at the startup Animorphic Systems. The technology was adapted to Java, and the Java HotSpot system was born. Sun acquired the startup and Bak ended up leading the HotSpot project until it successfully shipped in 1997. Next, Bak designed a lean and mean Java virtual machine for mobile phones, commercialized by Sun as CLDC HI. Bak left Sun again to pursue even smaller virtual machines. The startup OOVM was founded to create an always running Smalltalk platform for small embedded devices. The platform had powerful reflective features despite a memory footprint of 128KB. OOVM was acquired by Esmertec AG.

After the Self project, Ole Agesen implemented a Java-to-Self translator that, for a time, seemed to be the world's fastest Java system. Then he spearheaded a project at Sun incorporating exact garbage collection into Sun's original JVM; after that, he went to VMware, working on efficient software implementations of x86 CPUs. Many of the implementation techniques successful in dynamic languages can be reused for x86: it is really just a different kind of bytecode (x86) that is translated. More specifically, Ole has been on the team that designed and implemented the SMP version of VMware; more recently, he has worked on supporting 64-bit ISAs (x86-64).

Since graduating from Stanford, Craig Chambers has been a professor at the University of Washington, where he worked on language designs including Cecil, MultiJava, ArchJava, and EML, and on optimizing compiler techniques primarily targeting object-oriented languages. The language designs were inspired by Self's high level of simplicity and uniformity, while also incorporating features such as multiple dispatching and polymorphic, modular static type checking. The optimizing compiler research directly followed the Self optimizing dynamic compiler research, in some cases exploring alternative techniques such as link-time whole-program compilation as in the Vortex compiler, and in other cases applying (staged) dynamic compilation to languages such as C, as in the DyC project.

After Self, Bay-Wei Chang was at PARC for four years working on document editing, annotating, and reading interfaces for web, desktop, and mobile devices. For the past six years, Chang has been at the research group at Google working on bits of everything, including web characterization, mobile interfaces, e-mail interfaces, web search interfaces and tools, and advertising tools.

After Self, Urs Hölzle was at UCSB from 1994-99 as Assistant/Associate Professor. During that time, he also worked part-time with Lars Bak, first at Animorphic and then at Sun's Java organization on what became Sun's HotSpot JVM. Since 1999 Hölzle has been at Google in various roles (none involving dynamic compilation to date!), first as search engine mechanic and later as VP Engineering for search quality, hardware platforms, and as VP of Operations.

From the Self group, John Maloney went to work for Alan Kay for about six years, first at Apple's Advanced Technology Group and then at Walt Disney Imagineering R&D. (Alan Kay moved the entire group from Apple to Disney.) While there, John helped implement the Squeak Virtual Machine, notable because the VM itself was written (and debugged) in Smalltalk, then automatically translated into C code for faster execution. This technique resulted in an extremely portable, stable, and platform-independent virtual machine. Once they had the VM, John re-implemented Morphic in Smalltalk with very few design

changes from the Self version. The Morphic design has stood the test of time and has enabled a rich set of applications in Squeak, including the EToys programming system for children. In October 2002, John moved to the Lifelong Kindergarten Group at the MIT Media Lab, where he became the lead programmer for Scratch, a media-rich programming system for kids. Scratch is built on top of Squeak and Morphic. Is it currently in beta testing at sites around the world and will become publicly available in the summer of 2006.

10. Acknowledgments

The Self project resulted from the inspired creation of many individuals and the funding of many organizations, as mentioned in the Introduction. This lengthy paper about the Self project has resulted not only from the authors' efforts, but from the information and advice gleaned from many individuals. We especially wish to acknowledge the review committee chaired by Kathleen Fisher: each reviewer commented in great detail, most of them several times—our heartfelt thanks to each of you. Your diligence spoke of your desire to see a high-quality paper on the history of Self, and that desire in turn inspired us through the many arduous hours. Our special thanks to committee member Andrew Black, who was particularly thorough and thoughtful in his copious comments and suggestions.

11. Appendix: Self Release Announcements

11.1. Self 2.0

What: Self 2.0 Release
From: hoelzle@Xenon.Stanford.EDU (Urs Hoelzle)
Date: 10 Aug 92 21:08:25 GMT

Announcing Self Release 2.0

The Self Group at Sun Microsystems Laboratories, Inc., and Stanford University is pleased to announce Release 2.0 of the experimental object-oriented exploratory programming language Self.

Release 2.0 introduces full source-level debugging of optimized code, adaptive optimization to shorten compile pauses, lightweight threads within Self, support for dynamically linking foreign functions, changing programs within Self, and the ability to run the experimental Self graphical browser under OpenWindows.

Designed for expressive power and malleability, Self combines a pure, prototype-based object model with uniform access to state and behavior. Unlike other languages, Self allows objects to inherit state and to change their patterns of inheritance dynamically. Self's customizing compiler can generate very efficient code compared to other dynamically-typed object-oriented languages.

Self Release 2.0 runs on Sun-3's and Sun-4's, but no longer has an optimizing compiler for the Sun-3 (and therefore runs slower on the Sun-3 than previous releases).

This release is available free of charge and can be obtained via anonymous ftp from self.stanford.edu. Unlike previous releases, Release 2.0 includes all source code and is legally unencumbered (see the LICENSE file for legal information.) Also available for ftp are a number of papers published about Self.

Finally, there is a mail group for those interested in random ramblings about Self, self-interest@self.stanford.edu. Send mail to self-request@self.stanford.edu to be added to it (please do not send such requests to the mailing list itself!).

The Self Group at Sun Microsystems Laboratories, Inc. and Stanford University

11.2. Self 3.0

From: hoelzle@Xenon.Stanford.EDU (Urs Hoelzle)
Subject: Announcing Self 3.0
Date: 28 Dec 93 22:19:34 GMT

ANNOUNCING Self 3.0

The Self Group at Sun Microsystems Laboratories, Inc., and Stanford University is pleased to announce Release 3.0 of the experimental object-oriented programming language Self. This release provides simple installation, and starts up with an interactive, animated tutorial.

Designed for expressive power and malleability, Self combines a pure, prototype-based object model with uniform access to state and behavior. Unlike other languages, Self allows objects to inherit state and to change their patterns of inheritance dynamically. Self's customizing compiler can generate very efficient code compared to other dynamically-typed object-oriented languages.

The latest release is more mature than the earlier releases: more Self code has been written, debugging is easier, multiprocessing is more robust, and more has been added to the experimental graphical user interface which can now be used to develop code. There is now a mechanism (still under development) for saving objects in modules, and a source-level profiler.

The Self system is the result of an ongoing research project and therefore is an experimental system. We believe, however, that the system is stable enough to be used by a larger community, giving people outside of the project a chance to explore Self.

2 This Release

This release is available free of charge and can be obtained via anonymous ftp from Self.stanford.edu. Also available for ftp are a number of published papers about Self. There is a mail group for those interested in random ramblings about Self, Self-interest@Self.stanford.edu. Send mail to self-request@self.stanford.edu to be added to it (please do not send such requests to the mailing list itself!).

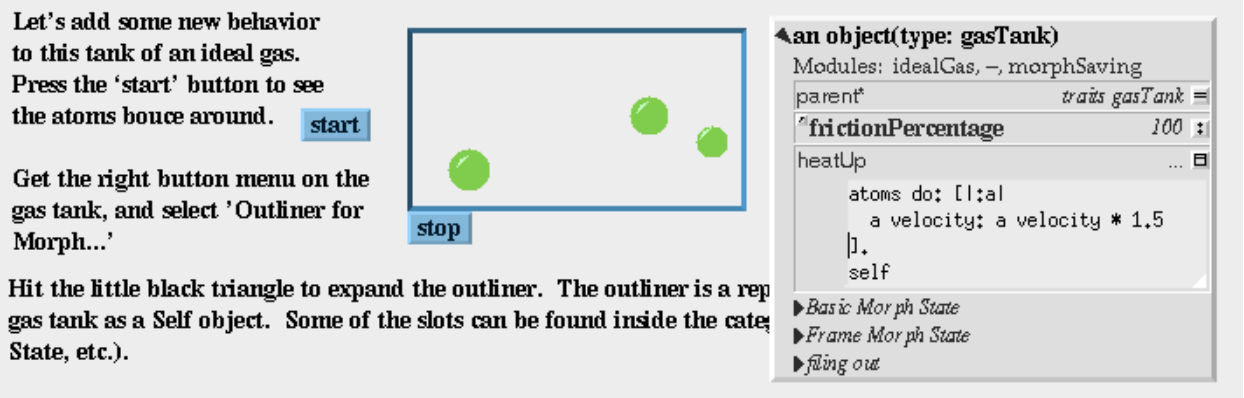
2.1 Implementation Status

Self currently runs on SPARC-based Sun workstations running SunOS 4.1.x or Solaris 2.3. The Sun-3 implementation is no longer provided.

2.2 Major Changes

Below is a list of changes and enhancements that have been made since the last release (2.0.1). Only the major changes are included.

- The graphical browser has been extended to include editing capabilities. All programming tasks may now be performed through the graphical user interface (the "ui"). Type-ins allow for expression evaluation, menus support slot editing, and methods can be entered and edited. If you are familiar with a previous version of the Self system, Section 14.1 of the manual entitled "How to Use Self 3.0" contains a quick introduction to the graphical user interface. The impatient might want to read that first.
- A mechanism - the transporter - has been added to allow arbitrary object graphs to be saved into files as Self source. The system has been completely modularized to use the transporter; every item of source now resides in a trans-



porter-generated module. Transport-generated files have the suffix .sm to distinguish them from "handwritten" files (.Self), though this may change as we move away from handwritten source. The transporter is usable but rough, we are still working on it.

- Every slot or object may now have an annotation describing the purpose of the slot. In the current system, annotations are strings used to categorize slots. We no longer categorize slots using explicit category parent objects. Extra syntax is provided to annotate objects and slots.
- A new profiler has been added, which can properly account for the time spent in different processes and the run-time system, and which presents a source-level profile including type information (i.e., methods inherited by different objects are not amalgamated in the profile, nor are calls to the same method from different sites). It also presents a consistent source-level view, abstracting from the various compiler optimizations (such as inlining) which may confuse the programmer.
- Privacy is not enforced, although the privacy syntax is still accepted. The previous scheme was at once too restrictive (in that there was no notion of "friend" objects) and too lax (too many object had access to a private slot). We hope to include a better scheme in the next release.
- The "new" compiler has been supplanted by the SIC ("simple inlining compiler"), and the standard configuration of the system is to compile first with a fast non-optimizing compiler and to recompile later with the SIC. Pauses due to compilation or recompilation are much smaller, and applications usually run faster.
- Characters are now single-byte strings. There is no separate character traits.
- Prioritized inheritance has been removed; the programmer must now manually resolve conflicts. We found the priority mechanism of limited use, and had the potential for obscure errors.

2.4 Bug Reports

Bug reports can be sent to self-bugs@self.stanford.edu. Please include an exact description of the problem and a short Self program reproducing the bug.

2.5 Documentation

This release comes with two manuals:
How to Use Self 3.0 (SelfUserMan.ps)
The Self Programmer's Reference Manual (progRef.ps)

Happy Holidays!

-- The Self Group

11.3. Self 4.0

Below is a redacted form of the Self 4.0 release announcement made on July 10, 1995. The text we do include has not been edited.

The Self Group at Sun Microsystems Laboratories, Inc., and Stanford University has made available Release 4.0 of the experimental object-oriented programming language Self.

This release of Self 4.0 provides simple installation, and starts up with an interactive, animated tutorial (a small piece of which is shown below).

Self 4.0 is, in some sense, the culmination of the Self project, which no longer officially exists at Sun. It allows novices to start by running applications, smoothly progress to building user interfaces by directly manipulating buttons, frames and the like, progress further to altering scripts, and finally to ascend to the heights of complete collaborative application development, all without ever stumbling over high cognitive hurdles.

Its user interface framework features automatic continuous layout, support for ubiquitous animation, direct-manipulation-based construction, the ability to dissect any widget you can see, and large, shared, two-dimensional spaces.

Its programming environment is based on an outliner metaphor, and features rapid turnaround of programming changes. It includes a plethora of tools for searching the system. Its debugger supports in-place editing. A structure editor supports some static type checking and helps visualize complex expressions. Finally, the programming environment features the new transporter, which eases the task of saving programs as source files.

Self 4.0 includes two applications: an experimental web browser, and an experimental Smalltalk system.

Major Changes in Self 4.0.

Below is a list of changes and enhancements that have been made since the last release (4.0). Only the major changes are included.

- This release contains an entirely new user interface and programming environment which enables the programmer to create and modify objects entirely within the environment, then save the objects into files. You no longer have to edit source files using an external editor. The environment includes a graphical debugger, and tools for navigation through the system.

- Any Self window can be shared with other users on the net: users each have their own cursor, and can act independently to grab and manipulate objects simultaneously. A Self window is actually a framed view onto a vast two-dimensional plane: users can move their frames across this surface, bringing them together to work on the same set of objects, or moving apart to work independently.
- A new version of the transporter, a facility for saving objects structure into files, has been used to modularize the system. The programming environment presents an interface to the module system which allows for straightforward categorization of objects and slots into modules, and the mostly-automatic saving of modules into files. Handwritten source files have almost completely disappeared.
- The environment has been constructed using a new, flexible and extensible user interface construction kit, based on "morphs." Morphs are general-purpose user interface components. An extensive collection of ready-built morphs is provided in the system, together with facilities to inspect, modify, and save them to files. We believe the morph-based substrate provides an unprecedented degree of directness and flexibility in user interface construction.
- An experimental Web browser has been written in Self and is included in the release. This browser supports collaborative net-surfing, and the buttons and pictures from Web pages can easily be removed and embedded into applications.
- A Smalltalk system is included in Self 4.0. This system is based on the GNU system classes, a translator that reads Smalltalk files and translates them to Self, and a Smalltalk user interface. The geometric mean of four medium-sized benchmarks we have tried suggests that this system runs Smalltalk programs 1.7 times faster than commercially available Smalltalk on a SPARCstation.
- Significant engineering has been done on the Virtual Machine to reduce the memory footprint and enhance memory management. For example, a 4.0 system containing a comparable collection of objects to that in the 3.0 release requires 40% less heap space. A SELF-level interface to the memory system is now available that enables SELF code to be notified when heap space is running low, and to expand the heap.
- The privacy syntax has been removed; in the previous release it was accepted but privacy was not enforced. The concept of privacy still exists, and is visible in the user interface, but is supported entirely through the annotation system.

SELF currently runs on SPARC-based Sun workstations using Solaris 2.3 or later, or SunOS 4.1.x. The compiler is an improved version of the one used in 3.0.

System requirements. To run SELF you will need a SPARC-based Sun computer or clone running SunOS 4.1.X or Solaris 2.3 or 2.4.

To use the programming environment you will need to run X Windows version 11 or OpenWindows on an 8-bit or deeper color display. The X server need not reside on the same host as SELF.

The SELF system as distributed, with on-line tutorial, Web browser and Smalltalk emulator, requires a machine with 48Mb of RAM or more to run well.

The user interface makes substantial demands of the X server. A graphics accelerator (such as a GX card) improves the responsiveness of the user interface significantly, and therefore we recommend that you use one if possible.

We hope that you enjoy using Self as much as we do.

-- The Self Group July 10, 1995

11.4. Self 4.3 (The latest release as of 2006)

The Power of Simplicity
Release 4.3

Adam Spitz, Alex Ausch, and David Ungar
Sun Microsystems Laboratories
June 30, 2006

Late-breaking news. Self now runs under Intel-based Macintoshes (as well as PowerPC-based and SPARC™-based systems), though it does not yet run on Windows or Linux. Additionally, the original Self user interface (UI1) has been resurrected, although its cartoon-animation techniques have not yet been incorporated into the default Self user interface (UI2). See the included release notes for a full list of changes.

Downloading. If you want to run Self 4.3, download and unpack one of the following:

- Self 4.3 for Mac OS X in compressed disk image format, or
- Self 4.3 for SPARC™ workstations from Sun Microsystems running the Solaris™ operating system in tar/gzip format

See the release notes for directions on how to run Self. (We're hoping that the procedure is fairly self-explanatory, though. If it's not, please contact us!)

If you also want to work on the Self virtual machine (most users will not want to do this), you will need to download one of the above packages, and you will also need one of the following:

- Virtual machine and Self sources in compressed disk image format, or
- Virtual machine and Self sources in tar/gzip format

What Self is. Self is a prototype-based dynamic object-oriented programming language, environment, and virtual machine centered around the principles of simplicity, uniformity, concreteness, and liveness. It was developed by the Self Group at Sun Microsystems Laboratories, Inc. and Stanford University.

Although Self is no longer an official project at Sun Microsystems Laboratories, we have seen many of Self's innovations adopted. The Morphic GUI framework has been incorporated into Squeak, and the virtual machine ideas provided the initial inspiration for the Java™ HotSpot™ performance engine. However, the language and especially the programming environment still provide a unique experience.

We have decided to do a new release because we have ported the virtual machine to the x86 architecture, so that it can run on the new Intel-based Macintosh computers (Mac Mini, MacBook,

iMac). The system is far from polished, but we have used Self on Mac OS X to do many hours of work on G4 Powerbooks and on the new Intel-based Macs.

Although our code is completely independent of theirs, we would be remiss if we did not mention Gordon Cichon and Harald Gliebe, who have also done an x86 port of Self. Their port runs on both Linux and Windows (which ours does not, yet - we would be thrilled if some kind soul were to port this latest version of Self to either of those platforms).

We hope that you will enjoy the chance to experience a different form of object-oriented programming.

Support. If you want to discuss Self with other interested people, there is a mailing list at self-interest@egroups.com. We would like to thank Jecel Assumpcao Jr. for investing the time and effort to deeply understand the Self system, and furthermore for his help in explaining Self to many folks on the Self mailing list. Jecel also hosts the Self Swiki.

For information on the programming environment (essentially unchanged for Self 4.3), please refer to the Web page on Self 4.0.

Supplemental Information.

- An HTML version of the Self tutorial, `Prototype-Based Application Construction Using Self 4.0', courtesy of Steve Dekorte. Thanks, Steve!
- In addition, see the Self bibliography for a listing of Self papers with on-line abstract

Acknowledgments.

This release was prepared by Alex Ausch, Adam Spitz and David Ungar. Kristen McIntyre helped with the PowerPC Mac port. Self owes its existence to the members of the Self Group and to the support of Sun Microsystems Laboratories, directed by Bob Sproull.

References

AC96 Martin Abadi and Luca Cardelli. *A Theory of Primitive Objects: Untyped and First-Order Systems.* Information and Computation, 125(2):78-102, 15 March 1996.

AFGH04 Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. *A Survey of Adaptive Optimization in Virtual Machines.* IBM Research Report RC23143 (W032-097), May 18, 2004.

Age95 Ole Agesen. *The Cartesian Product Algorithm.* ECOOP'95.

Age96 Ole Agesen. *Concrete Type Inference: Delivering Object-Oriented Applications.* PhD Dissertation, Computer Science, Stanford University,. 1996.

Age97 Ole Agesen. *The Design and Implementation of Pep, A Java™ Just-In-Time Translator.* Theory and Practice of Object Systems, 3(2), 1997, pp. 127-155.

AH95 Ole Agesen and Urs Hölzle. *Type Feedback vs. Concrete Type Inference: A Comparison of Optimization Techniques for Object-Oriented Languages.* OOPSLA'95.

AM95 Malcolm Atkinson and Ronald Morrison. *Orthogonally Persistent Object Systems.* The International Journal on Very Large Data Bases, Vol. 4, Issue 3, July 1995.

APS93 Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. *Type Inference of Self: Analysis of Objects with Dynamic and Multiple Inheritance.* Proc. ECOOP '93, pp. 247-267. Kaiserslautem, Germany, July 1993.

Assu Jecel Assumpcao Jr. *The Merlin Papers.* http://www.merlintec.com/lsi/mpapers.html

AU94 Ole Agesen and David Ungar. *Sifting Out the Gold: Delivering Compact Applications From an Exploratory Object-Oriented Environment,* OOPSLA'94.

Bair Anselm Baird-Smith. *OScheme Overview.* http://koala.ilog.fr/abaird/oscheme/oscheme.html.

BCFG99 Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. *The Jalapeño Dynamic Optimizing Compiler for Java.* Proceedings ACM 1999 Java Grande Conference.

BCR03 Patrick Baudisch, Edward Cutrell, and George Robertson. *High-Density Cursor: A Visualization Technique that Helps Users Keep Track of Fast-Moving Mouse Cursors.* INTERACT'03, IOS Press (c) IFIP, 2003, pp. 236-243.

BDB00 Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. *DYNAMO: A Transparent Dynamic Optimization System.* PLDI, June 2000, pp. 1-12.

BD81 A. Borning and R. Duisberg. *Constraint-Based Tools for Building User Interfaces.* ACM Transactions on Graphics 5(4) pp. 345-374 (October 1981).

Beau06 Gwenolé Beauchesne. The Official SheepShaver Home Page. http://sheepshaver.cebix.net. 2006.

BG93 Gilad Bracha, David Griswold. *Strongtalk: Typechecking Smalltalk in a Production Environment.* OOPSLA'93.

BHJ86 Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. *Object Structure in the Emerald System.* OOPSLA'86.

Blas91 G. Blaschek, *Type-Safe OOP with Prototypes: The Concept of Omega,* Structured Programming 12 (12) (1991) 1-9.

Blas94 G. Blaschek. *Object-Oriented Programming with Prototypes*. Springer-Verlag, New York, Berlin 1994.

Blom Ranier Blome. //www.pasteur.fr/~letondal/object-based.html, Object-based PLs.

Broo Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition.* Addison-Wesley Professional, Boston 1995.

BSUH87 William Bush, A. Dain Samples, David Ungar, and Paul Hilfinger. *Compiling Smalltalk-80 to a RISC.* SIGPLAN Notices Vol. 22, Issue 10, 1987. Also in SIGARCH Computer Architecture News 15(5), 1987, ASPLOS'87, and SIGOPS Operating Systems Review 21(4), 1987.

BU04 Gilad Bracha and David Ungar. *Mirrors: Design Principles for Meta-Level Facilities of Object-Oriented Programming Languages,* OOPSLA, 2004.

Card88 Luca Cardelli. *A Semantics of Multiple Inheritanc*e. Information and Computation 76, pp. 138-164, 1988.

Card95 Luca Cardelli. *A Language with Distributed Scope,* Computing Systems, 1995.

Cham Craig Chambers and the Cecil Group. The Cecil Language: Specification and Rationale. http://www.cs.washington.edu/research/projects/cecil/www/pubs/cecil-spec.html.

Cham92 Craig Chambers. *The Design and Implementation of the Self Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD Thesis, Computer Science Department, Stanford University, April 1992.

Cham92a Craig Chambers. *Object-Oriented Multi-Methods in Cecil*. ECOOP 1992.

Chan95 Bay-Wei Chang. *Seity: Object-Focused Interaction in the Self User Interface*. PhD dissertation, Stanford University, 1995.

CPGB94 Matthew Conway, Randy Pausch, Rich Gossweile, and Tommy Burnette. *Alice: A Rapid Prototyping System for Building Virtual Environments.* CHI'94 Conference Companion.

CU89 Craig Chambers and David Ungar. *Customization: Optimizing Compiler Technology for Self, A Dynamically-Typed Object-Oriented Programming Language.* PLDI 1989. Also in *20 Years of the ACM/SIGPLAN Conference on Programming Language Design and Implementation (1979-1999): A Selection*, 2003.

CU90 Craig Chambers and David Ungar. *Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs.* In Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation, White Plains, NY, June, 1990. Published as SIGPLAN Notices 25(6), June, 1990.

CU90a Bay-Wei Changs and David Ungar. *Experiencing Self Objects: An Object-Based Artificial Reality.* The Self Papers, Computer Systems Laboratory, Stanford University, 1990.

CU91 Craig Chambers and David Ungar. *Making Pure Object-Oriented Languages Practical.* In OOPSLA '91 Conference Proceedings, pp. 1-15, Phoenix, AZ, October, 1991.

CU93 Bay-Wei Chang, David Ungar. *Animation from Cartoons to the User Interface*. Sun Microsystems Laboratories TR95-33. Also in UIST, 1993.

CUCH91 Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle, *Parents Are Shared Parts of Objects: Inheritance and Encapsulation in Self.* Journal of Lisp and Symbolic Computation, 4(3), Kluwer Academic Publishers, June, 1991.

CUL89 Craig Chambers, David Ungar, and Elgin Lee. *An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes*. In OOPSLA '89 Conference Proceedings, pp. 49-70, New Orleans, LA, 1989. Published as SIGPLAN Notices 24(10), October, 1989.

CUS95 Bay-Wei Chang, David Ungar, and Randall B. Smith, *Getting Close to Objects*. In Burnett, M., Goldberg, A., and Lewis, T., editors, *Visual Object-Oriented Programming, Concepts and Environments*, pp. 185-198, Manning Publications, Greenwich, CT, 1995.

DB04 Jessie Deidecker and Dr. Werner Van Belle. *Actors in an Ad-Hoc Wireless Network Environment.* 2004.

Deid Jessie Deidecker. *Ambient Oriented Programming.* http://prog.vub.ac.be/amop/research/papers.html.

Dek06 Steve Dekorte. IO. http://www.iolanguage.com/

Deu83 L. Peter Deutsch. *The Dorado Smalltalk-80 Implementation: Hardware Architecture's Impact on Software Architecture*. In *Smalltalk-80: Bits of History, Words of Advice*, Glenn Krasner, ed. Addison-Wesley, 1983.

Deu88 L. Peter Deutsch. Richards benchmark. Personal communication, 1988.

DGBJ03 James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. *The Transmeta Code Morphing Software: Using Speculation, Recovery, and Adaptive Retranslation.* Proceedings of the First Annual IEEE/ACM International Symposium on Code Generation and Optimization March 2003, San Francisco, California. http://doi.ieeecomputersociety.org/10.1109/CGO.2003.1191529.

DMC92 C. Dony, J. Malenfant, and P. Cointe, *Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation*. In Proc. OOPSLA'92, pp. 201-217.

DN66 Ole-Johan Dahl and Kristen Nygaard. *SIMULA—an ALGOL-Based Simulation Language.* CACM 9(9), Sept. 1966, pp. 671-678.

Doe03 Osvaldo Pinali Doederlein. *The Tale of Java Performance*. Journal of Object Technology, Vol. 2, No. 5, Sept.-Oct. 2003. http://www.jot.fm/issues/issue_2003_09/column3.

DS84 L. Peter Deutsch and Allan M. Schiffman. *Efficient Implementation of the Smalltalk-80 System.* In Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages, pp. 297-302, Salt Lake City, UT, 1984.

Erns99 Erik Ernst. *gbeta, a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, University of Århus, Denmark, 1999.

Ernst Erik Ernst. *gbeta*. http://www.daimi.au.dk/~eernst/gbeta/

FG93 W. Finzer and L Gould. *Rehearsal World: Programming by Rehearsal*. In W. Finzer and L. Gould. *Watch What I Do: Programming by Demonstration*. MIT Press, 1993, pp 79-100.

FHM94 Kathleen Fisher, Furio Honsell, and John C. Mitchell. *A Lambda Calculus of Objects and Method Specialization.* Nordic Journal of Computing archive Volume 1, Issue 1, Pages: 3 - 37, 1994.

FQ03 Stephen J. Fink and Feng Qian. *Design, Implementation and Evaluation of Adaptive Recompilation with On-Stack Replacement.* International Symposium on Code Generation and Optimization (CGO'03), 2003

FS02 D. Flanagan and D. Shafer. *JavaScript: The Definitive Guide.* O'Reilly, 2002.

GBO99 T.R.G. Green, A. Borning, T. O'Shea, M. Minoughan, and R.B. Smith. *The Stripetalk Papers: Understandability as a Language Design Issue in Object-Oriented Programming Systems* in *Prototype-Based Programming: Concepts, Languages and Applications*, Noble, J., Taivalsaari, A., Moore, I., (eds), Springer (1999) pp. 47-62.

GR83 Adele Goldberg and David Robson, *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, MA, 1983.

GroF Group F: Avri Bilovich, Chris Budd, Graham Cartledge, Rhys Evans, Mesar Hameed, and Michael Parry. http://www.bath.ac.uk/~ma3mp/SelfHome.htm.

GSO91 W.W. Gaver, R.B. Smith., and T. O'Shea. *Effective Sounds in Complex Systems: The ARKola Simulation*. Proc. CHI '91 conference, pp 85-90.

HCU91 Urs Hölzle, Craig Chambers, and David Ungar. *Optimizing Dynamically-Typed Object-Oriented Programs using Polymorphic Inline Caches.* In ECOOP' 91 Conference Proceedings, pp. 21-38, Geneva, Switzerland, July, 1991.

HCU92 Urs Hölzle, Craig Chambers, and David Ungar. *Debugging Optimized Code with Dynamic Deoptimization,* in Proc. ACM SIGPLAN '92 Conferences on Programming Language Design and Implementation, pp. 32-43, San Francisco, CA (June 1992).

HKKH96 H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal, *Specifying Subject-Oriented Composition*, Theory and Practice of Object Systems, volume 2, number 3, 1996, Wiley & Sons.

HN88 J. Hennessy and P. Nye. *Stanford Integer Benchmarks*. Stanford University, 1988.

Hoar73 C. A. R. Hoare. *Hints on Programming Language Design.* SIGACT/SIGPLAN Symposium on Principles of Programming Languages, October 1973. First published as Stanford University Computer Science Dept. Technical Report No. CS-73-403, Dec. 1973.

Höl94 Urs Hölzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming.* PhD Thesis, Stanford University, Computer Science Department, 1994.

HU94 Urs Hölzle and David Ungar. *Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback.* In Proceedings of the SIGPLAN 94 Conference on Programming Language Design and Implementation, Orlando, FL, June, 1994.

HU94a Urs Hölzle and David Ungar. *A Third Generation Self Implementation: Reconciling Responsiveness with Performance.* In OOPSLA'94 Conference Proceedings, pp. 229-243, Portland, OR, October, 1994. Published as SIGPLAN Notices 29(10), October, 1994.

HU95 Urs Hölzle and David Ungar. *Do Object-Oriented Languages Need Special Hardware Support?* ECOOP'95.

HU96 Urs Hölzle and David Ungar. *Reconciling Responsiveness with Performance in Pure Object-Oriented Languages.* TOPLAS. 1996.

D'Hont Theo D'Hont. *About PROG.* http://prog.vub.ac.be/progsite.

Inga81 Daniel H. H. Ingalls. *Design Principles Behind Smalltalk.* BYTE Magazine, August 1981, pp. 286-298.

Iver79 Kenneth E. Iverson. *Operators.* ACM TOPLAS, Vol. 1 No. 2, October 1979, pp. 161-176.

John79 Ronald L. Johnston. *The Dynamic Incremental Compiler of APL\3000. Proceedings of the APL'79 Conference.* Published as APL Quote Quad 9(4), p. 82-87, 1979.

John88 Ralph E. Johnson, Justin O. Graver, and Lawrence W. Zurawski. TS: An Optimizing Compiler for Smalltalk. In *OOPSLA'88 Conference Proceedings,* pp. 18-26, San Diego, CA, 1988. Published as *SIGPLAN Notices 23(11),* November, 1988.

Jone89 Charles M. Jones. *Chuck Amuck: The Life and Times of an Animated Cartoonist.* Farrar Straus Gifoux, New York, 1989.

Kay93 Alan Kay. *The Early History of Smalltalk.* ACM SIGPLAN Notices Vol. 28, Issue 3, March 1993. Also HOPL-II, 1993.

Krin03 Chandra Krintz. *Coupling On-Line and Off-Line Profile Information to Improve Program Performance.* CGO'03.

Lee88 Elgin Lee. *Object Storage and Inheritance for Self.* Engineer's thesis, Electrical Engineering Department, Stanford University, 1988.

Len05 Mark Lentczner. *Welcome to Wheat.* www.wheatfarm.org

Lieb86 Henry Lieberman. *Using Prototypical Objects to Implement Shared Behavior in Object Oriented System.* OOPSLA'86.

Ling04 K Lyngfelt, *MorphR—A Morphic GUI in Ruby.* Masters Thesis, University of Trollhättan ÅE Uddevalla., 2004.

LJ87 G. Lakoff and M. Johnson. *Women, Fire, and Dangerous Things: What Categories Reveal About the Mind.* University of Chicago Press, 1987.

LTP86 WR LaLonde, DA Thomas, JR Pugh, *An Exemplar Based Smalltalk.* OOPSLA'86.

Mal01 John Maloney. *An Introduction to Morphic: The Squeak User Interface Framework.* In *Squeak: Open Personal Computing and Multimedia,* Mark Guzdial and Kim Rose (eds.)., Prentice Hall, 2001.

MMM90 Ole Lehrmann Madsen, Boris Magnusson, and Birger Møller-Pedersen, *Strong Typing of Object-Oriented Languages Revisited.* In ECOOP/OOPSLA'90 Conference Proceedings, pp. 140-149, Ottawa, Canada, October, 1990.

MMN93 Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language,* Addison-Wesley Publishing Co., Wokingham, England, 1993.

Moor Ivan Moore. Guru. http://selfguru.sourceforge.net/index.html

Moo95 Ivan Moore. *Guru - A Tool for Automatic Restructuring of Self Inheritance Hierarchies* at TOOLS USA 1995

Moo96 Ivan Moore. *Automatic Inheritance Hierarchy Restructuring and Method Refactoring.* OOPSLA'96.

Moo96a Ivan Moore. *Automatic Restructuring of Object-Oriented Programs.* PhD Thesis, Manchester University, 1996.

MC93 P. Mulet and P. Cointe, *Definition of a Reflective Kernel for a Prototype-Based Language.* In Proceedings of the 1st International Symposium on Object Technologies for Advanced Software, Kanazawa, Japan, Springer-Verlag, Berlin. S. Nishio and A. Yonezawa (eds.), pp. 128-144, 1993. http::/citeseer.ist.psu.edu/article/mulet93definition.html.

MC96 Ivan Moore and Tim Clement. *A Simple and Efficient Algorithm for Inferring Inheritance Hierarchies* at TOOLS Europe 1996.

MGD90 Brad A. Myers, Dario Giuse, Roger B. Dannenberg, Brad Vander Zanden, David Kosbie, Ed Pervin, Andrew Mickish, and Philippe Marchal. *Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment.* This article was printed as *Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces*, IEEE Computer. Vol. 23, No. 11. November, 1990. pp. 71-85. Translated into Japanese and reprinted in Nikkei Electronics, No. 522, March 18, 1991, pp. 187-205.

MMM97 Brad A. Myers, Richard G. McDaniel, Robert C. Miller, Alan S. Ferrency, Andrew Faulring, Bruce D. Kyle, Andrew Mickish, Alex Klimovitski, and Patrick Doane. *The Amulet Environment: New Models for Effective User Interface Software Development*. IEEE Transactions on Software Engineering, Vol. 23, no. 6. June, 1997. pp. 347-365.

MS95 John Maloney and Randall B. Smith, *Directness and Liveness in the Morphic User Interface Construction Environment*. UIST'95.

Mule95 Phillipe Mulet, *Réflexion & Langages à Prototypes*, PhD Dissertation, Ecole des Mines de Nantes, France, 1995.

Myer97 Brad Myers. *Amulet Overview.* http://www.cs.cmu.edu/afs/cs/project/amulet/www/amulet-overview.html

Pal90 Joseph Pallas. *Multiprocessor Smalltalk: Implementation, Performance, and Analysis.* PhD Dissertation. Stanford University, 1990.

Parn72 D. L. Parnas. *On the Criteria To Be Used in Decomposing Systems into Modules.* CACM, Vol. 15, No. 12, December 1972. pp. 1053-1058.

PKB86 J. M. Pendleton, S.I. Kong, E.W. Brown, F. Dunlap, C. Marino, D. M. Ungar, D.A. Patterson, and D. A. Hodges. *A 32-bit Microprocessor for Smalltalk.* IEEE Journal of Solid-State Circuits, Vol. 21, Issue 5, Oct. 1986, pp. 741-749.

PMH05 Ellen Van Paesschen, Wolfgang De Meuter, and Maja D'Hondt. *SelfSync: a Dynamic Round-Trip Engineering Environment.* In OOPSLA'05: Companion of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications. San Diego, U.S.A. ACM Press, 2005. http://prog.vub.ac.be/Publications/2005/vub-prog-tr-05-20.pdf.

RBJ97 Don Roberts, John Brant, and Ralph Johnson. *A Refactoring Tool for Smalltalk.* Theory and Practice of Object Systems, Vol. 3, Issue 4, pp. 253-263, 1997, John Wiley & Sons, Inc.

RMC91 George Robertson, Jock Mackinlay, and Stuart Card. Cone Trees: Animated 3D Visualizations of Hierarchical Information, ACM Computer Human Interface Conference, 1991.

RRS99 Ian Rogers, Alasdair Rawsthorne, Jason Sologlou. *Exploiting Hardware Resources: Register Assignment Across Method Boundaries.* First Workshop on Hardware Support for Objects and Microarchitectures for Java, 1999.

RS Brian T. Rice and Lee Salzman. *The Home of the Slate Programming Language*. http://slate.tunes.org/

SA05 Lee Salzman and Johnathan Aldrich. *Prototypes with Multiple Dispatch: An Expressive and Dynamic Object Model.* ECOOP 2005.

SC02 Benoît Sonntag and Dominique Colnet. *Lisaac: The Power of Simplicity at Work for Operating System.* ACM Fortieth International Conference on Tools Pacific: Objects for internet, mobile, and embedded applications, 2002.

SDNB03 Nathaneal Scharli, Stephane Ducassse, Oscar Nierstrasz, and Andrew Black. *Traits: Composable Units of Behavior.* ECOOP 2003.

Setal90 R.B. Smith,T. O'Shea, E. O'Malley, E. Scanlon, and J. Taylor. *Preliminary experiments with a distributed, multimedia problem solving environment,* in Proceedings of EC-CSCW `90, Gatwick, England, pp 19-34. Also appears in J. Bowers and S. Benford, eds., Studies in Computer Supported Cooperative Work: theory, practice and design, Elsevier, Amsterdam, (1991) pp 31-48.

Setal93 E. Scanlon, R.B. Smith, T. O'Shea, C. O'Malley, and J. Taylor. *Running in the Rain—Can a Shared Simulation Help To Decide?* Physics Education, Vol 28, March 1993, pp 107-113.

Setal99 R.B. Smith, M.J. Sipusic, and R.L. Pannoni. *Experiments Comparing Face-to-Face with Virtual Collaborative Learning.* Proceedings of Conference on Computer Support for Collaborative Learning 1999. Stanford University, Palo Alto, December 1999. pp 558-566.

SL93 B. Schwartz and M. Lentczner. *Direct Programming Using a Unified Object Model.* In OOPSLA'92 Addendum to the Proceedings. Published as OOPS Messenger, 4.2, (1993) 237.

SLST94 R. B. Smith, M. Lentczner, W. Smith, A. Taivalsaari, and D. Ungar, *Prototype-Based Languages: Object Lessons from Class-Free Programming (Panel)*, in Proc. OOPSLA'94, pp. 102-112 (October 1994). Also see the panel summary of the same title, in Addendum to the Proceedings of OOPSLA '94, pp. 48-53.

SLU88 Lynn Stein, Henry Lieberman, and David Ungar. *The Treaty of Orlando.* ACM SIGPLAN Notices, Vol. 23, Issue 5, May 1988. Also in W. Kim and F. Lochovsky (eds.), *Object-Oriented Concepts, Databases, and Applications.* ACM Press and Addison-Wesley.

Smi87 Randall B. Smith. *Experiences with the Alternate Reality Kit, an Example of the Tension Between Literalism and Magic.* Proc. CHI + GI Conference, pp 61-67, Toronto, (April 1987).

Smi91 Smith, R.B., *A Prototype Futuristic Technology for Distance Education*, in New Directions in Educational Technology, Scanlon, E., and O'Shea, T., (eds.) Springer, Berlin, (1991) pp 131-138.

Smi93 D. Smith. *Pygmalion: An Executable Electronic Blackboard*. In W. Finzer and L. Gould, *Watch What I Do: Programming by Demonstration*, MIT Press, 1993, pp 19-48.

Smi94 Walter Smith. *Self and the Origins of NewtonScript*. PIE Developers Magazine, July 1994. http://wsmith.best.vwh.net/Self-intro.html.

Smi95 Walter R. Smith. *Using a Prototype-based Language for User Interface: The Newton Project's Experience.* OOPSLA'95.

SMU95 Randall B. Smith, John Maloney, and David Ungar, *The Self-4.0 User Interface: Manifesting a System-wide Vision of Concreteness, Uniformity, and Flexibility.* OOPSLA '95.

SOSL97 E. Scanlon, T. O'Shea, R.B. Smith, and Y. Li. *Supporting the Distributed Synchronous Learning of Probability: Learning from an Experiment*, In R. Hall, N. Miyake, and N. Enyedy (eds.) Proceedings of CSCL'97, The Second International Conference on Computer Support for Collaborative Learning Toronto, December 10-14, 1997, pp 224-230.

SS79 Guy Steele, Jr. and Gerald Jay Sussman. *Design of LISP-based Processors, or SCHEME: A Dielectric LISP, or Finite Memories Considered Harmful, or LAMBDA: The Ultimate Opcode.* AI Memo No. 514, MIT AI Laboratory, March 1979.

SSP99 M.J. Sipusic, R.L. Pannoni, R.B. Smith, J. Dutra, J.F. Gibbons, and W.R. Sutherland. *Virtual Collaborative Learning: A Comparison between Face-to-Face Tutored Video Instruction (TVI) and Distributed Tutored Video Instruction (DTVI)*, Sun Laboratories Technical Report SMLI-TR-99-72.

Strou86 Bjarne Stroustrup. *An Overview of C++*. SIGPLAN Notices, Vol. 21, #10, October 1986, pp. 7-18

StV96 *Self: the Video*. Videotape by Sun Microsystems Laboratories, Sun Labs document #0448, Oct. 1996.

SU94 R.B. Smith and D. Ungar. *Us: A Subjective Language with Perspective Objects.* SML-94-0416.

SU95 Randall Smith and David Ungar. *Programming as an Experience: The Inspiration for Self*. In Proceedings of the 9th European Conference, Århus, Denmark, Aug. 1995. Published as Lecture Notes in Computer Science 952: ECOOP'95 — Object-Oriented Programming, Walther Olthoff (ed.) Springer, Berlin. pp 303-330.

SU96 Randall Smith and David Ungar. A *Simple and Unifying Approach to Subjective Objects*, in Theory and Practice of Object Systems, Vol. 2, Issue 3, Special Issue on subjectivity in object-oriented systems, 1996.

SUC92 Randall B. Smith, David Ungar, and Bay-Wei Chang. *The Use Mention Perspective on Programming for the Interface*, In Brad A. Myers, *Languages for Developing User Interfaces*, Jones and Bartlett, Boston, MA, 1992. pp 79-89.

Suth63 Ivan Sutherland. *Sketch Pad*, AFIPS Spring Joint Computer Conference, Detroit, 1963.

SWU97 Randall B. Smith, Mario Wolczko, and David Ungar. *From Kansas to Oz: Collaborative Debugging When a Shared World Breaks.* CACM, April, 1997. pp 72-78.

Tai92 Antero Taivalsaari. *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.

Tai93 Antero Taivalsaari. *A Critical View of Inheritance and Reusability in Object-Oriented Programming.* PhD dissertation, Jyvaskyla Studies in Computer Science, Economics and Statistics 23, University of Jyvaskyla, Finland, December 1993, 276 pages (ISBN 951-34-0161-8).

Tai93a Antero Taivalsaari, *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.

TC95 Bruce H. Thomas and Paul Calder. *Animating Direct Manipulation Interfaces.* UIST'95.

TC01 Bruce H. Thomas and Paul Calder. *Applying Cartoon Animation Techniques to Graphical User Interfaces. ACM Transactions on Computer-Human Interaction*, Vol. 8, No. 3, September 2001, Pages 198–222.

TD02 B. H. Thomas and V. Demczuk. *Which Animation Effects Improve Indirect Manipulation?* Interacting with Computers 14, 2002, pp. 211-229. Elsevier. www.elsevier.com/locat/intcom

Tha86 Chuck Thacker. *Personal Distributed Computing: The Alto and Ethernet Hardware.* ACM Conference on the History of Personal Workstations,1986.

Thom98 Bruce H. Thomas. *Warping to Enhance 3D User Interfaces.* APCHI, p. 169, Third Asian Pacific Computer and Human Interaction, 1998. http://doi.ieeecomputersociety.org/10.1109/APCHI.1998.704188

TJ84 Frank Thomas and Ollie Johnson. *Disney Animation: The Illusion of Life.* New York, Abbeville, 1984.

TK02 Robert Tolksdorf and Kai Knubben. *Programming Distributed Systems with the Delegation-Based Object-Oriented Language dSelf.* ACM Symposium on Applied Computing, 2002, pp. 927-931.

TSS88 C. P. Thacker, L. Stewart, and E. H. Sattterthwaite. *Firefly: A Multiprocessor Workstation*. IEEE Transactions on Computers, 37(8):909-920, Aug. 1988.

Ung84 David Ungar. *Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm.* ACM Symposium on Practical Software Development Environments, 1984. Also in SIGPLAN Notices Vol. 19, Issue 5, 1984 and SIGSOFT Software Engineering Notices Vol. 9, Issue 3, 1984.

Ung87 David Ungar. *The Design and Evaluation of a High-Performance Smalltalk System.* MIT Press, 1987.

Ung95 David Ungar. *Annotating Objects for Transport to Other Worlds.* OOPSLA'95.

US87 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.

USCH92 David Ungar, Randall B. Smith, Craig Chambers, and Urs Hölzle. *Object, Message, and Performance: How They Coexist in Self.* Computer, 25(10), pp. 53-64. (October 1992).

USA05 David Ungar, Adam Spitz, and Alex Ausch. *Constructing a Metacircular Virtual Machine in an Exploratory Programming Environment.* OOPSLA 2005.

WAU96 Mario Wolczko, Ole Agesen, and David Ungar. *Towards a Universal Implementation Substrate for Object-Oriented Languages.* Sun Labs 96-0506. http://www.sunlabs.com/people/mario/pubs/substrate.pdf, 1996. Also presented at OOPSLA'99 workshop on Simplicity, Performance and Portability in Virtual Machine Design.

Wha01 John Whaley. *Partial method compilation using dynamic profile information.* OOPSLA'01.

Wol96 Mario Wolczko. *Self Includes: Smalltalk. I*n *Prototype-Based Programming: Concepts, Languages and Applications,* J. Noble, A. Taivalsaari, and I. Moore, (eds), Springer (1999).