An Early History of Smalltalk

Alan Kay

VPRI Paper for Historical Context

Viewpoints Research Institute, 1025 Westwood Blvd 2nd flr, Los Angeles, CA 90024 t: (310) 208-0524

XI

Smalltalk Session

Chair: *Barbara Ryder*
Discussant: *Adele Goldberg*

THE EARLY HISTORY OF SMALLTALK

Alan C. Kay

Apple Computer
kay2@applelink.apple.com@Internet#

ABSTRACT

Most ideas come from previous ideas. The sixties, particularly in the ARPA community, gave rise to a host of notions about "human-computer symbiosis" through interactive time-shared computers, graphics screens, and pointing devices. Advanced computer languages were invented to simulate complex systems such as oil refineries and semi-intelligent behavior. The soon to follow paradigm shift of modern personal computing, overlapping window interfaces, and object-oriented design came from seeing the work of the sixties as something more than a "better old thing." That is, more than a better way: to do mainframe computing; for end-users to invoke functionality; to make data structures more abstract. Instead the promise of exponential growth in computing/$/volume demanded that the sixties be regarded as "*almost* a new thing" and to find out what the actual "new things" might be. For example, one would compute with a handheld "Dynabook" in a way that would not be possible on a shared main-frame; millions of potential users meant that the user interface would have to become a learning environment along the lines of Montessori and Bruner; and needs for large scope, reduction in complexity, and end-user literacy would require that data and control structures be done away with in favor of a more biological scheme of protected universal cells interacting only through messages that could mimic any desired behavior.

Early Smalltalk was the first complete realization of these new points of view as parented by its many predecessors in hardware, language, and user interface design. It became the exemplar of the new computing, in part, because we were actually trying for a qualitative shift in belief structures—a new Kuhnian paradigm in the same spirit as the invention of the printing press—and thus took highly extreme positions that almost forced these new styles to be invented.

CONTENTS

ALAN C. KAY

—To Dan Ingalls, Adele Goldberg and the rest of the Xerox PARC LRC gang

—To Dave Evans, Bob Barton, Marvin Minsky, and Seymour Papert

—To SKETCHPAD, JOSS, LISP and SIMULA, the four great progrmming conceptions of the sixties

INTRODUCTION

I am writing this introduction in an airplane at 35,000 feet. On my lap is a five-pound notebook computer—1992's "Interim Dynabook"—by the end of the year it sold for under $700. It has a flat, crisp, high-resolution bitmap screen, overlapping windows, icons, a pointing device, considerable storage and computing capacity, and its best software is object-oriented. It has advanced networking built in and there are already options for wireless networking. Smalltalk runs on this system, and is one of the main systems I use for my current work with children. In some ways this is more than a Dynabook (quantitatively), and some ways not quite there yet (qualitatively). All in all, pretty much what was in mind during the late sixties.

Smalltalk was part of this larger pursuit of ARPA, and later of Xerox PARC, that I called personal computing. There were so many people involved in each stage from the research communities that the accurate allocation of credit for ideas is intractably difficult. Instead, as Bob Barton liked to quote Goethe, we should "share in the excitement of discovery without vain attempts to claim priority."

I will try to show where most of the influences came from and how they were transformed in the magnetic field formed by the new personal computing metaphor. It was the attitudes as well as the great ideas of the pioneers that helped Smalltalk get invented. Many of the people I admired most at this time—such as Ivan Sutherland, Marvin Minsky, Seymour Papert, Gordon Moore, Bob Barton, Dave Evans, Butler Lampson, Jerome Bruner, and others—seemed to have a splendid sense that their creations, though wonderful by relative standards, were not near to the absolute thresholds that had to be crossed. Small minds try to form religions, the great ones just want better routes up the mountain. Where Newton said he saw further by standing on the shoulders of giants, computer scientists all too often stand on each other's toes. Myopia is still a problem when there are giants' shoulders to stand on—"outsight" is better than insight—but it can be minimized by using glasses whose lenses are highly sensitive to esthetics and criticism.

Programming languages can be categorized in a number of ways: imperative, applicative, logic-based, problem-oriented, and so on. But they all seem to be either an "agglutination of features" or a "crystalization of style." COBOL, PL/1, Ada, and the like, belong to the first kind; LISP, APL—and Smalltalk—are the second kind. It is probably not an accident that the agglutinative languages all seem to have been instigated by committees, and the crystalization languages by a single person.

Smalltalk's design—and existence—is due to the insight that everything we can describe can be represented by the recursive composition of a single kind of behavioral building block that hides its combination of state and process inside itself and can be dealt with only through the exchange of messages. Philosophically, Smalltalk's objects have much in common with the monads of Leibniz

and the notions of 20th century physics and biology. Its way of making objects is quite Platonic in that some of them act as idealizations of concepts—*Ideas*—from which *manifestations* can be created. That the Ideas are themselves manifestations (of the Idea-Idea) and that the Idea-Idea is a-kind-of Manifestation-Idea—which is a-kind-of itself, so that the system is completely self-describing—would have been appreciated by Plato as an extremely practical joke [Plato].

In computer terms, Smalltalk is a recursion on the notion of computer itself. Instead of dividing "computer stuff" into things each less strong than the whole—such as data structures, procedures, and functions that are the usual paraphernalia of programming languages—each Smalltalk object is a recursion of the entire possibilities of the computer. Thus its semantics are a bit like having thousands and thousands of computers all hooked together by a very fast network. Questions of concrete representation can thus be postponed almost indefinitely because we are mainly concerned that the computers behave appropriately, and are interested in particular strategies only if the results are off or come back too slowly.

Though it has noble ancestors indeed, Smalltalk's contribution is a new design paradigm—which I called *object-oriented*—for attacking large problems of the professional programmer, and making small ones possible for the novice user. Object-oriented design is a successful attempt to qualitatively improve the efficiency of modeling the ever more complex dynamic systems and user relationships made possible by the silicon explosion.

"We would know what they thought when they did it"
—Richard Hamming

"Memory and imagination are but two words for the same thing"
—Thomas Hobbes

In this history I will try to be true to Hamming's request as moderated by Hobbes' observation. I have had difficulty in previous attempts to write about Smalltalk because my emotional involvement has always been centered on personal computing as an amplifier for human reach—rather than programming system design—and we haven't got there yet. Though I was the instigator and original designer of Smalltalk, it has always belonged more to the people who made it work and got it out the door, especially Dan Ingalls and Adele Goldberg. Each of the LRGers contributed in deep and remarkable ways to the project, and I wish there was enough space to do them all justice. But I think all of us would agree that for most of the development of Smalltalk, Dan was the central figure. Programming is at heart a practical art in which real things are built, and a real implementation thus has to exist. In fact, many if not most languages are in use today not because they have any real merits but because of their existence on one or more machines, their ability to be bootstrapped,and so on. But Dan was far more than a great implementer; he also became more and more of the designer, not just of the language but also of the user interface as Smalltalk moved into the practical world.

Here, I will try to center focus on the events leading up to Smalltalk-72 and its transition to its modern form as Smalltalk-76. Most of the ideas occurred here, and many of the earliest stages of OOP are poorly documented in references almost impossible to find.

This history is too long, but I was amazed at how many people and systems that had an influence appear only as shadows or not at all. I am sorry not to be able to say more about Bob Balzer, Bob Barton, Danny Bobrow, Steve Carr, Wes Clark, Barbara Deutsch, Peter Deutsch, Bill Duvall, Bob Flegal, Laura Gould, Bruce Horn, Butler Lampson, Dave Liddle, William Newman, Bill Paxton, Trygve Reenskaug, Dave Robson, Doug Ross, Paul Rovner, Bob Sproull, Dan Swinehart, Bert Sutherland, Bob Taylor, Warren Teitelman, Bonnie Tennenbaum, Chuck Thacker, and John Warnock. Worse, I have omitted to mention many systems whose design I detested, but that generated

considerable, useful ideas and attitudes in reaction. In other words, "histories" should not be believed very seriously but considered as "FEEBLE GESTURES OFF" done long after the actors have departed the stage.

Thanks to the numerous reviewers for enduring the many drafts they had to comment on. Special thanks to Mike Mahoney for helping so gently that I heeded his suggestions and so well that they greatly improved this essay—and to Jean Sammet, an old, old friend, who quite literally frightened me into finishing it—I did not want to find out what would happen if I were late. Sherri McLoughlin and Kim Rose were of great help in getting all the materials together.

11.1 1960–1966—EARLY OOP AND OTHER FORMATIVE IDEAS OF THE SIXTIES

Though OOP came from many motivations, two were central. The large-scale one was to find a better module scheme for complex systems involving hiding of details, and the small-scale one was to find a more flexible version of assignment, and then to try to eliminate it altogether. As with most new ideas, it originally happened in isolated fits and starts.

New ideas go through stages of acceptance, both from within and without. From within, the sequence moves from "barely seeing" a pattern several times, then noting it but not perceiving its "cosmic" significance, then using it operationally in several areas; then comes a "grand rotation" in which the pattern becomes the center of a new way of thinking, and finally, it turns into the same kind of inflexible religion that it originally broke away from. From without, as Schopenhauer noted, the new idea is first denounced as the work of the insane, in a few years it is considered obvious and mundane, and finally the original denouncers will claim to have invented it.

True to the stages, I "barely saw" the idea several times circa 1961 while a programmer in the Air Force. The first was on the Burroughs 220 in the form of a style for transporting files from one Air Training Command installation to another. There were no standard operating systems or file formats back then, so some (to this day unknown) designer decided to finesse the problem by taking each file and dividing it into three parts. The third part was all the actual data records of arbitrary size and format. The second part contained the B220 procedures that knew how to get at records and fields to copy and update the third part. And the first part was an array of relative pointers into entry points of the procedures in the second part (the initial pointers were in a standard order representing standard meanings). Needless to say, this was a great idea, and was used in many subsequent systems until the enforced use of COBOL drove it out of existence.

The second barely seeing of the idea came just a little later when ATC decided to replace the 220 with a B5000. I did not have the perspective to really appreciate it at the time, but I did take note of its segmented storage system,

FIGURE 11.1 USAF ATG Randolph AFB B220 File Format ca. 1961

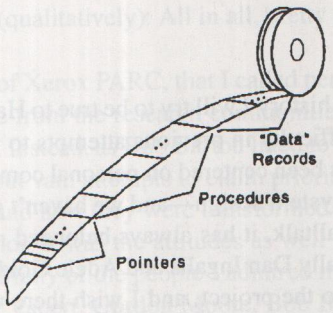


FIGURE 11.2 Gordon Moore's "Law"

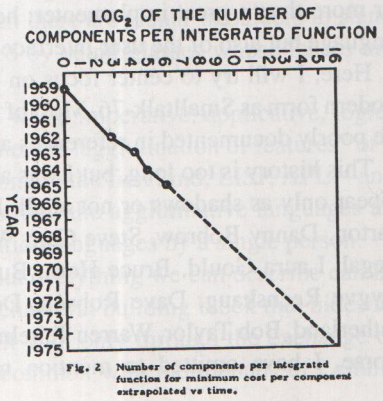


Fig. 2 Number of components per integrated function for minimum cost per component extrapolated vs time.

its efficiency of HLL compilation and byte-coded execution, its automatic mechanisms for subroutine calling and multiprocess switching, its pure code for sharing, its protection mechanisms, and the like. And, I saw that the access to its Program Reference Table corresponded to the 220 file system scheme of providing a procedural interface to a module. However, my big hit from this machine at this time was not the OOP idea, but some insights into HLL translation and evaluation [Barton 1961; Burroughs 1961].

After the Air Force, I worked my way through the rest of college by programming mostly retrieval systems for large collections of weather data for the National Center for Atmospheric Research. I got interested in simulation in general—particularly, of one machine by another—but aside from doing a one-dimensional version of a bit-field block transfer (bitblt) on a CDC 6600 to simulate word sizes of various machines, most of my attention was distracted by school, or I should say the theatre at school. While in Chippewa Falls helping to debug the 6600, I read an article by Gordon Moore that predicted that integrated silicon on chips was going to exponentially improve in density and cost over many years. At that time in 1965, standing next to the room-sized freon-cooled 10 mip 6600, his astounding predictions had little projection into my horizons.

11.1.1 Sketchpad and Simula

Through a series of flukes, I wound up in graduate school at the University of Utah in the Fall of 1966, "knowing nothing." That is to say, I had never heard of ARPA or its projects, or that Utah's main goal in this community was to solve the "hidden line" problem in 3D graphics, until I actually walked into Dave Evans's office looking for a job and a desk. On Dave's desk was a foot-high stack of brown covered documents, one of which he handed to me: "Take this and read it."

Every newcomer got one. The title was "Sketchpad: A man-machine graphical communication system"[Sutherland 1963]. What it could do was quite remarkable, and completely foreign to any use of a computer I had ever encountered. The three big ideas that were easiest to grapple with were: it was the invention of modern interactive computer graphics; things were described by making a "master drawing" that could produce "instance drawings"; control and dynamics were supplied by "constraints," also in graphical form, that could be applied to the masters to shape and interrelate parts. Its data structures were hard to understand—the only vaguely familiar construct was the embedding of pointers to procedures and using a process called reverse indexing to jump though them

FIGURE 11.3 When there was only one personal computer. Ivan at the TX-2 ca. 1962

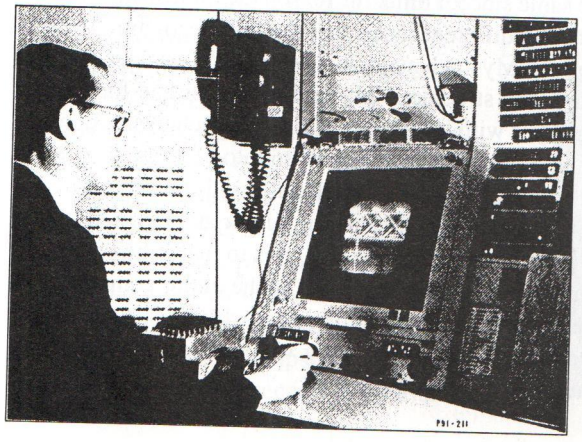


FIGURE 11.4 Drawing in Sketchpad

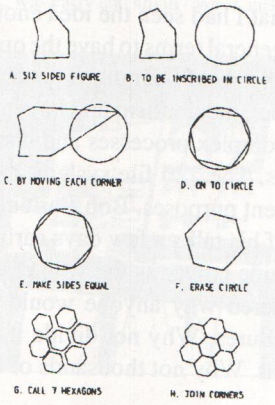


FIGURE 11.5 Programming with constraints

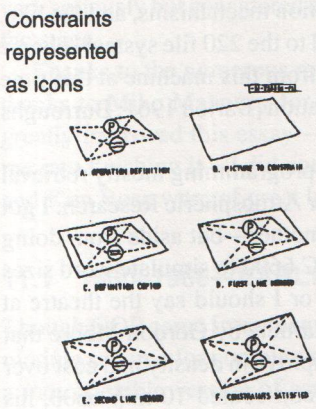
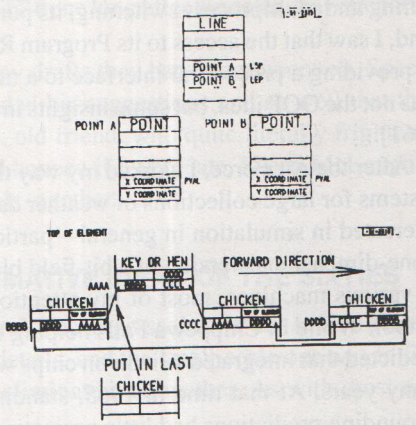


FIGURE 11.6 Sketchpad Structures



to routines, like the 220 file system [Ross1961]. It was the first to have clipping and zooming windows—one "sketched" on a virtual sheet about one third mile square!

Head whirling, I found my desk. On it was a pile of tapes and listings, and a note: "This is the Algol for the 1108. It doesn't work. Please make it work." The latest graduate student gets the latest dirty task.

The documentation was incomprehensible. Supposedly, this was the Case-Western Reserve 1107 ALGOL—but it had been doctored to make a language called Simula; the documentation read like Norwegian transliterated into English, which in fact it was. There were uses of words like *activity* and *process* that did not seem to coincide with normal English usage.

Finally, another graduate student and I unrolled the program listing 80 feet down the hall and crawled over it yelling discoveries to each other. The weirdest part was the storage allocator, which did not obey a stack discipline as was usual for ALGOL. A few days later, that provided the clue. What Simula was allocating were structures very much like the instances of Sketchpad. There were descriptions that acted like masters and they could create instances, each of which was an independent entity. What Sketchpad called masters and instances, Simula called activities and processes. Moreover, Simula was a procedural language for controlling Sketchpad-like objects, thus having considerably more flexibility than constraints (though at some cost in elegance) [Nygaard1966, 1983].

This was the big hit, and I have not been the same since. I think the reason the hit had such impact was that I had seen the idea enough times in enough different forms that the final recognition was in such general terms to have the quality of an epiphany. My math major had centered on abstract algebras with their few operations generally applying to many structures. My biology major had focused on both cell metabolism and larger scale morphogenesis with its notions of simple mechanisms controlling complex processes and one kind of building block able to differentiate into all needed building blocks. The 220 file system, the B5000, Sketchpad, and finally Simula, all used the same idea for different purposes. Bob Barton, the main designer of the B5000 and a professor at Utah, had said in one of his talks a few days earlier: "The basic principle of recursive design is to make the parts have the same power as the whole." For the first time I thought of the whole as the entire computer and wondered why anyone would want to divide it up into weaker things called data structures and procedures. Why not divide it up into little computers, as time-sharing was starting to? But not in dozens. Why not thousands of them, each simulating a useful structure?

FIGURE 11.7 "Generic block" showing procedural attachment

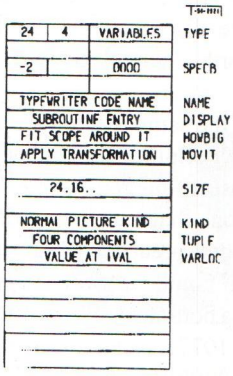
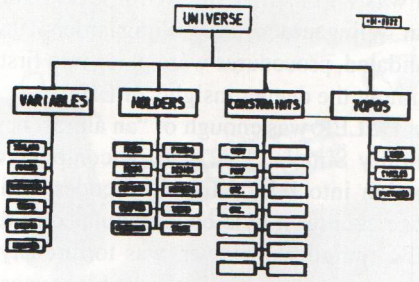


FIGURE 11.8 Sketchpad's "inheritance" hierarchy



I recalled the monads of Leibniz, the "dividing nature at its joints" discourse of Plato, and other attempts to parse complexity. Of course, philosophy is about opinion and engineering is about deeds, with science the happy medium somewhere in between. It is not too much of an exaggeration to say that most of my ideas from then on took their roots from Simula—but not as an attempt to improve it. It was the promise of an entirely new way to structure computations that took my fancy. As it turned out, it would take quite a few years to understand how to use the insights and to devise efficient mechanisms to execute them.

11.2 1967–69—THE FLEX MACHINE, A FIRST ATTEMPT AT AN OOP-BASED PERSONAL COMPUTER

Dave Evans was not a great believer in graduate school as an institution. As with many of the ARPA "contractors" he wanted his students to be doing "real things"; they should move through graduate school as quickly as possible; and their theses should advance the state of the art. Dave would often get consulting jobs for his students, and in early 1967, he introduced me to Ed Cheadle, a friendly hardware genius at a local aerospace company who was working on a "little machine." It was not the first personal computer—that was the LINC of Wes Clark—but Ed wanted it for noncomputer professionals; in particular, he wanted to program it in a higher level language, like BASIC. I said: "What about JOSS? It's nicer." He said: "Sure, whatever you think," and that was the start of a very pleasant collaboration we called the FLEX machine. As we got deeper into the design, we realized that we wanted to dynamically *simulate* and *extend*, neither of which JOSS (or any existing language that I knew of) was particularly good at. The machine was too small for Simula, so that was out. The beauty of JOSS was the extreme attention of its design to the end-user—in this respect, it has not been surpassed [Joss 1964, 1978]. JOSS was too slow for serious computing (but see also [Lampson 1966]), and did not have real procedures, variable

FIGURE 11.9 "The LINC was early and small" Wes Clark and the LINC, ca. 1962