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

—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.

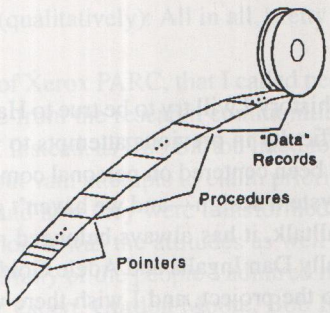
Thanks to the numerous reviewers for enduring the many drafts they had to comment on. Special thanks to Mike Mahoney for helping so gently that I heeded his suggestions and so well that they greatly improved this essay—and to Jean Sammet, an old, old friend, who quite literally frightened me into finishing 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.

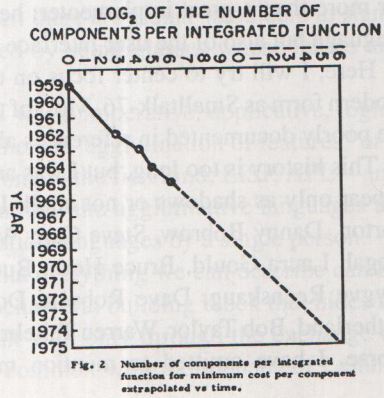
New ideas go through stages of acceptance, both from within and without. From within, the sequence moves from "barely seeing" a pattern several times, then noting it but not perceiving its "cosmic" significance, then using it operationally in several areas; then comes a "grand rotation" in which the 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.

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



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.

FIGURE 11.2 Gordon Moore's "Law"



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,

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

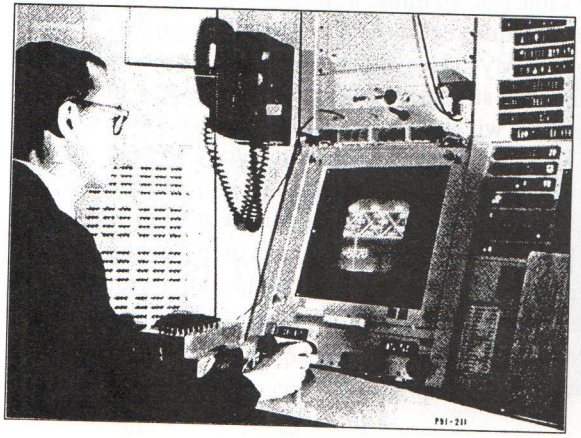


FIGURE 11.4 Drawing in Sketchpad

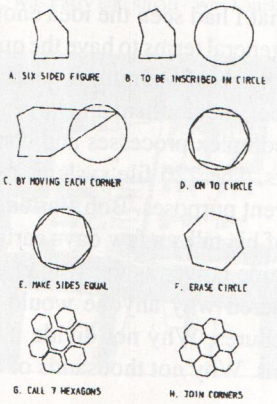


FIGURE 11.5 Programming with constraints

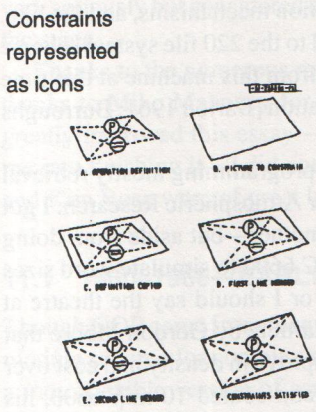
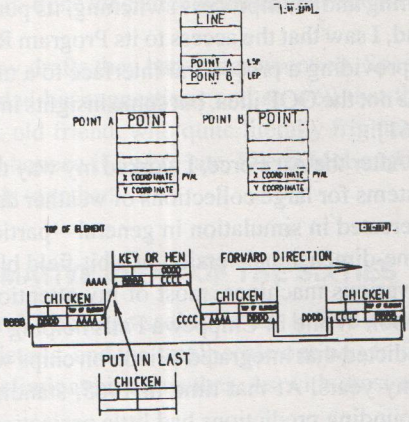


FIGURE 11.6 Sketchpad Structures



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

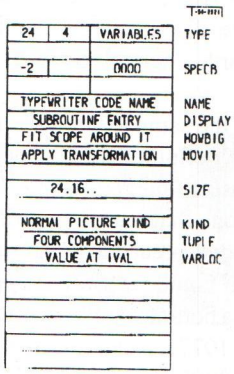
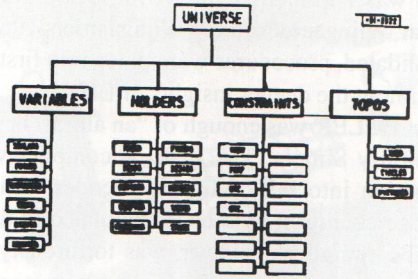


FIGURE 11.8 Sketchpad's "inheritance" hierarchy

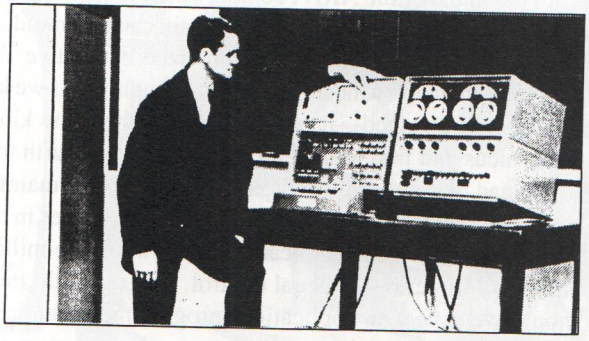


I recalled the monads of Leibniz, the "dividing nature at its joints" discourse of Plato, and other attempts to parse complexity. Of course, philosophy is about opinion and engineering is about deeds, with science the happy medium somewhere in between. It is not too much of an exaggeration to say that 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



scope, and so forth. A language that looked a little like JOSS but had considerably more potential power was Wirth's EULER [Wirth 1966]. This was a generalization of Algol along lines first set forth by van Wijngaarden [van Wijngaarden 1968] in which types were discarded, different features consolidated, procedures were made into first-class objects, and so forth—actually kind of LISPlike, but without the deeper insights of LISP.

But EULER was enough of "an almost new thing" to suggest that the same techniques be applied to simplify Simula. The EULER compiler was a part of its formal definition and made a simple conversion into B5000-like byte-codes. This was appealing because it suggested that Ed's little machine could run byte-codes emulated in the longish slow microcode that was then possible. The EULER compiler, however, was torturously rendered in an "extended precedence" grammar that actually required concessions in the language syntax (for example, "," could only be used in one role because the precedence scheme had no state space). I initially adopted a bottom-up Floyd-Evans parser (adapted from Jerry Feldman's original compiler-compiler [Feldman 1977]) and later went to various top-down schemes, several of them related to Schorre's META II [Schorre 1963] that eventually put the translator in the name space of the language.

The semantics of what was now called the FLEX language needed to be influenced more by Simula than by ALGOL or EULER. But it was not completely clear how. Nor was it clear how the user should interact with the system. Ed had a display (for graphing, and so forth) even on his first machine, and the LINC had a "glass teletype," but a Sketchpad-like system seemed far beyond the scope of what we could accomplish with the maximum of 16k 16-bit words that our cost budget allowed.

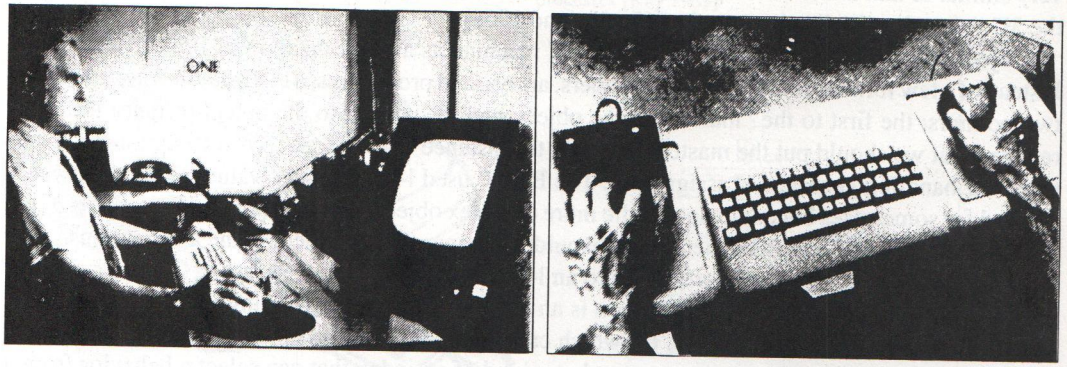
11.2.1 Doug Engelbart and NLS

This was in early 1967, and while we were pondering the FLEX machine, Utah was visited by Doug Engelbart. A prophet of Biblical dimensions, he was very much one of the fathers of what on the FLEX machine I had started to call "personal computing." He actually traveled with his own 16mm projector with a remote control for starting and stopping it to show what was going on (people were not used to seeing and following cursors back then). His notion of the ARPA dream was that the destiny of oNLine Systems (NLS) was the "augmentation of human intellect" via an interactive vehicle navigating through "thought vectors in concept space." What his system could do then—even by today's standards—was incredible. Not just hypertext, but graphics, multiple panes, efficient navigation and command input, interactive collaborative work, and so on. An entire conceptual world and world view [Engelbart 1968]. The impact of this vision was to produce in the minds of those who were "eager to be augmented" a compelling metaphor of what interactive computing should be like, and I immediately adopted many of the ideas for the FLEX machine.

In the midst of the ARPA context of human-computer symbiosis and in the presence of Ed's "little machine," Gordon Moore's "Law" again came to mind, this time with great impact. For the first time I made the leap of putting the room-sized interactive TX-2 or even a 10 mip 6600 on a desk. I was almost frightened by the implications; computing as we knew it could not survive—the actual meaning of the word changed—it must have been the same kind of disorientation people had after reading Copernicus and first looked up from a different Earth to a different Heaven.

Instead of at most a few thousand *institutional* mainframes in the world—even today in 1992 it is estimated that there are only 4000 IBM mainframes in the entire world—and at most a few thousand users trained for each application, there would be millions of *personal* machines and users, mostly outside of direct institutional control. Where would the applications and training come from? Why should we expect an applications programmer to anticipate the specific needs of a particular one of the millions of potential users? An *extensional* system seemed to be called for in which the end-users

FIGURE 11.10 A very modern picture: Doug Englebart, ca. 1967



would do most of the tailoring (and even some of the direct construction) of their tools. ARPA had already figured this out in the context of their early successes in time-sharing. Their larger metaphor of human-computer symbiosis helped the community avoid making a religion of their subgoals and kept them focused on the abstract holy grail of "augmentation."

One of the interesting features of NLS was that its user interface was parametric and could be supplied by the end-user in the form of a "grammar of interaction" given in their compiler-compiler TreeMeta. This was similar to William Newman's early "Reaction Handler" work in specifying interfaces by having the end-user or developer construct through tablet and stylus an iconic regular expression grammar with action procedures at the states (NLS allowed embeddings via its context-free rules). This was attractive in many ways, particularly William's scheme, but to me there was a monstrous bug in this approach. Namely, these grammars forced the user to be in a system state that required getting out of before any new kind of interaction could be done. In hierarchical menus or "screens" one would have to backtrack to a master state in order to go somewhere else. What seemed to be required were states in which there was a transition arrow to every other state—not a fruitful concept in formal grammar theory. In other words, a much "flatter" interface seemed called for—but could such a thing be made interesting and rich enough to be useful?

Again, the scope of the FLEX machine was too small for a miniNLS, and we were forced to find alternate designs that would incorporate some of the power of the new ideas, and in some cases to improve them. I decided that Sketchpad's notion of a general window that viewed a larger virtual world was a better idea than restricted horizontal panes and with Ed came up with a clipping algorithm

FIGURE 11.11 Multiple Panes and View Specs in NLS

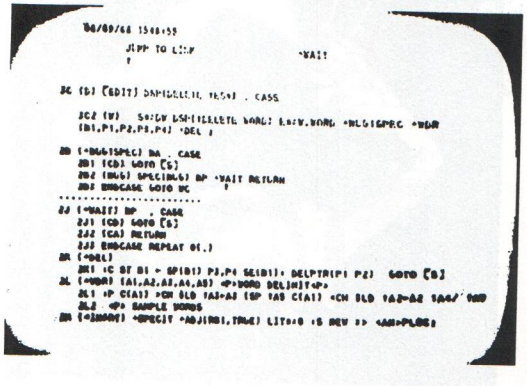
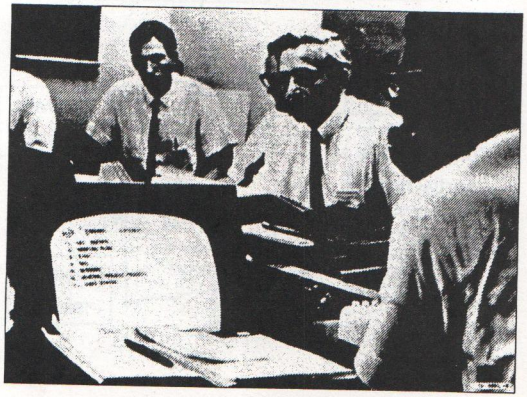


FIGURE 11.12 Collaborative work using NLS



very similar to that under development at the same time by Sutherland and his students at Harvard for the 3D "virtual reality" helmet project [Sutherland 1968].

Object references were handled on the FLEX machine as a generalization of B5000 descriptors. Instead of a few formats for referencing numbers, arrays, and procedures, a FLEX descriptor contained two pointers: the first to the "master" of the object, and the second to the object instance (later we realized that we should put the master pointer in the instance to save space). A different method was taken for handling generalized assignment. The B5000 used l-values and r-values [Strachey] which worked for some cases but could not handle more complex objects. For example: *a[55] := 0*, if *a* was a sparse array whose default element was *0* would still generate an element in the array because := is an "operator" and *a[55]* is dereferenced into an l-value before anyone gets to see that the r-value is the default element, regardless of whether *a* is an array or a procedure fronting for an array. What is needed is something like: *a(55, ':=', 0)*, which can look at all relevant operands before any store is made. In other words, := is not an operator, but a kind of an index that can select a behavior from a complex object. It took me a remarkably long time to see this, partly I think because one has to invert the traditional notion of operators and functions, and the like, to see that objects need to privately own all of their behaviors: *that objects are a kind of mapping whose values are its behaviors*. A book on logic by Carnap [Carnap 1947] helped by showing that "intensional" definitions covered the same territory as the more traditional extensional technique and were often more intuitive and convenient.

As in Simula, a coroutining control structure [Conway 1963] was used as a way to suspend and resume objects. Persistent objects such as files and documents were treated as suspended processes and were organized according to their ALGOL-like static variable scopes. These were shown on the screen and could be opened by pointing at them. Coroutining was also used as a control structure for looping. A single operator **while** was used to test the generators which returned **false** when unable to furnish a new value. Booleans were used to link multiple generators. So a "for-type" loop would be written as:

```
while i <= 1 to 30 by 2 ^ j <= 2 to k by 3 do j<- j * i;
```

where the ... **to** ... **by**... was a kind of coroutine object. Many of these ideas were reimplemented in a stronger style in Smalltalk later on.

Another control structure of interest in FLEX was a kind of event-driven "soft interrupt" called **when**. Its Boolean expression was compiled into a "tournament sort" tree that cached all possible

FIGURE 11.13 Flex when statement [Kay 1969]

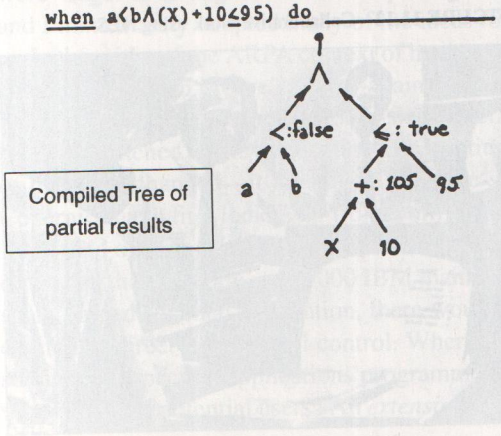


FIGURE 11.14 The first plasma panel

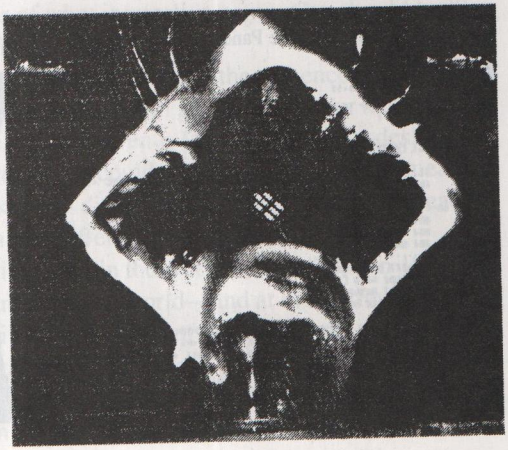


FIGURE 11.15 The FLEX machine self portrait, c. 1968 [Kay 1969]

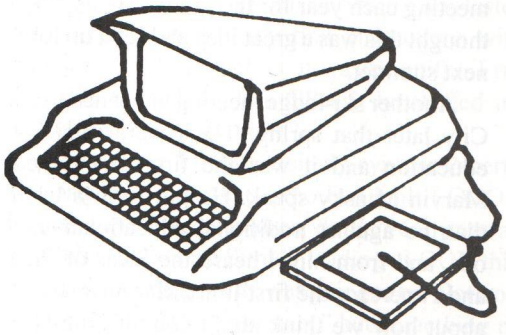
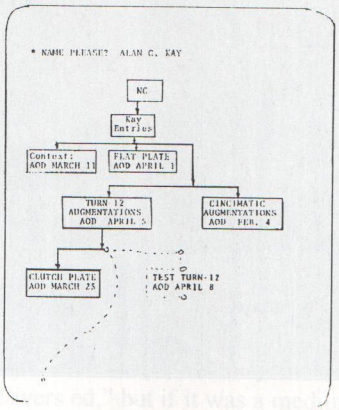


FIGURE 11.16 FLEX user interface. "Files" as suspended processes [Kay 1969]



intermediate results. The relevant variables were threaded through all of the sorting trees in all of the **when**s so that any change only had to compute through the necessary parts of the Booleans. The efficiency was very high and was similar to the techniques now used for spreadsheets. This was an embarrassment of riches with difficulties often encountered in event-driven systems. Namely, it was a complex task to control the *context* of just when the **when**s should be sensitive. Part of the Boolean expression had to be used to check the contexts, where I felt that somehow the structure of the program should be able to set and unset the event drivers. This turned out to be beyond the scope of the FLEX system and needed to wait for a better architecture.

Still, quite a few of the original FLEX ideas in their proto-object form did turn out to be small enough to be feasible on the machine. I was writing the first compiler when something unusual happened: the Utah graduate students got invited to the ARPA contractors' meeting held that year at Alta, Utah. Toward the end of the three days, Bob Taylor, who had succeeded Ivan Sutherland as head of ARPA-IPTO, asked the graduate students (sitting in a ring around the outside of the 20 or so contractors) if they had any comments. John Warnock raised his hand and pointed out that since the

FIGURE 11.17 FLEX machine window clipping [Kay 1969]

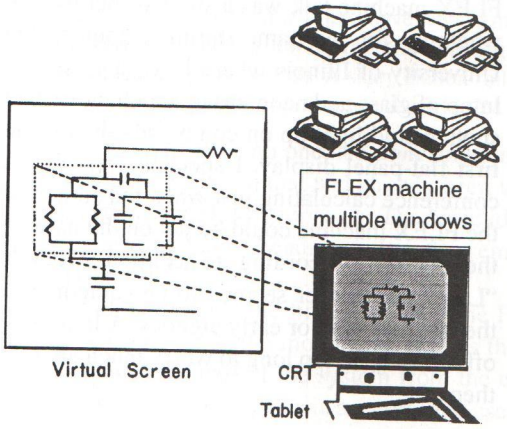


FIGURE 11.18 FLEX machine object structure [Kay 1969]

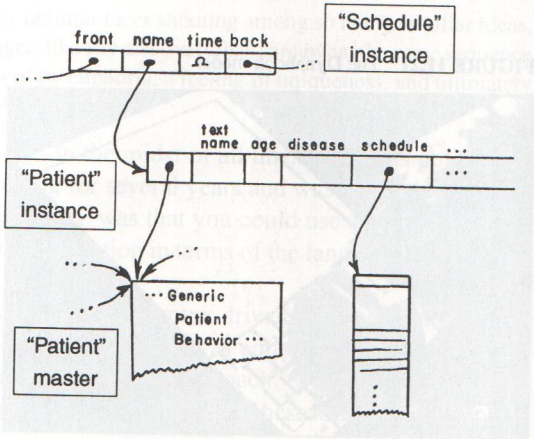


FIGURE 11.19 GRAIL

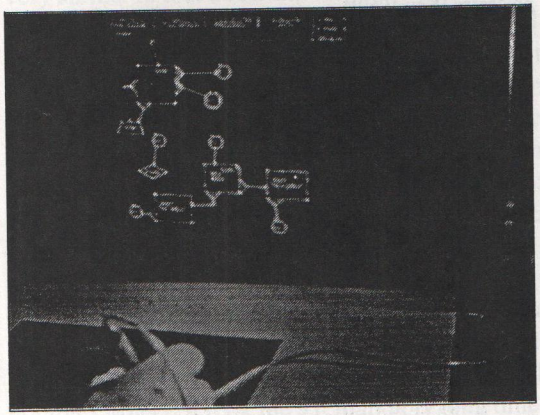


FIGURE 11.20 Seymour Papert and LOGO Turtle

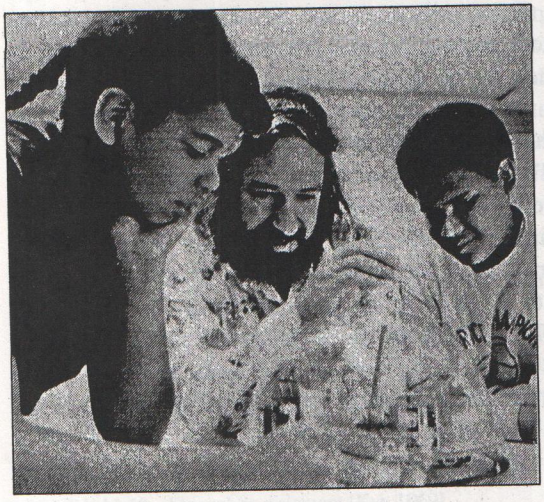
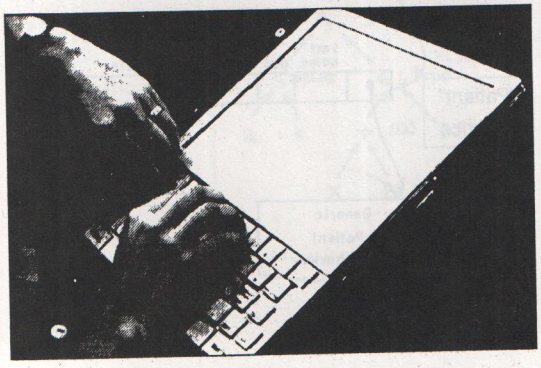


FIGURE 11.21 The Dynabook model



ARPA grad students would all soon be colleagues (and since we did all the real work anyway), ARPA should have a contractors-type meeting each year for the grad students. Taylor thought this was a great idea and set it up for the next summer.

Another ski-lodge meeting happened in Park City later that spring. The general topic was education and it was the first time I heard Marvin Minsky speak. He put forth a terrific diatribe against traditional educational methods, and from him I heard the ideas of Piaget and Papert for the first time. Marvin's talk was about how we think about complex situations and why schools are really bad places to learn these skills. He did not have to make any claims about computers+kids to make his point. It was clear that education and learning had to be rethought in the light of 20th century cognitive psychology and how good thinkers really think. Computing enters as a new representation system with new and useful metaphors for dealing with complexity, especially of systems [Minsky 1970].

For the summer 1968 ARPA grad students meeting at Allerton House in Illinois, I boiled all the mechanisms in the FLEX machine down into one 2'x3' chart. This included all of the "object structures," the compiler, the byte-code interpreter, i/o handlers, and a simple display editor for text and graphics. The grad students were a distinguished group that did indeed become colleagues in subsequent years. My FLEX machine talk was a success, but the big whammy for me came during a tour to the University of Illinois where I saw a 1" square lump of glass and neon gas in which individual spots would light up on command—it was the first flat-panel display. I spent the rest of the conference calculating just when the silicon of the FLEX machine could be put on the back of the display. According to Gordon Moore's "Law," the answer seemed to be sometime in the late seventies or early eighties. A long time off—it seemed too long to worry much about it then.

But later that year at RAND I saw a truly beautiful system. This was GRAIL, the graphical followon to JOSS. The first tablet (the famous RAND tablet) was invented by Tom Ellis [Davis 1964] in order to capture human gestures, and Gabe Groner wrote a program to efficiently recognize and respond to them [Groner 1966]. Though everything was fastened with bubble gum and the system crashed often, I have never forgotton my first interactions with this system. It was direct manipulation, it was analogical, it was modeless, it was beautiful. I realized that the FLEX interface was all wrong, but how could something like GRAIL be stuffed into such a tiny machine since it required all of a stand-alone 360/44 to run in?

A month later, I finally visited Seymour Papert, Wally Feurzig, Cynthia Solomon, and some of the other original researchers who had built LOGO and were using it with children in the Lexington schools. Here were children doing real programming with a specially designed language and environment. As with Simula leading to OOP, this encounter finally hit me with what the destiny of personal computing *really* was going to be. Not a personal dynamic *vehicle*, as in Engelbart's metaphor opposed to the IBM "railroads," but something much more profound: a personal dynamic *medium*. With a vehicle one could wait until high school and give "drivers ed," but if it was a medium, it had to extend into the world of childhood.

Now the collision of the FLEX machine, the flat-screen display, GRAIL, Barton's "communications" talk, McLuhan, and Papert's work with children all came together to form an image of what a personal computer really should be. I remembered Aldus Manutius who 40 years after the printing press put the book into its modern dimensions by making it fit into saddlebags. It had to be no larger than a notebook, and needed an interface as friendly as JOSS's, GRAIL's, and LOGO's, but with the reach of Simula and FLEX. A clear romantic vision has a marvelous ability to focus thought and will. Now it was easy to know what to do next. I built a cardboard model of it to see what it would look and feel like, and poured in lead pellets to see how light it would have to be (less than two pounds). I put a keyboard on it as well as a stylus because, even if handprinting and writing were recognized perfectly (and there was no reason to expect that it would be), there still needed to be a balance between the lowspeed tactile degrees of freedom offered by the stylus and the more limited but faster keyboard. Because ARPA was starting to experiment with packet radio, I expected that the Dynabook when it arrived a decade or so hence, would have a wireless networking system.

Early the next year (1969) there was a conference on Extensible Languages which almost every famous name in the field attended. The debate was great and weighty—it was a religious war of unimplemented, poorly thought out ideas. As Alan Perlis, one of the great men in Computer Science, put it with characteristic wit:

> It has been such a long time since I have seen so many familiar faces shouting among so many familiar ideas. Discovery of something new in programming languages, like any discovery, has somewhat the same sequence of emotions as falling in love. A sharp elation followed by euphoria, a feeling of uniqueness, and ultimately the wandering eye (the urge to generalize) [ACM 1969].

But it was all talk—no one had *done* anything yet. In the midst of all this, Ned Irons got up and presented IMP, a system that had already been working for several years and was more elegant than most of the nonworking proposals. The basic idea of IMP was that you could use any phrase in the grammar as a procedure heading and write a semantic definition in terms of the language as extended so far [Irons 1970].

I had already made the first version of the FLEX machine syntax-driven, but the meaning of a phrase was defined in the more usual way as the kind of code that was emitted. This separated the compiler-extensor part of the system from the end-user. In Irons' approach, every procedure in the system defined its own syntax in a natural and useful manner. I incorporated these ideas into the second

version of the FLEX machine and started to experiment with the idea of a direct interpreter rather than a syntax-directed compiler. Somewhere in all of this, I realized that the bridge to an object-based system could be in terms of each object as a syntax-directed interpreter of messages sent to it. In one fell swoop this would unify object-oriented semantics with the ideal of a completely extensible language. The mental image was one of separate computers sending requests to other computers that had to be accepted and understood by the receivers before anything could happen. In today's terms, every object would be a server offering services whose deployment and discretion depended entirely on the server's notion of relationship with the servee. As Liebniz said: "To get everything out of nothing, you only need to find one principle." This was not well thought out enough to do the FLEX machine any good, but formed a good point of departure for my thesis [Kay 1969], which as Ivan Sutherland liked to say, was "anything you can get three people to sign."

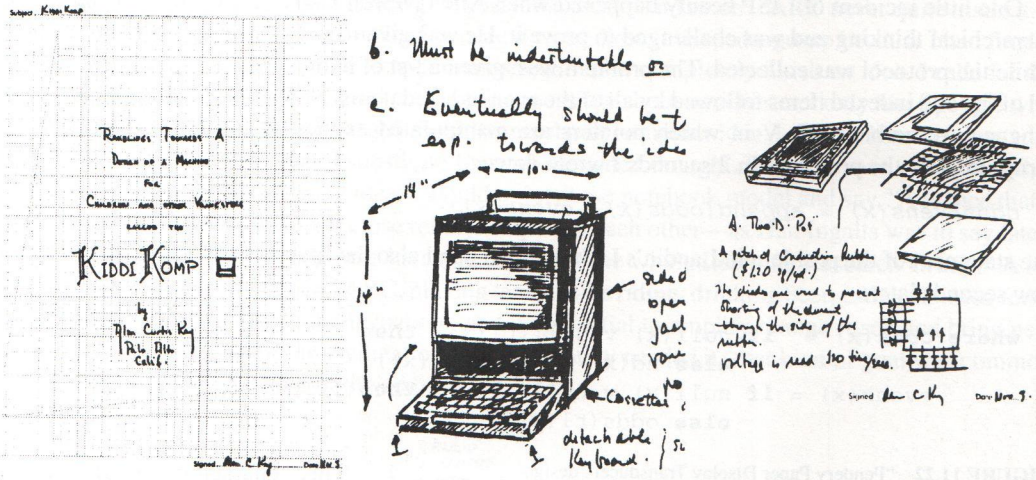
After three people signed it (Ivan was one of them), I went to the Stanford AI project and spent much more time thinking about notebook KiddyKomputers than AI. But there were two AI designs that were very intriguing. The first was Carl Hewitt's PLANNER, a programmable logic system that formed the deductive basis of Winograd's SHRDLU [Hewitt 1969]. I designed several languages based on a combination of the pattern matching schemes of FLEX and PLANNER [Kay 1970]. The second design was Pat Winston's concept formation system, a scheme for building semantic networks and comparing them to form analogies and learning processes [Winston 70]. It was kind of "object-oriented." One of its many good ideas was that the arcs of each net which served as attributes in AOV triples should themselves be modeled as nets. Thus, for example, a first order arc called LEFT-OF could be asked a higher order question such as "What is your converse?" and its net could answer: RIGHT-OF. This point of view later formed the basis for Minsky's frame systems [Minsky 1975]. A few years later I wished I had paid more attention to this idea.

That fall, I heard a wonderful talk by Butler Lampson about CAL-TSS, a capability-based operating system that seemed very "object-oriented" [Lampson 1969]. Unforgeable pointers (àla B5000) were extended by bit-masks that restricted access to the object's internal operations. This confirmed my "objects as server" metaphor. There was also a very nice approach to exception handling that reminded me of the way failure was often handled in pattern matching systems. The only problem—which the CAL designers did not see as a problem at all—was that only certain (usually large and slow) things were "objects." Fast things and small things, and the like, were not. This needed to be fixed.

The biggest hit for me while at SAIL in late '69 was to *really understand* LISP. Of course, every student knew about *car*, *cdr*, and *cons*, but Utah was impoverished in that no one there used LISP and hence, no one had penetrated the mysteries of *eval* and *apply*. I could hardly believe how beautiful and wonderful the *idea* of LISP was [McCarthy 1960]. I say it this way because LISP had not only been around enough to get some honest barnacles, but worse, there were deep flaws in its logical foundations. By this, I mean that the pure language was supposed to be based on functions, but its most important components—such as lambda expressions, quotes, and conds—were not functions at all, and instead were called special forms. Landin and others had been able to get quotes and conds in terms of lambda by tricks that were variously clever and useful, but the flaw remained in the jewel. In the practical language things were better. There were not just EXPRS (which evaluated their arguments), but FEXPRS (which did not). My next question was, why on earth call it a functional language? Why not just base everything on FEXPRS and force evaluation on the receiving side when needed? I could never get a good answer, but the question was very helpful when it came time to invent Smalltalk, because this started a line of thought that said "take the hardest and most profound thing you need to do, make it great, and then build every easier thing out of it." That was the promise of LISP and the lure of lambda—needed was a better "hardest and most profound" thing. Objects should be it.

11.3 1970–1972—XEROX PARC: THE KIDDIKOMP, MINICOM, AND SMALLTALK-71

In July 1970, Xerox, at the urging of its chief scientist, Jack Goldman, decided to set up a long-range research center in Palo Alto, California. In September, George Pake, the former chancellor at Washington University where Wes Clark's ARPA project was sited, hired Bob Taylor (who had left the ARPA office and was taking a sabbatical year at Utah) to start a "Computer Science Laboratory." Bob visited Palo Alto and we stayed up all night talking about it. The Mansfield Amendment was threatening to blindly muzzle the most enlightened ARPA funding in favor of direct military research, and this new opportunity looked like a promising alternative. But work for a company? He wanted me to consult and I asked for a direction. He said: follow your instincts. I immediately started working up a new version of the KiddiKomp that could be made in enough quantity to do experiments leading to the user interface design for the eventual notebook. Bob Barton liked to say that "good ideas don't often scale." He was certainly right when applied to the FLEX machine. The B5000 just did not directly scale down into a tiny machine. Only the byte-codes did, and even these needed modification. I decided to take another look at Wes Clark's LINC, and was ready to appreciate it much more this time [Clark 1965].



I still liked pattern-directed approaches and OOP so I came up with a language design called "Simulation LOGO" or SLOGO for short (I had a feeling the first versions might run nice and slow). This was to be built into a SONY "tummy trinitron" and would use a coarse bit-map display and the FLEX machine rubber tablet as a pointing device.

Another beautiful system that I had come across was Peter Deutsch's PDP-1 LISP (implemented when he was only 15) [Deutsch 1966]. It used only 2K (18-bit words) of code and could run quite well in a 4K machine (it was its own operating system and interface). It seemed that even more could be done if the system were byte-coded, run by an architecture that was hospitable to dynamic systems, and stuck into the ever larger ROMs that were becoming available. One of the basic insights I had gotten from Seymour was that you did not have to do a lot to make a computer an "object for thought" for children, but what you did had to be done well and be able to apply deeply.

Right after New Year's 1971, Bob Taylor scored an enormous coup by attracting most of the struggling Berkeley Computer Corp. to PARC. This group included Butler Lampson, Chuck Thacker, Peter Deutsch, Jim Mitchell, Dick Shoup, Willie Sue Haugeland, and Ed Fiala. Jim Mitchell urged

the group to hire Ed McCreight from CMU and he arrived soon after. Gary Starkweather was there already, having been thrown out of the Xerox Rochester Labs for wanting to build a laser printer (which was against the local religion). Not long after, many of Doug Englebart's people joined up—part of the reason was that they want to reimplement NLS as a distributed network system, and Doug wanted to stay with time-sharing. The group included Bill English (the co-inventor of the mouse), Jeff Rulifson, and Bill Paxton.

Almost immediately we got into trouble with Xerox when the group decided that the new lab needed a PDP-10 for continuity with the ARPA community. Xerox (which had bought SDS essentially sight unseen a few years before) was horrified at the idea of their main competitor's computer being used in the lab. They balked. The newly formed PARC group had a meeting in which it was decided that it would take about three years to do a good operating system for the XDS SIGMA-7 but that we could build "our own PDP-10" in a year. My reaction was "Holy Cow!" In fact, they pulled it off with considerable panache. MAXC was actually a microcoded emulation of the PDP-10 that used for the first time the new integrated chip memories (1K bits!) instead of core memory. Having practical in-house experience with both of these new technologies was critical for the more radical systems to come.

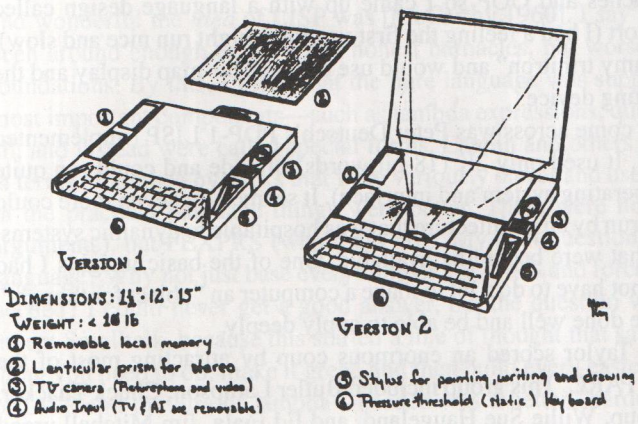
One little incident of LISP beauty happened when Allen Newell visited PARC with his theory of hierarchical thinking and was challenged to prove it. He was given a programming problem to solve while the protocol was collected. The problem was: given a list of items, produce a list consisting of all of the odd indexed items followed by all of the even indexed items. Newell's internal programming language resembled IPL-V in which pointers are manipulated explicitly, and he got into quite a struggle to do the program. In 2 seconds I wrote down:

```
oddsEvens(x) = append(odds(x), evens(x))
```

the statement of the problem in Landin's LISP syntax—and also the first part of the solution. Then a few seconds later:

```
where odds(x) =  if null(x) v null(tl(x)) then x
                 else hd(x) & odds(ttl(x))
      evens(x) = if null(x) v null(tl(x)) then nil
                 else odds(tl(x))
```

FIGURE 11.22 "Pendery Paper Display Transducer" design



This characteristic of writing down many solutions in declarative form and have them also be the programs is part of the appeal and beauty of this kind of language. Watching a famous guy much smarter than I struggle for more than 30 minutes to not quite solve the problem his way (there was a bug) made quite an impression. It brought home to me once again that "point of view is worth 80 IQ points." I wasn't smarter but I had a much better internal thinking tool to amplify my abilities. This incident

and others like it made it paramount that any tool for children should have great thinking patterns *and* deep beauty "built-in."

Right around this time we were involved in another conflict with Xerox management, in particular with Don Pendery, the head "planner." He really did not understand what we were talking about and instead was interested in "trends" and "what was the future going to be like" and how could Xerox "defend against it." I got so upset I said to him, "Look. *The best way to predict the future is to invent it.* Don't worry about what all those other people might do; this is the century in which almost any clear vision can be made!" He remained unconvinced, and that led to the famous "Pendery Papers for PARC Planning Purposes," a collection of essays on various aspects of the future. Mine proposed a version of the notebook as a "Display Transducer," and Jim Mitchell's was entitled "NLS on a Minicomputer."

Bill English took me under his wing and helped me start my group as I had always been a lone wolf and had no idea how to do it. One of his suggestions was that I should make a budget. I am afraid that I really did ask Bill, "What's a budget?" I remembered at Utah, in pre-Mansfield Amendment days, Dave Evans saying to me as he went off on a trip to ARPA, "We're almost out of money. Got to go get some more." That seemed about right to me. They give you some money. You spend it to find out what to do next. You run out. They give you some more. And so on. PARC never quite made it to that idyllic standard, but for the first half-decade it came close. I needed a group because I had finally realized that I did not have all the temperaments required to completely finish an idea. I called it the Learning Research Group (LRG) to be as vague as possible about our charter. I only hired people who got stars in their eyes when they heard about the notebook computer idea. I did not like meetings: did not believe brainstorming could substitute for cool sustained thought. When anyone asked me what to do, and I did not have a strong idea, I would point at the notebook model and say, "Advance that." LRG members developed a very close relationship with each other—as Dan Ingalls was to say later: "...the rest has enfolded through the love and energy of the whole Learning Research Group." A lot of daytime was spent outside of PARC, playing tennis, bike riding, drinking beer, eating Chinese food, and constantly talking about the Dynabook and its potential to amplify human reach and bring new ways of thinking to a faltering civilization that desperately needed it (that kind of goal was common in California in the aftermath of the sixties).

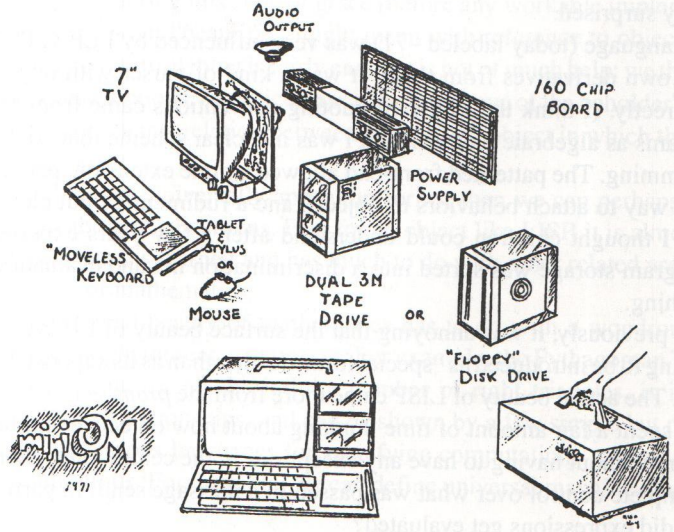


FIGURE 11.23 Smalltalk-71 programs

```
to T 'and' :y do 'y'
to F 'and' :y do F

to 'factorial' 0 is 1
to 'factorial' :n do 'n*factorial n-1'

to 'fact' :n do 'to 'fact' n do factorial n. ^ fact n'

to :e 'is-member-of [] do F
to :e 'is-member-of :group
        do 'if e = first of group then T
              else e is-member-of rest of group'

to 'cons' :x :y is self
to 'hd' ('cons' :a :b) do 'a'
to 'hd' ('cons' :a :b) '<-' :c do 'a <- c'
to 'tl' ('cons' :a :b) do 'b'
to 'tl' ('cons' :a :b) '<-' :c do 'b <- c'

to :robot 'pickup' :block
       do 'robot clear-top-of block.
       robot hand move-to block.
       robot hand lift block 50.
       to 'height-of block do 50'
```

In the summer of '71 I refined the KiddiKomp idea into a tighter design called miniCOM. It used a bit-slice approach like the NOVA 1200, had a bit-map display, a pointing device, a choice of "secondary" (really tertiary) storages, and a language I now called "Smalltalk"—as in "programming should be a matter of ..." and "children should program in ...".The name was also a reaction against the "IndoEuropean god theory" where systems were named Zeus, Odin, and Thor, and hardly did anything. I figured that "Smalltalk" was so innocuous a label that if it ever did anything nice people would be pleasantly surprised.

This Smalltalk language (today labeled –71) was very influenced by FLEX, PLANNER, LOGO, META II, and my own derivatives from them. It was a kind of parser with object-attachment that executed tokens directly. (I think the awkward quoting conventions came from META.) I was less interested in programs as algebraic patterns than I was in a clear scheme that could handle a variety of styles of programming. The patterned front-end allowed simple extension, patterns as "data" to be retrieved, a simple way to attach behaviors to objects, and a rudimentary but clear expression of its *eval* in terms that I thought children could understand after a few years experience with simpler programming. Program storage was sorted into a discrimination net and evaluation was straightforward pattern-matching.

As I mentioned previously, it was annoying that the surface beauty of LISP was marred by some of its key parts having to be introduced as "special forms" rather than as its supposed universal building block of functions. The actual beauty of LISP came more from the *promise* of its metastructures than its actual model. I spent a fair amount of time thinking about how objects could be characterized as universal computers without having to have any exceptions in the central metaphor. What seemed to be needed was complete control over what was passed in a message send; in particular, *when* and in *what environment* did expressions get evaluated?

An elegant approach was suggested in a CMU thesis of Dave Fisher [Fisher 1970] on the synthesis of control structures. ALGOL60 required a separate link for dynamic subroutine linking and for access to static global state. Fisher showed how a generalization of these links could be used to simulate a wide variety of control environments. One of the ways to solve the "funarg problem" of LISP is to associate the proper global state link with expressions and functions that are to be evaluated later so that the free variables referenced are the ones that were actually implied by the static form of the language. The notion of "lazy evaluation" is anticipated here as well.

Nowadays this approach would be called *reflective design*. Putting it together with the FLEX models suggested that all that should be required for "doing LISP right" or "doing OOP right" would be to handle the mechanics of invocations between modules without having to worry about the details of the modules themselves. The difference between LISP and OOP (or any other system) would then be what the modules could contain. A universal module (object) reference—à la B5000 and LISP—and a message holding structure—which could be virtual if the senders and receivers were simpatico—that could be used by all would do the job.

If all of the fields of a messenger structure were enumerated according to this view, we would have:

```
GLOBAL:                 the environment of the parameter values
SENDER:                 the sender of the message
RECEIVER:               the receiver of the message
REPLY-STYLE:            wait, fork, ...?
STATUS:                 progress of the message
REPLY:                  eventual result (if any)
OPERATION SELECTOR:     relative to the receiver
# OF PARAMETERS:
P1
...
PN
```

This is a generalization of a stack frame, such as is used by the B5000, and very similar to what a good intermodule scheme would require in an operating system such as CAL-TSS—a lot of state for every transaction, but useful to think about.

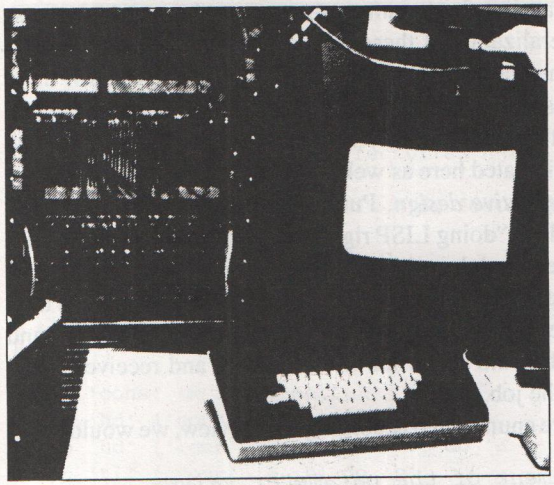
Much of the pondering during this state of grace (before any workable implementation) had to do with trying to understand what "beautiful" might mean with reference to object-oriented design. A subjective definition of a beautiful thing is fairly easy but is not of much help: we think a thing beautiful because it evokes certain emotions. The cliche has it lie "in the eye of the beholder" so that it is difficult to think of beauty as other than a relation between subject and object in which the predispositions of the subject are all-important.

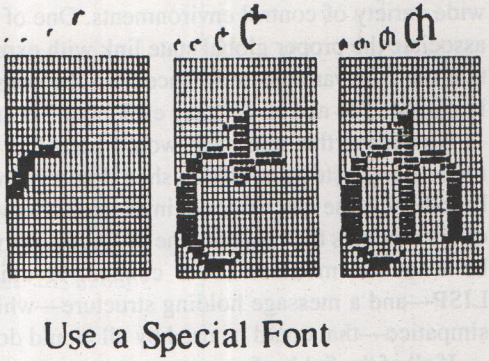
If there are such things as universally appealing forms, then we can perhaps look to our shared biological heritage for the predispositions. But, for an object like LISP, it is almost certain that most of the basis of our judgment is learned and has much to do with other related areas that we think are beautiful, such as much of mathematics.

One part of the perceived beauty of mathematics has to do with a wondrous synergy between parsimony, generality, enlightenment, and finesse. For example, the Pythagorean Theorem is expressible in a single line, is true for all the infinite number of right triangles, is incredibly useful in understanding many other relationships, and can be shown by a few simple but profound steps.

When we turn to the various languages for specifying computations we find many to be general and a few to be parsimonious. For example, we can define universal machine languages in just a few

FIGURE 11.24 The "Old Character Generator"—early 1972





Use a Special Font

instructions that can specify anything that can be computed. But most of these we would not call beautiful, in part because the amount and kind of code that has to be written to do anything interesting is so contrived and turgid. A simple and small system that can do interesting things also needs a "high slope"—that is, a good match between the degree of interestingness and the level of complexity needed to express it.

A fertilized egg that can transform itself into the myriad of specializations needed to make a complex organism has parsimony, generality, enlightenment, and finesse—in short, beauty, and a beauty much more in line with my own esthetics. I mean by this that Nature is wonderful at *both* elegance and practicality—the cell membrane is partly there to allow useful evolutionary kludges to do their necessary work and still be able to act as components by presenting a uniform interface to the world.

One of my continual worries at this time was about the size of the bit-map display. Even if a mixed mode was used (between fine-grained generated characters and coarse-grained general bit-map for graphics) it would be hard to get enough information on the screen. It occurred to me (in a shower, my favorite place to think) that FLEXtype windows on a bit-map display could be made to appear as overlapping documents on a desktop. When an overlapped one was refreshed it would appear to come to the top of the stack. At the time, this did not appear as the wonderful solution to the problem but it did have the effect of magnifying the effective area of the display enormously, so I decided to go with it.

To investigate the use of video as a display medium, Bill English and Butler Lampson specified an experimental character generator (built by Roger Bates) for the POLOS (PARC OnLine Office System) terminals. Gary Starkweather had just gotten the first laser printer to work and we ran a coax over to his lab to feed him some text to print. The "SLOT machine" (Scanning Laser Output Terminal) was incredible. The only Xerox copier Gary could get to work on went at 1 page per second and could not be slowed down. So Gary just made the laser run at that rate with a resolution of 500 pixels to the inch!

The character generator's font memory turned out to be large enough to simulate a bit-map display if one displayed a fixed "strike" and wrote into the font memory. Ben Laws built a beautiful font editor and he and I spent several months learning about the peculiarities of the human visual system (it is decidedly nonlinear). I was very interested in high-quality text and graphical presentations because I

FIGURE 11.25 The first painting system—Summer '72

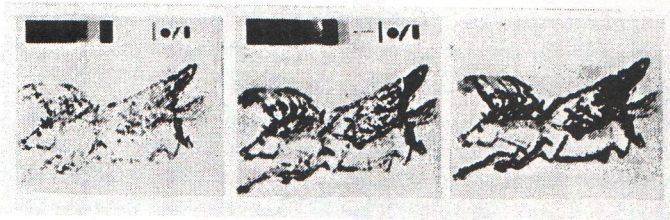


FIGURE 11.26 Portrait of the Xerox "RISK" executive

thought it would be easier to get the Dynabook into schools as a "trojan horse" by simply replacing school books rather than to try to explain to teachers and school boards what was really great about personal computing.

Things were generally going well all over the lab until May of 1972 when I tried to get resources to build a few miniCOMs. A relatively new executive ("X") did not want to give them to me. I wrote a memo explaining why the system was a good idea (see Appendix I), and then had a meeting to discuss it. "X" shot it down completely, saying among other things that we had used too many green stamps getting Xerox to fund the time-shared MAXC and this use of resources for personal machines would confuse them. I was shocked. I crawled away back to the experimental character generator and made a plan to get four more made and hooked to NOVAs for the initial kid experiments.

I got Steve Purcell, a summer student from Stanford, to build my design for bit-map painting so the kids could sketch as well as display computer graphics. John Shoch built a line-drawing and gesture recognition system (based on Ledeen's [Newman and Sproull 1972]) that was integrated with the painting. Bill Duvall of POLOS built a miniNLS that was quite remarkable in its speed and power. The first overlapping windows started to appear. Bob Shur (with Steve Purcell's help) built a 2 1/2 D animation system. Along with Ben Laws' font editor, we could give quite a smashing demo of what we intended to build for real over the next few years. I remember giving one of these to a Xerox executive, including doing a portrait of him in the new painting system, and wound it up with a flourish, declaring: "And what's really great about this is that it only has a 20% chance of success. We're taking a risk just like you asked us to!" He looked me straight in the eye and said, "Boy, that's great, but just make sure it works." This was a typical executive notion about risk. He wanted us to be in the "20%" one hundred percent of the time.

That summer while licking my wounds and getting the demo simulations built and going, Butler Lampson, Peter Deutsch, and I worked out a general scheme for emulated HLL machine languages. I liked the B5000 scheme, but Butler did not want to have to decode bytes, and pointed out that in as much as an 8-bit byte had 256 total possibilities, what we should do is map different meanings onto different parts of the "instruction space." This would give us a "poor man's Huffman code" that would be both flexible and simple. All subseqent emulators at PARC used this general scheme.

I also took another pass at the language for the kids. Jeff Rulifson was a big fan of Piaget (and semiotics) and we had many discussions about the "stages" and what iconic thinking might be about. After reading Piaget and especially Jerome Bruner, I was worried that the directly symbolic approach taken by FLEX, LOGO (and the current Smalltalk) would be difficult for the kids to process because evidence existed that the symbolic stage (or mentality) was just starting to switch on. In fact, all of the educators that I admired (including Montessori, Holt, and Suzuki) all seemed to call for a more

FIGURE 11.27 Children with Dynabooks from "A Personal Computer for Children of All Ages" [Kay 1972]

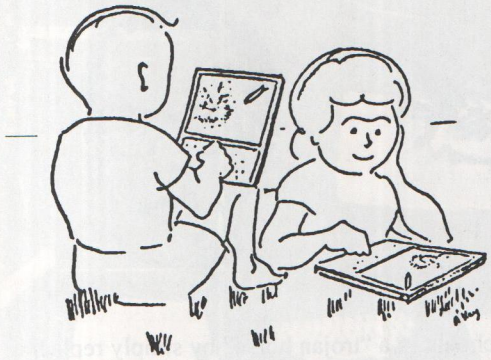
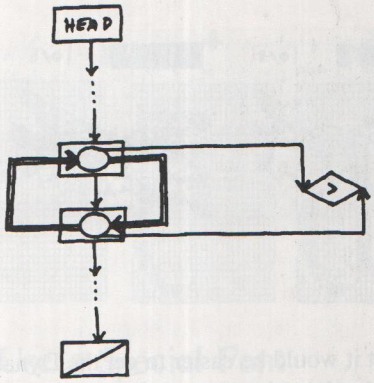


FIGURE 11.28 Iconic Bubble Sort from 1972 LRG Plan [Kay 1972b]



figurative, more iconic approach. Rudolph Arnheim [Arnheim 1969] had written a classic book about visual thinking, and so had the eminent art critic Gombrich [Gombrich 1960]. It really seemed that something better needed to be done here. GRAIL wasn't it, because its use of imagery was to portray and edit flowcharts, which seemed like a great step backwards. But Rovner's AMBIT-G held considerably more promise [Rovner 1968]. It was kind of a visual SNOBOL [Farber 1964] and the pattern matching ideas looked like they would work for the more plannerlike scheme I was using.

Bill English was still encouraging me to do more reasonable-appearing things to get higher credibility, such as making budgets and writing plans and milestone notes, so I wrote a plan that proposed over the next few years that we would build a real system on the character generators *cum* NOVAs that would involve OOP, windows, painting, music, animation, and "iconic programming." The latter was deemed to be hard and would be handled by the usual method for hard problems, namely, give them to grad students.

"Simple things should be simple, complex things should be possible."

11.4 1972–76—THE FIRST REAL SMALLTALK (–72), ITS BIRTH, APPLICATIONS, AND IMPROVEMENTS

In Sepember, within a few weeks of each other, two bets happened that changed most of my plans. First, Butler and Chuck came over and asked: "Do you have any money?" I said, "Yes, about $230K for NOVAs and CGs. Why?" They said, "How would you like us to build your little machine for you?" I said, "I'd like it fine. What is it?" Butler said: "I want a '$500 PDP-10,' Chuck wants a '10 times faster nova,' and you want a 'kiddicomp.' What do you need on it?" I told them most of the results we had gotten from the fonts, painting, resolution, animation, and music studies. I asked where this had come from all of a sudden and Butler told me that they wanted to do it anyway, that Executive "X" was away for a few months on a "task force" so maybe they could "sneak it in," and that Chuck had a bet with Bill Vitic that he could do a whole machine in just three months. "Oh," I said.

The second bet had even more surprising results. I had expected that the new Smalltalk would be an iconic language and would take at least two years to invent, but fate intervened. One day, in a typical PARC hallway bull session, Ted Kaehler, Dan Ingalls, and I were standing around talking about programming languages. The subject of power came up and the two of them wondered how large a language one would have to make to get great power. With as much panache as I could muster, I

asserted that you could define the "most powerful language in the world" in "a page of code." They said. "Put up or shut up."

Ted went off to CMU but Dan was still around egging me on. For the next two weeks I got to PARC every morning at four o'clock and worked on the problem until eight, when Dan, joined by Henry Fuchs, John Shoch, and Steve Purcell showed up to kibbitz the morning's work.

I had originally made the boast because McCarthy's self-describing LISP interpreter was written in itself. It was about "a page," and as far as power goes, LISP was the whole nine-yards for functional languages. I was quite sure I could do the same for object-oriented languages *plus* be able to do a reasonable syntax for the code à la some of the FLEX machine techniques.

It turned out to be more difficult than I had first thought for three reasons. First, I wanted the program to be more like McCarthy's second nonrecursive interpreter—the one implemented as a loop that tried to resemble the original 709 implementation of Steve Russell as much as possible. It was more "real." Second, the intertwining of the "parsing" with message receipt—the evaluation of parameters that was handled separately in LISP—required that my object-oriented interpreter re-enter itself "sooner" (in fact, much sooner) than LISP required. And, finally, I was still not clear how *send* and *receive* should work with each other.

The first few versions had flaws that were soundly criticized by the group. But by morning 8 or so, a version appeared that seemed to work (see Appendix II for a sketch of how the interpreter was designed). The major differences from the official Smalltalk-72 of a little bit later were that in the first version symbols were byte-coded and the receiving of return-values from a send was symmetric—that is, receipt could be like parameter binding—this was particularly useful for the return of multiple values. For various reasons, this was abandoned in favor of a more expression-oriented functional return style.

Of course, I had gone to considerable pains to avoid doing any "real work" for the bet, but I felt I had proved my point. This had been an interesting holiday from our official "iconic programming" pursuits, and I thought that would be the end of it. Much to my surprise, only a few days later, Dan Ingalls showed me the scheme *working* on the NOVA. He had coded it up (in BASIC!), added a lot of details, such as a token scanner, a list maker, and the like, and there it was—running. As he like to say: "You just do it and it's done."

It evaluated 3+4 *v e r y s l o w l y* (it was "glacial," as Butler liked to say) but the answer always came out 7. Well, there was nothing to do but keep going. Dan loved to bootstrap on a system that "always ran," and over the next ten years he made at least 80 major releases of various flavors of Smalltalk.

In November, I presented these ideas and a demonstration of the interpretation scheme to the MIT AI lab. This eventually led to Carl Hewitt's more formal "Actor" approach [Hewitt 1973]. In the first Actor paper the resemblence to Smalltalk is at its closest. The paths later diverged, partly because we were much more interested in making things than theorizing, and partly because we had something no one else had: Chuck Thacker's Interim Dynabook (later known as the "ALTO").

Just before Chuck started work on the machine I gave a paper to the National Council of Teachers of English [Kay 1972c] on the Dynabook and its potential as a learning and thinking amplifier—the paper was an extensive rotogravure of "20 things to do with a Dynabook" [Kay 1972c]. By the time I got back from Minnesota, Stewart Brand's *Rolling Stone* article about PARC [Brand 1972] and the surounding hacker community had hit the stands. To our enormous surprise it caused a major furor at Xerox headquarters in Stamford, Connecticut. Though it was a wonderful article that really caught the spirit of the whole culture, Xerox went berserk, forced us to wear badges (over the years many were printed on t-shirts), and severely restricted the kinds of publications that could be made. This

FIGURE 11.29 BILBO, the first "Interim Dynabook," and "Cookie Monster," the first graphics it displayed. April 1973

1. Everything is an *object*

2. Objects communicate by sending and receiving *messages* (in terms of objects)

3. Objects have their *own memory* (in terms of objects)

4. Every object is an *instance* of a *class* (which must be an object)

5. The class holds the shared *behavior* for its instances (in the form of objects in a program list)

6. To eval a program list, control is passed to the first object and the remainder is treated as its message

was particularly disastrous for LRG, because we were the "lunatic fringe" (so-called by the other computer scientists), were planning to go out to the schools, and needed to share our ideas (and programs) with our colleagues such as Seymour Papert and Don Norman.

Executive "X" apparently heard some harsh words at Stamford about us, because when he returned around Christmas and found out about the interim Dynabook, he got even more angry and tried to kill it. Butler wound up writing a masterful defense of the machine to hold him off, and he went back to his "task force."

Chuck had started his "bet" on November 22, 1972. He and two technicians did all of the machine except for the disk interface which was done by Ed McCreight. It had a ~500,000 pixel (606x808) bitmap display, its microcode instruction rate was about 6MIPs, it had a grand total of 96kb, and the entire machine (exclusive of the memory) was rendered in 160 MSI chips distributed on two cards. It was beautiful [Thacker 1972, 1986]. One of the wonderful features of the machine was "zero-overhead" tasking. It had 16 program counters, one for each task. Condition flags were tied to interesting events (such as "horizontal retrace pulse," and "disk sector pulse," and so on). Lookaside logic scanned the flags while the current instruction was executing and picked the highest priority program counter to fetch from next. The machine never had to wait, and the result was that most hardware functions (particularly those that involved I/O (such as feeding the display and handling the disk) could be replaced by microcode. Even the refresh of the MOS dynamic RAM was done by a task. In other words, this was a coroutine architecture. Chuck claimed that he got the idea from a lecture I had given on coroutines a few months before, but I remembered that Wes Clark's TX-2 (the Sketchpad machine) had used the idea first, and I probably mentioned that in the talk.

In early April, just a little over three months from the start, the first Interim Dynabook,

known as "Bilbo," greeted the world and we had the first bit-map picture on the screen within minutes: the Muppets' Cookie Monster that I had sketched on our painting system.

Soon Dan had bootstrapped Smalltalk across, and for many months it was the sole software system to run on the Interim Dynabook. Appendix III has an "acknowledgements" document I wrote from this time that is interesting in its allocation of credits and the various priorities associated with them. My $230K was enough to get 15 of the original projected 30 machines (over the years some 2000 Interim Dynabooks were actually built). True to Schopenhauer's observation, Executive "X" now decided that the Interim Dynabook was a good idea and he wanted *all but two* for his lab (I was in the other lab). I had to go to considerable lengths to get our machines back, but finally succeeded.

By this time most of Smalltalk's schemes had been sorted out into six main ideas that were in accord with the initial premises in designing the interpreter. The first three principles are what objects "are about"—how they are seen and used from "the outside." These did not require any modification over the years. The last three—objects from the inside—were tinkered with in every version of Smalltalk (and in subsequent OOP designs). In this scheme, (1 and 4) imply that classes are objects and that they must be instances of themself. (6) implies a LISPlike universal syntax, but with the receiving object as the first item followed by the message. Thus c_i <– *de* (with subscripting rendered as "°" and multiplication as "*") means:

receiver | *message*
c | ° *i* <– *d***e*

The *c* is bound to the receiving object, and *all* of ° *i* <– *d***e* is the message to it. The message is made up of a literal token "°", an expression to be evaluated in the sender's context (in this case *i*), another literal token <–, followed by an expression to be evaluated in the sender's context (*d***e*). Because "LISP" pairs are made from two element objects they can be indexed more simply: *c hd*, *c tl*, and *c hd* <– *foo*, and so on.

"Simple" expressions like a+b and 3+4 seemed more troublesome at first. Did it really make sense to think of them as:

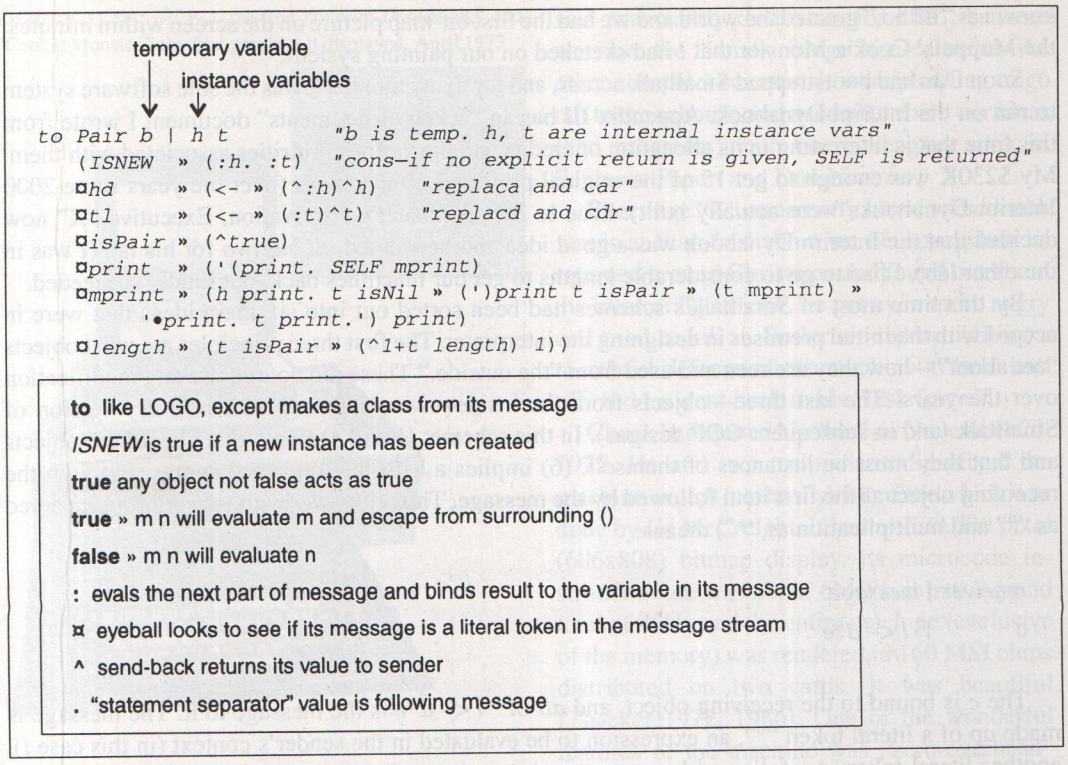
receiver | *message*
a | + *b*
3 | + *4*

It seemed silly if only integers were considered, but there are many other metaphoric readings of "+", such as:

"*kitty*" | + "*kat*" => "*kittykat*"

$$\begin{bmatrix}3 & 4 & 5\\6 & 7 & 8\end{bmatrix} \mid + 4 \Rightarrow \begin{bmatrix}7 & 8 & 9\\10 & 11 & 12\end{bmatrix}$$

This led to a style of finding *generic behaviors* for message symbols. "Polymorphism" is the official term (I believe derived from Strachey), but it is not really apt as its original meaning applied only to functions that could take more than one type of argument. An example class of objects in Smalltalk-72, such as a model of cons pairs, would look like:



Because control is passed to the class before any of the rest of the message is considered—the class can decide *not* to receive at its discretion—complete protection is retained. Smalltalk-72 objects are "shiny" and impervious to attack. Part of the environment is the binding of the SENDER in the "messenger object" (a generalized activation record) that allows the receiver to determine differential privileges (see Appendix II for more details). This looked ahead to the eventual use of Smalltalk as a network OS (see [Goldstein and Bobrow 1980]), and I don't recall it being used very much in Smalltalk-72.

One of the styles retained from Smalltalk-71 was the comingling of function and class ideas. In other words, Smalltalk-72 classes looked like and could be used as functions, but it was easy to produce an instance (a kind of closure) by using the object *ISNEW*. Thus factorial could be written "extensionally" as:

```
to fact n (^if :n=0 then 1 else n*fact n-1)
```

or "intensionally", as part of class integer:

```
(... ¤! » (^:n=0 » (1) (n-1)! )
```

Of course, the whole idea of Smalltalk (and OOP in general) is to define everything *intensionally*. And this was the direction of movement as we learned how to program in the new style. I never liked this syntax (too many parentheses and nestings) and wanted something flatter and more grammar-like as in Smalltalk-71. To the right is an example syntax from the notes of a talk I gave around then. We will see something more like this a few years later in Dan's design for Smalltalk-76. I think something similar happened with LISP—that the "reality" of the straightforward and practical syntax **you could program in** prevailed against the flights of fancy that never quite got built.

FIGURE 11.30 Proposed Smalltalk-72 syntax

```
Pair :h :t
   hd <- :h
   hd          » h
   tl <- :t    » t
   isPair      » true
   print       » '(print. SELF mprint.
   mprint      » h print. if t isNil then ') print
                        else if t isPair then t mprint
                        else '•print. t print. ') print.
   length      » 1 + if t isList then t length else 0
```

11.4.1 Development of the Smalltalk-72 System and Applications

The advent of a real Smalltalk on a real machine started off an explosion of parallel paths that are too difficult to intertwine in strict historical order. Let me first present the general development of the Smalltalk-72 system up to the transistion to Smalltalk-76, and then follow that with the several years of work with children who were the primary motivation for the project. The Smalltalk-72 interpreter on the Interim Dynabook was not exactly zippy ("majestic" was Butler's pronouncement), but was easy to change and quite fast enough for many real-time interactive systems to be built in it.

Overlapping windows were the first project tackled (with Diana Merry) after writing the code to read the keyboard and create a string of text. Diana built an early version of a bit field block transfer (bitblt) for displaying variable pitch fonts and generally writing on the display. The first window versions were done as real 2 1/2D draggable objects that were just a little too slow to be useful. We decided to wait until Steve Purcell got his animation system going to do it right, and opted for the style that is still in use today, which is more like "2 1/4D." Windows were perhaps the most redesigned and reimplemented class in Smalltalk because we did not quite have enough compute power to just do the continual viewing to "world coordinates" and refreshing that my former Utah colleagues were starting to experiment with on the flight simulator projects at Evans & Sutherland. This is a simple powerful model but it is difficult to do in real-time even in 2 1/2D. The first practical windows in

FIGURE 11.31 One of the "first build" ALTOs

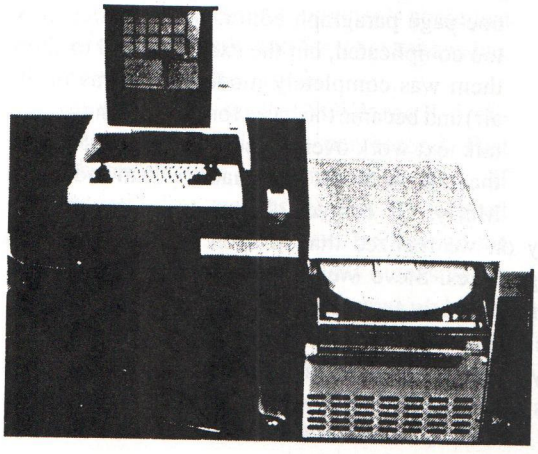


FIGURE 11.32 Early Smalltalk Windows on Interim Dynabook

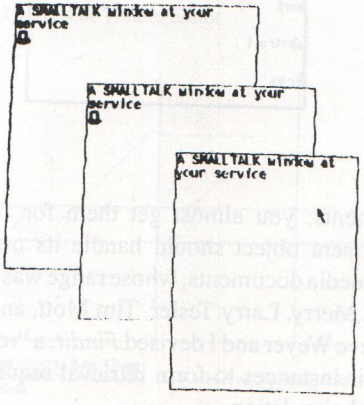


FIGURE 11.33 Turtles

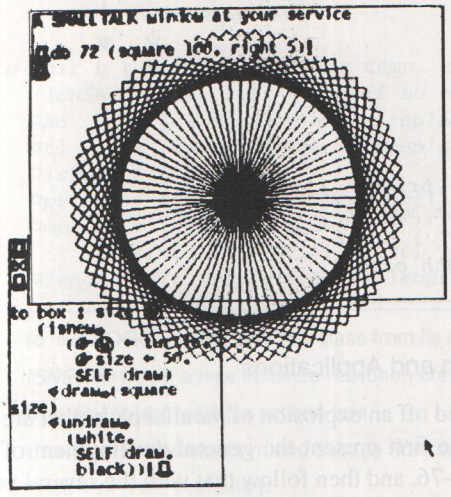
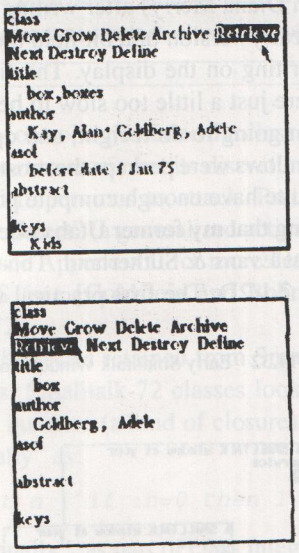


FIGURE 11.33 Findit retrieval by example



Smalltalk used the GRAIL conventions of sensitive corners for moving, resizing, cloning, and closing. Window scheduling used a simple "loopless" control scheme that threaded all the windows together.

One of the next classes to be implemented on the Interim Dynabook (after the basics of numbers, strings, and so on) was an object-oriented version of the LOGO turtle implemented by Ted. This could make many turtle instances that were used both for drawing and as a kind of value for graphics transformations. Dan created a class of "commander" turtles that could control a troop of turtles. Soon the turtles were made so they could be clipped by the windows.

John Shoch built a mouse-driven structured editor for Smalltalk code.

Larry Tesler (then working for POLOS) did not like the modiness and general approach of NLS, and he wanted both to show the former NLSers an alternative and to conduct some user studies (almost unheard of in those days) about editing. This led to his programming *miniMOUSE* in Smalltalk, the first real WYSIWYG galley editor at PARC. It was modeless (almost) and fun to use, not just for us, but for the many people he tested it on. (I ran the camera for the movies we took and remember their delight and enjoyment.) *miniMOUSE* quickly became an alternate editor for Smalltalk code and some of the best demos we ever gave used it.

One of the "small program" projects I tried on an adult class in the Spring of '74 was a one-page paragraph editor. It turned out to be too complicated, but the example I did to show them was completely modeless (it was in the air) and became the basis for much of the Smalltalk text work over the next few years. Most of the improvements were made by Dan and Diana Merry. Of course, objects mean multimedia documents; you almost get them for free. Early on we realized that in such a document, each component object should handle its own editing chores. Steve Weyer built some of the earliest multimedia documents, whose range was greatly and variously expanded over the years by Bob Flegal, Diana Merry, Larry Tesler, Tim Mott, and Trygve Reenskaug.

Steve Weyer and I devised *Findit*, a "retrieval by example" interface that used the analogy of classes to their instances to form retrieval requests. This was used for many years by the PARC library to control circulation.

The sampling synthesis music I had developed on the NOVA could generate three high-quality real-time voices. Bob Shur and Chuck Thacker transfered the scheme to the Interim Dynabook and achieved 12 voices in real-time. The 256 bit generalized input that we had specified for low speed devices (used for the mouse and keyboard) made it easy to connect 154 more to wire up two organ keyboards and a pedal. Effects such as portamento and decay were programmed. Ted Kaehler wrote TWANG, a music capture and editing system, using a tabulature notation that we devised to make music clear to children [Kay 1977a]. One of the things that was hard to do with sampling was the voltage controlled operator (VCO) effects that were popular on the "Well Tempered Synthesizer." A summer later, Steve Saunders, another of our bright summer students, was challenged to find a way to accomplish John Chowning's *very* non-real-time FM synthesis in real-time on the ID. He had to find a completely different way to think of it than "FM," and succeeded brilliantly with eight real-time voices that were integrated into TWANG [Saunders 1977].

Chris Jeffers (who was a musician and educator, not a computer scientist) knocked us out with OPUS, the first real-time score capturing system. Unlike most systems today it did not require metronomic playing but instead took a first pass looking for strong and weak beats (the phrasing) to establish a local model of the likely tempo fluctuations and then used curve fitting and extrapolation to make judgments about just where in the measure, and for what time value, a given note had been struck.

The animations on the NOVA ran 3–5 objects at about 2–3 frames per second. Fast enough for the *phi* phenomenon to work (if double buffering was used), but we wanted "Disney rates" of 10–15 frames per second for 10 or more large objects and many more smaller ones. This task was put into the ingenious hands of Steve Purcell. By the Fall of '73 he could demo 80 ping-pong balls and 10 flying horses running at 10 frames per second in 2 1/2 D. His next task was to make the demo into a general

FIGURE 11.34 Retrieved HyperDocument

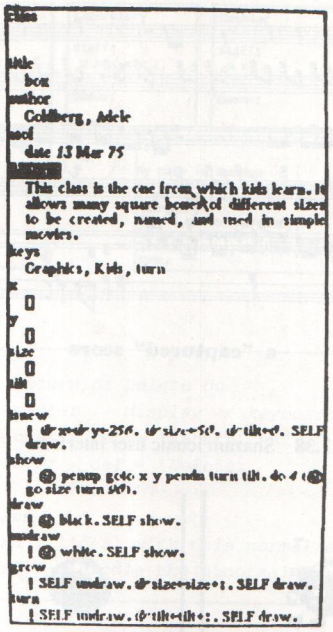


FIGURE 11.35 TWANG Music System

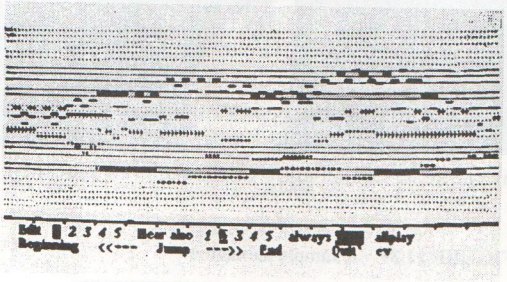


FIGURE 11.36 FM Timbre Editor

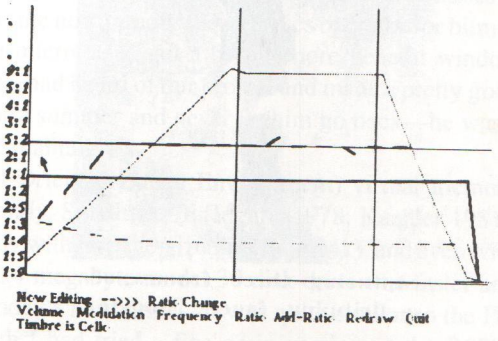
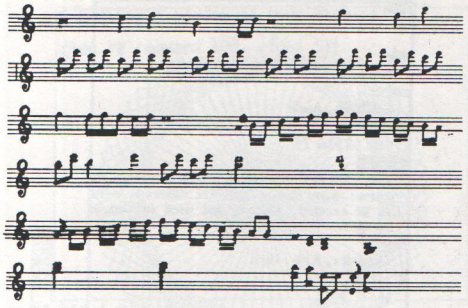


FIGURE 11.37 OPUS Score Capture



a "captured" score

FIGURE 11.38 Shazam iconic user interface

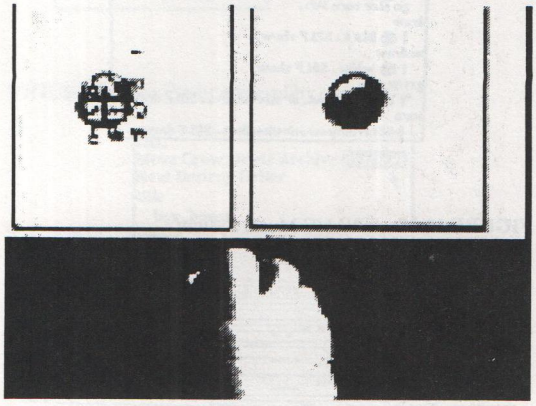
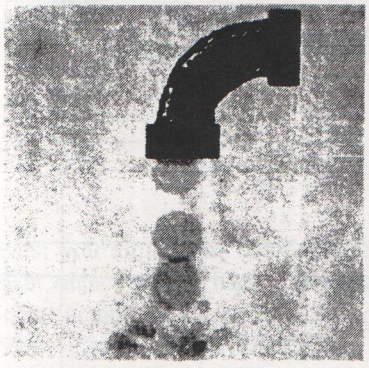


FIGURE 11.39 A sample animation



capturing three frames of a dripping faucet movie

systems facility from which we could construct animation systems. His chaos system started working in May '74, just in time for summer visitors Ron Baecker, Tom Horseley, and professional animator Eric Martin to visit and build Shazam, a marvelously capable and simple animation system based on Ron's GENESYS thesis project on the TX-2 in the late sixties [Baecker 1969].

The main thesis project during this time was Dave Smith's PYGMALION [Smith 1975], an essay into iconic programming (no, we hadn't quite forgotten). One programmed by showing the system how changes should be made, much as one would illustrate on a blackboard with another programmer. This program became the starting place from which many subsequent "programming by example" systems took off.

I should say something about the size of these programs. PYGMALION was the largest program ever written in Smalltalk-72. It was about 20 pages of code—all that would fit in the interim dynabook ALTO—and is given in full in Smith's thesis. All the other applications were smaller. For example, the Shazam animation system was written and revised several times in the summer of 1974, and finally wound up as a 5–6 page application that included its icon-controlled multiwindowed user interface.

Given its roots in simulation languages, Simpula, a simple version of the SIMULA sequencing set approach to schedulingwas easy to write in a few pages. By this time we had decided that coroutines could be rendered more cleanly by scheduling individual methods as separate simulation phases. The generic SIMULA example was a job shop. This could be generalized into many useful forms, such as a hospital with departments of resources serving patients (see to the right). The children did not care for hospitals but saw they could model amusement parks, such as Disneyland, their schools, the stores they and their parents shopped in, and so forth. Later this model formed the basis of the Smalltalk Sim-kit, a high-level end-user programming environment (described ahead).

FIGURE 11.40 PYGMALION iconic programming

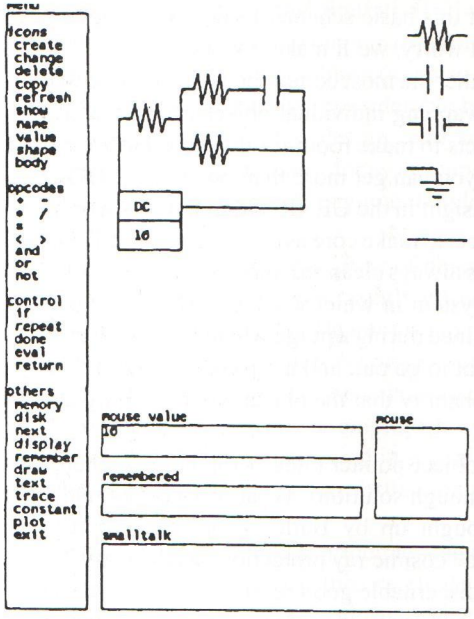
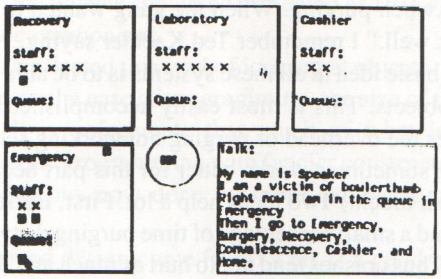


FIGURE 11.41 "Simpula" hospital simulation




```
(until Return or Delete do
  ('character <- display <- keyboard.
  character = ret » (Return)
  character = del » (Delete)
  )
then case
  Return : ('deal with this normal exit')
  Delete : ('handle the abnormal exit'))
```

Many nice "computer sciency" constructs were easy to make in Smalltalk-72. For example, one of the controversies of the day was whether to have **goto**s or not (we did not), and if not, how could certain very useful control structures—such as multiple exits from a loop—be specified? Chuck Zahn at SLAC proposed an *event-driven case* structure in which a set of events could be defined so that when an event is encountered, the loop will be exited and the event will select a statement in a case block [Zahn 1974; Knuth 1974]. Suppose we want to write a simple loop that reads characters from the keyboard and outputs them to a display. We want it to exit normally when the <return> key is struck and with an error if the <delete> key is hit. Appendix IV shows how John Shoch defined this control structure.

11.4.2 The Evolution Of Smalltalk-72

Smalltalk-74 (sometimes known as FastTalk) was a version of Smalltalk-72 incorporating major improvements that included providing a real "messenger" object, message dictionaries for classes (a step toward real class objects), Diana Merry's bitblt (the now famous 2D graphics operator for bitmap graphics) redesigned by Dan and implemented in microcode, and a better, more general window interface. Dave Robson, while a student at UC Irvine, had heard of our project and made a pretty good stab at implementing an OOPL. We invited him for a summer and never let him go back—he was a great help in formulating an official semantics for Smalltalk.

The crowning addition was the OOZE (Object-Oriented Zoned Environment) virtual memory system that served Smalltalk-74, and more importantly, Smalltalk-76 [Ingalls 1978; Kaehler 1981]. The ALTO was not very large (128–256K), especially with its page-sized display (64k), and even with small programs, we soon ran out of storage. The 2.4 megabyte model 30 disk drive was faster and larger than a floppy and slower and smaller than today's hard drives. It was quite similar to the HP direct contact disk of the FLEX machine on which I had tried a fine-grain version of the B5000

segment swapper. It had not worked as well as I wanted, despite a few good ideas as to how to choose objects when purging. When the gang wanted to adapt this basic scheme, I said: "But I never got it to work well." I remember Ted Kaehler saying, "Don't worry, we'll make it work!"

The basic idea in all these systems is to be able to gather the most comprehensive possible working set of objects. This is most easily accomplished by swapping individual objects. Now the problem becomes the overhead of purging nonworking set objects to make room for the ones that are needed. (Paging sometimes works better for this part because you can get more than one object (**OOZE**) in each disk touch.) Two ideas help a lot. First, Butler's insight in the GENIE OS that it was worthwhile to expend a small percentage of time purging dirty objects to make core as clean as possible [Lampson 1966]. Thus crashes tend not to hurt as much and there is always clean storage to fetch pages or objects from the disk into. The other is one from the FLEX system in which I set up a stochastic decision mechanism (based on the class of an object) that determined during a purge whether to throw an object out. This had two benefits: important objects tended not to go out, and a mistake would just bring it back in again with the distribution, ensuring a low probablity that the object would be purged again soon.

The other problem that had to be taken care of was object-pointer integity (and this is where I had failed in the FLEX machine to come up with a good enough solution). What was needed really was a complete *transaction*, a brand new technique (thought up by Butler?) that ensured recovery regardless of when the system crashed. This was called "cosmic ray protection" as the early ALTOs had a way of just crashing once or twice a day for no discernable good reason. This, by the way did not particularly bother anyone as it was fairly easy to come up with *undo* and *replay* mechanisms to get around the cosmic rays. For pointer-based systems that had automatic storage management, this was a bit more tricky.

Ted and Dan decided to control storage using a Resident Object Table that was the only place machine addresses for objects would be found. Other useful information was stashed there as well to help LRU ageing. Purging was done in background by picking a class, positioning the disk to its instances (all of a particular class were stored together), then running through the ROT to find the dirty ones in storage and stream them out. This was pretty efficient and, true to Butler's insight, furnished a good-sized pool of clean storage that could be overwritten. The key to the design though (and the implementation of the transaction mechanism) was the checkpointing scheme. This ensured that there was a recoverable image no more than a few seconds old, regardless of when a crash might occur. OOZE swapped objects in just 80KB of working storage and could handle about 65K objects (up to several megabytes worth, more than enough for the entire system, its interface, and its applications).

11.4.3 "Object-oriented" Style

This is probably a good place to comment on the difference between what we thought of as OOP-style and the superficial encapsulation called "abstract data types" that was just starting to be investigated in academic circles . Our early "LISP-pair" definition is an example of an abstract data type because it preserves the "field access" and "field rebinding" that is the hallmark of a data structure. Considerable work in the 1960s was concerned with generalizing such structures. The "official" computer science world started to regard Simula as a possible vehicle for defining *abstract data types* (even by one of its inventors [Dahl 1970]), and it formed much of the later backbone of ADA. This led to the ubiquitous stack data-type example in hundreds of papers. To put it mildly, we were quite amazed at this, because to us, what Simula had whispered was something much stronger than simply reimplementing a weak and *ad hoc* idea. What I got from Simula was that you could now replace

bindings and assignment with *goals*. The last thing you wanted any programmer to do is mess with internal state even if presented figuratively. Instead, the objects should be presented as *sites of higher level behaviors more appropriate for use as dynamic components*.

Even the way we taught children (see also ahead) reflected this way of looking at objects. Not too surprisingly, this approach has considerable bearing on the ease of programming, the size of the code needed, the integrity of the design, and so on. It is unfortunate that much of what is called "object-oriented programming" today is simply old style programming with fancier constructs. Many programs are loaded with "assignment-style" operations now done by more expensive attached procedures.

Where does the special efficiency of object-oriented design come from? This is a good question given that it can be viewed as a slightly different way to apply procedures to data structures. Part of the effect comes from a much clearer way to represent a complex system. Here, the constraints are as useful as the generalities. Four techniques used together—persistant state, polymorphism, instantiation, and methods-as-goals for the object—account for much of the power. None of these requires an "object-oriented language" to be employed—ALGOL 68 can almost be turned to this style—an OOPL merely focuses the designer's mind in a particular, fruitful direction. However, doing encapsulation right is a commitment not just to abstraction of state, but to eliminate state-oriented metaphors from programming.

Perhaps the most important principle—again derived from operating system architectures—is that when you give someone a structure, rarely do you want them to have unlimited privileges with it. Just doing type-matching is not even close to what is needed. Nor is it terribly useful to have some objects protected and others not. Make them all first class citizens and protect all.

I believe that the much smaller size of a good OOP system comes not just by being gently forced to come up with a more thought-out design. I think it also has to do with the "bang per line of code" you can get with OOP. The object carries with it a lot of significance and intention, its methods suggest the strongest kinds of goals it can carry out, and its superclasses can add up to much more code-functionality being invoked than most procedures-on-data-structures. Assignment statements—even abstract ones—express very low-level goals, and more of them will be needed to get anything done. Generally, we don't want the programmer to be messing around with state, whether simulated or not. The ability to instantiate an object has a considerable effect on code size as well. Another way to think of all this is: though the late-binding of automatic storage allocation does not do anything a programmer cannot do, its presence leads to both simpler and more powerful code. OOP is a late binding strategy for many things and all of them together hold off fragility and size explosion much longer than the older methodologies. In other words, human programmers are not Turing machines—and the less their programming systems require Turing machine techniques the better.

11.4.4 Smalltalk And Children

Now that I have summarized the "adult" activities (we were actually only semiadults) in Smalltalk up to 1976, let me return to the summer of '73, when we were ready to start experiments with children. None of us knew anything about working with children, but we knew that Adele Goldberg and Steve Weyer who were then with Pat Suppes at Stanford had done quite a bit and we were able to entice them to join us.

Because we had no idea how to teach object-oriented programming to children (or anyone else), the first experiments Adele did mimicked LOGO turtle graphics, and she got what appeared to be very similar results. That is to say, the children could get the turtle to draw pictures on the screen, but there

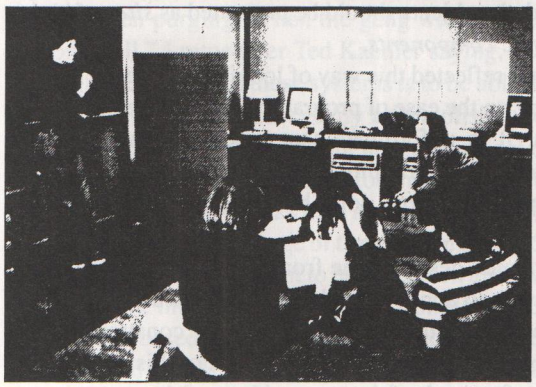


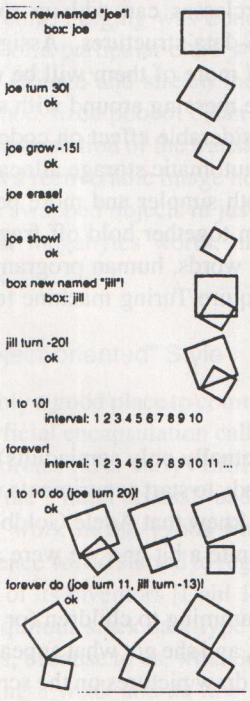
FIGURE 11.42 Adele holding forth at Jordan Middle School

seemed to be little happening beyond surface effects. At that time I felt that because the content of personal computing was interactive tools, the content of this new kind of authoring literacy should be the creation of interactive *tools* by the children. Procedural turtle graphics just wasn't it.

Then Adele came up with a brilliant approach to teaching Smalltalk as an object-oriented language: the "Joe Book." I believe this was partly influenced by Minsky's idea that you should teach a programming language holistically from working examples of serious programs.

Several instances of the class box are created and sent messages, culminating in a simple multiprocess animation. After getting kids to guess what a box might be like—they could come surprisingly close—they would be shown:

```
to box | x y size tilt
(¤draw    »  (@ place x y turn tilt. square size.)
¤undraw   »  (@ white. SELF draw. @ black)
¤turn     »  (SELF undraw. 'tilt <- tilt + :. SELF draw)
¤grow     »  (SELF undraw. 'size <- size + :. SELF draw)
ISNEW     »  (SELF undraw. 'size <- size + :. SELF draw)
```



What was so wonderful about this idea were the myriad children's projects that could spring off the humble boxes. And some of the earliest were tools! This was when we got really excited. For example, Marion Goldeen's (12 years old) painting system was a full-fledged tool. A few years later, so was Susan Hamet's (12 years old) OOP illustration system (with a design that was like the MacDraw to come). Two more were Bruce Horn's (15 years old) music score capture system and Steve Putz's (15 years old) circuit design system. Looking back, this could be called another example in computer science of the "early success syndrome." The successes were real, but they were not as general as we thought. They would not extend into the future as strongly as we hoped. The children were chosen from the Palo Alto schools (hardly an average background) and we tended to be much more excited about the successes than the difficulties. In part, what we were seeing was the "hacker phenomenon," that, for any given pur-

suit, a particular 5% of the population will jump into it naturally, whereas the 80% or so who can learn it in time do not find it at all natural.

We had a dim sense of this, but we kept on having relative successes. We could definitely see that learning the mechanics of the system was not a major problem. The children could get most of it themselves by swarming over the ALTOS with Adele's JOE book. The problem seemed more to be that of design.

It started to hit home in the Spring of '74 after I taught Smalltalk to 20 PARC nonprogrammer adults. They were able to get through the initial material faster than the children, but just as it looked like an overwhelming success was at hand, they started to crash on problems that did not look to me to be much harder than the ones on which they had just been doing well. One of them was a project thought up by one of the adults, which was to make a little database system that could act like a card file or rolodex. They could not even come close to programming it. I was very surprised because I "knew" that such a project was well below the mythical "two pages" for end-users within we were working. That night I wrote it out, and the next day I showed all of them how to do it. Still, none of them were able to do it by themselves. Later, I sat in the room pondering the board from my talk. Finally, I counted the number of nonobvious ideas in this little program. They came to 17. And some of them were like the concept of the arch in building design: very hard to discover, if you do not already know them.

FIGURE 11.43 The suthor in the Interim Dynabook playroom. Working with the kids was my favorite part of this Romance

FIGURE 11.44 Adele's planning template for Smalltalk (top) New behavior added by child (bottom)

Message the box can receive	English description of the action the box will carry out	Smalltalk description
new	It creates a new box that needs its own *pen* to draw the new box on the display, and that must remember its size whose first value is 50. Then it draws itself on the display screen	**pen new name "pal". number new name "size". size value 50. SELF draw.**
draw	The box has its pen draw a square on the screen at the pen's current location and orientation. The length of its four sides is **size.**	**do 4 (pal draw size. pal turn 90).**
undraw	Erase the box.	**pal white. SELF draw. pal black.**
grow	After erasing itself, the box instance retrieves a message which is interpreted as an increment of its size. It then redraws itself as a bigger or smaller square.	**SELF undraw.** **size increase by :. SELF draw.**
turn	After erasing itself, the box instance retrieves a message which is interpreted as an increment of its orientation. Note, since the pen, rather than the box, remembers the orientation, the box has to tell the pen to turn.	**SELF undraw.** **pal turn :. SELF draw.**

Message the box can receive	English description of the action the box will carry out	Smalltalk description
moveto	After erasing itself, the box instance retrieves two messages which are interpreted as the new coordinates of the box. Note, since the pen, rather than the box, remembers the location, the box has to tell the pen to place itself at the new location.	**SELF undraw.** **pal place at :. SELF draw.**

The connection to literacy was painfully clear. It is not enough to just learn to read and write. There is also a *literature* that renders *ideas*. Language is used to read and write about them, but at some point the organization of ideas starts to dominate mere language abilities. And it helps greatly to have some powerful ideas under one's belt to better acquire more powerful ideas [Papert 1971, 1971a, 1973]. So, we decided we should teach *design*. And Adele came up with another brilliant stroke to deal with this. She decided that what was needed was an intermediary between the vague ideas about the problem and the very detailed writing and debugging that had to be done to get it to run in Smalltalk. She called the intermediary forms *design templates*.

Using these, the children could look at a situation they wanted to simulate, and decompose it into classes and messages without having to worry just how a method would work. The method planning

FIGURE 11.45 Marion Goldeen's painting program (top) Susan Hamet's OO Illustrator (bottom)

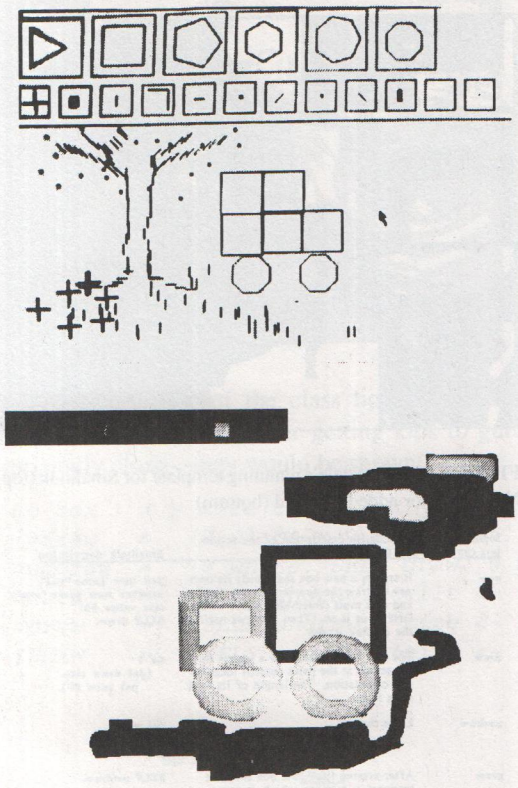
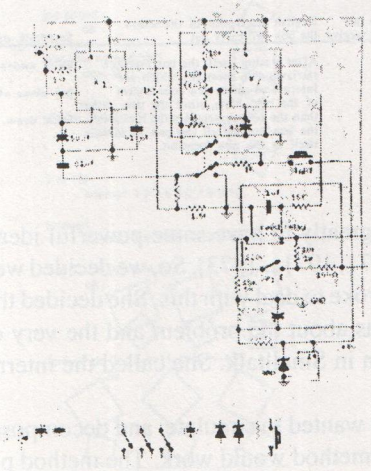


FIGURE 11.46 Cirrcuit design system by Steve Putz (age 15)



could then be done informally in English, and these notes would later serve as commentaries and guides to the writing of the actual code. This was a terrific idea, and it worked very well.

But not enough to satisfy us. As Adele liked to point out, it is hard to claim success if only some of the children are successful—*and* if a maximum effort of both children and teachers was required to get the successes to happen. Real pedagogy has to work in much less idealistic settings and be considerably more robust. Still, *some* successes are qualitatively different from *no* successes. We wanted more, and started to push on the inheritance idea as a way to let novices build on frameworks that could only be designed by experts. We had good reason to believe that this could work because we had been impressed by Lisa van Stone's ability to make significant changes to Shazam (the five or six page Smalltalk animation tool done by relatively expert adults). Unfortunately, inheritance—though an incredibly powerful technique—has turned out to be very difficult for novices (and even professionals) to deal with.

At this point, let me do a look back from the vantage point of today. I am now pretty much convinced that our design template approach was a good one after all. We just did not apply it longitudinally enough. I mean by this that there is now a large accumulation of results from many attempts to teach novices programming [Soloway 1989]. They all have similar stories that seem to have little to do with the various features of the programming languages used, and everything to do with the difficulties novices have thinking the special way that good programmers think. Even with a much better interface than we had then (and have today), it is likely that this area is actually more like writing than we wanted it to be. Namely, for the "80%," it really has to be learned gradually over a period of years in order to build up the structures that need to be there for design and solution look-ahead.

The problem is not to get the kids to do stuff—they love to *do*, even when they are not

sure exactly what they are doing. This correlates well with studies of early learning of language, when much rehearsal is done regardless of whether content is involved. Just *doing* seems to help. What is difficult is to determine *what* ideas to put forth and how *deeply* they should penetrate at a given child's developmental level. This confusion still persists for reading and writing of natural language—and for mathematics—despite centuries of experience. And it is the main hurdle for teaching children programming. When, in what order and depth, and how should the powerful ideas be taught?

Should we even try to teach programming? I have met hundreds of programmers in the last 30 years and can see no discernable influence of programming on their general ability to think well or to take an enlightened stance on human knowledge. If anything, the opposite is true. Expert knowledge often remains rooted in the environments in which it was first learned—and most metaphorical extensions result in misleading analogies. A remarkable number of artists, scientists, and philosophers are quite dull outside of their specialty (and one suspects within it as well). The first siren's song we need to be wary of is the one that promises a connection between an interesting pursuit and interesting thoughts. The music is not in the piano, and it is possible to graduate Julliard without finding or feeling it.

I have also met a few people for whom computing provides an important new metaphor for thinking about human knowledge and reach. But something else was needed besides computing for enlightenment to happen.

Tools provide a path, a context, and almost an excuse for developing enlightenment, but no tool ever contained it or can dispense it. Cesare Pavese observed: to know the world we must construct it. In other words, *we make not just to have, but to know*. But the having can happen without most of the knowing taking place.

Another way to look at this is that knowledge is in its least interesting state when it is first being learned. The representations—whether

FIGURE 11.47 Tangram designs are created by selecting shapes from a "menu" displayed at the top of the screen. This system was implemented in Smalltalk by a fourteen-year old girl [Kay 1977a]

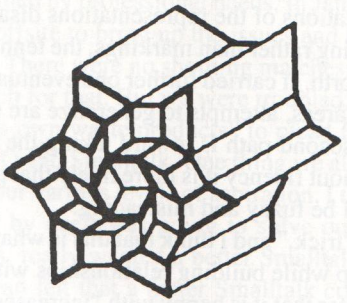


FIGURE 11.48 SpaceWar by Dennis (age 12)

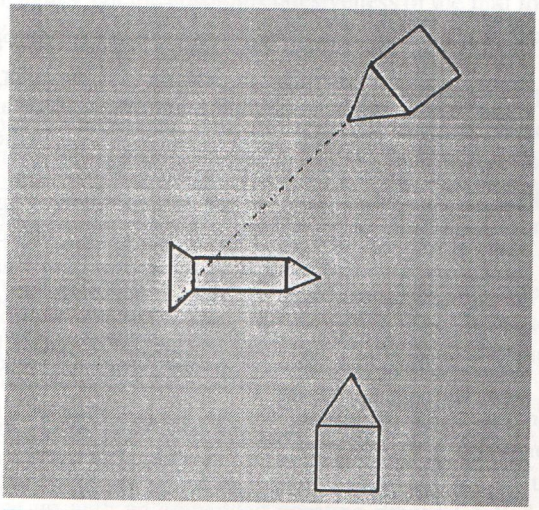
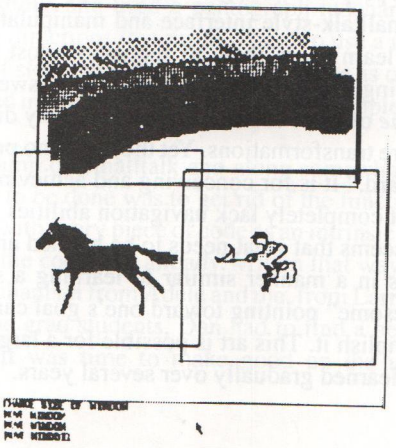


FIGURE 11.49 Shazam modified to "group" multiple images by Lisa van Stone (age 12)



markings, allusions, or physical controls—get in the way (almost take over as goals) and must be laboriously and painfully interpreted. From here there are several useful paths, two of which are important and intertwined.

The first is *fluency*, which in part is the process of building mental structures that make the interpretations of the representations disappear. The letters and words of a sentence are experienced as meaning rather than markings, the tennis racquet or keyboard becomes an extension of one's body, and so forth. If carried further one eventually becomes a kind of expert—but without deep knowledge in other areas, attempts to generalize are usually too crisp and ill-formed.

The second path is toward taking the knowledge as a *metaphor* than can illuminate other areas. But without fluency it is more likely that prior knowledge will hold sway and the metaphors from this side will be fuzzy and misleading.

The "trick," and I think that this is what liberal arts eduation is supposed to be about, is to get fluent *and* deep while building relationships with other fluent deep knowledge. Our society has lowered its aims so far that it is happy with "increases in scores" without daring to inquire whether any important threshold has been crossed. Being able to read a warning on a pill bottle or write about a summer vacation is not literacy and our society should not treat it so. Literacy, for example, is being able to fluently read and follow the 50-page argument in Paine's *Common Sense* and being able (and happy) to fluently write a critique or defense of it. Another kind of 20th century literacy is being able to hear about a new fatal contagious incurable disease and instantly know that a disastrous exponential relationship holds and early action is of the highest priority. Another kind of literacy would take citizens to their personal computers where they can fluently and without pain build a systems simulation of the disease to use as a comparison against further information.

At the liberal arts level we would expect that connections between each of the fluencies would form truly powerful metaphors for considering ideas in the light of others.

The reason, therefore, that many of us want children to understand computing deeply and fluently is that like literature, mathematics, science, music, and art, it carries special ways of thinking about situations that, in contrast with other knowledge and other ways of thinking critically, boost our ability to understand our world.

We did not know then, and I am sorry to say from 15 years later that these critical questions still do not yet have really useful answers. But there are some indications. Even very young children can understand and use interactive *transformational tools*. The first ones are their hands! They can readily extend these experiences to computer objects and making changes to them. They can often imagine what a proposed change will do and not be surprised at the result. Two- and three-year-olds can use the Smalltalk-style interface and manipulate object-oriented graphics. Third graders can (in a few days) learn more than 50 features—most of these are transformational tools—of a new system including its user interface. They can answer any question whose answer requires the application of just *one* of these tools. But it is extremely difficult for them to answer any question that requires *two* or more transformations. Yet they have no problem applying sequences of transformations, exploring "forward." It is for conceiving and achieving even modest goals requiring several changes that they almost completely lack navigation abilities.

It seems that what needs to be learned and taught is how to package transformations in two's and three's in a manner similar to learning a strategic game such as checkers. The vague sense of a "threesome" pointing toward one's goal can be a setup for the more detailed work that is needed to accomplish it. This art is possible for a large percentage of the population, but for most, it will need to be learned gradually over several years.

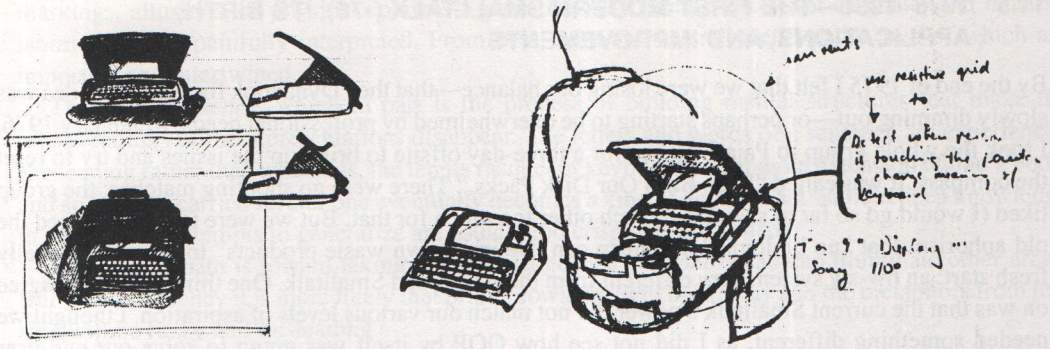
11.5 1976–1980—THE FIRST MODERN SMALLTALK (-76), ITS BIRTH, APPLICATIONS, AND IMPROVEMENTS

By the end of 1975 I felt that we were losing our balance—that the "Dynabook for children" idea was slowly dimming out—or perhaps starting to be overwhelmed by professional needs. In January 1976, I took the whole group to Pajaro Dunes for a three-day offsite to bring up the issues and try to reset the compass. It was called "Let's Burn Our Disk Packs." There were no shouting matches; the group liked (I would go so far to say: *loved*) each other too much for that. But we were troubled. I used the old aphorism that "no biological organism can live in its own waste products" to plead for a really fresh start: an hw-sw system very different from the ALTO and Smalltalk. One thing we all did agree on was that the current Smalltalk's power did not match our various levels of aspiration. I thought we needed something different, as I did not see how OOP by itself was going to solve our end-user problems. Others, particularly some of the grad students, really wanted a better Smalltalk that was faster and could be used for bigger problems. I think Dan felt that a better Smalltalk could be the vehicle for the different system I wanted, but could not describe clearly. The meeting was not a disaster, and we went back to PARC still friends and colleagues, but the absolute cohesiveness of the first four years never re-jelled. I started designing a new small machine and language I called the *NoteTaker* and Dan started to design Smalltalk-76.

The reason I wanted to "burn the disk packs" is that I had a very McLuhanish feeling about media and environments: that once we have shaped tools, in his words, they turn around and "reshape us." Of course this is a great idea if the tools are *really* good and aimed squarely at the issues in question. But the other edge of the sword cuts as deep—that inadequate tools and environments still reshape our thinking in spite of their problems, in part, because we *want* paradigms to guide our goals. Strong paradigms such as LISP and Smalltalk are so compelling that they *eat their young*: when you look at an application in either of these two systems, they resemble the systems themselves, not a new idea. When I looked at Smalltalk in 1975, I was looking at something great, but I did not see an end-user language; I did not see a solution to the original goal of a "reading" and "writing" computer medium for children. I wanted to stop, dynamite everything, and start from scratch again.

The *NoteTaker* was to be a "laptop" that could be built in a few years using the (almost) available 16K RAMs (a vast improvement over the 1K RAMs that the ALTO employed). A laptop could not use a mouse (which I hated anyway) and a tablet seemed awkward (not a lot of room and the stylus could flop out of reach when let go), so I came up with an embedded pointing device I called a "tabmouse." It was a relative pointer and had an *up* sensor so it could be stroked like a mouse and would also stay where you left it, but it felt like a stylus and used a pantograph mechanism that eliminated the annoying hysteresis bias in the x and y directions that made it hard to use a mouse as a pen. I planned to use a multiprocessor architecture of slow but highly integrated chips as originally specified for the Dynabook and wanted a new bytecoded interpreter for a friendlier and simpler system than Smalltalk-72.

Meanwhile Dan was proceeding with his total revamp of Smalltalk and along somewhat similar lines [Ingalls 1978]. The first major thing that needed to be done was to get rid of the function/class dualism in favor of a completely intensional definition with every piece of code as an intrinsic method. We had wanted that from the beginning (and most of the code was already written that way). There was a variety of strong desires for a real inheritance mechanism from Adele and me, from Larry Tesler, who was working on desktop publishing, and from the grad students. Dan had to find a better way than Simula's very rigid compile-time conception. It was time to make good on the idea that



"everything was an object," which included all of the internal "systems" objects like "activation records," and the like. We were all agreed that the flexible syntax of the earlier Smalltalks was too flexible, and this level of extensibility was not desirable. All the extensions we liked used various keyword schemes, so Dan came up with a combination keyword/operator syntax that was very flexible, but allowed the language to be read unambiguously by both humans and the machine. This allowed a FLEX machine-like byte-code compiler and efficient interpreter to be defined that ran up to 180 times as fast as the previous direct interpreter. The OOZE VM system could be modified to handle the new objects and its capacity was well matched to the ALTO's RAM and disk.

11.5.1 Inheritance

A word about inheritance. Simula-I had neither classes as objects nor inheritance. Simula-67 added the latter as a generalization to the ALGOL-60 <block> structure. This was a great idea. But it did have some drawbacks: minor ones, like name clashes in multiple threaded lists (no one uses threaded lists anymore), and major ones, like a rigidity in the extended type structures, a need to qualify types, only a single path of inheritance, and difficulty in adapting to an interactive development system with incremental compiling and other needs for instant changes. Then there were a host of problems that were really outside the scope of Simula's goals: having to do with various kinds of modeling and inferencing that were of interest in the world of artificial intelligence. For example, not all useful questions could be answered by following a static chain. Some of them required a kind of "inheritance" or "inferencing" through dynamically bound "parts" (that is, instance variables). Multiple inheritance also looked important but the corresponding possible clashes between methods of the same name in different superclasses looked difficult to handle, and so forth.

On the other hand, because things can be done with a dynamic language that are difficult with a statically compiled one, I just decided to leave inheritance out as a feature in Smalltalk-72, knowing that we could simulate it back using Smalltalk's LISPlike flexibility. The biggest contributer to these AI ideas was Larry Tesler, who used what is now called "slot inheritance" extensively in his various versions of early desktop publishing systems. Nowadays, this would be called a "delegation-style" inheritance scheme [Lieberman]. Danny Bobrow and Terry Winograd during this period were designing a "frame-based" AI language called KRL which was "object-oriented" and I believe was influenced by early Smalltalk. It had a kind of multiple inheritance—called *perspectives*—that permitted an object to play multiple roles in a very clean way. Many of these ideas a few years later went into PIE, an interesting extension of Smalltalk to networks and higher level descriptions by Ira Goldstein and Bobrow [Goldstein and Bobrow 1980].

By the time Smalltalk-76 came along, Dan Ingalls had come up with a scheme that was Simula-like in its semantics but could be incrementally changed on the fly to be in accord with our goals of close interaction. I was not completely thrilled with it because it seemed that we needed a better theory about inheritance entirely (and still do). For example, inheritance and instancing (which is a kind of inheritance) muddles both pragmatics (such as factoring code to save space) and semantics (used for way too many tasks such as: specialization, generalization, speciation, and so forth). Alan Borning employed a multiple inheritance scheme in Thinglab [Borning 1977] which was implemented in Smalltalk-76. But no comprehensive and clean multiple inheritance scheme appeared that was compelling enough to surmount Dan's original Simula-like design.

Meanwhile, the running battle with Xerox continued. There were now about 500 ALTOs linked with Ethernets to each other and to Laserprinter and file servers, that used ALTOs as controllers. I wrote many memos to the Xerox planners trying to get them to make plans that included personal computing as one of their main directions. Here is an example:

A Simple Vision of the Future

A Brief Update of My 1971 Pendery Paper

In the 1990's there will be millions of personal computers. They will be the size of notebooks of today, have high-resolution flat-screen reflective displays, weigh less than ten pounds, have ten to twenty times the computing and storage capacity of an *Alto*. Let's call them *Dynabooks*.

The purchase price will be about that of a color television set of the era, although most of the machines will be given away by manufacturers who will be marketing the content rather than the container of personal computing.

...

Though the *Dynabook* will have considerable local storage and will do most computing locally, it will spend a large percentage of its time hooked to various large, global information utilities which will permit communication with others of ideas, data, working models, as well as the daily chit-chat that organizations need in order to function. The communications link will be by private and public wires and by packet radio. Dynabooks will also be used as servers in the information utilities. They will have enough power to be entirely shaped by software.

The Main Points of This Vision

- There need only be a few hardware types to handle almost all of the processing activity of a system.
- Personal Computers, Communications Links, and Information Utilities are the three critical components of a Xerox future.

...

In other words, the *material* of a computer system is the computer itself, *all* of the *content* and *function* is fashioned in software.

There are two important guidelines to be drawn from this:

» Material: If the design and development of the hardware computer material is done as carefully and completely as Xerox's development of special light-sensitive alloys, then only one or two computer designs need to be built... Extra investment in development here will be vastly repaid by simplifying the manufacturing process and providing lower costs through increased volume.

» Content: Aside from the wonderful generality of being able to continously shape new content from the same material, *software* has three important characteristics:

- the *replication* time and cost of a content-function is *zero*
- the *development* time and cost of a content-function is *high*
- the *change* time and cost of a content-function can be *low*

Xerox ***must*** take these several points seriously if it is to survive and prosper in its new business area of information media. If it does, the company has an excellent chance for several reasons:

- Xerox has the financial base to cover the large development costs of a small number of very powerful computer-types and a large number of software functions.
- Xerox has the marketing base to sell these functions on a wide enough scale to garner back to itself an incredible profit.
- Xerox has working for it an impressively large percentage of the best software designers in the world.

In 1976, Chuck Thacker designed the ALTO III that would use the new 16k chips and be able to fit on a desktop. It could be marketed for about what the large cumbersome special purpose "word-processors" cost, yet could do so much more. Nevertheless, in August of 1976, Xerox made a fateful decision: not to bring the ALTO III to market. This was a huge blow to many of us—even me, who had never really really thought of the ALTO as anything but a stepping stone to the "real thing." In 1992, the world market for personal computers and workstations was $90 million—twice as much as the mainframe and mini market, and many times Xerox's 1992 gross. The most successful company of this era—Microsoft—is not a hardware company, but a software company.

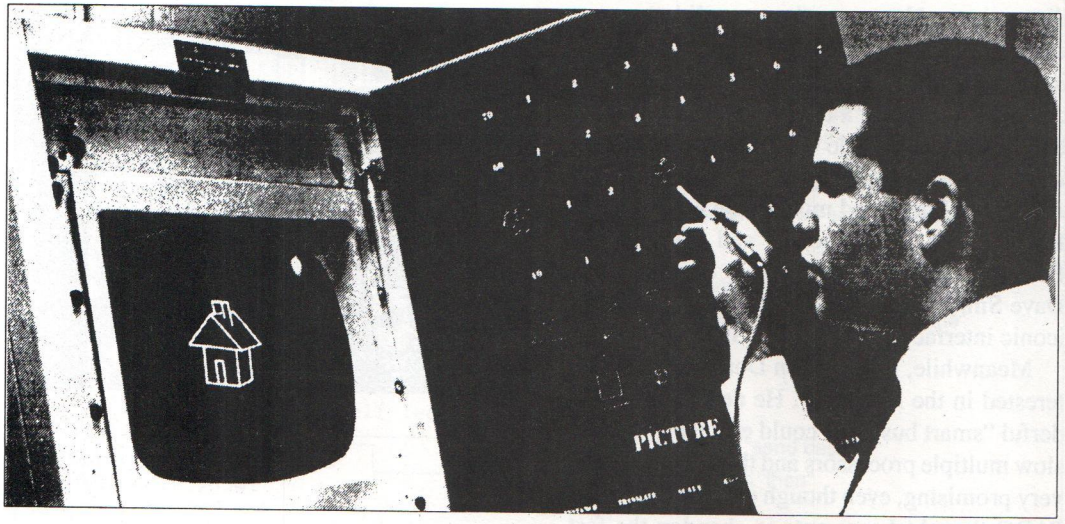
11.5.2 The Smalltalk User Interface

I have been asked by several of the reviewers to say more about the development of the "Smalltalk-style" overlapping window user interface in as much as there are now more than 20 million computers in the world that use its descendents. A decent history would be as long as this chapter, and none has been written so far. There is a summary of some of the ideas in [Kay 1989]—let me add a few more points.

All of the elements eventually used in the Smalltalk user interface were already to be found in the sixties—as different ways to access and invoke the functionality provided by an interactive system. The two major centers of ideas were Lincoln Labs and RAND Corporation—both ARPA funded. The big shift that consolidated these ideas into a powerful theory and long-lived examples came because the LRG focus was on children. Hence, we were thinking about learning as being one of the main effects we wanted to have happen. Early on, this led to a 90-degree rotation of the purpose of the user interface from "access to functionality" to "environment in which users learn by doing." This new stance could now respond to the echos of Montessori and Dewey, particularly the former, and got me, on rereading Jerome Bruner, to think beyond the children's curriculum to a "curriculum of the user interface."

The particular aim of LRG was to find the equivalent of writing—that is, learning and thinking by doing in a medium—our new "pocket universe." For various reasons I had settled on "iconic programming" as the way to achieve this, drawing on the iconic representations used by many ARPA projects in the sixties. My friend Nicholas Negroponte, an architect, was extremely interested in how environments affected people's work and creativity. He was interested in embedding the new computer

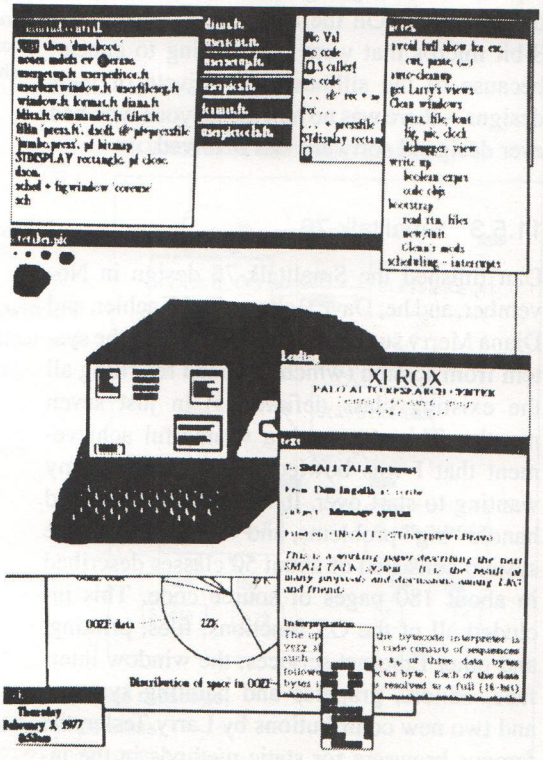
FIGURE 11.50 Paul Rovner showing the iconic "Lincoln Wand" ca. 1968



magic in familar surroundings. I had quite a bit of theatrical experience in a past life, and remembered Coleridge's adage that "people attend 'bad theatre' hoping to forget, people attend 'good theatre' *aching to remember*." In other words, it is the ability to evoke the audience's own intelligence and experiences that makes theatre work.

Putting all this together, we want an apparently free environment in which exploration causes desired sequences to happen (Montessori); one that allows kinesthetic, iconic, and symbolic learning—"*doing* with *images* makes *symbols*" (Piaget and Bruner); the user is never trapped in a mode (GRAIL); the magic is embedded in the familiar (Negroponte); and which acts as a magnifying mirror for the user's own intelligence (Coleridge). It would be a great finish to this story to say that having articulated this we were able to move straightforwardly to the design as we know it today. In fact, the UI design work happened in fits and starts in between feeding Smalltalk itself, designing children's experiments, trying to understand iconic construction, and just playing around. In spite of this meandering, the context almost forced a good design to turn out anyway. Just about everyone at PARC at this time had opinions about the UI, ours and theirs. It is impossible to

FIGURE 11.51 The last Smalltalk-72 interface



give detailed credit for the hundreds of ideas and discussions. However, the consolidation can certainly be attributed to Dan Ingalls, for listening to everyone, contributing original ideas, and constantly building a design for user testing. I had a fair amount to do with setting the context, inventing overlapping windows, and so on, and Adele and I designed most of the experiments. Beyond that, Ted Kaehler and visitor, Ron Baecker, made highly valuable contributions. Dave Smith designed SmallStar, the prototype iconic interface for the Xerox Star product.

Meanwhile, I had gotten Doug Fairbairn interested in the *Notetaker*. He designed a wonderful "smart bus" that could efficiently handle slow multiple processors and the system looked very promising, even though most of the rest of PARC thought I was nuts to abandon the fast bipolar hw of the ALTO. But I could not see that bipolar was ever going to make it into a laptop or Dynabook. On the other hand, I hated the 8-bit micros that were just starting to appear, because of the silliness and naivete of their designs—there was no hint that anyone who had ever designed software was involved.

11.5.3 Smalltalk-76

Dan finished the Smalltalk-76 design in November, and he, Dave Robson, Ted Kaehler, and Diana Merry successfully implemented the system from scratch (which included rewriting all the existing class definitions) in just seven months. This was such a wonderful achievement that I was bowled over in spite of my wanting to start over. It was fast, lively, could handle "big" problems, and was great fun. The system consisted of about 50 classes described in about 180 pages of source code. This included all of the OS functions, files, printing, and other Ethernet services, the window interface, editors, graphics and painting systems, and two new contributions by Larry Tesler, the famous browsers for static methods in the inheritance hierarchy, and dynamic contexts for debugging in the run-time environment. In every way it was the consolidation of all our

FIGURE 11.52 Ted Kaehler's iconic painting interface

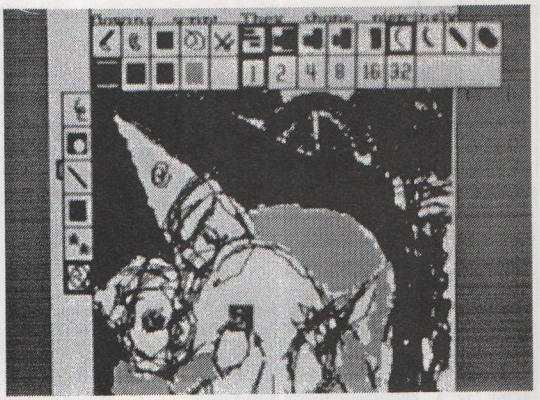
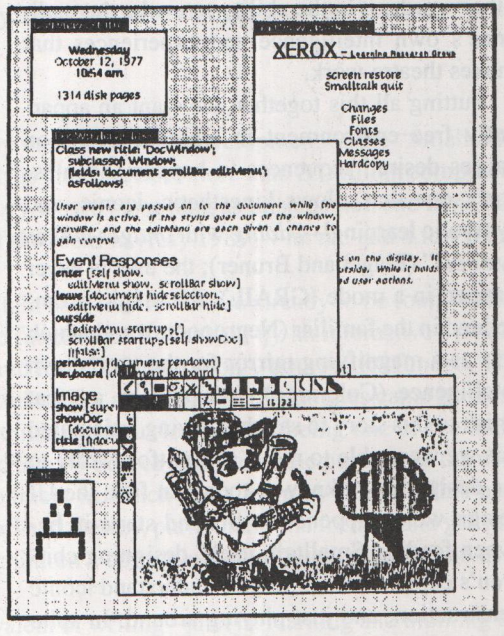


FIGURE 11.53 *Smalltalk-76 User Interface with a variety of* applications, including a clock, font editor, painting and illustration editor with iconic menus and programmable radio buttons, a word processor document editor, and a class editor showing window interface code.



```
Class new title: 'Window';
  fields: 'frame';
  asFollows!
```

This is a superclass for presenting windows on the display. It holds control until the stylus is depressed outside. While it holds control, it distributes messages to itself based on user actions.

```
Scheduling
startup
  [frame contains: stylus =>
    self enter.
    repeat:
      [frame contains: stylus loc =>
        [keybard active => [self keyboard]
        stylus down => [self pendown]]
      self outside => []
      stylus down => [^self leave]]]
  ^false]
Default Event Responses
enter [self show]
leave
outside [^false]
pendown
keyboard [keyboard next. frame flash]
Image
show
  [frame outline: 2.
  titleframe put: self title at: frame origin + title loc.
  titleframe complement]
... etc.
```

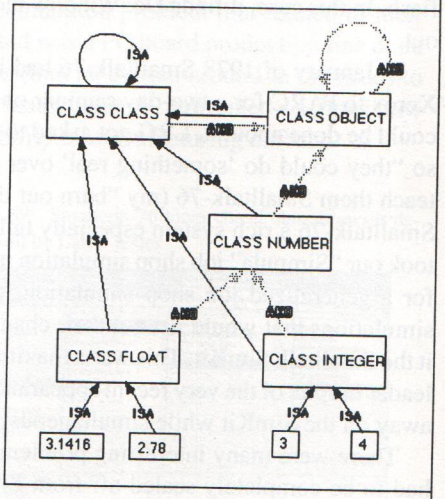
: means keyword whose following expression will be sent "by value"

: means keyword whose following expression will be sent "by name"

^ means "send back"

=> means "then"

FIGURE 11.54 Smalltalk-76 Metaphysics



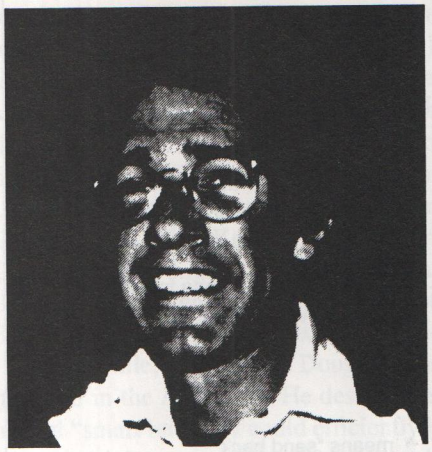

```
Class new title: 'DocWindow';
  subclassof: Window;
  fields: 'document scrollbar edit Menu';
  asFollows!
```

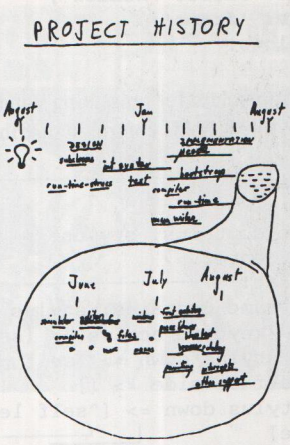
User events are passed on to the document while the window is active. If the stylus goes out of the window, scrollbar and the editMenu are each given a chance to gain control.

```
Event Responses
enter [self show.editMenu show.scrollbar
show]
leave [document hideselection.editMenu
hide.scrollbar hide]
outside
  [editMenu startup => []
  scrollbar startup => [self showDoc]
  ^flase]
pendown [document pendown]
keyboard [document keyboard]
Image
show [super show.self showDoc]
showDoc [doucument showin: frame at: scrollbar position]
title [^document title]
```

super means delegate message to next higher superclass

FIGURE 11.55 Dan Ingalls, the main implementer of Smalltalk, creator of Smalltalk-76, and his implementation plan (right)





ideas and yearnings about Smalltalk in one integrated package. All Smalltalks since have resembled this conception very closely. In many ways, as Tony Hoare once remarked about ALGOL, Dan's Smalltalk-76 was a great improvement on its successors!

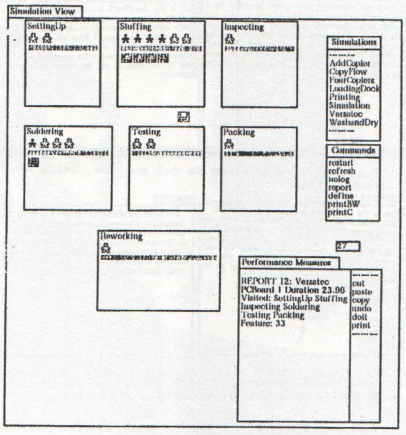
Here are two stylish ST-76 classes written by Dan. Notice, particularly in class Window, how the code is expressed as goals for other objects (or itself) to achieve. The superclass Window's main job is to notice events and distribute them as messages to its subclasses. In the example, a document window (a subclass of DocWindow) is going to deal with the effects of user interactions. The Window class will notice that the keyboard is active and send a message to itself that will be intercepted by the subclass method. If there is no method the character will be thrown away and the window will flash. In this case, it finds DocWindow method: **keyboard**, which tells the held document to check it out.

In January of 1978 Smalltalk-76 had its first real test. CSL had invited the top ten executives of Xerox to PARC for a two-day seminar on software, with a special emphasis on complexity and what could be done about it. LRG got asked to give them a hands-on experience in end-user programming so "they could do 'something real' over two 1 1/2 hour sessions." We immediately decided *not* to teach them Smalltalk-76 (my "burn our disk packs" point in spades), but to create in two months in Smalltalk-76 a rich system especially tailored for adult nonexpert users (Dan's point in trumps). We took our "Simpula" job shop simulation model as a starting point and decided to build a user interface for a generalized job shop simulation tool that the executives could make into specific dynamic simulations that would act out their changing states by animating graphics on the screen. We called it the Smalltalk SimKit. This was a maximum effort and everyone pitched in. Adele became the design leader in spite of the very recent appearance of a new baby. I have a priceless memory of her debugging away on the SimKit while simultaneously nursing Rachel!

There were many interesting problems to be solved. The system itself was straightforward but it had to be completely sealed off from Smalltalk proper, particularly with regard to error messages. Dave Robson came up with a nice scheme (almost an expert system) to capture complaints from the bowels of Smalltalk and translated them into meaningful SimKit terms. There were many user interface details—some workaday, such as making new browsers that could only look at the four SimKit classes (Station, Worker, Job, Report), and some more surprising as when we tried it on ten PARC nontechnical adults of about the same age and found that they could not read the screen very well. The small fonts our thirtysomething year-old eyes were used to did not work for those in their

FIGURE 11.56 Jack Goldman finally uses the system he paid for all those years (with Alan Borning helping)

FIGURE 11.57 An end-user simulation by a Xerox executive, in SimKit. Total time including training: 3 hours



fifties. This led to a nice introduction to the system in which the executives were encouraged to customize the screen by choosing among different fonts and sizes with the side effect that they learned how to use the mouse unselfconsciously.

On the morning of the "big day" Ted Kaehler decided to make a change in the virtual memory system OOZE to speed it up a little. We all held our breaths, but such was the clarity of the design and the confidence of the implementers that it did work, and the executive hands-on was a howling success. About an hour into the first session one of the VPs (who had written a few programs in FORTRAN 15 years before) finally realized he was programming and mused "so it's finally come to this." Nine out of the ten executives were able to finish a simulation problem that related to their specific interests. One of the most interesting and sophisticated was a PC board production line done by the head of a Xerox-owned company using actual figures (that he carried around in his head) to prime a model that could not be solved easily by closed form mathematics—it revealed a serious flaw in the disposition of workers, given the line's average probability of manufacturing defects.

FIGURE 11.58 Alan Borning's Thinglab, a constraint-based iconic problem solver

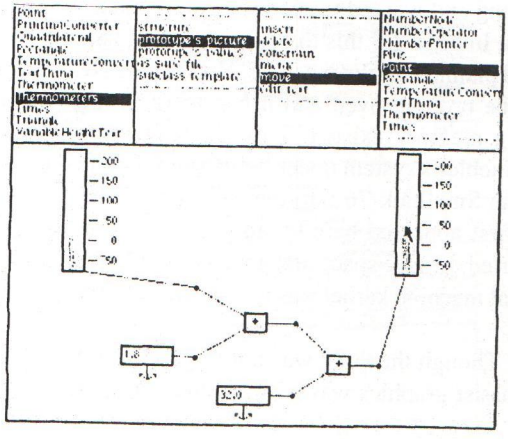


FIGURE 11.59 Smalltalk-76 hierarchical class browser designed and built by Larry Tesler

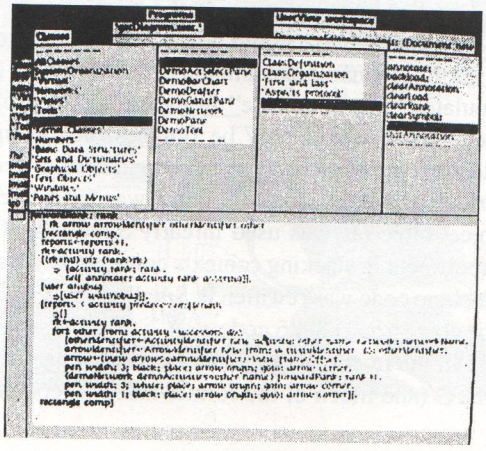
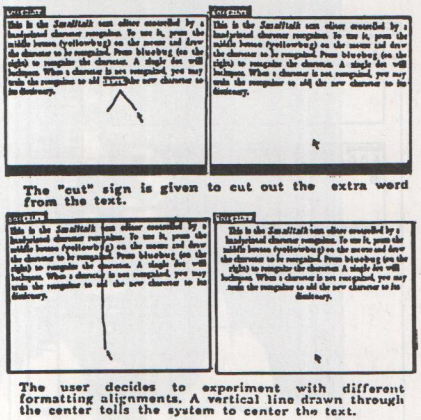


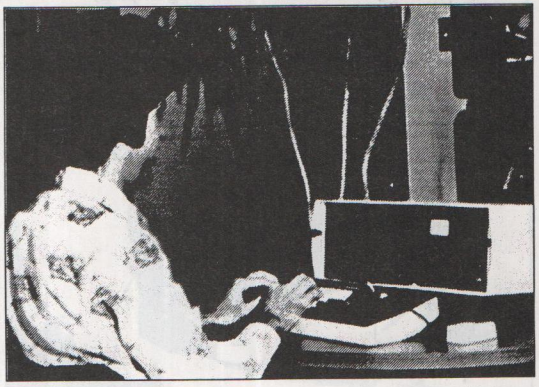
FIGURE 11.60 The author' pen-based interface for ST-



The "cut" sign is given to cut out the extra word from the text.

The user decides to experiment with different formatting alignments. A vertical line drawn through the center tells the system to center the text.

FIGURE 11.61 Doug Fairbain using his *NoteTaker*



Another important system done at this time was Alan Borning's Thinglab [Borning1979]—the first serious attempt to go beyond Ivan Sutherland's Sketchpad. Alan devised a very nice approach for dealing with constraints that did not require the solver to be omniscient (or able to solve Fermat's last theorem).

We could see that the "pushing" style of Smalltalk could eventually be relaced by a "pulling" style that was driven by changes to values that different methods were based on. This was an old idea but Thinglab showed how the object-oriented definition could be used to automatically limit the contexts for event-driven processing. And we soon discovered that "prototypes" were more hospitable than classes and that multiple inheritance would be well served if there were classes for methods that knew generally what they were supposed to be about (inspired by Pat Winston's second order models).

Meanwhile, the *NoteTaker* was getting more real, bigger, and slower. By this time the Western Digital emulation-style chips I hoped to use showed signs of being "diffusion-ware," and did not look like they would really show up. We started looking around for something that we could count on, even if it did not have a good architecture. In 1978, the best candidate was the Intel 8086, a 16-bit chip (with many unfortunate remnants of the 8008 and 8080), but with (barely) enough capacity to do the job—we would need three of them to make up for the ALTO, one for the interpreter, one for bitmapped graphics, and one for I/O (networking, and so on).

Dan had been interested in the *NoteTaker* all along and wanted to see if he could make a version of Smalltalk-76 that could be the *NoteTaker* system. In order for this to happen, it would have to run in 256K (the maximum amount of RAM that we had planned for the machine. None of the NOVA-like emulated "machine-code" from the ALTO could be brought over, and it had to fit in memory as well—there would only be floppies, no swapping memory existed. This challenge led to some excellent improvements in the system design. Ted Kaehler's system tracer (which could write out new virtual memories from old ones) was used to clone Smalltalk-76 into the *NoteTaker*. The indexed object table (as was used in early Smalltalk-80) first appeared here to simplify object access. An experiment in stacking contexts contiguously was tried: to save space and gain speed. Most of the old machine code was rewitten in Smalltalk and the total machine kernel was reduced to 6K bytes of (the not very strong) 8086 code.

All the re-engineering had an interesting effect. Though the 8086 was not as good at bitblt as the ALTO (and much of the former machine code to assist graphics was now in Smalltalk), the overall

FIGURE 11.62 Design for *NoteTaker* Interface [Kay 1979]

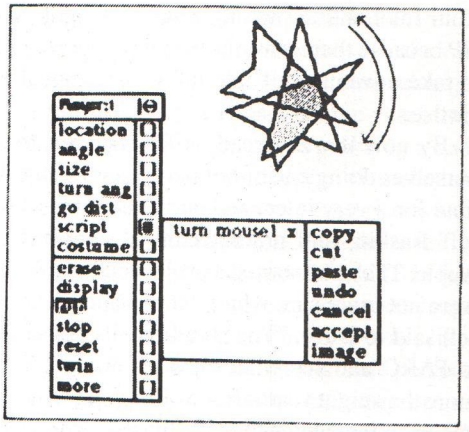


FIGURE 11.63 Diana Merry at her trusty ALTO

interpreter was about twice as fast as the ALTO version (because not all the Smalltalk byte-code interpreter would fit into the 4K microcode memory on the ALTO). With various kinds of tricks and tuning, graphics display was "largely compensated" (in Dan's words). This was mainly because the ALTO did not have enough microcode memory to take in all of the Smalltalk emulation code—some of it had to be rendered in emulated "NOVA" code which forced two layers of interpretation. In fact, the *NoteTaker* worked extremely well, though it would have crushed any lap. It had hopped back on the desk, and looked suspiciously like miniCOM (and several computers that would appear a few years later). It really did run on batteries and several of us had the pleasure of taking *NoteTaker* on a plane and running an object-oriented system with a windowed interface at 35,000 feet.

We eventually built about 10 of the machines, and though in many senses an engineering success, what had to be done to make them had once again squeezed out the real end-users for whom it was originally aimed. If Xerox (and PARC) as a whole had believed in these smaller scale ideas, we could have put much more silicon muscle behind the dreams and successfully built them in the '70s when

FIGURE 11.64 What Steve Jobs saw. Multiviews on complex structures by Trygve Reenskaug

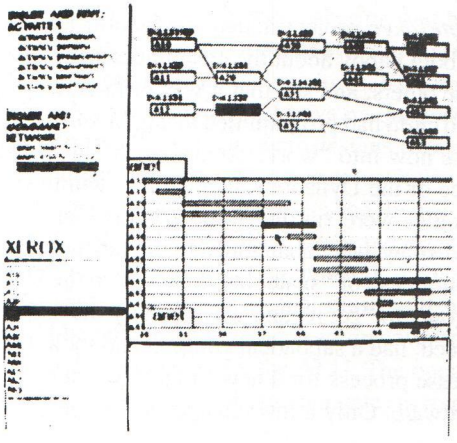


FIGURE 11.65 Multimedia documents by Bob Flegal and Diana Merry

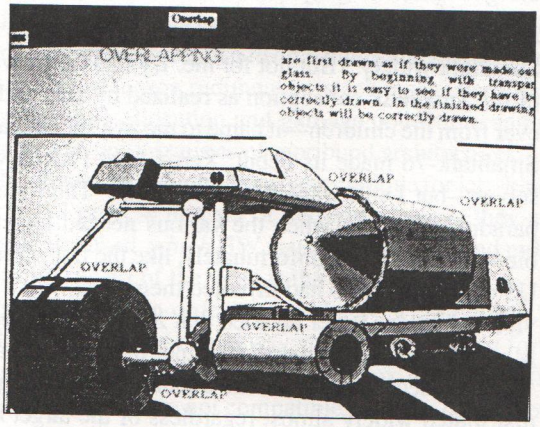


FIGURE 11.66 Dave Robson

they were first possible. It was a bitter disappointment to have to get the wrong kind of CPU from Intel and the wrong kind of display from HP because there was not enough corporate will to take advantage of internal technological expertise.

By now it was already 1979, and we found ourselves doing one of our many demos, but this time for a very interested audience: Steve Jobs, Jeff Raskin, and other technical people from Apple. They had started a project called *Lisa* but were not quite sure what it should be like, until Jeff said to Steve, "You should really come over to PARC and see what they are doing." Thus, more than eight years after overlapping windows had been invented and more than six years after the ALTO started running, the people who could really do something about the ideas finally got to see them. The machine used was the Dorado, a very fast "big brother" of the ALTO, whose Smalltalk microcode had been largely written by Bruce Horn, one of our original "Smalltalk kids" who was still only a teenager. Larry Tesler gave the main part of the demo with Dan sitting in the copilot's chair and Adele and I watched from the rear. One of the best parts of the demo was when Steve Jobs said he did not like the blt-style scrolling we were using and asked if we could do it in a smooth continuous style. In less than a minute Dan found the methods involved, made the (relatively major) changes, and scrolling was now continuous! This shocked the visitors, especially the programmers among them, as they had never seen a really powerful incremental system before.

Steve tried to get and/or buy the technology from Xerox (which was one of Apple's minority venture capitalists), but Xerox would neither part with it nor come up with the resources to continue to develop it in-house by funding a better *NoteTaker* cum Smalltalk.

"The greatest sin in Art is not Boredom, as is commonly supposed, but lack of Proportion"
—Paul Hindemith

11.6 1980–1983—THE RELEASE VERSION OF SMALLTALK (-80)

As Dan said, "The decision not to continue the *NoteTaker* project added motivation to release Smalltalk widely." But not for me. By this time I was both happy about the cleanliness and elegance of the Smalltalk conception as realized by Dan and the others, and sad that it was farther away than ever from the children—it came to me as a shock that no child had programmed in any Smalltalk since Smalltalk-76 made its debut. Xerox (and PARC) were now into "workstations" as things in themselves—but I still wanted "playstations." The romance of the Dynabook seemed less within grasp, paradoxically, just when the various needed technologies were starting to be commercially feasible—some of them, unfortunately, like the flat-screen display, were abandoned to the Japanese by the US companies who had invented them. This was a major case of "snatching defeat from the jaws of victory." Larry Tesler decided that Xerox was never going to "get it" and was hired by Steve Jobs in May 1980 to be a principal designer of the *Lisa*. I agreed, had a sabbatical coming, and took it.

Adele decided to drive the documentation and release process for a new Smalltalk that could be distributed widely almost regardless of the target hardware. Only a few changes had to be made to

the *NoteTaker* Smalltalk-78 to make a releasable system. Perhaps the change that was most ironic was to turn the custom fonts that made Smalltalk more readable (and were a hallmark of the entire PARC culture) back into standard pedestrian ASCII characters. According to Peter Deutsch this "met with heated opposition within the group at the time, but has turned out to be essential for the acceptance of the system in the world." Another change was to make blocks more like lambda expressions; as Peter Deutsch was to observe nine years later: "In retrospect, this proliferation of different kinds of instantiation and scoping was probably a bad idea." The most puzzling idea—at least to me as a new outsider—was the introduction of metaclasses (really just to make instance initialization a little easier—a very minor improvement over what Smalltalk-76 did quite reasonably already). Peter's 1989 comment is typical and true: "Metaclasses have proven confusing to many users, and perhaps in the balance more confusing than valuable." In fact, in their PIE system, Goldstein and Bobrow had already implemented in Smalltalk an "observer language," somewhat following the view-oriented approach I had been advocating and in some ways like the "perspectives" proposed in KRL [Goldstein]. Once one can view an instance via multiple perspectives, even "semi-metaclasses" such as Class Class and Class Object are not really necessary because the object-role and instance-of-a-class-role are just different views and it is easy to deal with life-history issues including instantiation. This was there for the taking (along with quite a few other good ideas), but it was not adopted. My guess is that Smalltalk had moved into the final phase I mentioned at the beginning of this story, in which a way of doing things finally gets canonized into an inflexible belief structure.

11.6.1 Coda

One final comment. Hardware is really just software crystallized early. It is there to make program schemes run as efficiently as possible. But far too often the hardware has been presented as a given and it is up to software designers to make it appear reasonable. This has caused low-level techniques and excessive optimization to hold back progress in program design. As Bob Barton used to say: "Systems programmers are high priests of a low cult."

One way to think about progress in software is that a lot of it has been about finding ways to *late-bind*, then waging campaigns to convince manufacturers to build the ideas into hardware. Early hardware had wired programs and parameters; random access memory was a scheme to late-bind them. Looping and indexing used to be done by address modification in storage; index registers were a way to late-bind. Over the years software designers have found ways to late-bind the locations of computations—this led to base/bounds registers, segment relocation, paging MMUs, migratory processes, and so forth. Time-sharing was held back for years because it was "inefficient"—but the manufacturers would not put MMU's on the machines; universities had to do it themselves! Recursion late-binds parameters to procedures, but it took years to get even rudimentary stack mechanisms into CPUs. Most machines still have no support for dynamic allocation and garbage collection, and so forth. In short, most hardware designs today are just re-optimizations of moribund architectures.

From the late-binding perspective, OOP can be viewed as a comprehensive technique for late-binding as many things as possible: the *mix* of state and process in a set of behaviors, *where* they are located, *what* they are called, *when* and *why* they are invoked, *which* HW is used, and so on, and more subtle, the strategies used in the OOP scheme itself. The art of the wrap is the art of the trap.

Consider the two cases that must be handled efficiently in order to completely wrap objects. It would be terrible if $a+b$ incurred *any* overhead if a and b were bound, say, to "3" and "4" in a form that could be handled by the ALU. The operation should occur full speed using look-aside logic (in the simplest scheme a single *and* gate) to trap if the operands are not compatible with the ALU. Now

all elementary operations that have to happen fast have been wrapped without slowing down the machine.

The second case happens if the trap has determined the objects in question are too complicated for the ALU. Now the HW has to dynamically find a method that can handle the objects. This is very similar to indexing—the class of one of the objects is "indexed" by the desired method-selector in a slightly more general way. In other words, the VIRTUAL-ADDRESS of a method is <class><selector>. Because most HW today does a virtual address translation of some kind to find the real address—a trap—it is quite possible to hide the overhead of the OOP dispatch in the MMU overhead that has already been rationalized.

Again, the whole point of OOP is *not* to have to worry about what is *inside* an object. Objects made on different machines and with different languages *should* be able to talk to each other—and will *have to* in the future. Late-binding here involves trapping incompatibilities into *re*compatibility methods—a good discussion of some of the issues is found in [Popek 1984].

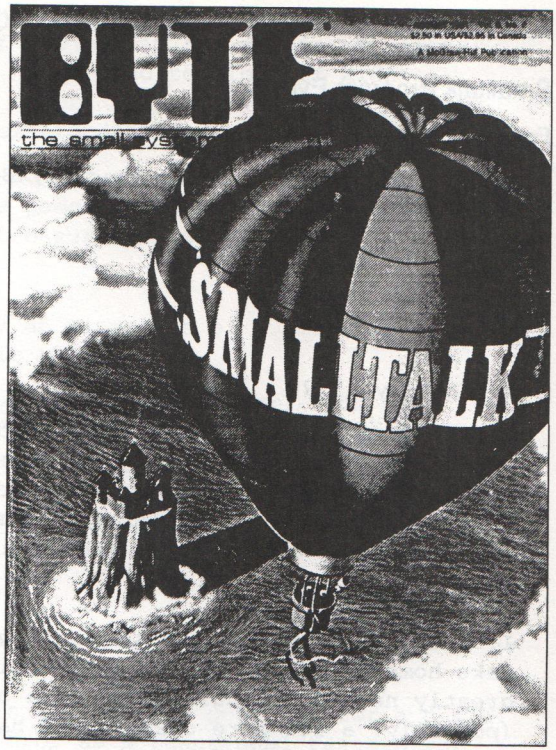
Staying with the metaphor of late-binding, what further late-binding schemes might we expect to see? One of the nicest late-binding schemes that is being experimented with is the *metaobject protocol* work at Xerox PARC [Kiczales 1991]. The notion is that the language designer's choice for the internal representation of instances, variables, and the like, may not cover what the implementer needs. So within a *fixed* semantics they allow the implementer to give the system strategies—for example, using a hashed lookup for slots in an instance instead of direct indexing. These are then efficiently compiled and extend the base implementation of the system. This is a direct descendant of similar directions from the past of Simula, FLEX, CDL, Smalltalk, and Actors.

Another late-binding scheme that is already necessary is to get away from direct protocol matching when a new object shows up in a system of objects. In other words, if someone sends you an object from halfway around the world it will be unusual if it conforms to your local protocols. At some point it will be easier to have it carry even more information about itself—enough so its specifications can be "understood" and its configuration into your mix done by the more subtle matching of *inference*.

A look beyond OOP as we know it today can also be done by thinking about late-binding. Prolog's great idea is that it does not need bindings to values in order to carry out computations. The variable is an object, and a web of partial results can be built to be filled in when a binding is finally found. Eurisko constructs its methods—and modifies its basic strategies—as it tries to solve a problem. Instead of a problem looking for methods, the methods look for problems—and Eurisko looks for the methods of the methods. This has been called "opportunistic programming"—I think of it as a drive for more enlightenment, in which problems get resolved as part of the process.

This higher computational finesse will be needed as the next paradigm shift—that of pervasive networking—takes place over the next five years. Objects will gradually become active agents and will travel the networks in search of useful information and tools for their managers. Objects brought back into a computational environment from halfway around the world will not be able to confugure themselves by direct protocol matching as do objects today. Instead, the objects will carry much more information about themselves in a form that permits *inferential* docking. Some of the ongoing work in specification can be turned to this task.

Tongue in cheek, I once characterized progress in programming languages as a kind of "sunspot" theory, in which major advances took place about every 11 years. We started with machine code in 1950, then in 1956 FORTRAN came along as a "better old thing" which, if looked at as "almost a new thing," became the precursor of ALGOL-60 in 1961. In 1966, SIMULA was the "better old thing," which if looked at as "almost a new thing" became the precursor of Smalltalk in 1972.



Everything seemed set up to confirm the "theory" once more: In 1978, Eurisko was in place as the "better old thing" that was "almost a new thing." But 1983—and the whole decade—came and went without the "new thing." Of course, such a theory is silly anyway—and yet, I think the enormous commercialization of personal computing has smothered much of the kind of work that used to go on in universities and research labs, by sucking the talented kids towards practical applications. With companies so risk-adverse towards doing their own HW, and the HW companies betraying no real understanding of sw, the result has been a great step backwards in most repects.

A twentieth century problem is that technology has become too "easy." When it was hard to do *anything*, whether good or bad, enough time was taken so that the result was usually good. Now we can make things almost trivially, especially in software, but most of the designs are trivial as well. This is inverse vandalism: the making of things because you can. Couple this to even less sophisticated buyers and you have generated an exploitation marketplace similar to that set up for teenagers. A counter to this is to generate enormous dissatisfaction with one's designs, using the entire history of human art as a standard and goad. Then the trick is to decouple the dissatisfaction from self-worth—otherwise it is either too depressing or one stops too soon with trivial results.

I will leave the story of early Smalltalk in 1981 when an extensive series of articles on Smalltalk-80 was published in *Byte* magazine [Byte 1981], followed by Adele's and Dave Robson's books [Goldberg 1983] and the official release of the system in 1983. Now programmers could easily implement the virtual machine without having to reinvent it, and, in several cases, groups were able to roll their own *image* of basic classes. In spite of having to run almost everywhere on moribund hw architectures, Smalltalk has proliferated amazingly well (in part because of tremendous optimization efforts on these machines) [Deutsch 1989]. As far as I can tell, it still seems to be the most widely used system that claims to be object-oriented. It is incredible to me that no one since has come up with a qualitatively better idea that is as simple, elegant, easy to program, practical, and comprehensive. (It is a pity that we did not know about PROLOG then or vice versa; the combinations of the two languages done subsequently are quite intriguing.)

While justly applauding Dan, Adele, and the others who made Smalltalk possible, we must wonder at the same time: where are the Dans and Adeles of the '80s and '90s who will take us to the next stage?

APPENDIX I: PERSONAL COMPUTER MEMO

Smalltalk Program Evolution

From a Memo on the "KiddiKomputer":

To: Butler Lampson, Chuck Thacker, Bill English, Jerry Elkind, George Pake

Subject: "KiddiKomputer"

Date: May 15, 1972

****

4. January 1972

The Reading Machine. Another attempt to work on the actual problem of a personal computer. Every part of this gadget (except display) is buildable now but requires some custom chip design and fabrication. This is discussed more completely later on. A meeting was held with all three labs to try to stimulate invention of the display.

B. Utility

1. I think the uses for a personal gadget as an editor, reader, take-home-context, intelligent terminal, etc. are fairly obvious and greatly needed by adults. The idea of having kids use it implies (possibly) a few more constraints having to do with size, weight, cost, and capacity. I have been begging this question under the assumptions that a size and weight that are good for kids will be super acceptable to adults, and that the gadget will almost inescapably have CPU power to burn (more than PDP-10): implies larger scale use by adults can be gotten by buying more memory and maybe a cache.

2. Although there are many "educational" things that can be done once the device is built, I have had four basic projects in mind from the start.

a. Teaching "thinking" (à la Papert) through giving the kids a franchise for the strategies, tactics, and model visualization that are the fun (and important) part of the design and debugging of programs. Fringe benefits include usage as a medium for symbols allowing editing of text and pictures.

b. Teaching "models" through "simulation" of systems with similar semantics and different syntax. This could be grouped with (a) although the emphasis is a bit different. The initial two systems would be music and programming and would be an extension of some stuff I did at Utah in 1969-1970 with the organ/computer there.

c. Teaching "interface" skills such as "seeing" and "hearing." The initial "seeing" project would be an investigation into how reading might be taught via combining iconic and audible representation of works in a manner reminiscent of Bloomfield and Moore. This would

require a corollary inquiry into why good readers do so much better than average readers. A farther off project in the domain of sight would be an investigation into the nature and topology of kids' internal models for objects and an effort to perserve iconic imagery from being totally replaced by a relational model.

d. Finding out what children would do (if anything) "unofficially" during non-school hours with such a gadget through invisible 'demons", which are little processes that watch surreptitiously.

3. Second Level Projects

a. The notion of evaluation (partly an extension of 2.a.) represents an important plateau in "algorithmic thinking."

b. Iconic programming. If we believe Piaget and Bruner, kids deal mostly with icons before the age of 8 rather than symbolic references. Most people who teach programming say there is a remarkable difference between 3rd and 4th grades. Whatever an iconic programming language is, it had better be considerably more stylish and viable than GRAIL and AMBIT/G. I feel that this is a way to reach very young kids and is tremendously important.

****

C. The Viability Of miniCOM

It was noted earlier that miniCOM is only barely portable for a child. Does it have a future for adults and/or as a functional testbed for kids? If only one is needed, the answer seems to be no since ~$15k will simulate its function in a non-portable fashion. If more than one is necessary (say 10 or more), then the cheapest way to get functions of this kind is to design and build it.

Rationalizations for building a bunch of them:

1. It will allow us to find out some things not predictable or discoverable by any other path.

A perfect case in point is our character generator through which we have found some absolutely astounding and unsuspected things about human perception and raster scan television which will greatly further display design. It has paid its way already.

2. The learning experiments not involving portability can be done for a reasonable cost and will allow us to get into the real world which is absolutely necessary for the future of learning research at PARC.

3. It will foster some new thoughts in small computer system design.

It has already sparked the original "jaggies" investigation. The minimal nice serifed character fonts were done because of cost and space limitations. There are some details which have been handwaved into the woodwork which really neeed to be solved seriously: philosophy of instruction set, compile or interpret, mapping, and I/O control.

4. It will be a useful "take home" editor and terminal for PARC people. It is absurd to think of using a multidimensional medium during the day (NLS, etc.), then at night going home to a 1D AJ or worse: dumping structured ideas on paper.

5. It is not unreasonable to think of the gadget as an attempt at a cost-effective node for a future office system. As such, it should be developed in parallel with the more exotic and greatly more expensive luxury system.

6. It is not clear that the more ideal device (A.4.), requiring custom chip design, can be done well without us knowing quite a bit more about this kind of system.

APPENDIX II: SMALLTALK INTERPRETER DESIGN

When I set out to win the bet, I realized that many of the details that have to be stated explicitly in McCarthy's elegant scheme can be *finessed*. For example, if there were objects that could handle various kinds of partial message receipt, such as *evaluated*, *unevaluated*, *literal*, and the like, then there would be no need to put any of those details in the *eval*. This is analogous to not having cond as a "special form," but instead to finding a basic building block in which COND can be defined like any other subpart.

One way to do this was to use the approach of Dave Fisher, in which the no-man's land of control structures is made accessable by providing a protected way to access and change the relationships of the static and dynamic environment [Fisher 1970]. In an object-based scheme, the protection can be provided by the objects themselves and many of Fisher's techniques are even easier to use. The effect of all this is to extend the *eval* by *distributing* it: both to the individual objects that participate in it and dynamically as the language is extended.

I also decided to ignore the metaphysics of objects even though it was clear that, unlike Simula, in a full blown OOPL classes had to exist at run-time as "first-class" objects—indeed, there should be nothing but first-class objects. So there had to be a "class-class" whose instances were classes, class-class had to be an instance of itself, there had to be a "class-object" that would terminate any subclassing that might be done, and so forth. All of this could be part of the argument concerning what I *didn't* have to show to win the bet.

The biggest problem remaining was that I wanted to have a much nicer syntax than LISP and I did not want to use any of my precious "half-page" to write even a simple translator. Somehow the *eval* had to be designed so that syntax got specified as part of the use of the system, not in its basic definition.

I wanted the interpretation to go from left to right. In an OOP, we can choose to interpret the syntax rule for expressions as meaning: the first element will be evaluated into the instance that will receive the message, and *everything* that follows will be the message. What should expressions like *a+b* and c_i <– *de* mean? From past experience with FLEX, the second of these had a clear rendering in object-oriented terms. The *c* should be bound to an object, and *all* of $_i$ <– *de* would be thought of as the message to it. Subscripting and multiplication are implicit in standard mathematical orthography—we need explicit symbols, say "°" and "*". This gives us:

receiver	message
c	° *i* <– *d***e*

The message is made up of a literal token "°", an expression to be evaluated in the sender's context (in this case *i*), another literal token <–, followed by an expression to be evaluated in the sender's context (*d***e*). "LISP" pairs are made from two element objects and can be indexed more simply: *c hd*, *c tl*, and *c hd* <– *foo*, and so forth.

The expression *3+4* seemed more troublesome at first. Did it really make sense to think of it as:

receiver	message
3	*+4*

We are so used to thinking of "+" and "*" as operators, function machines. On the other hand, there are many senses of "+" and "*" that go beyond simple APLish generalizations of scalar operators to arrays—for example in matrix and string algebras. From this standpoint it makes great sense to let the objects in question decide what the token "+" means in a particular context. This means that *3+4*5...* should be thought of as *3|+4*5...*, and that the way class number chooses to receive messages should be arranged so that the next subexpression is handled properly. For example, 3 could check to see if a token (like +, or *) follows and then ask to have the rest of the message evaluated to get its next input. This would force *4*5...* to be the new sending , as *4|*5*, and so on. Not only are fewer parentheses needed but proglike sequential evaluation is a byproduct.

By this point I had been able to finesse and argue away most of the programming that seemed to be required of the *eval*. To summarize:

- message receipt would be done by objects in the midst of normal code
- control structures would be handled by objects that could addess the context objects
- the context objects (that acted like stack frames, schedulers, and so on) could be simulated by standard objects and thus wouldn't have to be specified in the eval
- variable dereferencing and storage would be done by having variables be objects and sending them the messages *value* and <-.
- the evaluation of a code body would be done by starting evaluation of its first item
- methods would be realized by the control structure in the cl6.alss code body. This would implement protection, would make the externals of an object entirely virtual, and permit very flexible messaging schemes
- Smalltalk's metaphysics would be covered by making everything an object, and didn't have to be specified now
- and so forth

This also means that useful elements such as lists, atoms, control structures, quote, receivers (such as "receive evaluated," "is the next token this?," and so on), and the like do not have to be defined in the kernel interpreter, as they can be realized quite simply as instances of normal classes with escapes to metacode.

What seemed to remain for the *eval* was simply to show of what a message send actually consisted. For this system a send is the equivalent not of a postman delivering a letter, but simply delivering a notice of where the letter was to be found. It is up to the receiving object to do something about it. In fact, it could ignore the request, complain about it, invoke inferential processes elsewhere, or simply handle it with one of its own messages.The final thing I had to do was to extend the uniform syntax idea of *receiver message* to cover all cases, including message receipt and simple control structures. So, we need some objects to pattern match and evaluate, to return and define, and so on.

The "LISP" code body would not need any escapes to lower-level code and could look something like:

```
(¤hd     » (¤<-   (^:h)^h)   "replaca and car where h is an instance variable"
¤tl      » (¤<-   (:t)^t)    "replaca and cdr where t is an instance variable"
¤isPair  » (^true)
¤length  » (t isPair  (^1+t length) 1)
...)                         "etc"
```

I hope this is clear enough. For example, if *c* is bound to a cons pair,

c hd <– 3+4

FIGURE 11A.1 Used in the first interpreter definition

¤	eyeball	looks to see if its message is a literal token in the message stream
:	evl-bind	evals the next part of message and binds result to its message
:	unval-bind	picks up next part of message unevaled and binds to its message
^	send-back	returns its value to the sender
'	quote	overides any metainterpretation of its message

would be dealt with as follows: Control is passed to that object and the first test is to see if the symbol *hd* appears in the message (¤*hd* »). It does. The next check is for an "assignment" token (¤<– »). It is there. Last, we want to evaluate the rest of the message (we get 7), bind the value to the internal instance variable t and, finally, return this value to the sender (^*:t*). So this is like: (REPLACA C (PLUS 3 4)).

This is getting a little ahead of the story in that not all of these ideas were thought out in this detail, but I want to show the context in which I was thinking, and it seemed quite clear at the time that things would come out all right if I pushed in this direction. This stuff is similar to mathematical or musical thinking where many things can be done "ahead of time" if one's intuition whispers that "you're on the right track." The compass setting felt right; I could "see" that all these things would eventually work out just because of "what objects were."

To motivate the next part, let us examine the classic evaluation of *3+4* using a *nonrecursive* evaluator. For code, we use *arrays* of pointers and expect that some of the pointers will be encoded for literal objects (an old LISP trick). We need good old program counters "PC" that we can bump along over the code. The wrinkle of delayed receipt of message (not evaling and passing arguments at *send* time) will require us to manipulate *both* the program counter of the sender and the receiver as the message is reeled in. One way I worked it out was as a before-after diagram for "3+4".

We start in the middle of a method of some class of objects and we need to evaluate "3+4". The essentials of the *eval* are those that successfully take us into the method of "3" in class integer. Because all methods are only in terms of sends and all sends are done in a similar manner, this is enough. It is like an induction proof in which we assume "n" and show how to get to "n+1".

Note that the various auxiliary objects (such as 'peek') have to responsibly move the sender's program counter when receiving part of the message.

I have hand-evaluated this nonrecursive version in a number of cases and it seems to work pretty well, but there are probably some bugs. If a reader feels prompted to come up with an even nicer, tidier, and smaller scheme, I would be glad to look at it.

FIGURE 11A.2 Defined when the first "real" implementation was done

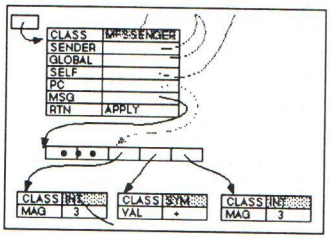
to	define	likeLOGO, except can make a class from it message
ISNEW	testinst	is true if a new instance has been created
=	equals	true only if its receiver and parameter are the same object
»	then	receiver=true: evals next part of message and exits receiver=false: skips over the next part of message and continues evaling
.	fence	"statement" separator. Quits applying its receiver; starts evaling its arg

FIGURE 11A.3 The One Pager

e (the environment) will be bound to the current Messenger object

result holds the result of a send, usually to be *applied* to next part of message

"Before"




```
eval: if null(•MSG)      then 'reuslt <- nil; goto apply;
      if escale(e•MSG)   then goto escapes;
      if atom(•MSG)      the 'result <- lookupvalue(e,e•MSG); goto apply;
      if notlist(e•MSG) the 'result <- e•MSG; goto apply;
evlist: 'e <- Table(CLASS, MESSENGER,
                    SENDER, e,
                    GLOBAL, e•GLOBAL,
                    SELF,   e•SELF,
                    PC,     1,
                    MSG,    e•MSG•PC)
                    RTN,    APPLY);
              goto eval;

apply: 'e <- e•SENDER;
       e•PC <- e•PC + 1
       if e•PC > length(MSG) then goto dispathrtn;
       if e•MSG•PC = '.      then e•PC <- e•PC + 1; goto evlist;
       if e•MSG•PC = '»      then if result = 'false
                                  then e•PC <- e•PC + 2; goto evlist;
                                  else e•PC <- e•PC + 1;
                                       'e <- Table(CLASS   MESSENGER,
                                                   SENDER, e,
                                                   GLOBAL, GLOBAL,
                                                   SELF    result,
                                                   PC,     t,
                                                   MSG,    e•MSG•PC,
                                                   RTN,    FROMTTRUE);
                                       goto eval;
      'e <- Table( CLASS,  MESSENGER,
                   SENDER, e,
                   GLOBAL, GLOBAL,
                   SELF,   reuslt,
                   PC,     1,
                   MSG,    reuslt•CLASS•CODE,
                   RTN,    APPLY);

fromTURE: 'e <- e•SENDER•SENDER; goto dispatchrtn;

fromEYE: putvalue(E•GLOBAL, e•p, reuslt); goto apply;

dispatchrtn: select e•RTN
                  case APPLY:    goto apply;
                  case FROMTRUE: goto fromTRUE;
                  case FROMEYE:  goto fromEYE;
escapes: select e•MSG•PC+1
                  etc...
```

"During"

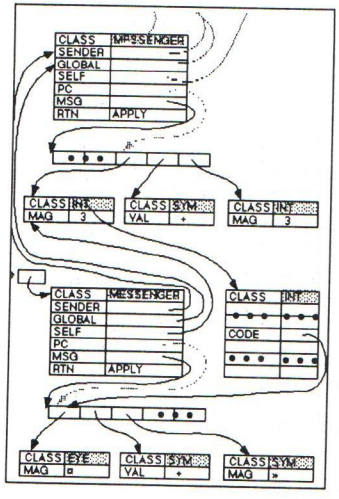
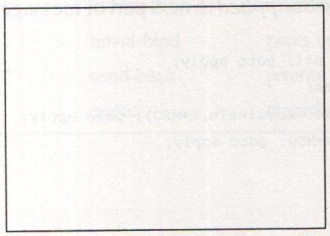


FIGURE 11A.3 The "One Pager" *(cont.)*

"After"



```
to ¤ (metacodefor(if e•SNDR•MSG(PC)=e•SNDR•SNDR•MSG(PC)
                        then bump(e•SNDR•SNDR•PC);result <- TRUE
                        else result <- FALSE;
                        goto apply))
to : p (: p. metacodefor(set up a new context and eval sender))
to : p v (metacodefor('v <- e•SNDR•SNDR•MSG•PC;
                        if nil(e•'p <- e•SNDR•MSG•PC)
                             then result <- v
                              else e•p <- result <- v;
                        goto apply;))
to ^ b (: b. metacodefor('return <- e•b; goto apply))
```

APPENDIX III: ACKNOWLEDGMENTS

1971

Chris Jeffers, + ?

1972

Chris Jeffers, John Shoch, Steve Purcell, Bob Shur, Bonny Tennenbaum, Barbara Deutsch

1973

A document written by me shortly after Smalltalk-72 started working

ACKNOWLEDGMENTS

Latest revision: March 23, 1973

Much of the philosophy on which our work is based was inspired by the ideas of Seymour Papert and his group at MIT.

The Dynabook (ka 71) is a godchild of Wes Clark's LINC (cl 1962) and a lineal descendent of the FLEX machine (ka 67, 68, 69).

The "interim Dynabook" (known as the ALTO (Th 71, Mc 71) is the beautiful creation of Chuck Thacker and Ed McCreight of the Computer Science Lab. at PARC.

SMALLTALK is basically a synthesis of wellknown ideas for programming languages and machines which have appeared in the last 15 years.

The Burroughs B5000 (ba 61) (B60) had many design ideas well in advance of its time (and still not generally appreciated): compact "addressless code; a uniform semantics for names (the PRT), automatic coprocesses, "capability" protection (also by the PRT and Descriptors_, virtual segmented memory, the ability to call a subroutine from "either side" of the assignment arrow, etc.

The notions of code as a data structure; intensional properties of names (property lists of attribute-value pairs on atoms); evaluation

with respect to arbitrary environments; etc., are found in LISP, probably the greatest single design for a programming language yet to appear. SMALLTALK is definitely "LISPlike".

The SIMULAs ('65 and '67) combined Conway's notions of software coroutines (1963—hardware version had appeared in the B5000 3 years earlier), ALGOL-60, and Hoare's ideas about record classes (ca.1964) into an epistemology that allowed a class to have any number of parallel instantiations (or activation records) containing local state including a separate program counter. Most of the operations for a SIMULA '67 class are held intrinsically as procedures local to the class definition.

The FLEX machine and its language ('67-69) took the SIMULA ideas (discarding most of the AGOLishness), moved "type" from a variable onto the objects (ala B5000 and EULER), formed a total identification between "coprocesses" and "data", consolidating notions such as arrays, files, lists, "subroutine" files (ala SDS-940) etc., into one idea. The "user as a process" also appeared here. A start was made to allow proccesses to determine their own input syntax, an idea held by many (notably Irons, Leavenworth, etc.)

The Control Definition Language of Dave Fisher (1970) provided many ideas, solutions and approaches to the notion of control. It, with FLEX, is the major source for the semantics of SMALLTALK. It is a "soulmate" to FLEX; independently worrying about many of the same problems and very frequently arriving at cleaner, neater ways to do things. Many of Dave;s ideas are used including the provision for many orthogonal paths to external environments, and that control is basically a matter of organizing these environments. SMALLTALK removes Fisher's need for a compiler to provide a mapping between nice syntax and semantics and offers other improvements over his schemes such as total local control of the format of an instance, etc.

An extemporaneous talk by R.S. Barton at Alta ski lodge (1968) about computers as communications devices and how everything one does can easily be portrayed as sending messages to and fro, was the real genesis of the current version of SMALLTALK.

The fact that kids were to be the users, and the simplicity and ease of use of the already existing LOGO, whose own parents were LISP and JOSS (which set a standard for the esthetics for interaction that has not yet been surpassed), provided lots of motivation to have programs and transactions appear as simple as possible—i.e. moving from left to right, procedures gather their own messages, etc. It is no accident that simple SMALLTALK programs look a bit like LOGO!

Problems discovered years ago in "lefthand calls" prompted SMALLTALK to make "store" intensional—i.e. a <- b, means "call 'a' with a message consisting of the token'<-' and symbol 'b'. If anyone can make the right decision for what this means, it must be the object bound to 'a'. The early fall of 1972 saw an evaluator for SMALLTALK, and the idea that '+', '-', etc., should also be intensional. This led to an entire philosophy of use (unlike SIMULA '67) to put EVERYTHING in

class definitions including the so-called "infix operators". This message idea allows messages to have a wide range of form since all messages can be received incrementally.

"Control of control" allows control structures to be defined, The language SMALLTALK itself thus avoids "primitives" such as "loop...pool", synchronous and asynchronous "ports", interrupts, backtracking, parallel evel and return, etc. All of these can be easily simulated when needed.

****

These are the main influences on our language. There were many other minor and negative influences from other existing languages and ideas too numerous to mention except briefly in the references,

****

This particular version of SMALLTALK was designed through the summer and early fall of 1972 and was aided by discussions with Steve Purcell, Dan Ingalls, Henry Fuchs, Ted Kaehler, and John Schoch. From the proceeding acknowledgements it can be seen as a consolidation of good ideas into one simple ides:

Make the PARTS (object, subroutines, I/O, etc.) have the same properties and power as the WHOLE (such as a computer).

This is the basic principle of recursive design, SMALLTALK recurs on the notion of "computer" rather than of "subroutine."

A talk on SMALLTALK was given at the AI lab at MIT (Nov 1972) which discussed the process structure and the new, intentional way to look at properties, messages, and "infix operators". This led to the just published formal "actors model of computation" of Hewitt, *et. al.* (1973).

****

Dan Ingalls of our group at PARC, the implementor of SMALLTALK, has revealed many design flaws through his several, excellent quick "throw away" implementation of the language. SMALLTALK could not have existed without his help, virtuosity, and good cheer.

The original design of the "painting editor" was by Alan Kay. It was implented and tremendously improved by Steve Purcell.

The "Animator" was designed and implemented by Bob Shur and Steve Purcell.

Line graphics and the hand-character recognizer were done by John Shoch.

"Music:" was designed and implemented by Alan Kay.

The design and implementation of the font editor was by Ben Laws (POLOS).

We would like to thank CSL and POLOS in general for a great deal of all kinds of help.

1976

Learning Research Group

Alan Kay, Head, Adele Goldberg, Dan Ingalls, Chris Jeffers, Ted Kaehler, Diana Merry, Dave Robson, John Shoch, Dick Shoup, Steve Weyer

Students

Barbara Deutsch, Tom Horsley, Steve Purcell, Steve Saunders, Bob Shur, David C. Smith, Radia Perlman

Child Interns

Dennis Burke (age 12), Marian Goldeen (age 13), Susan Hammet (age 12), Bruce Horn (age 15), Lisa Jack (age 12), Kathy Mansfield (age 12), Steve Putz (age 15)

Visitors

Ron Baecker, Eric Martin, Bonnie Tenenbaum

Help from Other Groups at PARC

Patrick Baudelaire, Dave Boggs, Bill Bowman, Larry Clark, Jim Cucinitti, Peter Deutsch, Bill English, Bob Flegal, Ralph Kimball, Butler Lampson, Bob Metcalfe, Mike Overton, Alvy Ray Smith, Bob Sproull, Larry Tesler, Chuck Thacker, Truett Thach

APPENDIX IV: EVENT DRIVEN LOOP EXAMPLE

First we make a class for events:

```
to event | mycode

(ISNEW    »  ('mycode <- array 3.
              mycode[2] <- 'done.)
¤newcode »  (mycode[1] <- :.)
¤is       »  (ISIT eval)
mycode eval)
```

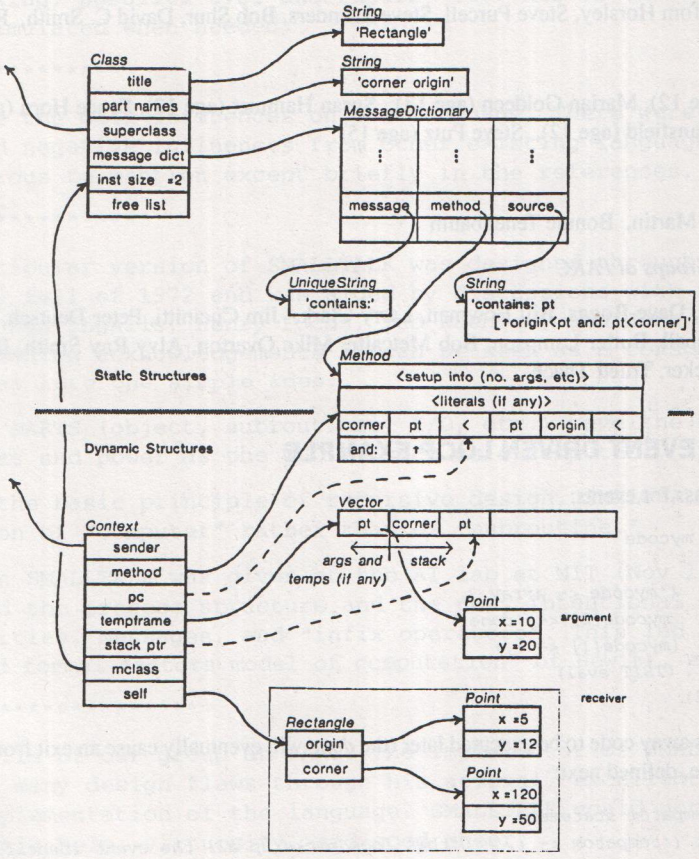
Each event stores away code to be executed later (the *done* will eventually cause an exit from the driving loop in the *until* structure, defined next:

```
to until tempatom statement
    (repeat ('tempatom <- :.    "this loop picks up all the event identifiers
(unevaled)"
             tempatom <- event."an indirect store to whatever was in the message"
             ¤or » (again) done)
   (¤do    »  ('statement <- :))       "the loop body to be evaled"
   (¤case  »  (repeat ('tempatom <- :. "pick up an event-case label"
                 (tempatom eval is event »
                   (¤:. tempatom eval newcode :.)"pick up the corresponding code"
                 done)))
  repeat (statement eval)) "execute body until an event is encountered and run"
                    "the event will then force exit from the until loop"
```

This kind of playing around was part of the general euphoria that came with having a really extensible language. It is like the festooning of type faces that happens when many fonts are suddenly available. We had both, and our early experimentation sometimes got pretty baroque. Eventually we calmed down and started to focus on fewer, simpler structures of higher power.

APPENDIX V: SMALLTALK-76 INTERNAL STRUCTURES

This shows how Smalltalk-76 was implemented. In the center, between "static" and "dynamic" lies a byte-compiled method of Class Rectangle. Slightly above it is the source text string written by the programmer. The method tests to see whether a point is contained in the rectangle. In the dynamic part, the program counter is just starting to execute the first less-than. This general scheme goes all the way back to the B5000 and the FLEX machine, but is considerably more refined.



APPENDIX VI: SMALLTALK INTERPRETER DESIGN DOCUMENTATION

General Landscape of the Sixties

The almost-OOP idea started early, as designers found that the straightforward material-and-recipe metaphors of data structures and procedures broke down all too readily under stress.

1962–1964	representation independence (wrapping)	USAF ATG file system	unknown
1960–1964	protection, HLL Arch, many firsts	B5000	Barton
1962–1963	graphics, simulation, classes, instances	Sketchpad	Sutherland
1962–1963	coroutines	B220 COBOL Comp.	Conway

1964–1967	general simulation: proc classes & instances	Simula	Nygaard, Dahl
1967–1969	subsuming programming constructs	FLEX	Kay
1967–*	inheritance	Simula-67	Nygaard, Dahl
1967	"dataless programming"	??	Balzer
1969–1971	capability operating system	CAL-TSS	Lampson-Sturgis

What these designers had in common—and what set them apart from others in the field—was an interest in finding simple general structures to *finesse* difficulties, rather than expending enormous efforts to produce large amounts of code.

In addition to these initial gropings towards the OOP idea, several other early nonOOP systems contributed important ideas and stylistic directions.

1959–1963	kernel elegance, self-describing, etc.	LISP	McCarthy
1959–1966	ease of end-user authoring	JOSS	Shaw
1961–1966	generalizations of PLs	GA, EULER, APL, etc.	vanWijn..., Wirth
1967–1969	pattern-based inferencing	PLANNER	Hewitt
1968–1970	definition of control structures	CDL	Fisher

The other major set of cross-currents at work in the sixties was Licklider's dream of man-machine symbiosis. This led to the formation of the ARPA IPT office that in the sixties funded most of the advanced computer research in the US. Perhaps as important is that these visions were also adopted by other funders such as ONR and NIH. Shortly there appeared on the scene time-sharing, computer graphics, advanced networking, and many other new ways to think about the technology. In addition to several of the above listings (for example, Sketchpad, JOSS), specific contributions to the invention of personal computing were:

1962	first true personal computer	LINC	Clark
1962–1967?	metaphor of personal computing	NLS	Englebart
1964–1966	modern operating system	GENIE	Lampson
1964–	programming for children	LOGO	Feurzig, Papert
1967–1969	first OO desktop PC	FLEX machine	Cheadle-Kay
196?–?	conceptions of end-user interaction	ARC-MACH	Negroponte
1967–1969	gesture/modeless GUI, iconic prog.	GRAIL	Ellis, Groner, *et al.*
1968	early iconic programming	AMBIT-G	Rovner

Finally, there was a considerable background radiation of ideas generated by the enormous productivity of the sixties. It was still possible back then to be aware of just about everything that was going on and most ideas contributed, even if only to help avoid unpromising paths. For example:

1960–1969	syntax dir. proc., extensible lang., etc.	FSL, META II, IMP, EL1	Feldman, Shorre, Irons, Wegbreit
1960–1969	formal models of prog.	CPL, ISWIM, GEDANKEN	Strachey, Landin, Reynolds
1963–1969	postALGOL, DS definitions	ALGOL-W, -X,-68, etc.	Wirth, Hoare, vanWijn..., etc.
1963–1970	AI representation development	GAS, ***, QA4, etc.	Minsky, Evans, Winston, Hewitt

In other words, there was an entire landscape of activity going on even when the filter is restricted to the 10% or so that was actually interesting.

Smalltalk Documentation

After PARC got set up, it was customary to write a report every six months and a two-year plan every September. I have many of these, but will not list them specifically. All the things described in the text of this chapter were documented in the six-month report that followed, which (in our case) always contained lots of screen shots of the latest wonders. I should also mention that there is an extensive collection of slides, 16mm film (most of it shot by me) both in my files and at Xerox, and many hours of video tapes of all aspects of our work.

Nov 1970	KiddiKomp notebook
Spring 1971	First of several SLOGO documents
June 1971	Display Transducers, (A Pendery Paper for PARC Planning Purposes)
Sum 1971	"Brown Lab Book" (lost or mislaid)
Sum 1971	miniCOM documents
Sum 1971	early Smalltalk-71 programs
Sum 1971	first LRG plan
Fall 1971	first LRG report
Fall 1971	**FJCC panel abstract on mainframes as dinosaurs that will be replaced by personal computers**
Win 1972	report on the wonderful world of fonts (lost or mislaid, most material duped in "May 1972")
May 1972	formal miniCOM proposal (per "X"), I have first draft
Sum 1972	Drafts of A Personal Computer For Children Of All Ages, many Smalltalk-71 programs
Aug 1972	**Final Draft of APCFCOAA (Smalltalk-71 programs removed as per D. Bobrow suggestion)** ACM Nat'l Conf. Boston
Sept 1972	LRG plan about "iconic programming," etc. Xerox probably still has. (Jack Goldman files?)
Sept 1972	First Smalltalk-72 "one page" interpreter demonstration
Oct 1972	First Smalltalk-72 programs run (D. Ingalls)
Nov 1972	**A Dynamic Medium For Creative Thought, NCTE Conference** "20 things to do with a Dynabook" This was pretty much what I wanted the group to accomplish (except for music, etc.)
Nov 1972	Presentation to MIT AI Lab of Smalltalk and its interpreter
Nov 1972	Stewart Brand *Rolling Stone* article on PARC—caused blackout of LRG pubs until 1975.
Win 1973	Smalltalk "Bluebook": general documentation, teaching sequences, alternate syntaxes, etc
Win 1973	First of many Smalltalk bootstrap files (D. Ingalls)—so readable, they were used as a manual
1973–1975	Many application documentation notes, written by Ted Kaehler, Diana Merry, etc.
1974	minimouse (Larry Tesler)
??	
1975	**Blackout lifted. Smalltalk-72 Handbook (with Adele), PARC TR-?**
1975	Dynamic Personal Media (with Adele) proposal to NSF for funding longitudinal studies of children
1975	**PYGMALION *** (David Canfield Smith), Ph.D. Thesis, Stanford, also as ???**
1976	**Dynamic Personal Media (with Adele)** PARC TR-?, a fairly complete rendering of work through 1975
1976	First *NoteTaker* documents
1976	*Findit* (with Steve Weyer), presented at "Dulles" learning center
1977	**Smalltalk in the Classroom (with Adele), PARC TRs, report on work with kids**
???	**Marion Goldeen, etc., articles on her experiences (*Creative Computing*?)**
1977	**Personal Dynamic Media (with Adele) *IEEE Computer*, March.**
1977	Smalltalk-76 Listing (Dan Ingalls + Dave Robson, Ted Kaehler, Diana Merry) The first complete running system
1977	Smalltalk-76 Documentation (Larry Tesler) with included applications

1977	More *NoteTaker* documents (with Doug Fairbairn)
1977	**THINGLAB *** (Alan Borning), Ph.D. Thesis, Stanford**
1977	**Microelectronics and the personal computer, *Scientific American*, Sept.**
1978	**Smalltalk-76 (Dan Ingalls), ACM POPL Conference, January**
1979	**TinyTalk (Larry Tesler and Kim McCall) where?**
1979	**Programming Your Own Computer, Science Year '79, *World Book* Computer Infotech State of the Art Report**
1979	**The Programming Language Smalltalk-72 (John Shoch), *****, Paris**
1980	**Infotech State of the Art Report**
1981	**Smalltalk Issue, *Byte* Magazine**
1983	**Goldberg and Robson**

+ ca 1975–1966 ???, 3D masters thesis in ST-72

+ ca 1978–1980 Gould and Finzer, Rehearsal World

+ ca 1978 Ingalls, Horn, etc., Smalltalk-78, Dorado Smalltalk, etc.

+ ca 1979ish Goldstein & Bobrow, PIE documents

+ ca 1980 Steve Weyer Thesis

Historical Views Of PARC and LRG

19**	Ted Nelson Ravings
19**	Tim Rentsch paper
1984	Smalltalk Implementation History, D. Ingalls
198**	Tools for Thought, H. Rheingold, good profile on early ARPA, Engelbart, etc.
198**	IEEE Computer, Tekla Perry, et al., excellent portrait of PARC
1989	Smalltalk, Past, Present, and Future, P. Deutsch
19**	*Fumbling the Future*, disorganized, and inaccurate portrait of PARC

Smalltalk Spinoffs

Hewitt Actors papers

D-LISP (User interface)

MIT LISP machines

Rosetta Smalltalk

Methods and Smalltalk-V

etc.

REFERENCES

[ACM, 1969] ACM SIGPLAN, *Conference on Extensible Languages*, May 1969.

[Arnheim,1969] Arnheim, Rudolf, *Visual Thinking*, Berkeley: University of California Press, 1969.

[Balzer, 1967] Balzer, R.M. Dataless programming. *Proceedings of the FJCC*, July 1967.

[Barton, 1961] Barton, R.S. A new approach to the functional design of a digital computer, in *Proceedings of the WJCC*, May 1961.

[Baecker, 1969] Baecker, Ronald M. Interactive computer-mediated animation, Dept. of Electrical Engineering, Ph.D. thesis, MIT, 1969, Supervisor: Edward L. Glaser.

[Borning, 1979] Borning, Alan. Thinglab—A constraint-oriented simulation laboratory, Xerox Palo Alto Research Center, #SSL-79-3, July 1979.

[Brand, 1972] Brand, Stewart. Fanatic life & symbolic death among the computer bums, *Rolling Stone Magazine*, Dec. 1972.

[Burroughs,1961] Burroughs Corp. The Descriptor—a definition of the B5000 information processing system, Detroit, Michigan, Bulletin No. 5000-20002-P, Feb. 1961.

[Byte, 1981] *Byte Magazine*, Issue on Smalltalk , Christopher Morgan, Ed., Vol. 6, No. 8, Aug. 1981.

[Carnap, 1947] Carnap, Rudolf. *Meaning and Necessity, A Study in Semantics and Modal Logic*, Chicago:University of Chicago Press, 1947.

[Clark, 1962] Clark, Wesley A. The General Purpose Computer in the Life Sciences Laboratory, in *Engineering and the Life Sciences*, NAS-NRC Report, Washington DC, Apr. 1962.

[Clark ,1965] _____, and Molnar, C.E. A Description of the LINC, in *Computers in Biomedical Research*, Vol. 1, Chapter 2, R.W. Stacy and B.D. Waxman, Ed., New York: Academic Press, 1965.

[Conway, 1963] Conway, Melvin E. Design of a separable transition-diagram compiler, in *Communications of the ACM*, Vol. 6, No. 7, July 1963, pp. 396–408.

[Dahl, 1970] Dahl, O.-J., Decomposition and Classification in Programming Languages, in *Linguaggi nella Società e nella Tecnica*. Milan: Edizioni di Comunita, 1972.

[Dahl, 1970] Dahl, O.-J. and Hoare, C. A. R., Hierarchical Program Structures, in Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R., *Structured Programming*, New York: Academic Press, pp. 175–220.

[Davis, 1964] Davis, M. R., and Ellis, T. O. The RAND tablet: A man-machine graphical communication device, report .#RM-4122-ARPA, CA: RAND, 1964.

[Deutsch, 1966] Deutsch, L.P. LISP for the PDP-1, in *The Programming Language LISP; Its Operation and Applications*, Edmund C. Berkeley and Daniel G. Bobrow, Eds., Cambridge, MA: M.I.T. Press, ix, p. 382.

[Deutsch, 1989] _____. The past, present, and future of smalltalk, in *Proceedings of the 3rd European Conference on Object Oriented Programming*, Cambridge University Press, 1989.

[Engelbart, 1968] Engelbart, Douglas C. and English, William K., A research center for augmenting human intellect, in *Proceedings of the FJCC*, Vol. 33, Part one, Dec. 1968, pp. 395–410.

[Farber,1964 Farber, D. J., Griswald, R. E. and Polensky, F. P., *"SNOBOL, a string manipulation language"* JACM 11, 1964, pp. 21–30.

[Feldman, 1977] Feldman, Jerome A. A formal semantics for computer languages and its application in a compiler-compiler, in *Communications of the ACM*, Jan. 1977, pp. 3–9.

[Fisher, 1970] Fisher, David Allen. Control structures for programming languages, Ph.D. thesis, Department of Computer Science, Carnegie Mellon University, 1970.

[Goldberg, 1983] Goldberg, Adele, Kay, Alan C. and Robson, D. *Smalltalk-80: The Language and its Implementation*, Reading, MA: Addison Wesley, 1983.

[Goldstein, 1980] Goldstein, I. and Bobrow, D., PIE Papers.

[Gombrich,1960] Gombrich, E. H. *Art & Illusion: A Study in the Psychology of Pictorial Representation*, New York: Pantheon Books, 1960.

[Groner, 1966] Groner, Gabriel. Real-time recognition of hand printed text, CA: RAND, Report #RM-5016-ARPA, Oct. 1966.

[Hewitt, 1969] Hewitt, Carl E. Planner: A language for manipulating models and proving theorems in a robot, Cambridge: MIT, Project MAC., AI memo #168, 1969.

[Hewitt, 1973] _____, P. Bishop, Greif, I. Smith, B. Matson, T. and Steiger, R. ACTOR induction and meta-evaluation, in *Conference Record of ACM Symposium on Principles of Programming Languages*, 1–3 Oct. 1973, New York: ACM, 1973, pp.153–168.

[Ingalls 1978] Ingalls, D. The Smalltalk-76 Programming System, Design and Implementation, in 5th ACM *Symposium on Principles of Programming Languages*, Tucson, AZ, Jan. 1978

[Irons, 1970] Irons, E. T. Experience with an extensible language, in *Communications of the ACM*, vol. 13, no. 1, Jan. 1970, pp. 31–40.

[Joss,1964] Shaw, J. C. JOSS: *A Designer's View of an Experimental Online Computer System*, CA: RAND, #P-2922, 1964.

[Joss, 1978] _____, JOSS Session, in *History of Programming Languages*, ed. Richard L. Wexelblat, New York: Academic Press, xxiii, Chapter X, 1981. Conference: History of Programming Languages Conference, Los Angeles, CA, 1978.

[Kaehler, 1981] Kaehler, Edwin B. Virtual memory for an object-oriented langauge, *Byte*, Aug. 1981.

[Kay, 1968] Kay, Alan C. FLEX: a flexible extensible language, M.S. thesis, University of Utah, May 1968.

[Kay, 1969] _____. The reactive engine, Ph.D. thesis, University of Utah, September 1969.

[Kay, 1970] _____. Ramblings towards a KiddiKomp, in *Stanford AI Project Lab Notebook*, Nov. 1970.

[Kay, 1972b _____. Learning research group 3 year plan, Xerox Palo Alto Research Center, May 1972.

[Kay, 1972c] _____. A personal computer for children of all ages, in *Proceedings of the ACM National Conference*, Boston, Aug. 1972.

[Kay, 1977a] _____ and Goldberg, Adele Personal dynamic media, *IEEE Computer*, Vol. 10, March 1977, pp. 31–41. Reprinted in *A History of Personal Workstations*, New York: Academic Press, 1988.

[Kay, 1979] _____. Programming your own computer, Science Year 1979, *World Book Encyclopedia*, 1979.

[Kiczales, 1991] Kiczales, Gregor, Des Rivieres, Jim Bobrow, Daniel G. *The Art of the Metaobject Protocol*, Cambridge, MA: MIT Press, viii, 335 p. 1991.

[Knuth, 1971] Knuth, Donald E. and Floyd, Robert W. Notes on avoiding 'go to'statements, in *Information Processing Letters*, Vol., 1, No. 1, Feb. 1971.

[Lampson, 1966] Lampson, B.T. Project GENIE documentation, Computer Center, U.C.Berkeley, Oct. 1966.

[Lampson, 1969] _____. An overview of the CAL time-sharing system, Computer Center, U.C. Berkeley, Sept. 1969. Originally entitled On reliable and extendable operating systems, Sept. 5, 1969.

[McCarthy, 1960] McCarthy, John P. Part 1, Recursive functions of symbolic expressons and their computation by machine, in *Communications of the ACM*, Vol. 3, Number 4, April 1960, pp. 184–195.

[Minsky, 1974] Minsky, Marvin. A framework for representing knowledge, MIT, Artificial Intelligence Laboratory Memo No. 306, June 1974. Reprinted in *The Psychology of Computer Vision*, New York: McGraw-Hill, 1975.

[Newman, 1973] Newman, W. M., and Sproull, R. F. *Principles of interactive computer graphics*, New York: McGraw-Hill, 1973.

[Nygaard, 1966] Nygaard, Kristen, and Dahl, Ole-Johan. Simula—an ALGOL-based simulation language, in *Communicatons of the ACM*, IX, 9, Sept. 1966, pp. 671–678.

[Plato] Plato. Timaeus & Phaedrus: *The Dialogues of Plato*, translated by Benjamin Jowett, Great Books of the Western World, Robert Maynard Hutchins, Ed., Encyclopedia Britannica, Inc., 1952.

[Popek, 1984] Popek, G., *et al.*, *The Locus Distributed Operating System*, Cambridge: MIT Press, 1984.

[Ross, 1960] Ross, D.T., and Ward, J. E. Picture and pushbutton languages, chapter 8 of *Investigations in Computer-Aided Design*, interim engineering report 8436-IR-1, Electrical Systems Lab, MIT, May 1960.

[Ross, 1961] _____. A generalized technique for symbol manipulation and numerical calculation, in *Communications of the ACM*, Vol. 4, No. 3, March 1961, pp. 147–150.

[Rovner, 1968] Rovner, P. D. An AMBIT/G programming language implementation, MIT Lincoln Laboratory, Lexington, MA, June 1968.

[Schorre, 1963] Shorre, D. V. META II— A syntax-oriented compiler writing language, UCLA computing facility,

[Shoch, 1979] Shoch, J. F. 1979, An overview of the programming language Smalltalk-72, in *SIGPLAN Notices*, vol. 14, no. 9, Sept. 1979, pp. 64–73.

[Soloway, 1989] Soloway, Elliot and James C. Spohrer, Ed., *Studying the Novice Programmer*, New Jersey: Lawrence Erlbaum Associates, Inc., 1989.

[Smith, 1975] Smith, David Canfield. *Pygmallion*, Ph.D. thesis, Stanford University, 1975.

[Strachey,*] Strachey, Christopher. Toward a formal semantics, United Kingdom.

[Sutherland, 1963] Sutherland, Ivan C. Sketchpad: A man-machine graphical communication system, MIT Lincoln Laboratory, Technical Report 296, Jan. 1963.

[Sutherland, 1963a _____. ibid, in *Proceedings of the SJCC*, Vol. 23, 1963, pp. 329–346.

[Sutherland, 1968] _____. A head-mounted three dimensional display, in *Proceedings of the FJCC*, 1968, p. 757.

[Thacker, 1972] Thacker, C.P. A personal computer with microparallel processing, Xerox Palo Alto Research Center, Dec. 1972.

[Thacker, 1986] _____. Personal distributed computing: the ALTO and ethernet hardware, in *A History of Personal Workstations*, Adele Goldberg, Ed., New York: ACM Press, 1988, pp. 267–290.

[Van Wijngaarden, 1968] Van Wijngaarden, A., Ed., Draft Report on ALGOL 68, Mathematisch Centrum, MR 93, Amsterdam, The Netherlands, 1968.

[Wirth, 1966] Wirth, N.K. and Weber, H. EULER: A generalization of ALGOL, and its formal definition: Part I, in *Communications of the ACM*, Vol. 9, No. 1, Jan. 1966, pp. 13–25.

[Winston, 1970] Winston, Patrick H. Learning structural descriptions from examples, Ph.D. thesis, MIT, Jan. 1970.

[Zahn, 1974] Zahn, C. T, Jr. A control statement for natural top-down structured programming, in *Proceedings of the Colloque sur la Programmation*, April 1974, Paris. A revised version of this paper appears, under the same title, in *Programming Symposium*, vol. 19 of the Lecture notes in Computer Science, B. Robinet, Ed., Berlin: Springer Verlag, 1974, pp. 170–180.

TRANSCRIPT OF PRESENTATION

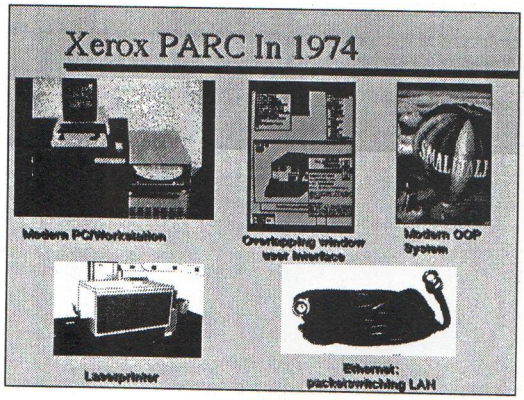
SESSION CHAIR BARBARA RYDER: This interesting biography was given to me by Dr. Kay. Alan Kay is a former professional musician, who majored in mathematics and molecular biology while an undergraduate at the University of Colorado, but spent most of his time working on theatrical productions, and as a systems programmer for the National Center of Atmospheric Research. In 1966 he joined the University of Utah's ARPA Project under Dave Evans and Bob Barton. First contributions were as a member of the original continuous-tone, 3D graphics group. Early on, the confluence of Sketchpad and Simula, with his background in math and biology, brought forth a vision of what

he later called object-oriented programming. Then, with Ed Cheadle at Memcor and Montec Corporation, he designed and built the FLEX Machine, the first object-oriented personal computer. He received his masters and Ph.D. degrees in 1968 and 1969 (both with distinction) for this work. In 1968, struck by the first flat panel display, the GRAIL system at Rand, and Papert's work on LOGO, Kay conceived of the Dynabook, a notebook-sized intimate computer "For Children of All Ages," and set out to realize it. He spent his first post-doc year at the Stanford AI project, and became one of the founders of Xerox PARC when it was set up in 1970. Between the years of 1971 and 1981, Kay was at Xerox PARC serving as the head of the Learning Research Group, principal scientist, and finally Xerox Fellow. In his own terms, he invented, co-invented, inspired, admired, coerced,and tantrumed, with a host of great colleagues, much of personal computing or as he called it—a new paradigm for human–computer collaboration. He was the original designer of the bit-mapped screen, overlapping window interface, painting graphics, and Smalltalk, the first completely object-oriented language. From 1982 to 1984, he was the Chief Scientist, and then Senior Vice President, of Atari, where he set up a large scale research organization. From 1984 to the present, he has been at Apple Computer Corporation, where he currently is an Apple Fellow. There he started, and is currently the head, of the Vivarium Project, a large scale, long-range investigation of children's learning and ability to be amplified by computers. He is codesigner of a number of language environments, including Playground and Constructo. Beyond computers, music is Kay's special passion. He has been a professional jazz musician, a composer, an amateur classical pipe organist, harpsichordist, and chamber pianist. And the last is, according to him, in a decreasing order of ability and interest. His sole professional membership is in the International Society of Organ Builders. He has built several musical instruments, including a collaboration with a master organ builder on a late 17th century instrument for his home.

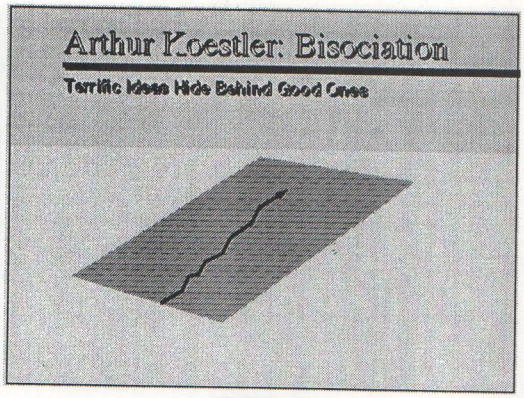
ALAN C. KAY My job is to get to the ten-minute video tape in twenty-five minutes. I'm going to give a different kind of talk, I think. The paper I wrote "The Early History of Smalltalk" was written to be readable and to be read, and I hope that you will read it at some point, if you haven't already. It is fairly detailed, and I can't actually cover that ground today. It's probably just as well. There is also an interesting ACM policy in writing papers, I discovered, and that is that they won't let you put footnotes in. So the idea is that you either put them in the body of the text or leave them out. So I left them out, because the reason for footnotes is so you can have a nice flow of writing. But what I thought I would do, is do the footnotes in this talk. I'm a little worried about the time limit. I remember after graduate school I was going around looking for jobs and somebody asked me, "Can you give a two-hour lecture on the FLEX machine?" I said, "I don't know. I'd never been that brief."

One of the things I discovered about this exercise is what Henry Ford said, which is "history is bunk," because every time I tried writing the paper, I wrote a different one. Every time I worked on this talk, I came up with a different one. They are all sort of true, and they are all sort of false. There's not only revisionism here, but there's also the emotional tinge that you have from day to day, in how you remember things.

(SLIDE 1) At PARC in 1974—forget about these firsts—the word "first" doesn't matter; arguably, we had the first PC workstation, first overlapping window interface, first modern OOP, first laser printer, and first packet-switching LAN. One of the interesting things about PARC—and I'm going to get into this a little bit—is that, for example, the PC workstation was done by a couple of people in the Computer Science Lab with some data from me. Our group, the Learning Research Group, did the user interface and Smalltalk. The Ethernet was done by one person in one lab and one person in another lab. And the laser printer was done by a person in yet a third lab. And in fact, there was no supreme plan of any kind, having to do with management coordination. The whole shebang was done



SLIDE 1

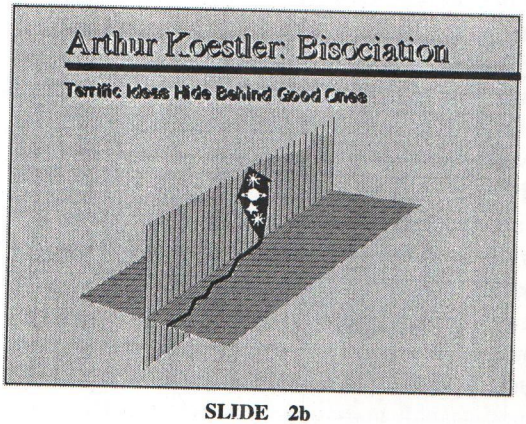


SLIDE 2a

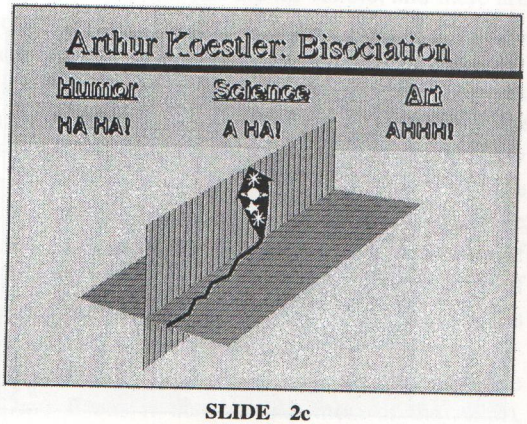
by a bunch of people who were actually kind of friendly and just decided to do it. It wasn't to say that there wasn't a dream or a vision; there was a very strong dream from ARPA, and a very strong vision. One of the joys at PARC is that, for many years, we got along without a lot of management. There was a *romance* about this, and what I want to talk to you about, is the romance of design, or how do you actually do it, or why do things sometimes turn out in a completely different way than normal improvement.

(SLIDE 2a) Arthur Koestler wrote a book called *The Act of Creation* which had quite a bit of influence on us. One of his models was that we think relative to contexts, which are more or less consistent sets of beliefs. Kuhn calls them "paradigms." Creativity-in-a-context normally happens and turns out results that are improvements on the kinds of things that you are doing that are in accord with your belief structures. Now, he calls this "normal science" or "evolutionary science." I'd like to explore the other kind, what he calls "revolutionary science." The way I think about this is, the two things that make life worth living are love and "holy shits." Once you have a "holy shit," it's hard not to want to find more. And in Koestler's diagram, a "holy shit" is sort of represented like this.

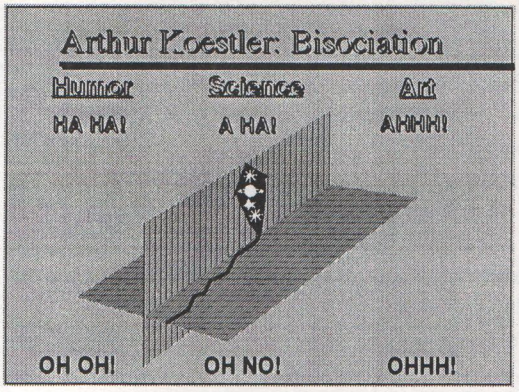
(SLIDE 2b) It is a funny thing that terrific ideas tend to hide behind good ones. The real enemy of a terrific idea is a good one, because the good ones have so many reasons for staying with them. But, all of a sudden something happens, and all of a sudden you are catapulted into a completely different



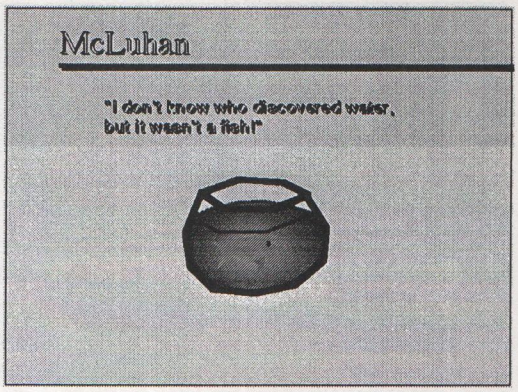
SLIDE 2b



SLIDE 2c



SLIDE 2d



SLIDE 3a

set of beliefs. Koestler's theory of creation is that most creation is sort of like a joke—a joke is where you are led around one path and all of a sudden it reveals you are in a completely different situation.

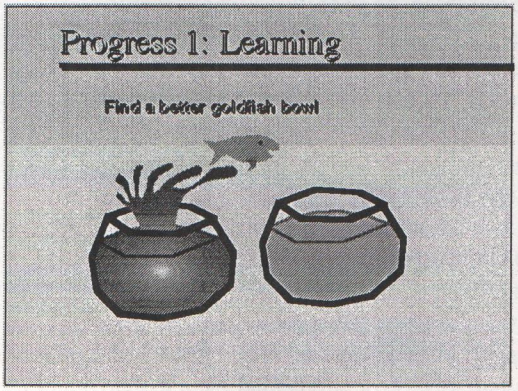
(SLIDE 2c) He even came up with emotional reactions: like if it's a joke it's "ha," if it's science, its "a ha," and if it's art, it's "aah." I thought I would balance that, because that was too optimistic.

(SLIDE 2d) There was also "uh oh," "oh no" and "oh!" So there is a surprise of being catapulted into a different context.

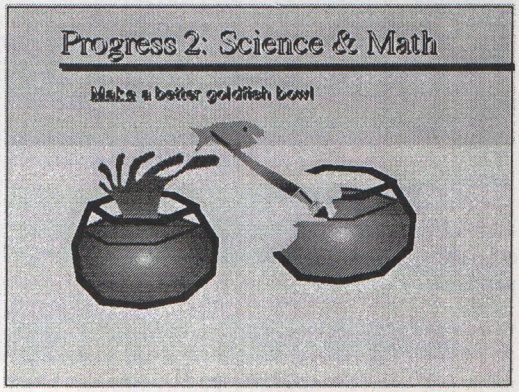
(SLIDE 3a) McLuhan had a different way of looking at this. He said "I don't know who discovered water but it wasn't a fish." So the fish doesn't know what color water he is in, because his nervous system has normalized away what's in his environment. We can think of that water as being beliefs.

(SLIDE 3b) One of the things that humans have learned to do is to use better goldfish bowls. Some of them are called culture. The cultural beliefs that we have can be much more powerful than what our brains are born with. Going from being situated in one goldfish bowl to another is about as tough as being totally creative; maybe it's tougher. Many people who make the jump become even more religious about the new bowl; they forget it's a bowl.

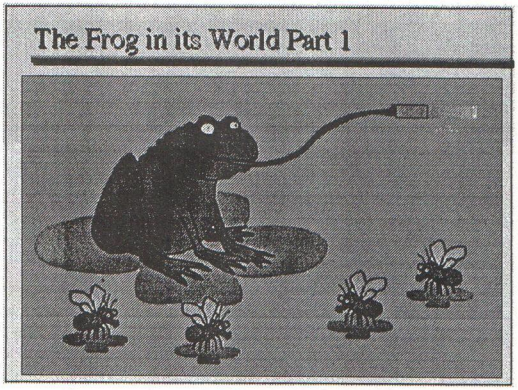
(SLIDE 3c) And of course, one of the things that we do occasionally in science, is that we make a better goldfish bowl. So as we're jumping, we are sketching away at the bowl, and we hope that we get all the water in there before we hit.



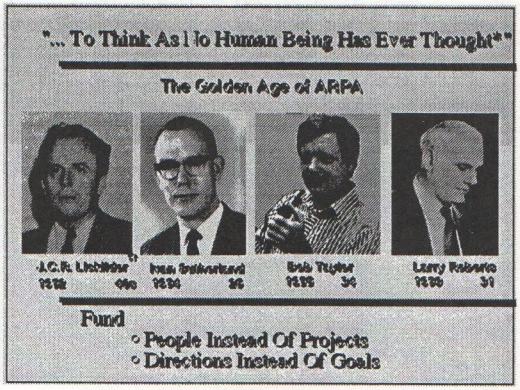
SLIDE 3b



SLIDE 3c



SLIDE 4



SLIDE 5

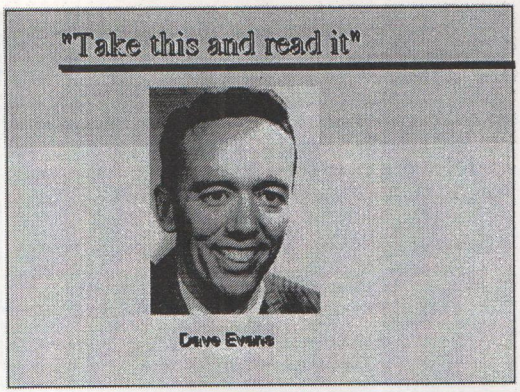
Now this one is mainly done by crazy people, and by a few scientists, and some artists. One of the problems with this is that when you go from one goldfish bowl to another—particularly when you make the goldfish bowl yourself—every goldfish bowl forces you to see things its way. It's like the man in Moliere's play, who suddenly discovered that he had been speaking prose all of his life; it changed his realization of what he had been doing. So, the idea is that if you are going to do this, you better make sure that the end result is really good, because the religious feeling you are going to get when you get over here is going to be the same regardless of whether it is good or bad. It is just the self-confirming seductiveness that a belief-structure has.

(SLIDE 4) Another metaphor I like is frogs. In fact, here at MIT they discovered that the frog's eyes do not tell the frog's brain about much. If you take the frog's normal food, paralyze it (the frog) with a little bit of chloroform, (and then) put flies out in front of the frog, the frog will not eat them, because it can't see that a fly is food. If you flip a little piece of cardboard at the frog that is organized longitudinally, the frog will try and eat it every time, because the food is an oblong moving shape. I think that for us scientists, one of the ways of looking at this, is thinking of these flies as ideas. They are in front of us all the time, but we can't see the darn things.

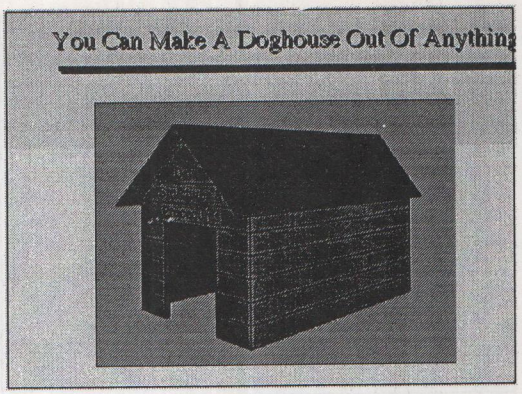
So there's a fundamental principle in all this stuff, and that is that all nervous systems are unconscious with regard to most of the environment, all of the time. One of the most important things to realize, when you are trying to do this stuff, is that whatever you think you believe in almost certainly is not going to help you. You somehow have to make the bowl that you are in, visible, and there are various ways of doing it. You can only learn to see, when you realize that you are blind. Of course, one of the things we know about eyes is that they don't evolve when there is no light.

(SLIDE 5) Here are some candle bearers. This is one of the most amazing decades I think in this century for funding that these four guys created in the sixties. Notice that only Lick (Licklider) was over forty when he took over. Ivan (Sutherland) was twenty-six, Bob Taylor was thirty-four, and Larry Roberts was thirty-one. They were taken out of the ARPA community, and their basic idea is that we will fund people instead of projects, because we can't really judge projects. So we will fund people instead, and we will fund directions instead of goals. That way, these guys didn't have to do a lot of peer-review stuff. But, what they were interested in, was the energy and verve, and probability of success, of the people that they were funding. In spite of the fact that this is one of the best examples of success in funding, very few companies and very few institutions do it this way.

(SLIDE 6) There are lots of cool characters out there that helped the future happen. Here is Dave Evans. One of the most important things about Dave Evans is that he. like most of the ARPA



SLIDE 6



SLIDE 7

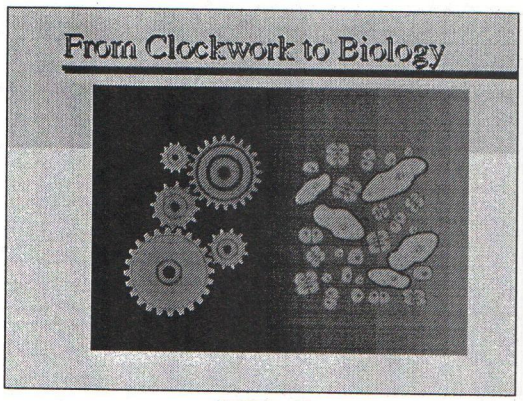
Contractors, treated his graduate students as absolute colleagues. If you couldn't be a colleague, he would ease you out. For instance, I flew 140,000 miles as a graduate student; he actually had a travel budget for graduate students. His idea was (that) we had to go around and find out what people are doing; don't wait to read it in a paper. He had been an architect of one of the earliest tagged architectures, the Bendix G-20, in the the '50s. And one of the things I got from him, is this idea that, whenever you are working on what appears to be the main part of a problem, you are probably off, because that's the most obvious path. What you should be worrying about are the exception conditions. So one of his ideas was basically always leave an "extra bit." That extra bit is all you need, because if you can trap anywhere, then whatever you are trying to do straightforwardly won't kill you in the end. This is a terrific general metaphor.

(SLIDE 7) The context at Utah was large scale 3-D. It wasn't just inventing continuous tone graphics. It was large-scale architecture and CAD. The size of the data structures and stuff they were talking about was many, many orders of magnitude larger than any of the computing that was going on at the time. One of the first things that struck me, was the disparity between what Evans wanted to do, and the way computing was being done then. Which reminded me of how you can make a dog house out of pretty much anything except maybe match sticks, because the materials are so strong compared to the size and structure you are building. Programming in the '50s, particularly in the '60s, could get away with having fragile data structures and procedures, because they weren't that complicated; it wasn't that big a deal; you could track down things. When I learned how to program in the early '60s we only got the machine for five minutes every other day, and you couldn't touch it. So you had to do everything by desk checking.

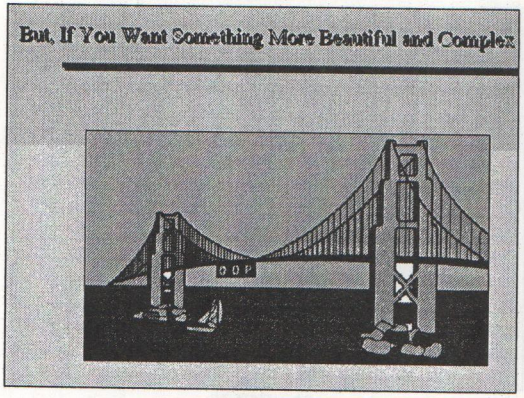
(SLIDE 8) But this isn't what they are talking about. So this is the mechanistic view that still persists today. Probably the worst thing you ever could do is to teach somebody algorithms as their first programming course, because it gives people a completely limited idea of what computers are about. It's like anybody could do physics in the middle ages, you just get a hat. There is not that much to know.

(SLIDE 9) But, if you want to do something that has hundreds and hundreds of thousands, or millions and millions of parts, you need something like a principle of architecture. That set up some stress.

One of the most fortunate things that happened to me when I went to Utah, was that I got there too late for one semester and too early for another semester. So there were actually two months when I had nothing to do. Dave Evans did something really amazing; he said, "What would you like to do in



SLIDE 8



SLIDE 9

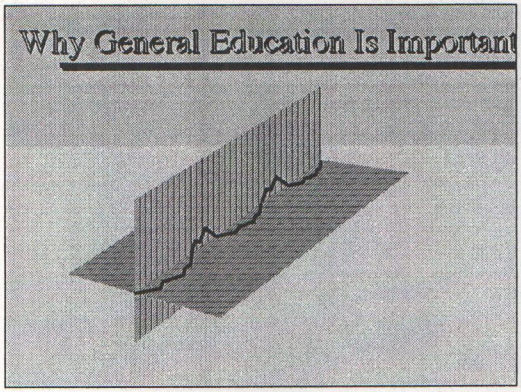
those two months?" I'd never had anybody ask me that in any school situation. I said, "Well, I'd like to actually read things. I'd like to try to understand what's going on," because I actually didn't know anything.

(SLIDE 10) Now, go back to Koestler a second. Why is general education important? Well, if you don't have general education, you don't have any blue ideas, because when you are thinking along, what actually happens is your little thought patterns in one context are bumping into these other contexts. There are little excursions in there, and then rejections usually, and they come back to earth. You need some sort of forcing function. When a forcing function happens, you have two choices. One is you can decide to keep on going and automate the old stuff that you are working on using the new technique better, or there might be this possibility that this is an entirely new way of doing things—and you can actually leap up into this new context and the world is not the same again. For instance, in Maxwell's equations, the cruxpoint was Lorentz's transform, which was a solution of Maxwell's equations that didn't make any sense in a Newtonian world, and Einstein had to go to a different context in order to make sense of them.

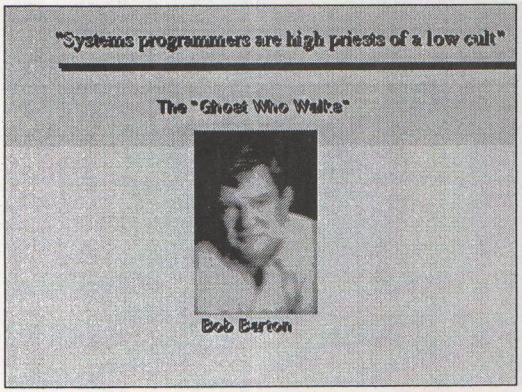
Simula, I think, was one of the greatest cruxpoints that we have had in many years, certainly the best one since LISP. The question is, what was it?

The most normal way was to think of it as a better old thing, and it certainly was—certainly better than ALGOL and ALGOL-X. One of the things you could do—as Barbara talked about—is we could use this as a stimulus to construct something that was better than the better old things, and get abstract data types. Fortunately, I didn't know anything about computer science then. I've not been that innocent since. So, I actuallv had no interest in improving ALGOL because I didn't even know ALGOL very well. I had only programmed in machine code and a little bit of FORTRAN. And so, when I saw Simula, it immediately made me think of tissues. There are lots of little independent things—Simula called them processes—floating around in communication with each other, and my little bump transformed me from thinking of computers as mechanisms, to thinking of them as things that you could make biological-like organisms in, that is, things were up in the ten^{9} to ten^{13} level of complexity.

So, my path from then on, was this way—towards something that was a different way. It's not necessarily that this was a better way of going, but I am just trying to say that I just couldn't look a regular programming language in the face after that encounter with Simula—procedures seemed now to be the worst possible way of doing anything.



SLIDE 10



SLIDE 11

And I had some aiders and abetters: Bob Barton, who is known as the "ghost who walks" at Utah, because he was seen outside of the Merrill Engineering building and inside of Merrill Engineering building, but he was never seen entering or leaving Merrill Engineering Building.

(SLIDE 11) This is actually not a picture of him, because I don't think he has ever been photographed. I'm not sure that he would actually show up on film. So, I actually found a picture of somebody else that looked like him, and I processed it. Barton was this big huge guy who spoke his mind. I remember the first course I took from him. He came stomping into the room and said, "Well, there are a few things known about advanced systems design. I expect you to read them and learn them. But my job is to firmly disabuse you of all the fondly held notions that you might have brought into this classroom." And so what he proceeded to do was to shoot down everything that we believed in. This is something I believe Marvin did that was a service to MIT students. It was extremely liberating. Because by the time you got done with this, those of us who would still survive, there was nothing that was sacred. We could do anything we wanted. Just because other people were doing it didn't mean anything. He was so obnoxious, that even the faculty complained about him to Dave Evans. And Dave Evans said to them: "We don't care if they are prima donnas, as long as they can sing."

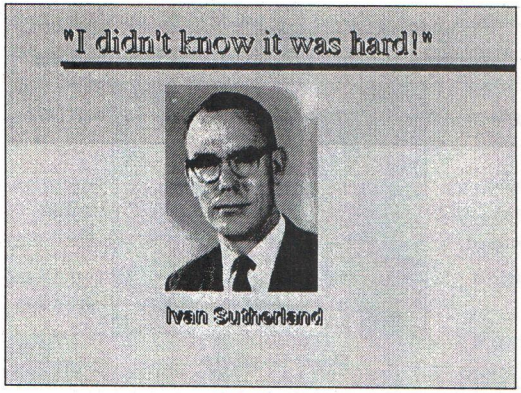
That was Bob Barton.

(SLIDE 12) Here is Ivan. I once asked him, "How could you possibly have done the first object-oriented software system, the first graphics system, and the first constraint solving system, all in one year?" He said, "Well, I didn't know it was hard." In fact, according to Wes Clark, the genesis of Sketchpad was when Ivan went out to Lincoln Labs and saw how bad the display was on the TX-2. And instead of rejecting it like other people had, he asked, "What else can it do?" That's a question we could all try asking for ourselves. And what else it could do, it could simulate the things it was showing pictures of, not just these thick papers, but to actually simulate.

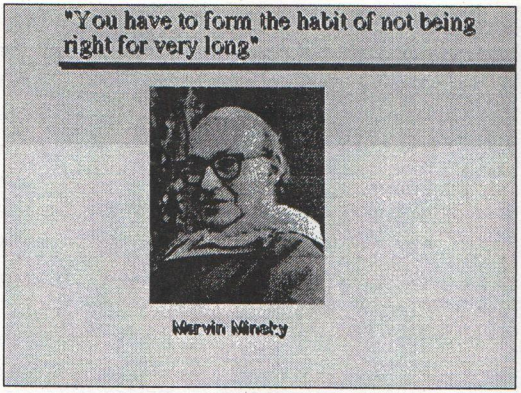
(SLIDE 13) Marvin (Minsky). "You have to form the habit of not being right for very long." One of the things that Marvin used to say is you should never just form distinctions, you should form "tri-stinctions." So, don't just divide it up into two things, try to find a third way to divide it up, and that's probably the way you should go.

The reason I am showing things, is, because this stuff is more a matter of attitude than intelligence. The attitude that these guys had led to really good results. Classical science we think of as being given a universe and we work to discover its rules.

(SLIDE 14) The best book on complex system design. Think about it.



SLIDE 12



SLIDE 13

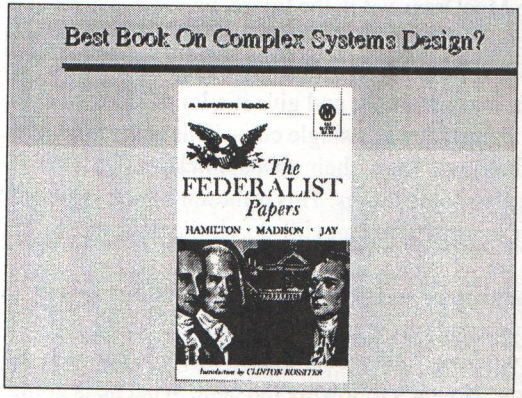
(SLIDES 15 and 16) Just two more people here.

I want to say that when you are doing something real, you've got to have some real people make it happen; and in many ways, Smalltalk is more of Dan's language than mine. I gave you some of the basis for the way we did it. I should mention one other thing, and that is that one of the great things about PARC is that we didn't have to worry about anybody's hardware. We just built everything. When you do that, you don't have to worry about certain efficiency things. Dan, if you know him, is one of the most even-tempered guys I've ever met. I only saw him really angry once, and that was right after he had done Smalltalk-76. I went into him and said, "Dan, now I know what we really should do." He just could not believe I would do this, after he had spent seven months getting this new system going. That was the kind of relationship we had.

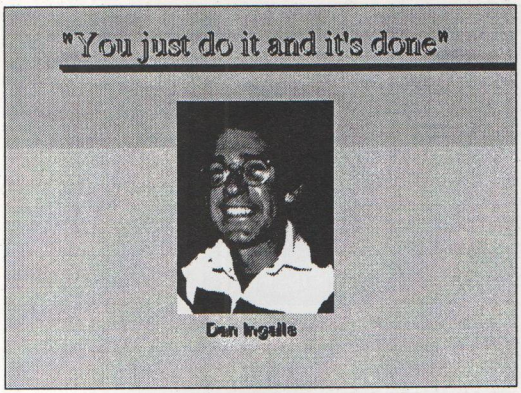
And then finally, Adele, who said, "Isn't this illegal?" when we wanted to take the ALTOs down to the school, and PARC wouldn't let us—even though that was in the plan. We got our station wagon and loaded up some ALTOs in that station wagon, and took them down to the school. I called up George Pake and said, "Guess what I just did?" So, I'm the person who corrupted Adele.

(VIDEO) I want to show you a quick video montage of things in Smalltalk. This goes quickly, so I'll just talk as fast as I can. Here is the early ALTOs, with the cardboard protector on the display.

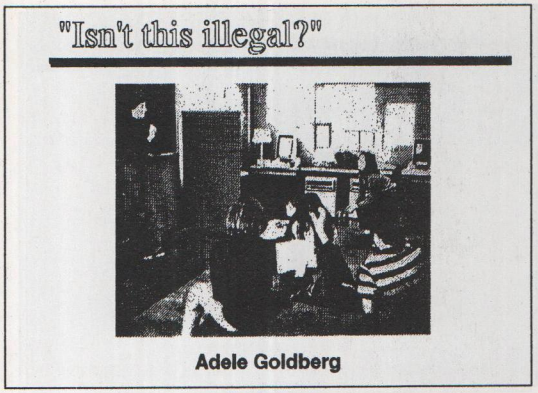
Here is one of the earliest groups of children, and I wonder who that might be, watching what they are doing. I don't recognize the hair style.



SLIDE 14



SLIDE 15



SLIDE 16

Here is how we taught—this was devised by Adele—here's how we taught kids Smalltalk. We gave them an instance of a class, which is called traditionally "Joe." We said, "Joe, grow fifty," now we say "Joe grow minus fifty." "Joe turn thirty-five." And now we make another instance of box called "Jill". That shows up too and we can talk to it separately. So, we say, "Jill, turn whatever." Then we can make a little infinite repeat loop, that gives messages to them. Ultimately, we get a little movie. This gives the kids the idea that these are separate entities but have similar behavior. Make a more complicated movie, has them changing size as well as rotating. They could play around with these. What was interesting about this way of going about it, is you could build many interesting things from these boxes.

Here is one of the very first Smalltalk classes, Marion Goldeen, age twelve. One of the things she discovered was that if you took out the erase part in the draw methods, that you could make little brushes out of things. Each one of those is a box at the top. She has generalized them from squares into polygons. What it is doing now is following the mouse. Now she is getting a hexagon, and again it follows the mouse; it's a particular brush type. As you can see, she has made quite a few . . . . This is like MacPaint; it wasn't completely original; she had seen one of the early painting systems that we had done. This is a very short Smalltalk program done by a twelve-year old.

Then, we got this idea to have Marion teach the class. She is a year older here. She is thirteen, teaching twelve-year olds.

Here is an example from 1975, done by Susan Hamet who is also age twelve. This is a little bit more elaborate; this is like MacDraw, yet in the future, because instead of painting here, what she has is objects being maintained on the screen, (so) she can talk to them separately. She has a little menu over there, and a little window for feeding back and getting parameters. So she is going to change the size of this box on the left, make it small, and give it a lot of sides so she will get a circle. Each one of these things is actually an instance of a single class. This is the kind of thing we were shooting for, which is to have kids do their own tools, their own applications.

Now, the grownups are also doing things. This is the animation system done by Steve Purcell—unfortunately, quite superior to what you can still do on the Macintosh. The ALTO is very powerful doing this.

And then some adult animators and computer people did this system that is called Shazam. This is about a five and a half to six page program in Smalltalk 72. On the left there is an animation window, on the right is a painting window, little iconic menus there. What he is going to do is give the thing he just drew a path to follow. Now it's following the path. What he is going to do is single step it now.

He is pointing to the stair steps. What he wants, is to get it down to the bottom, and he wants to replace the bottom one with a squashed one. Now, now he starts the animation again. And he has gotten a transparent cell; it's overlaid on the one he's got there. So he has actually not destroyed the old one; he is painting on another transparent layer. He is using that as the center. You can see that the painting is being inserted in, as he is painting. What animators want to do is put in enough similarity so you will think that the ball is deforming continuously, even though it's just one frame.

Another twelve-year old. Now she added a feature to the adult system. The feature is to be able to take two animated images and to combine them into one, which is not in the original system. So she picks that up; when she lets it go it's going to merge with the thing there, so now she gets a jockey and a race horse that will animate together.

One more thing in Smalltalk-72. This is a timbre editing system for frequency modulation. This was the first real-time FM system. It was done by Steve Saunders and Ted Kaehler. What you are doing is producing that timbre as a woodwind, and here it is as a kind of a bowed string. The ALTO could do eight real-time voices without any special hardware at all. So here is the C minor Fugue by Bach. This is about 1975.

Now, transition to Smalltalk-76, which is actually much more of a language for adult programmers. One of the first really nifty things that was done in it, was this "simulation kit," which was a tool for novices. This grew out of our work going all the way back to Simula. It's kind of a job shop simulation where you can design the icons. This happened to be done by a Xerox Versatech executive, and represented a production line in the Versatech, and would automatically animate the simulation as it went along.

This was mainly designed by Adele. It was done in 1978. This is a system that was done in Smalltalk, called Playground. The ideas are very similar to stuff we were doing at PARC, except we decided that instead of just having the kids do applications, we wanted them to do complex systems. Because one of the things that the computer does better than anything else is (it can be) a medium for understanding complex systems.

This is all done by children. It is a clownfish that has about twenty-five separate processes running in it. This is all highly parallel programming. What the clownfish is doing is acclimating itself to see an enemy. It also forages for food, tries to mate in the spring, it does shelter. It has four or five drives and only one body. In this foraging path, it attracts the attention of the shark. The shark chases it, its predator avoidance routine takes over the body and it goes for its sea anemone, which it has gotten acclimated to. The shark will not go there because the sea anemone will sting it. That was a more complicated program than most college people ever write, because they almost never write anything that is highly parallel or even parallel at all. One of the ways of thinking about this, my last sentence, is that the Greeks held that the visual arts were the imitation of life; we know that creativity is a lot more than that. But the computer arts are the imitation of creation itself. That's what we should aspire to when we try and design these systems.

TRANSCRIPT OF DISCUSSANT'S REMARKS

SESSION CHAIR BARBARA RYDER: Adele Goldberg joined Xerox PARC in June 1973, working with Alan Kay in the Learning Research Group as a new Ph.D. with experience in the use of computers and education, both for young children, college students, and adults. Her initial responsibility was to determine how to effectively teach children and adults about the various Smalltalk systems, and to plan and manage experiential projects as feedback to the language and environment design groups. When Kay left on sabbatical, Goldberg became group manager for the System Concepts Group. One goal for the group was to create a Smalltalk system that would run on standard workstations and be

distributed outside Xerox. This goal was accomplished in 1983. At this time, Goldberg authored and coauthored several books on the Smalltalk-80 system. Earlier in 1981, Goldberg served as guest editor of the August 1981 issue of *Byte Magazine*, which introduced the general public to the ideas behind Smalltalk. In the early 1980s, Goldberg became manager of the System Concepts Laboratory, one of six research labs at PARC. This lab was located both in Palo Alto and in Portland. Research programs were pursued on the use of video to facilitate group design when members of the group are located in remote sites, essentially initiating research on collaborative computing. While working at Xerox PARC, Goldberg was active in the ACM. She served as Editor-in-Chief of *Computing Surveys* from 1979 until 1982, National Secretary from 1982 to 1984, and President from 1984 to 1986. During that time, she founded ACM Press, initiated the History Series, edited the ACM Press Book on *The History of Personal Workstations*, and helped found the OOPSLA Conference now managed by SIGPLAN. In 1987, Goldberg convinced Xerox to form a separate business to leverage the research on object-oriented technology. ParcPlace Systems was founded in 1988. From 1988 through 1992, she served as CEO and President of ParcPlace. She is currently Chairman of the Board. Goldberg's current technical activities focus on organizational and cultural issues in the introduction of new technology into large organizations' software engineering groups.

ADELE GOLDBERG: As Alan has pointed out in his paper and presentation, Smalltalk is not the story of the evolution of the design of a language or language mechanisms. But rather, it's a story of culture, of organization and vision.

Early in my acquaintance with Alan, I recall someone asking him to give a talk to predict the future of computing. He said, "No, I'm not going to do that. I don't predict the future—I make it."

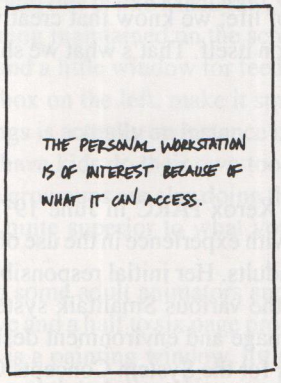
Lesson number one from Alan. Pick a dream, make it happen.

In this first regard, the Smalltalk invention and development history is different from many other languages. It represents an attempt to realize a dream.

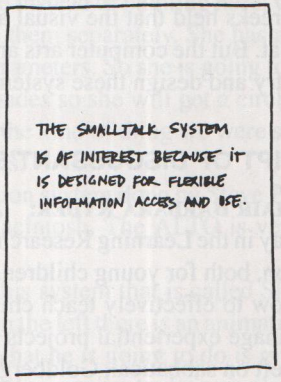
(SLIDE 1) The dream starts with the idea, startling for those days, that computers are simply *not* interesting. What is interesting is what people can do with computers, and what they can access with them.

(SLIDE 2) The scheme then, the key to flexible information access and use, would be software, and that software we named Smalltalk.

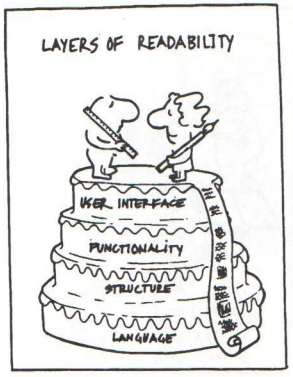
Of course, to make dreams happen, you have to be empowered, and you have to empower. Alan once told us, that to his dismay, his vision of the future was hard, and he simply could not do it by



SLIDE 1



SLIDE 2



SLIDE 3

SLIDE 4

himself. So he went to Xerox and set out to hire an unorthodox set of people—each one too naive and full of the dream, to know the problem was hard—and each one some expansion of some part of Alan. In private, we use to joke about which part each of us represented. I suppose that in truth, we weren't supposed to be a part of his body, but of his mind—sort of an early form of Vulcan mind meld.

Lesson number two from Alan. Hire people who don't know that what you want them to do is hard.

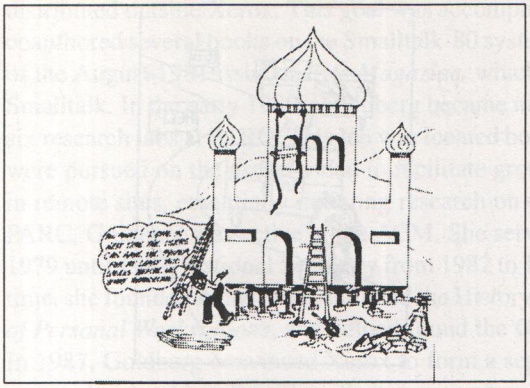
In the early days, I was the representative user. Maybe you could call me the Learning Police. Is what we have created learnable outside the group? Lots of clever computer science can't live up to that requirement.

(SLIDE 3) In a way, it was very fortunate that the research was set in the context of working with kids, because it introduced pedagogy as a quality objective. It motivated a number of significant language changes to support learning by doing, and learning by reading. Doing meant refining from existing useful objects—at every layer of the system. This readability as a quality objective allowed us to see the software problem that we were trying to solve—not as a problem in creating a programming language—but rather as a problem of creating a system for creating systems. These systems would define a higher-level language, created by their developers to be writeable and readable by anyone who understood the domain in which it was going to be used. Aiding readability became a critical challenge for us and any user interface and other environment support.

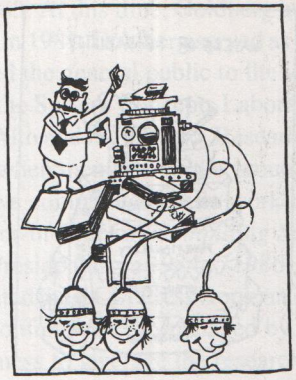
(SLIDE 4) One of my self-imposed tasks was to get people to understand that the ability to teach Smalltalk to children in no way implied that we had solved the learnability problem for adults. Kids could be guided through the system as it was constructed. The several educational resource centers I created convinced me of this point.

But adults would come to the system with their own ideas of what they wanted to do. We would have to be responsive to their way of thinking—of helping them to create appropriate models for their worlds. Clearly, the simulation or information modeling capabilities of our system would be the key determiner of success. I still believe this today. I do not believe I succeeded, however, in getting the group to understand the unfortunate distinction between so-called kids and so-called adults.

(SLIDE 5) By 1979, we were in systems programming mode, trying to move software ideas forward without a similar change in hardware capability. As Alan pointed out in his paper, the *NoteTaker* project had been canceled despite—in our eyes—major proof-of-concepts success. We created the Smalltalk Hall of Shame that same year, and we inducted as the first member the then head of PARC, who declared that no one was interested in carrying computers around.



SLIDE 5



SLIDE 6

It was time to publish what we knew, and reorganize the research project. Perhaps it was time to admit the problem was hard, and find some more naive people to help us. Also, this form of completion was important as a morale builder, since considerable frustration was building up at the lack of, at least, internal Xerox acknowledgment of the group's contributions.

(SLIDE 6) Larry Tesler and I made another run at the business players that year—a project called Monk. Simple idea and it was very doable. Xerox technicians were not using up-to-date manuals, nor good diagnostic tools when repairing copiers. We wanted to give them an interim Dynabook, with a world of copier information literally under the machine with them. They didn't get the idea and the project did not get funded.

Steve Jobs came to visit. He did get the idea. He came back—with the entire Lisa programming team. After a two-hour hallway argument, I was forced by my managers' manager (the head of the then divided PARC) to give them an in-depth demo. They took the idea. This manager, too, was inducted into the Smalltalk Hall of Shame.

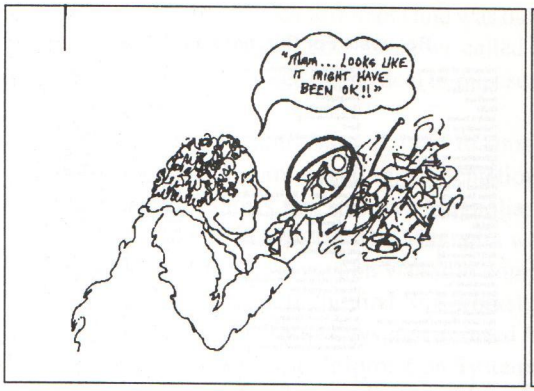
(SLIDE 7) These experiences communicated one hard lesson—we did not have a coherent act. We had lots of bits and pieces for good demos, but, the integrated whole was still in our heads. The best way to share was going to be by publishing the Smalltalk system itself, not a book or papers about a dream where we would risk people getting the schemes wrong. Of course, this was very risky. It violated Alan's critical admonishment, "Do not have customers; they expect support."

> Lesson number three from Alan. If you don't share your results, you can burn the disk packs and start over.

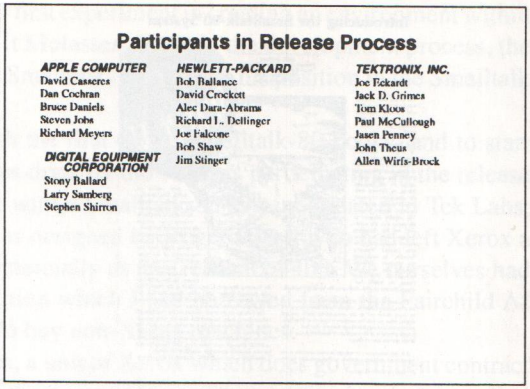
But as individuals, and as a group, we wanted to feel that we finished something in a form that would get a fair evaluation in the computing world. We knew Smalltalk would run on the new "personal" computers because the *NoteTaker* was 8086-based. And we know we could not, alone, do all the work to take Smalltalk to people using these personal computers.

(SLIDE 8) To put out a system meant stopping research and doing enough work to have something we could release to people like us—researchers at universities and advanced development people in commercial companies. But which companies? Burt Sutherland suggested the answer—companies with the ability to create the hardware we wanted. We wanted to influence hardware extensions that would enhance the media accessible by a Smalltalk-like system.

So, we picked companies that built hardware systems, but which had in-house software teams. We signed up Apple, Hewlett-Packard, DEC, and Tektronix. Thus began the saga of creating the shareable



SLIDE 7



SLIDE 8

version of the system called Smalltalk-80, as a multivendor project. Again, Alan wasn't there to tell us the next lesson, although I'm sure it's something he would say.

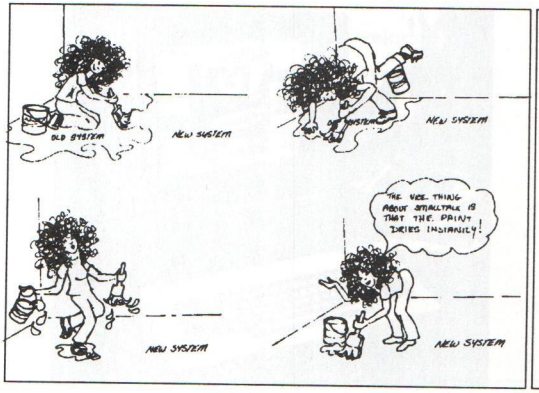
Lesson number four from Alan. You have to be careful what you wish for—because you *will* get it.

(SLIDE 9) The release process had three parts to it. We redesigned the language and environment subsystem by subsystem, giving the engineers in the other companies access as we proceeded. We were able to rebuild the system in this manner partially because of the ability to cause all existing objects of one class to, essentially and instantaneously, *become* objects of another class.

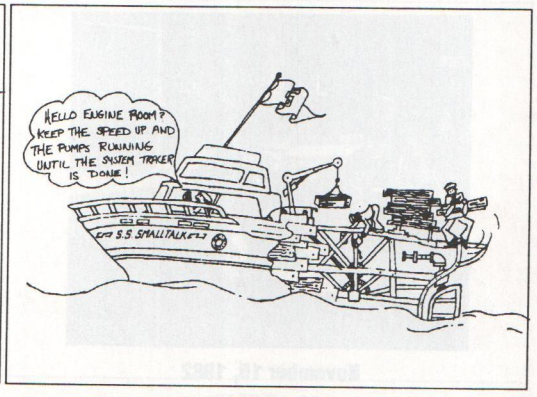
(SLIDE 10) Ted Kaehler, the group cartoonist, likened this to the process of launching a ship while it was still being constructed.

The engineers in the four companies worked with us to define the formal specification of the virtual machine architecture. They each implemented the bytecode interpreter and devised various memory management schemes. Later these efforts were published in the book *Smalltalk 80: Bits of Wisdom, Words of Advice*, edited by Glenn Krasner, who managed the systems work in our research laboratory.

Dave Robinson and I acted as documenters, writing the description of the system as it unfolded, and making sure each part was sufficiently explainable. We wrote three books about the language. We only published the third.



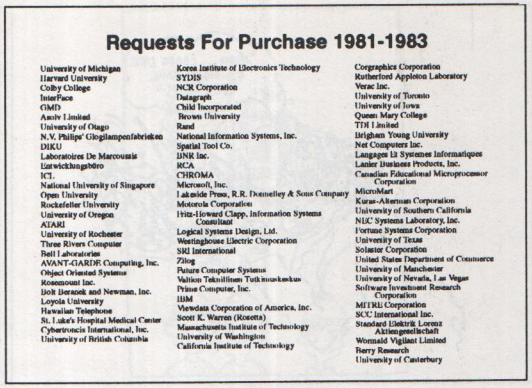
SLIDE 9



SLIDE 10


Introducing the Smalltalk-80 System

SLIDE 11


Requests For Purchase 1981-1983

University of Michigan
Harvard University
Colby College
InterFace
GMD
Amdy Limited
University of Otago
N.V. Philips' Gloeilampenfabrieken
DIKU
Laboratoires De Marcoussis
Entwicklungsbüro
ICL
National University of Singapore
Open University
Rockefeller University
University of Oregon
ATARI
University of Rochester
Three Rivers Computer
Bell Laboratories
AVANT-GARDE Computing, Inc.
Object Oriented Systems
Rosemount Inc.
Bolt Beranek and Newman, Inc.
Loyola University
Hawaiian Telephone
St. Luke's Hospital Medical Center
Cybertronics International, Inc.
University of British Columbia
Korea Institute of Electronics Technology
SYDIS
NCR Corporation
Datagraph
Child Incorporated
Brown University
Rand
National Information Systems, Inc.
Spatial Tool Co.
INRIA Inc.
RCA
CHROMA
Microsoft, Inc.
Lakeside Press, R.R. Donnelley & Sons Company
Motorola Corporation
Fritz-Howard Clapp, Information Systems Consultant
Logical Systems Design, Ltd.
Westinghouse Electric Corporation
SRI International
Zilog
Future Computer Systems
Valtion Teknillinen Tutkimuskeskus
Prime Computer, Inc.
IBM
Wavetek Corporation of America, Inc.
Scott K. Warren (Rosetta)
Massachusetts Institute of Technology
University of Washington
California Institute of Technology
Cegraphics Corporation
Rutherford Appleton Laboratory
Verac Inc.
University of Toronto
University of Iowa
Queen Mary College
TDI Limited
Brigham Young University
Net Computers Inc.
Languges I3 Systemes Informatiques
Lanier Business Products, Inc.
Canadian Educational Microprocessor Corporation
MicroMart
Kurss-Alterman Corporation
University of Southern California
NEC Systems Laboratory, Inc.
Fortune Systems Corporation
University of Texas
Solanier Corporation
United States Department of Commerce
University of Manchester
University of Nevada, Las Vegas
Software Investment Research Corporation
MITRE Corporation
SCC International Inc.
Standard Elektrik Lorenz Aktiengesellschaft
Wormald Vigilant Limited
Berry Research
University of Canterbury

SLIDE 12

(SLIDE 11) The earliest practice in publication came in 1981. Chris Morgan, then Editor-in-Chief of *Byte Magazine*, offered me all the feature pages of the August 1981 issue. Dan Ingalls liked hot air balloons as a way to announce that Smalltalk was finally traveling off its isolated island, an island Chris had described two years before.

(SLIDE 12) The result was shocking. I started to get letters from around the world—from universities and commercial organizations—asking to purchase the Smalltalk system. There was even an article called "Mostly Smalltalk" in the TWA *Ambassador Magazine*.

(SLIDE 13) Two more Xerox disappointments clouded the work effort. Because of some problems with the introduction of the Star system in 1981, we were asked to join another group in making recommendations. On July 21, 1982, Dan Ingalls and I formally submitted a plan to build a low-cost 68000-based Smalltalk-80 machine, that would serve as the entry level system of the Star family. We called it *Twinkle*. The proposal was ignored. It reappeared in 1984 as a variety of Apple. Apparently suggesting freehand graphics and a Smalltalk base was too controversial for Xerox. The Xerox manager responsible for delivering the proposal had simply not bothered, and he too was inducted into the Smalltalk Hall of Shame.

In order to share the Smalltalk-80 result, the then head of the Office Systems Group obligated us to build a version for the Star workstation. The project was started in (September) 1982 by Frank Zdybel, and joined later by Paul Mc Cullough. It was completed in the spring of 1984. It was written


November 15, 1982

SLIDE 13

SLIDE 14

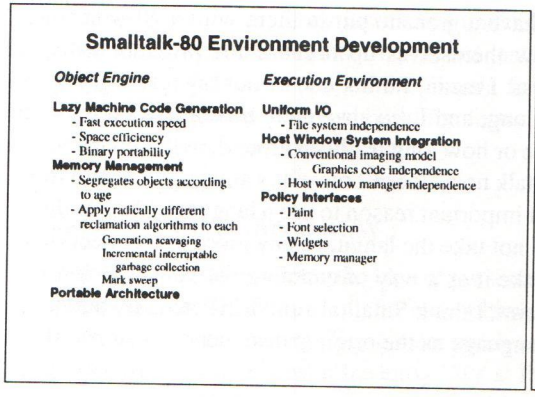
in the Mesa programming environment and was our first experiment in creating an environment within an environment. In honor of how it ran, we called it Molasses. Early in the development process, the business head tried to renege on the deal to release Smalltalk, and earned his position in the Smalltalk Hall of Shame.

(SLIDE 14) It took until May of 1983 to finish the first of the Smalltalk-80 books, and to start rolling out the first of commercial implementations done by the vendors participating in the release process. This slide shows the Tekronix Magnolia, with a Smalltalk-80 system, created in Tek Labs, primarily by Allen Wirfs-Brock. The Magnolia was designed by Roger Bates, who had left Xerox a short time before. The next version was sold commercially as the Tekronix 4400. We ourselves had done a native system port to the first Sun workstation which I had borrowed from the Fairchild AI Lab because we were, in those days, not allowed to buy non-Xerox machines.

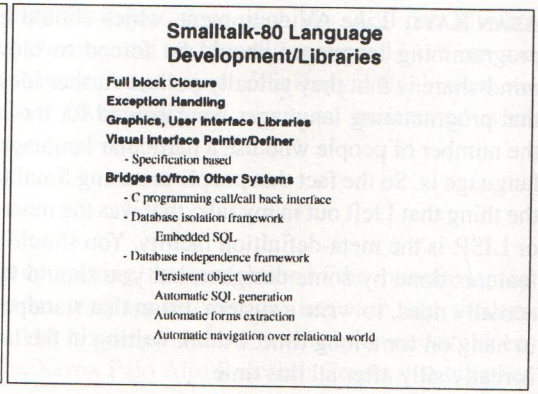
Meanwhile, Xerox Special Information Systems, a unit of Xerox which does government contract work, had taken early Smalltalk releases and used them to create an office automation system called the Analyst. The Analyst-84 was being readied for broad deployment in the CIA. And so by 1985, we tried to get Xerox to face up to a cold truth: the CIA was in production using an Analyst workstation based on a research result. This was *not* a good idea. The base system had to be engineered; a real business investment was needed. Visits from the Administrative Head of the CIA to PARC did not change the mind of any Xerox business decision maker.

But, in 1986, with a new head of the Office Systems Group, and some friendly ears, an internal task force set out to evaluate the marketplace for Smalltalk. Their conclusion? Do it! Bill Spencer, then head of PARC, approved an experiment to start a business internally, while Xerox sorted out what to do. We moved three engineers from the Dallas Development Center in July, and some of the researchers refocused on rethinking the implementation technology. In November of 1986, Peter Deutsch and shortly after, Allan Shiffman, joined and started the complete redesign of the virtual machine technology. In February 1987, myself and two of the research team formed ParcPlaceHolders, a corporate shell that began a 13-month negotiation to form ParcPlace Systems, Inc., as an independent company.

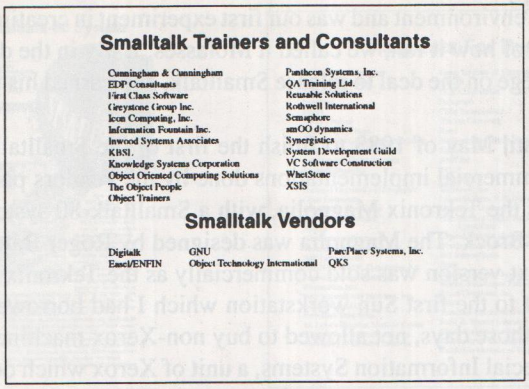
(SLIDE 15) After five years, the language base has not changed a lot, although the implementation technology has been completely redesigned and implemented. It is the first truly reusable, fully-compiled, Smalltalk system. Creating an application on one of 11 ParcPlace platforms means that your application runs on 11, without your doing any more work. But those ports are compiled to the host instruction set, access host operating system features, and host windowing and graphics. We have



SLIDE 15



SLIDE 16



SLIDE 17

changed the virtual machine and the object engine; we have extended the notion of late binding to run-time compilation and execution in the context of late binding of application-level specification of policies for graphics and for user interaction.

(SLIDE 16) The language was modified to have closure of blocks. An exception handling capability was added. And a number of libraries, as well as bridges to C, and to external databases, were built. We ship two different development environments, two additional kits of libraries and tools, on all 11 platforms. Since the fall of 1986, we have released five complete revisions of the system.

(SLIDE 17) Today, there are numerous Smalltalk vendors and service companies. A few months ago, ANSI created X3J20 as the official Smalltalk Standards Committee. According to last week's *Business Week* article, "The Buzz is About Smalltalk," and the Analyst workstation is still alive and well in the CIA.

So, 10 years, almost to the month, after its first publication, and two months short of a 20-year anniversary of my arrival at Xerox, Smalltalk continues to be a surprisingly persistent object.

TRANSCRIPT OF QUESTION AND ANSWER SESSION

For ALAN KAY and ADELE GOLDBERG from JEFF SUTHERLAND (Object Databases): Smalltalk is used by 10 percent of software product developers, C++ by 50 percent. What do you see as the future mind-share for Smalltalk, particularly in end-users SWOPS?

ALAN KAY: Like AV equipment, which should have a grenade put in them with a 20-year timer, programming languages should be forced to blow themselves up because the problem with the mind-share is that they actually pollute further ideas. I mean, I think one of our biggest problems is that programming languages hang around for too long, and I see absolutely no correlation between the number of people who use a particular language or how long it's been around, with how good the language is. So the fact that people are using Smalltalk now, doesn't mean it's any good either. I think the thing that I left out in my talk, that was the most important reason to use a language like Smalltalk or LISP, is the meta-definition facility. You should not take the language any more as a collection of features done by some designer, but you should take it as a way of enabling the language that you actually need, to write a system. From that standpoint, I think Smalltalk and LISP actually are going to hang on for a long time. I think writing in the language as the original designers envisioned them is really silly after all this time.

ADELE GOLDBERG: Let me add to this. I really despise all these statistics. Languages are designed for specific purposes. Each one of them has its role to meet the design objectives. If you take a market segment where the language is appropriate, then you talk about penetration, you'll have some interesting numbers. The market segment that's interesting for Smalltalk, is the current COBOL community of MIS people, where C++ has zero penetration, where Smalltalk has now got about five percent, which I think is at least four and a half percent more than anybody expected, maybe more. What is going on here, is that people invent languages to express their ideas. And what really needs to happen, is to make it easier and easier for people who know their domain to know what they want to do, to express their ideas, and invent their own programming language. That's why that simulation kit that Alan showed you is so interesting. It is a framework for expressing the adventure of simulation, in which people who know something about the events and the domains that they are interested in could express their own ideas about what they want to see happen.

For ALAN KAY from JIM KIRK (Affiliation not mentioned): In your talk, you nominated object-oriented programming as a programming system with thousands of parts. In your paper you endorsed object-oriented programming as a way for children and others to think about computers. Are these two notions, in fact, related? Are these two notions, in fact, the same?

KAY: Give me a few minutes, and I can relate any two things. One of the ways of boiling down what our aspirations were, was that every time we come along with a new medium for capturing our ideas, it also has some modes of thought that go along with it. I think the modes of thought that are fallacious about computing, have to do with complex systems with many interacting parts. And that's something that's very hard to think about in using old mathematics, and old medium(s). I think that one of the aims for teaching children thinking, is that one of the good modes for them to think about are very complex things that have ecological relationships to each other. So, I think giving children a way of attacking complexity, even though for them complexity may be having a hundred simultaneously executing objects—which I think is complex enough for anybody—gets them into that space in thinking about things that I think is more interesting than just simple input/output mechanisms.

SCOTT GUTHREY (Schlumberger): Did the D machine initiative help or hinder Smalltalk?
(Ryder: Perhaps in your answer, you'll explain what the D machines were)

KAY: The D machines were talked about this morning a bit. They were a series of follow-ons to the ALTP. I think in many ways, one of the things that hurt Smalltalk—and this is absolutely my own personal view—is that the first D machine was late. And it was late, partially because Xerox just did not want to put hundreds of ALTOs in there and we came right back and said now we need machines that are five times faster. They just didn't understand the implications of things. And part of what happened when we did Smalltalk-76 was not just a cleanup, but we actually regressed—if I can use that word—back towards some of the Simula formations, to get more efficiency. I think what we should have done was kept on building faster and faster machines until we understood what object-oriented design was really about.

BIOGRAPHY OF ALAN C. KAY

Dr. Alan Kay is best known for the idea of personal computing, the conception of the intimate laptop computer, and the inventions of the now ubiquitous overlapping-window interface and modern object-oriented programming. These were catalyzed by his deep interests in education and children. He led one of several groups in the early '70s at the Xerox Palo Alto Research Center that together

developed these ideas into: modern workstations and the forerunners of the Macintosh, Smalltalk, EtherNet, Laserprinting, and network "client-servers."

Before Xerox, Kay was a member of the University of Utah ARPA research team that developed 3D graphics. His Ph.D. in 1969 was awarded for the development of the first graphical object-oriented personal computer. His undergraduate degrees were in Mathematics and Molecular Biology (from the University of Colorado in 1966). As a member of the ARPA community, he also participated in the early design of the ARPANet (which became the Internet). After Xerox he was Chief Scientist of Atari, and from 1984 has been a Fellow at Apple Computer.

He is a Fellow of the American Academy of Arts and Sciences, and the Royal Society of Arts. He has received numerous awards and prizes including the J-D Warnier Prix D'Informatique and the ACM Systems Software Award.

A former professional jazz guitarist and composer, he is now an amateur classical pipe-organist.