

THE EARLY HISTORY OF SMALLTALK

Alan C. Kay
 Apple Computer
 kay2@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 mainframe; 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 which almost forced these new styles to be invented.

Table Of Contents

I. 1960-66—Early oop and other formative ideas of the sixties 2

 B220 File System 2

 SketchPad & Simula 4

II. 1967-69—The FLEX Machine, an oop-based personal computer 6

 Doug Englebart and NLS 6

III. 1970-72—XEROX PARC 12

 KiddiKonn 12

 miniCOM 12

 Smalltalk-71 12

 Overlapping Windows 12

 Font Editing, Painting, Animation, Music 12

 Byte Codes 12

 Iconic Programming 12

IV. 1972-76—XEROX PARC: The first real Smalltalk (72) 17

 The two bets: birth of Smalltalk and Interim Dynabook 17

 Smalltalk-72 Principles 17

 The Smalltalk User Interface 17

 Development of the Smalltalk Applications & System 17

Evolution of Smalltalk: ST-74, ooze storage management

Smalltalk and Children 29

V. 1976-80—The first modern Smalltalk (-76) 29

 "Let's burn our disk packs" 29

 The NoteMaker 29

 Smalltalk-76 29

 Inheritance 29

 More Troubles With Xerox 29

 ThingLab 29

 Apple Demo 29

VI. 1980-83—The release version of Smalltalk (-80) 38

 Transformations 38

 Coda 38

References Cited In Text 41

 Appendix I: KiddiKonn Memo 41

 Appendix II: Smalltalk-72 Interpreter Design 45

 Appendix III: Acknowledgements 47

 Appendix IV: Event Driven Loop Example 50

 Appendix V: Smalltalk-76 Internal Structures 53

- 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 4 great programming conceptions of the sixties

Introduction

I'm 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, etc. But they all seem to be either an "agglutination of features" or a "crystallization of style". COBOL, PL/1, Ada, etc., belong to the first kind; LISP, APL—and Smalltalk—are the second kind. It is probably not an accident that the agglutinative languages

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

HOPL-III/4/93/MA,USA
 © 1993 ACM 0-89791-571-2/93/0004/0069...\$1.50

all seem to have been instigated by committees, and the crystallization 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—like data structures, procedures, and functions which 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 Lxrcrs 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, etc. 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.

1. 1960-66—Early oop and other formative ideas of the sixties

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

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

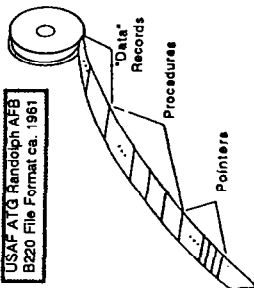
True to the stages, I “barely saw” the idea several times ca. 1961 while a programmer in the Air Force. The first was on the Burroughs 220 in the form of a style for transporting files from one Air Training Command installation to another. There were no standard operating systems or file formats back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

The second barely-seeing of the idea came just a little later when ATC decided to replace the 220 with a B5000. I didn't 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, etc. 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 fut from this machine at this time was not the OOP idea, but some insights into HLL translation and evaluation. [Barton, 1961] [Burroughs, 1961]

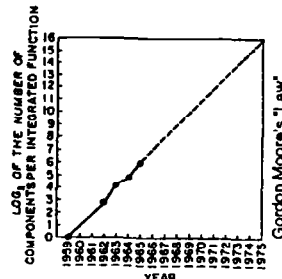
After the Air Force I worked my way through the rest of college by programming mostly retrieval systems for large collections of weather data for the National Center for Atmospheric Research. I got interested in simulation in general—particularly of one machine by another—but aside from doing a one-dimensional version of a bit-field block transfer (bibbit) 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 which predicted that integrated silicon on chips was going to exponentially improve in density and cost over many years [Moore 65]. 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.

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 ARIA 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



USAFATG Randolph AFB
B220 File Format ca. 1961



Gordon Moore's "Law"

walked into Dave Evans' 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 inter-relate 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 through them to routines like the 220 file system [Ross, 1961]. It was the first to have clipping and zooming windows—one "sketched" on a virtual sheet about 1/3 mile square!

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

"generic block" showing
procedural attachment

Sketchpad's inheritance
hierarchy

Sketchpad Structures

Drawing in Sketchpad

Programming with constraints

Constraints
represented
as icons

Constraints
merged
with picture

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 didn't 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) [Nygaard, 1966; Nygaard, 1983].

This was the big hit, and I've 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 algo-

bras 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?

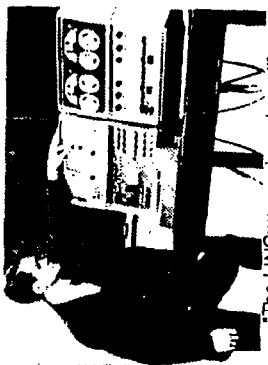
I recalled the monads of Leibniz, the "dividing nature at its joints" discourse of Plato, and other attempts to parse complexity. Of course, philosophy is about opinion and engineering is about deeds, with science the happy medium somewhere in between. It is not too much of an exaggeration to say that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

II. 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 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 surpassed [Joss, 1964, Joss, 1978]. Joss was too slow for serious computing (but cf. Lamson 65), did not have real procedures, variable 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 1963] in which types were discarded, different features consolidated, procedures were made into first class objects, and so forth. Actually kind of Lisp-like, 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 85000-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 tortuously rendered in an "extended precedence" grammar that actually required concessions in the language syntax (e.g. " 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 Shorre's META [Shorre 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, etc.) even on his first machine, and the LINC



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

had a "glass teletype", but a Sketchpad-like system seemed far beyond the scope that we could accomplish with the maximum of 16k, 16-bit words that our cost budget allowed.

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, etc. An entire conceptual world and world view (Engelbart 68). 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 MP 6600 on a desk. I was almost frightened by the implications: computing as we knew it couldn't 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.

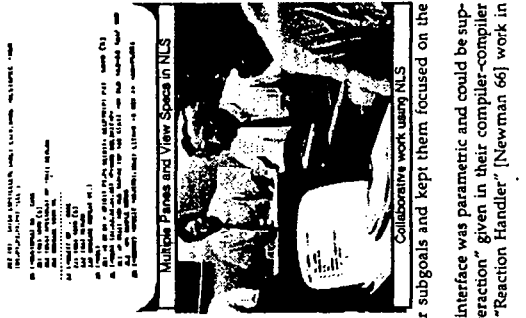
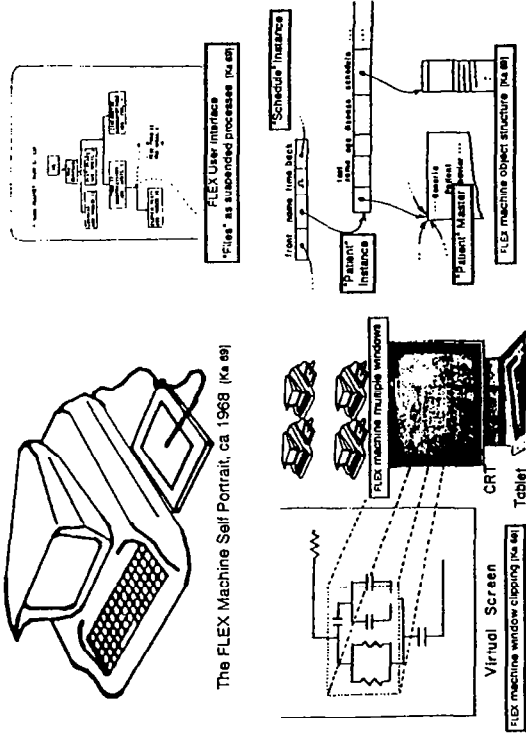
Instead of at most a few thousand institutional mainframes in the world—even today in 1992 it is estimated that there are only 4000 IBM mainframes in the entire world—and at most a few thousand users trained for each application, there would be millions of personal machines and users, mostly outside 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 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" [Newman 66] 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 which 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 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 BS5000 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 BS5000 used l-values and r-values (Strachey) which worked for some cases but couldn't handle more complex objects. For example: $a[55] := 0$, if a 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, etc., 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[Ca] 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 corouting control structure[Conway, 1963] was used as a way to suspend and resume objects. Persistent objects like 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. Corouting 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;
```

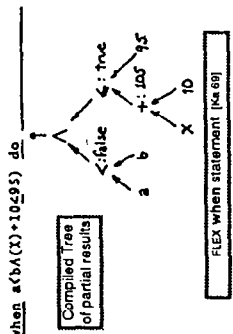
where the ... to ... by ... was a kind of corouting object. Many of these ideas were reimplemented in a stronger style in Smalltalk later on.

Another control structure of interest in FLEX was a kind of when statement (6b)(x) * 10 95) do event-driven "soft interrupt" called when. Its boolean expression was compiled into a "tournament sort" tree that cached all possible intermediate results. The relevant variables were threaded through all of the sorting trees in all of the whens 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 whens 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. Towards the end of the three days, Bob Taylor, who had succeeded Ivan Sutherland as head of ARPA-INFO, 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 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 didn't 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 70].

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 U 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.

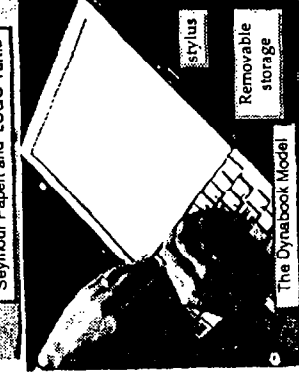
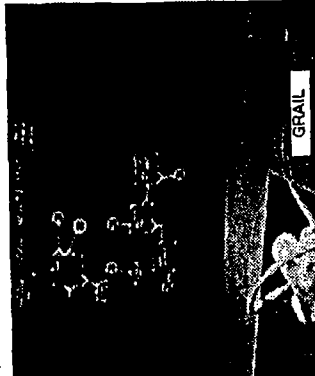
But later that year at RAND I saw a truly beautiful system. This was GRAIL, the graphical follow-on to JOSS. The first tablet (the famous RAND tablet) was invented by Tom Ellis [Davis 1964] in order to capture human gestures, and Gabe Groner wrote a program to efficiently recognize and respond to them [Groner 1966]. Though everything was fastened with bubble gum and the system crashed often, I have never forgotten my first interactions with this system. It was direct manipulation, it was analogical, it was modelless, 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 LOCO 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' GRAIL's, and LOCO'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 low-speed tactile degrees of freedom offered by the stylus and the more limited but faster keyboard. Since 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 next year (1969) there was a conference on Extensible Languages in 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 eyes (the urge to generalize) [ACM 59].



Seymour Papert and LOGO Turtle

The Dynabook Model

Removable storage stylus

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 that 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 where 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 69], which as Ivan Sutherland liked to say was "anything you can get three people to sign".

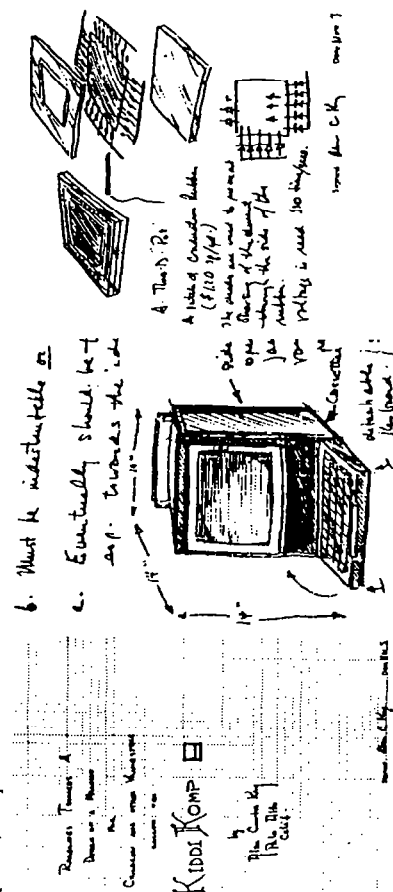
After three people signed it (Ivan was one of them), I went to the Stanford AI project and spent much more time thinking about notebook KiddyKomputers than AI. But there were two AI designs that were very intriguing. The first was Carl Hewitt's PLANNER, a programmable logic system that formed the deductive basis of Winograd's SHRDLU [Sussman 69, Hewitt 69]. I designed several languages based on a combination of the pattern matching schemes of FLEX and PLANNER [Kay 70]. 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 75]. 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]. Unforgable pointers (ala 85000) 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 which 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, etc., weren't. 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 consds—were not functions at all, and instead were called special forms. Landin and others had been able to get quotes and consds 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.

III. 1970-72—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 directly 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 didn't 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 SLOCO 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 didn't 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 Years 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 Engelbart'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:

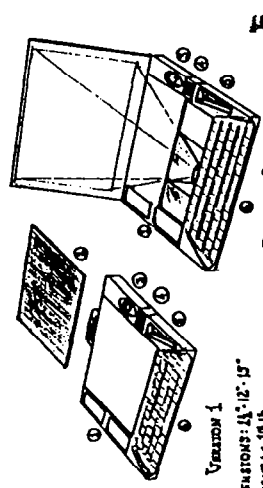
```
oddsEvens(x) = append(odds(x), evens(x))
where odds(x) = if null(x) v null(tl(x)) then x
                else hd(x) & odds(tl(x))
                evens(x) = if null(x) v null(tl(x)) then nil
                          else odds(tl(x))
```

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

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 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 Penderly the head "planner". He really didn't 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 "Penderly 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".

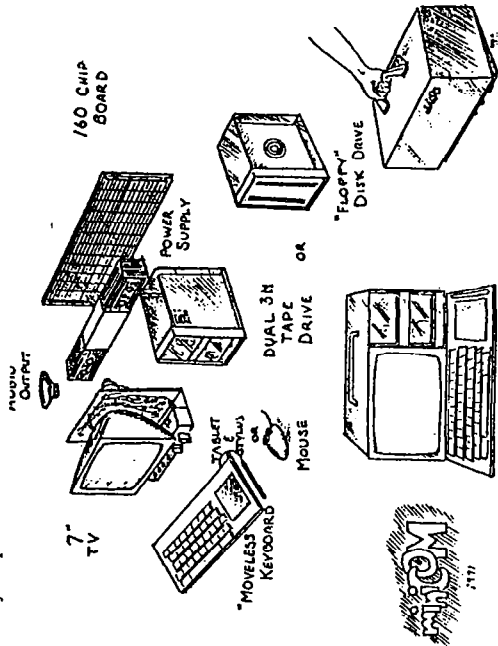
Bill English took me under his wing and helped me start my group as I had always been a lone wolf and had no idea how to do it. One of his suggestions was that I should make a budget. I'm afraid that I really did ask Bill, "What's a budget?". I remembered at Utah, in pre-Mansfield Amendment days, Dave 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 of 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 that got stars in their eyes when they heard about the notebook computer idea. I didn't like meetings: didn't believe brainstorming could substitute for cool sustained thought. When anyone asked me what to do, and I didn't have a strong idea, I would point at the notebook model



- ① Removable local memory
- ② Local hard drive for Xerox
- ③ TV camera (photographic and video)
- ④ Audio input (TV) (not in research)
- ⑤ Stylus for pointing, writing and drawing
- ⑥ Pressure sensitive (thin) light board

and say, "Advance that". LRG members developed a very close relationship with each other—as Dan Ingalls was to say later: "...the rest has unfolded through the love and energy of the whole Learning Research Group". A lot of daytime was spent outside of PARC, playing tennis, bikeriding, 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).

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 "Indo-European 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, LOCO, 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

```
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 'lo fact n do factorial n ^ fact n'
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 'tl' ('cons' a b) do 'b'
to 'r' ('cons' a b) do 'b'
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
```

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 70] 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—ala B5000 and LISP—and a message holding structure—which could be virtual if the senders and receivers were sympatico—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*
- RECIPIENT: *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 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 cliché 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 a thing 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 judgement 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 of 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 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.

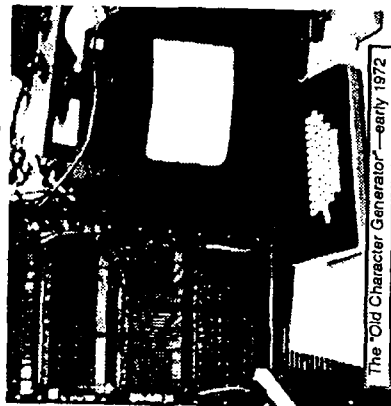
A fertilized egg that can transform itself into the myriad of specializations needed to make a complex organism has parsimony, generality, enlightenment, and finesse—in short, beauty, and a beauty

much more in line with my own esthetics. I mean by this that Nature is wonderful both at elegance and practicality—the cell membrane is partly there to allow useful evolutionary kluges to do their necessary work and still be able to act as component by presenting a uniform interface to the world.

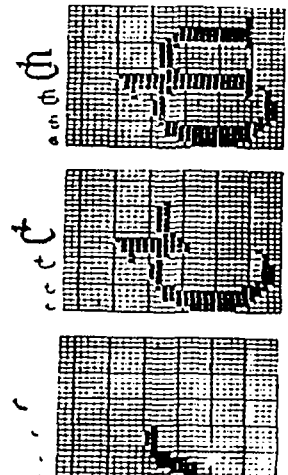
One of my continual worries at this time was about the size of the bit-map display. Even if a mixed mode was used (between fine-grained generated characters and coarse-grained general bit-map for graphics) it would be hard to get enough information on the screen. It occurred to me (in a shower, my favorite 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 PLOLOS (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 "stor machine" (Scanning Laser Output Terminal) was incredible. The only Xerox copier Gary could get to work on went at 1 page a 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 non-linear). I was very interested in high-quality text and graphical presentations because I 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.



The Old Character Generator—early 1970s



Use a Special Font

Things were generally going well all over the lab until May of 72 when I tried to get resources to build a few minicomps. 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 II), 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 4 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 72]) that was integrated with the printing. Bill Duvall of PLOLOS built a minibus 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 execu-



Portrait of the Xerox "HISK" executive



The First Painting System—Summer 72

ive, 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 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.

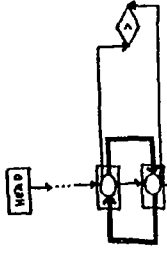
That summer while licking my wounds and getting the demo simulations built and going, Butler Lampton, Peter Deutsch and I worked out a general scheme for emulated HLL machine languages. I liked the B5000 scheme, but Butler did not want to have to decode bytes, and pointed out that since an 8-bit byte 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 subsequent emulators at PARC used this general scheme.

I also took another pass at the language for the kids. Jeff Rullison 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, LOCO (and the current Smalltalk) would be difficult for the kids to process since 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 figurative, more iconic approach. Rudolph Arnhem [Arnhem 69] had written a classic book about visual thinking, and so had the eminent art critic Gombrich [Gombrich **]. It really seemed that something better needed to be done here. CRAL 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 68]. It was kind of a visual SNOBOL. [Farber 63] and the pattern matching ideas looked like they would work for the more PLANNER-like scheme I was using.

Bill English was still encouraging me to do more reasonable appearing things to get higher credibility, like making budgets, 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.



Children with Dynabooks from "A Personal Computer For Children Of All Ages (Ka 72a)"



Iconic Bubble Sort from 1972 LRG Plan (Ka 72a)

*Simple things should be simple, complex things should be possible

IV. 1972-76—The first real Smalltalk (-72), its birth, applications, and improvements

In Sept, 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 CCS. 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 Vitek that he could do a whole machine in just 3 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 bullsession, 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 back 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 Stoch, and Steve Purcell showed up to kibbitz the morning's work.

I had originally made the boast because McCarthy's self-describing LSP interpreter was written in itself. It was about "a page", and as far as power goes, LSP 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 non-recursive 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 which 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 III 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—i.e. 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, etc., and there it was—running. As he like to say: "You just do it and it's done".

It evaluated 3+4 *VERY SLOWLY* (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 73]. In the first Actor paper the resemblance 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 72c] 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 72c]. By the time I got back from Minnesota, Stewart Brand's *Rolling Stone* article about PARC [Brand, 1972] and the surrounding 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 was particularly disastrous for LRC, since 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 defence 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 (506x808) bitmap display, its microcode instruction rate was about 6MIPS, it had a grand total of 128k, 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", etc.). 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 (like 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.

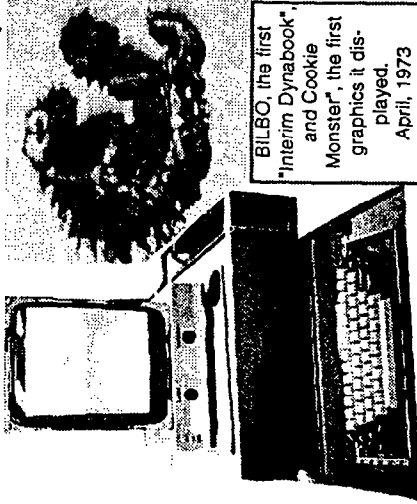
In early April, just a little over three months from the start, the first Interim Dynabook, known as 'Bilbo', greeted the world and we had the first bit-map picture on the screen within minutes: the Muppets' Cookie Monster that I had sketched on our painting system.

Soon Dan had bootstrapped Smalltalk across, and for many months it was the sole software system to run on the Interim Dynabook. Appendix I 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 & 4) imply that classes are objects and that they must be instances of themselves. (6) implies a Lisp-like universal syntax, but with the receiving object as the first item followed by the message. Thus $c_i \leftarrow d_e$ (with subscripting rendered as "o" and multiplication as "x") means:

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

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



receiver | message
c | o i <- d * e

The c is bound to the receiving object, and all of o i <- d * e is the message to it. The message is made up of a literal token "o", 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). Since "Lisp" pairs are made from 2 element objects they can be indexed more simply: c had, c tl, and c hd <- foo, etc. "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 | 1 + b
3 | 1 + 4

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

"kitty" | + "kai" => "kittykat"

3	4	5
6	7	8
9	10	11
12	13	14

 =>

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:

likeCOCO, except makes a class from its message isNEW is true if a new instance has been created true any object not false acts as true true * m n will evaluate m and n escape from surrounding() false * m n will evaluate n	temporary variable to Pair b 1 h t (isNEW * (h .:)) ohd * (o <- * (^h) ^h) otl * (o <- * (t) ^t) oisPair * (^time) oprint * ((print SELF mprint) oimprint * (h print, t isNil * (') print) t isPair * (t mprint) * * print, t print, ') print) olength * (t isPair * (^?+ t, length) 1)	instance variables b is temp, h, t are internal instance vars cons-if no explicit return is given, SELF is returned replica and car replacd and cdr	send-back returns its value to sender in the message stream statement separator value is following message
---	---	---	---

Since 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 "message object" (a generalized activation record) which 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 & 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:
 (... o! * (^: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 gram-

```

Pair h: i
hd <- h
if <- i
if isPair
print
mprint
length
> h
> t
> true
> ' print. SELF mprint.
> h print. if t isNil then ' print
else if t isPair then t mprint
else ' print. t print. ' print.
length
> 1 + if t isLast then t length else 0
  
```

mar-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 USP—that the "reality" of the straightforward and practical syntax you could program in prevailed against the flights of fancy that never quite got built.

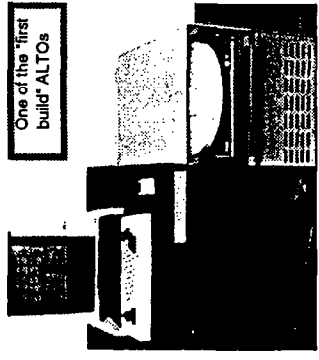
Development Of The Smalltalk-72 System And Applications

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 transition to Smalltalk-76, and then follow that with the several years of work with children that 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.

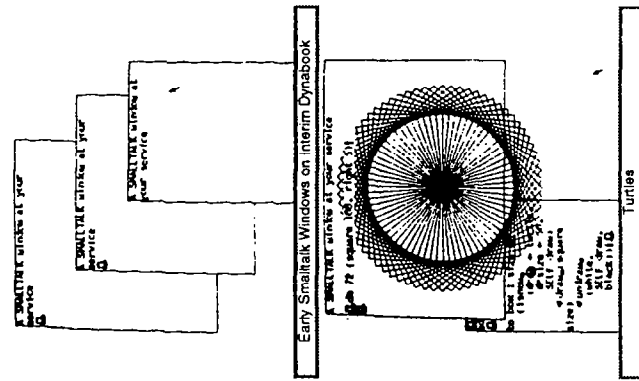
Overlapping windows were the first project tackled (with Diana Merry) after writing the code to read the keyboard and create a string of text. Diana built an early version of a bit field block transfer (bitbit) for displaying variable pitch fonts and generally writing on the display. The first window versions were done as real 21/2D drag-able 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 "21/4D". Windows were perhaps the most redesigned and reimplemented class in Smalltalk because we didn't 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 21/2D. The first practical windows in 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 of the windows together.

One of the next classes to be implemented on the Interim Dynabook (after the basics of numbers, strings, etc.) was an object-oriented version of the *LOCO* 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 Schoch built a mouse-driven structured editor for Smalltalk code.



One of the "first" built ALTOs



Larry Tesler (then working for *ROLOS*) did not like the modish and general approach of NLS, and he wanted both 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 *mini-MOUSE* in Smalltalk, the first real WYSIWYG galley editor at PARC. It was modelless (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 modelless (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 multi-media 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 multi-media documents, whose range was greatly and variously expanded over the years by Bob Fliegel, Diana Merry, Larry Tesler, Jim Mott, and Trygve Reenskaug.

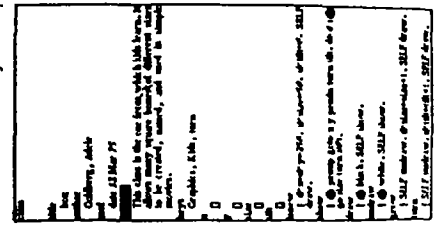
Steve Weyer and I devised *Findit*, a "retrieval by example" interface that used the analogy of classes to their instances to form retrieval requests. This was used for many years by the *PARC* library to control circulation.

The sampling synthesis music I had developed on the *NOVA* could generate 3 high-quality real-time voices. Bob Shur and Chuck Thacker transferred 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 8 real-time voices that were integrated into *TWANG* [Saunders *].

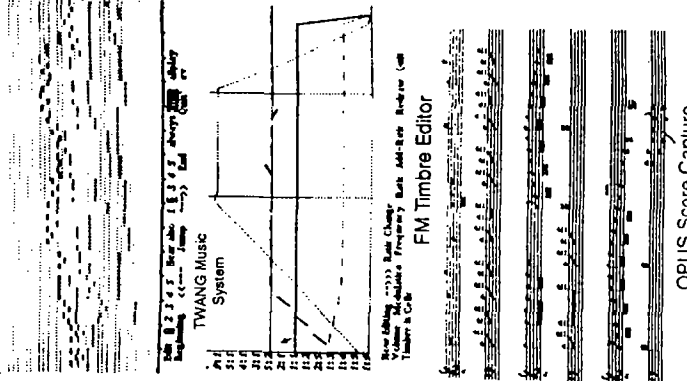
Chris Jeffers (who was a musician and educator, not a computer scientist) knocked us out with *OPUS*, the first real-time score capturing system. Unlike most systems



Findit Retrieval By Example



Retrieved HyperDocument



OPUS Score Capture

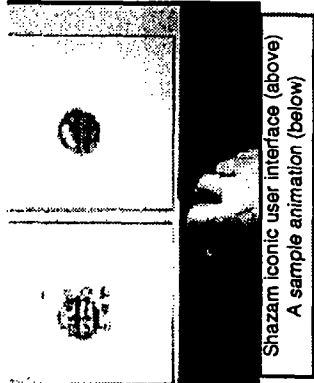
today it did not require metronomic playing but instead took a first pass looking for strong and weak beats (the phrasing) to establish a local model of the likely tempo fluctuations and then used curve fitting and extrapolation to make judgements about just where in the measure, and for what time value, 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 a 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 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 GENESIS thesis project on the TX-2 in the late sixties [Baecker 69].

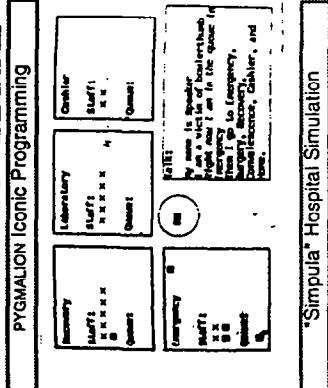
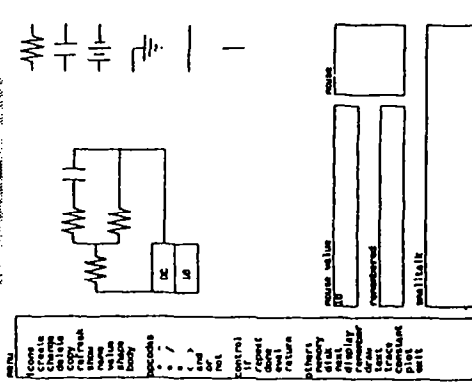
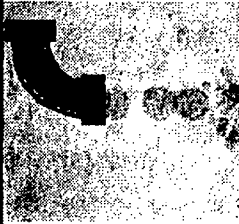
The main thesis project during this time was Dave Smith's PYGMALION [Smith 75], an essay into iconic programming (no, we hadn't quite forgotten). One programmed by showing the system how changes should be made, much as one would illustrate on a blackboard with another programmer. This program became the starting place from which many subsequence 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 of 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 which included its icon-controlled multindowed user interface.

Given its roots in simulation languages, it was easy to write in a few pages. Simpula, a simple version of the SIMULA sequencing set approach to scheduling. By this time we had decided that coroutines could be more cleanly be rendered 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, like Disneyland, their schools, the stores they and their parents shopped in, and so forth. Later this model formed the



Shazam iconic user interface (above)
A sample animation (below)



basis of the smalltalk Sim-kit, a high-level end-user programming environment (described ahead).

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 gotos or not (we didn't), 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.

```
(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')
)
```

The Evolution Of Smalltalk-72

Smalltalk-74 (sometimes known as FastTalk) was a version of Smalltalk-72 incorporating major improvements which included providing a real "messenger" object, message dictionaries for classes (a step towards real class objects), Diana Merry's bitbit (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 OOP. 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 OOSE (Object Oriented Zoned Environment) virtual memory system that served Smalltalk-74, and more importantly, Smalltalk-76 [Ing 78, Kae '7]. 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 BS000 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 of 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 non-working 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 or not 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 insuring a low probability that the object would be purged again soon.

The other problem that had to be taken care of was object-pointer integrity (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 aging. 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, fur-



Adele holding forth at Jordan Middle Sch.

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.

Since we had no idea how to teach object-oriented programming to children (or anyone else), the first experiment Adele did mimicked LOCO 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 seemed to be little happening beyond surface effects. At that time I felt that since the content of personal computing was interactive tools, that 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 with a simple multiprocessing animation. After getting kids to guess what a box might be like—they could come surprisingly close—they would be shown:

to box 1 x y size tilt
 (draw * (@ place x y turn tilt, square size.)
 undraw * (@ white, SELF draw, @ black)
 corner * (SELF undraw, tilt <- tilt + ., SELF draw)
 origin * (SELF undraw, size <- size + ., SELF draw)
 isnew * (SELF undraw, size <- size + ., SELF draw)
 What was so wonderful about this idea were the myriad of 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 Coldeen's (12 yrs old) painting system was a full-fledged tool. A few years later, so was Susan Hamet's (12 yrs old) OOP illustration system (with a design that was like the MacDraw to come). Two more were Bruce Horn's (15 yrs old) music score capture system and Steve Putz's (15 yrs 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 weren't as general as we thought. They wouldn't 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 pursuit, a particular 5% of the population will jump into it naturally, while 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

over the ALTOS with Adele's JOE book. The problem

ished 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 they came up with. This insured that there was a recoverable image no more than a few seconds old, regardless of when a crash might occur. OOOE 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).

"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 60s was concerned with generalizing such structures (DSP '1). 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, since 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 (cf. 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, etc. 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.

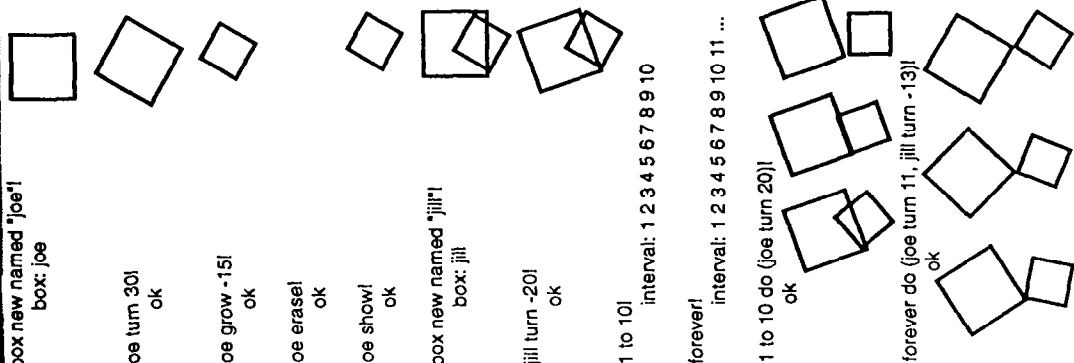
Where does the special efficiency of object-oriented design come from? This is a good question given that it can be viewed as a slightly different way to apply procedures to data-structures. Part of the effect comes from a much clearer way to represent a complex system. Here, the constraints are as useful as the generalities. Four techniques used together—persistent state, polymorphism, instantiation, and methods-as-goals for the object—account for much of the power. None of these require an "object-oriented language" to be employed—ALCOL 68 can almost be turned to this style—an OOP merely focuses the designer's mind in a particularly 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 isn't even close to what's 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, 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 doesn't do anything a programmer can't do, its presence leads both to 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 aren't Turing machines—and the less their programming systems require Turing machine techniques the better.

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 chil-



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 didn't look to me to be much harder than the ones they had just been doing well on. 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 couldn't 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 we were working within. 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 don't already know them.

The connection to literacy was painfully clear. It isn't 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 70s]. 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 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,

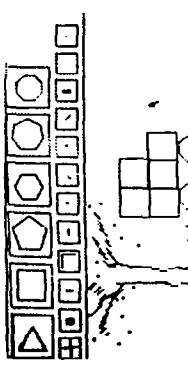


The author in the interim Dynabook playground. Working with the kids was my favorite part of this Romance

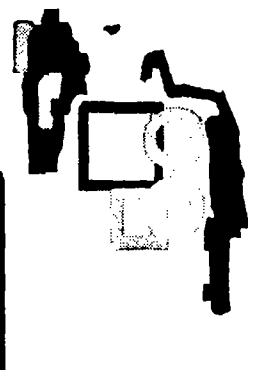
Message	English description of the action	Smalltalk description
make	To create a new box that needs to work with the other boxes in the system. The box is created in the system. The length of the box is determined by the user.	newBox := Box new
add	The box has to get other objects to work with it. The box is added to the system. The length of the box is determined by the user.	addBox := Box new
erase	After erasing itself, the box has to be removed from the system. The box is removed from the system. The length of the box is determined by the user.	eraseBox := Box new
print	After printing itself, the box has to be removed from the system. The box is removed from the system. The length of the box is determined by the user.	printBox := Box new
write	After writing itself, the box has to be removed from the system. The box is removed from the system. The length of the box is determined by the user.	writeBox := Box new

Adele's planning template for Smalltalk (above)
New behavior added by child (below)

Message	English description of the action	Smalltalk description
make	After making itself, the box has to be removed from the system. The box is removed from the system. The length of the box is determined by the user.	newBox := Box new
add	After adding itself, the box has to be removed from the system. The box is removed from the system. The length of the box is determined by the user.	addBox := Box new
erase	After erasing itself, the box has to be removed from the system. The box is removed from the system. The length of the box is determined by the user.	eraseBox := Box new
print	After printing itself, the box has to be removed from the system. The box is removed from the system. The length of the box is determined by the user.	printBox := Box new
write	After writing itself, the box has to be removed from the system. The box is removed from the system. The length of the box is determined by the user.	writeBox := Box new



Marion Golden's painting program (above)
Susan Harrel's OO instructor (below)



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'm now pretty much convinced that our design template approach was a good one after all. We just didn't 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 looking ahead. 41

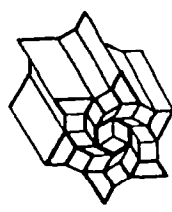
The problem is not to get the kids to do stuff—they love to do, even when they are not sure exactly what they are doing. This correlates well with studies of early learning of language, when much rehearsal is done regardless of whether content is involved. Just doing seems to help. What is difficult is to determine what ideas to put forth and how deeply they should penetrate at a given child's developmental level. This is a 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, 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.

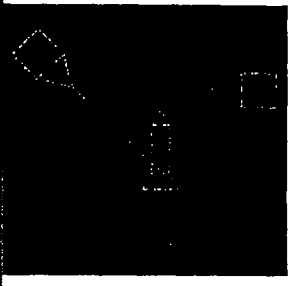
I have also met a few people for whom computing provides an important new metaphor for thinking about human knowledge and reach. But something else was needed besides computing for enlightenment to happen. Tools provide a path, a context, and almost an excuse for developing enlightenment, but no tool ever contained it or can dispense it. Cesare Pavese observed: to know the



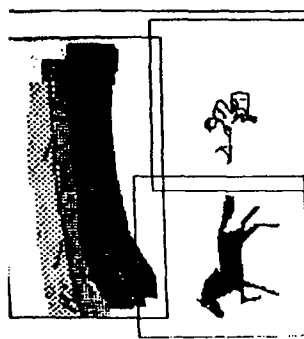
Circuit design system by Steve Putz (age 15)



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 77)



SpaceWar by Dennis (age 12)



SHAZAM modified to "group" multiple images by Lisa van Stone (age 12)

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 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 disappear the interpretations of the representations. The letters and words of a sentence are experienced as meaning rather than markings, the tennis racket 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 towards 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 education 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 defence 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'm 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. 3rd 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 up transformations in two and threes in a manner similar to learning a strategic game like checkers. The vague sense of a "three-some" pointing towards one's goal can be a set up 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.

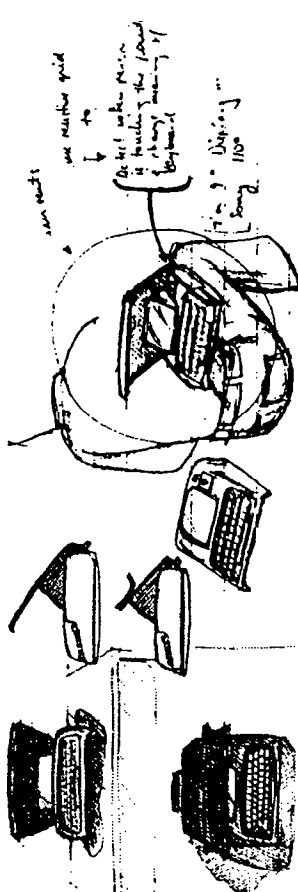
V. 1976-80—The first modern Smalltalk (-76), its birth, applications, and improvements

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

reilly fresh start: a 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 reformed. 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've 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 like 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 enduser 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 couldn't use a floppy (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 (in 78). 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 were 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", etc. We were all agreed that the flexible syntax of the earlier Smalltalks was too flexible, and this level of extensibility was not desirable. All of 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

Inheritance

A word about inheritance. Simula-1 had neither classes as objects nor inheritance. Simula-67 added the later as a generalization to the ALCOU-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, 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" (i.e. 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, since 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 Lisp-like flexibility. The biggest contributor 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 84]. 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*—which 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 & 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, etc.). Alan Borning employed a multiple inheritance scheme in Thinglab [Borning 77] 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 Pendency 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 *Altix*. 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 chat-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 continuously 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 to 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 are 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—seven I, 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 main-frame 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.

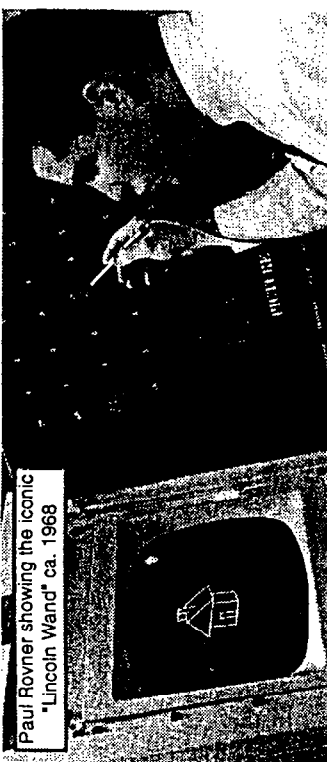
The Smalltalk User Interface

I have been asked by several of the reviewers to say more about the development of the "Smalltalk-style" overlapping window user interface since there are now more than 20 million computers in the world that use its descendants. 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 89]—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 corp—both ARPA funded. The big shift that consolidated these ideas into a powerful theory and long-lived examples came because the LRC 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 LRC 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 peoples' work and creativity. He was interested in embedding the new computer magic in familiar 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

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



'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 & 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 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, etc., 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 [Smith 83].

Meanwhile, I had gotten Doug Fairbairn interested in the *NoteLaker*. 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 couldn't 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.

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 of 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 runtime environment. In every way it was the consolidation of all of our 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.

Class new title: 'Window';
 fields: 'frame';
 asfollow:
 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.

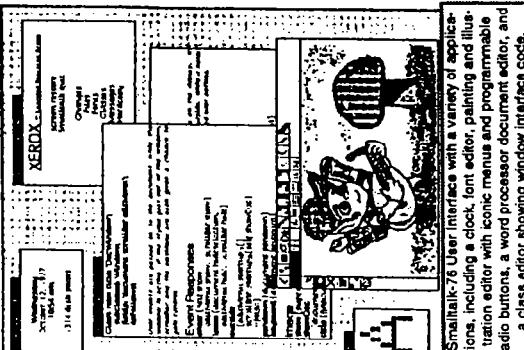
Scheduling startup
 [frame contains: stylus => self enter.
 repeat
 [frame contains: stylus loc => keyboard active => [self keyboard] stylus down => [self pendown]] self outside => [self leave]] ^false

Default Event Responses
 enter [self show]
 leave outside (^ false)
 keyboard [keyboard next: frame flash] image show
 [frame outline: 2
 titleframe put: self title at: frame origin + title loc.
 titleframe complement]
 ... etc.

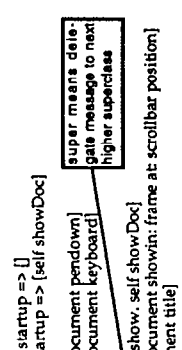
Class new title: 'DocWindow';
 subclassof: Window;
 fields: 'document scrollbar editMenu';
 asfollow:
 User centers are passed on to the document while the window is active. If the stylus goes out of the window, scrollbar and the editMenu are each given a chance to gain control.

Event Responses
 enter [self show: editMenu show: scrollbar show]
 leave [document hideSelection: editMenu hide: scrollbar hide] outside
 [editMenu startup => [scrollbar startup => [self showDoc] ^false]
 pendown [document pendown] keyboard [document keyboard] image [super means delete message to next higher superclass]
 show [super show: self showDoc] showDoc [document showin: frame at: scrollbar position] title [^document title]

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 user 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 which will be intercepted by the subclass method. If there is no method the character will be thrown away and the window will



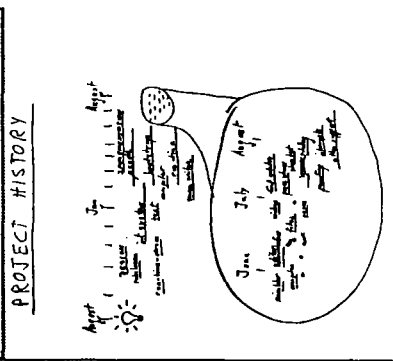
Smalltalk-76 User Interface with a variety of applications, including a clock, font editor, painting and illustration editor with iconic menus and programmable radio buttons, a word processor document editor, and a class editor showing window interface code.



Smalltalk-76 Metaobjects



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



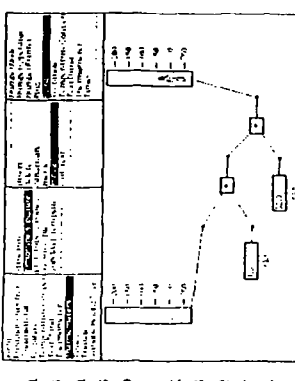
Jack Goldman finally uses the system he paid for all those years (with Alan Borning helping)

End-user simulation by a Xerox executive in SimKit. Total time including training: 3 hours. Includes a table with columns for 'Task', 'Time', and 'Status'.

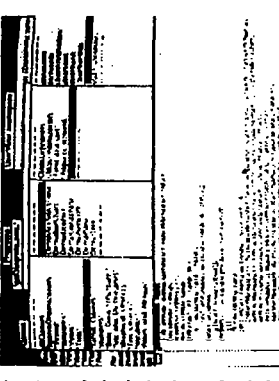
flash. In this case, it finds DocWindow method: keyboard, which tells the held document to check it out. In January of 1978 Smalltalk-76 had its first real test. CSL had invited the top ten executives of Xerox to PARC for a two day seminar on software, with a special emphasis on complexity and what could be done about it. LRC 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 triumphs). 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, like 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 small fonts our thirtysomething year-old eyes were used to didn't work for those in their 50s. 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.

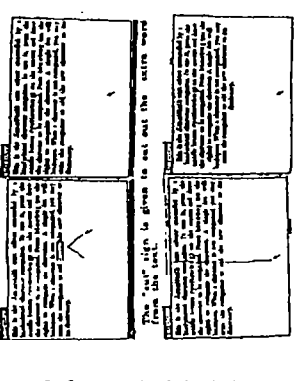
On the morning of the "big day" Ted Kaehler decided to make a change in the virtual memory system COZE to speed it up a little. We all held our breaths, but such was the clarity of the design and the confidence of the implementers that it did work, and the executive hands-on was a howling success. About 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



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



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



The author's pen-based interface for ST-76



Doug Fairbairn using his NoteTaker

the line's average probability of manufacturing defects. Another important system done at this time was Alan Borning's Thinglab [Borning,1979]—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 relaxed 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 2nd order models).

Meanwhile, the NoteTaker was getting realer, 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 didn't 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, etc).

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 rewritten in Smalltalk and the total machine kernel was reduced to 6K bytes of the not very strong) 8086 code.

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

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

VI, 1980-83—The release version of Smalltalk (-80)

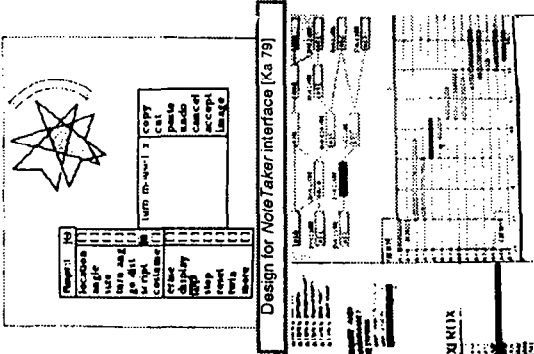
As Dan said "the decision not to continue the NoteTaker project added motivation to release Smalltalk widely". But not for me. By this time I was both happy about the cleanliness and elegance of the Smalltalk conception as realized by Dan and the others, and sad that it was farther away than ever from 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, 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 which, 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 strange 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" like Class Class and Class Object are not really necessary since 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 wasn't 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.

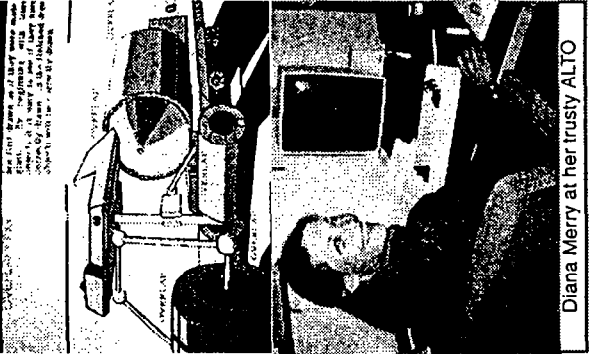
Coda

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, migration processes, and so forth. Time-sharing was held back for years because it was "inefficient"—but the manufacturers wouldn't 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 crus. Most machines still have no support for dynamic allocation and garbage



What Steve Jobs saw. Multiviews on complex structures by Trygve Reenskaug (above) Multimedia documents by BobFlegal and Diana Merry (below)



Diana Merry at her trusty ALTO

sated" (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 70's when 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 weren't 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 teen-ager. 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 didn't like the bit-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 would come up with the resources to continue to develop it in house by funding a better NoteTaker cum Smalltalk.

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, etc., 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 aren't 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 questions 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>*. Since 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 *iscompatibility* methods—a good discussion of some of the issues is found in [Popok,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, etc., 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, CDI, 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 doesn't need bindings to values in order to carry out computations [Col **]. 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 [Lenat **] 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 configure 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 [Guttag **] [Goguen **].

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 respects.

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 83]. 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's a pity that we didn't 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 that made Smalltalk possible, we must wonder at the same time: where are the Dans and Adeles of the '80s and '90s) that will take us to the next stage?



Dave Robson



References Cited In The Text

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

[Arnheim, 1969] Arnheim, Rudolf, *Visual Thinking*. Berkeley: University of California Press, 1969, ISBN 0520013786.

[Balzer, 1967] Balzer, R.M., *Database programming*. *Proceedings of the FCC*, 1967.

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

[Baecher, 1969] Baecher, Ronald M., *Interactive computer-mediated animation*, Dept. of Electrical Engineering, PhD thesis, MIT, 1969, Supervisor: Edward L. Glaeser.

[Bitzer, 1966] Bitzer, D.L., and Slotow, H.G., *The plasma display panel — a digitally addressable display with inherent memory*, in *Proceedings of the FCC*, November 1966.

[Bobrow, 1977] Bobrow, D.C., and Winograd, T., *An overview of KRL, a knowledge representation language*, in *Cognitive Science*, Vol. 1, (1) (pp. 3-46), 1977.

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

[Bruner, 1960] Bruner, Jerome S., *The Process of Education*, Harvard/Belknap Press, 1960.

[Bruner 1966] Bruner, R.M., *Database programming*, Harvard/Belknap Press, 1966, ISBN 0-674-89700-5.

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

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

[Bush, 1945] Bush, Vannevar, 1945, *A scientist looks at tomorrow as we may think*, *Atlantic Monthly*, Vol 176, No. 1, (p. 101), July 1945.

[Byte, 1981] *Byte Magazine*, Issue on Smalltalk, Christopher Morgan, ed., Volume 6, number 8, August, 1981.

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

[Colmerauer, 1978] Colmerauer, Alain., *Metamorphosis grammars, in Natural Language Communication with Computers*. Bolc, L., ed., (pp. 133-189), West Germany: Springer-Verlag, 1978, ISBN: 3 540 08911 X.

[Colmerauer, 1981] _____ et al., *Last steps towards an ultimate PROLOG*, in *Proceedings of the 7th International Joint Conference on Artificial Intelligence*, Vol. 2, (pp. 947-948), August 1981. Available from the American Association for Artificial Intelligence, Menlo Park, CA.

[Colmerauer, 1983] _____ PROLOG in 10 figures, in *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, Vol. 1, Distributed by William Kaufmann Inc, Los Altos, CA. (pp. 487-499), August 1983.

[Clark, 1957] Clark, Wesley A., *The Lincoln tx-2 computer development*, in *Proceedings of the WJCC*, (pp. 143-145), February 1957.

[Clark, 1962] _____ *The General Purpose Computer in the Life Sciences Laboratory*, in *Engineering and the Life Sciences*, NAS-NRC Report, Washington DC, April 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., Academic Press, New York, 1965.

[Clark, 1966] _____ *Programming the LINC*, Computer-Systems Lab, Washington University, St. Louis, Technical Report, 1966.

[Clark, 1988] _____ *The LINC was early and small*, in *A History of Personal Workstations*, Adele Goldberg, ed., New York: New York, ACM Press, (pp. 347-391), 1988, ISBN 020 111 2590.

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

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

[Dahl, 1972] Dahl, O.-J., and Hoare, C.A.R., *Hierarchical Program Structure*. In Dah

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

[Deutsch, 1973] _____ *A lisp machine with very compact programs*, in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, Stanford, CA, 1973.

[Deutsch, 1983] _____ *The dorado Smalltalk-80 implementation: hardware architecture's impact on software architecture*, in *Smalltalk-80 Bits of History, Words of Advice*, Krasner, C., ed., Addison-Wesley, (pp. 113-126), 1983.

[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 FCC*, Vol. 33, Part one, (pp. 395-410), December 1968.

[Farber, 1964] Farber, D.J., Griswald, R.E., Polensky, F.P., *"SNOBOL, a String Manipulation Language"* JACM 11, 1964, 21-30.

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

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

[Goldberg, 1977] Goldberg, Adele and Kay, Alan C., *Teaching Smalltalk (2 papers): Methods for teaching the programming language Smalltalk and Smalltalk in the classroom*, Xerox Palo Alto Research Center, June 1977.

[Goldberg, 1978] _____ *Smalltalk simulation kit documentation*, Xerox Palo Alto Research Center, LRG Internal Note, Feb 1978.

[Goldberg, 1983] _____ and Robson, D., *Smalltalk-80: The Language and its Implementation*. Addison Wesley, Reading, Mass., 1983.

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

[Groner, 1966] Groner, Gabriel, *Real-time recognition of hand printed text*, CA: RAND, Report #RM-5016-ARP A, October 1966.

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

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

[Hewitt, 1977] _____ and Baker, Henry Jr., *Actors in continuous functionals*, Cambridge, MA, MIT, Laboratory for Computer Sciences, 1977, MIT/LCS/TR-194, MIT, Laboratory for Computer Sciences, Technical Report #194.

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

[Ingalls, 1981] _____ *The smalltalk graphics kernel*, *Byte*, Vol. 6, Number 8, (p. 168), August, 1981.

[Ingalls, 1983] _____ *The evolution of the smalltalk virtual machine*, in *Smalltalk-80 Bits of History, Words of Advice*, Krasner, G., ed., Addison-Wesley, (pp 9-28), 1983.

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

[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. ISBN: 0127450408. Conference: History of Programming Languages Conference (1978: Los Angeles, Calif.)

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

[Kay, 1969] Kay, Alan C., *Flex: a flexible extensible language*, M.S. thesis, University of Utah, May 1968.

[Kay, 1970] _____ *The reactive engine*, PhD thesis, University of Utah, September 1969.

[Kay, 1971] _____ *Ramblyng towards a KiddiKomp*, in *Stanford AI Project Lab Notebook*, November 1970.

[Kay, 1971a] _____ *Display transducers*, in *Pendery Papers for Parc Planning Purposes*, Xerox Palo Alto Research Center, June 1971.

[Kay, 1971b] _____ *Draft design for miniCOM*, in *PARC Lab Book*, Xerox Palo Alto Research Center, August 1971.

[Kay, 1972] _____ *Computer Structures-Past Present and Future*, Panel paper, in *Proceedings of the FCC*, Vol. 39, November 1971

[Kay, 1972a] _____ *MiniCOM proposal*, in *PARC Lab Book*, Xerox Palo Alto Research Center, May 1972.

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

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

[Kay, 1972d] _____ *A dynamic medium for creative thought*, in *Proceedings of the National Council of Teachers of English Conference*, Minneapolis, November 1972.

[Kay, 1976] _____ *Smalltalk Blue Book*, Fall 1972.

[Kay, 1976] _____, Goldberg, Adele., ed., *Smalltalk Instruction Manual*, SSL-76-6, May 1976.

[Kay, 1977]	_____ 1977, Microelectronics and the personal computer, <i>Scientific American</i> , (pp. 125-136) September 1977.	[Papert, 1976]	_____ Abelson, H., Bamberger, J. and Goldstein, I., LOGO Progress Report 1973-1975, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 356, LOGO Memo 22, 1976.
[Kay, 1977a]	_____ and Goldberg, Adele., Personal dynamic media, <i>IEEE Computer</i> , Vol. 10, (pp. 31-41), March 1977. Reprinted in <i>A History of Personal Workstations</i> , Academic Press, 1988.	[Papert, 1976]	_____ Proposal to the National Science Foundation: An Evaluative Study of Modern Technology in Education, Appendix One: LOGO memo 8, Appendix Two: LOGO memo 77, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory, memo 371, LOGO memo 26, 1976.
[Kay, 1979]	_____ Programming your own computer, <i>Science Year 1979, World Book Encyclopedia</i> , 1979.	[Papert, 1976a]	_____ ; Solomon, C.I., A Case Study of a Young Child doing Turtle Graphics in LOGO, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 375, LOGO Memo 28, 1976.
[Kay, 1984]	_____ 1984, Computer software, <i>Scientific American</i> , September 1984.	[Papert, 1979]	_____ ; Wait, D., DiSessa, A., Weir, S., Final Report of the Brookline LOGO Project. Part II: Project Summary and Data Analysis, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory Memo 545, LOGO Memo 53, 1979.
[Kay, 1990]	_____ User interface: a personal view, in <i>The Art of Human-Computer Interface Design</i> , ed., Brenda Laurel, Addison-Wesley Publishing Co., 1990 (pp. 191-207) ISBN 0 201 51797 3.	[Perry, 1985]	Perry, Tekla, "Inside the PARC: the 'Information Architects'" IEEE Spectrum, October 1985.
[Kay, 1991]	_____ 1991, Computers, networks, and learning, <i>Scientific American</i> , Vol. 265, No. 3, (pp. 138-148) September 1991.	[Plato]	Plato, Timaeus & Phaedrus: <i>The Dialogues of Plato</i> , translated by Benjamin Jowett, Great Books of the Western World, Robert Maynard Hutchins, ed., Encyclopedia Britannica, Inc., 1951.
[Kiczales, 1991]	Kiczales, Gregor, Des Rivieres, Jim; Bobrow, Daniel G., <i>The Art of the Metaobject Protocol</i> , Cambridge, Mass.: MIT Press, viii, 335 pp.: 1991, ISBN 0262111586.	[Popek, 1984]	Popek, G., et. al., <i>The Locus Distributed Operating System</i> , Cambridge: MIT Press, 1984.
[Knuth, 1971]	Knuth, Donald E and Floyd, Robert W., Notes on avoiding 'go to' statements, in <i>Information Processing Letters</i> , volume 1, number 1, February 1971.	[Ross, 1960]	Ross, D.T., and Ward, J.E., Picture and pushbutton languages, chapter 8 of <i>Investigations in Computer-Aided Design</i> , internem engineering report 8436-IR-1, Electrical Systems Lab., MIT, May 1960.
[Knuth, 1974]	_____ Structured programming with 'go to' statements, in <i>ACM Computing Surveys</i> , vol. 6, no. 4, (pp. 261-301), December 1974.	[Ross, 1961]	_____ A generalized technique for symbol manipulation and numerical calculation, in <i>Communications of the ACM</i> , Vol. 4, no. 3, (pp. 147-150) March 1961.
[Krasner, 1983]	Krasner, Glenn, ed., <i>Smalltalk-80: Bits of History, Words of Advice...</i> , Addison-Wesley, 1983, ISBN 0 201 116 69 3.	[Rover, 1968]	Rover, P.D., An AMBIT/G programming language implementation, MIT Lincoln Laboratory, Lexington, Mass., June 1968.
[Lampson, 1966]	Lampson, J., CAL reference manual, Project GENIE documentation, Computer Center, UC Berkeley, 1966.	[Saunders, 1977]	Saunders, Steven E., Improved FM audio synthesis methods for real-time digital music generation, in <i>Computer Music Journal</i> , Vol. 1, No. 1, February, 1977. Reprinted in <i>Computer Music</i> , Roads, C. and Strawn, J., editors, Cambridge: MIT Press, 1985.
[Lampson, 1966a]	_____ A user machine in a time sharing system, in <i>Proceedings of the IEEE</i> , 54(12): (pp.1744-1766), December 1966.	[Schorre, 1963]	Shorre, D.V., META II— A syntax-oriented compiler writing language, UCLA computing facility, Shoch, J.F., 1979, An overview of the programming language Smalltalk-72, in <i>SIGPLAN Notices</i> , vol. 14, no. 9, (pp. 64-73), September 1979.
[Lampson, 1969]	_____ An overview of the CAL time-sharing system, Computer Center, U.C. Berkeley, September 1969. Originally entitled On reliable and extendable operating systems, September 5, 1969.	[Shoch, 1979]	Shoch, J.F., 1979, An overview of the programming language Smalltalk-72, in <i>SIGPLAN Notices</i> , vol. 14, no. 9, (pp. 64-73), September 1979.
[Lampson, 1972]	_____ Why Alto?, in <i>PARC Lab Book</i> , Xerox Palo Alto Research Center.	[Soloway, 1989]	Soloway, Elliot and Spohrer, James C., ed., <i>Studying the Notion Programmer</i> , New Jersey: Lawrence Erlbaum Associates, Inc., 1989, ISBN 0-8058-002.
[Lampson, 1988]	_____ Personal distributed computing: alto and ethernet software, in <i>A History of Personal Workstations</i> , Adele Goldberg, ed., New York: New York, ACM Press, 1988, ISBN 020 2590.	[Smith, 1975]	Smith, David Canfield, <i>Pigmalion</i> , PhD thesis, Stanford Univ., 1975
[Landin, 1965]	Landin, P.J., A correspondence between ALGOL 60 and Church's lambda notation: Part 1, in <i>Communications of the ACM</i> , Vol. 8, No. 2, February 1965.	[Strachey]	Strachey, Christopher, <i>Toward a formal semantics</i> , United Kingdom.
[Landin, 1966]	_____ The next 700 programming languages, in <i>Communications of the ACM</i> , Vol. 9, No. 3, March 1966. (pp. 157-164).	[Sutherland, 1963]	Sutherland, Ivan C., Sketchpad: A man-machine graphical communication system, MIT Lincoln Laboratory, Technical Report 296, January 1963.
[Licklider, 1960]	Licklider, J.C.R., Man-computer symbiosis, in <i>IRE Transactions on Human Factors in Electronics</i> , HFE-1: 4-11, 1960.	[Sutherland, 1963a]	_____ ibid., in <i>Proceedings of the SJCC</i> , Vol. 23, (pp. 329-346), 1963.
[LRG, 1976]	Learning Research Group, Dynamic Personal Media, Xerox Palo Alto Research Center, Report #SSL-76-7, June 1976.	[Sutherland, 1968]	_____ A head-mounted three dimensional display, in <i>Proceedings of the FCC</i> , (p. 757), 1968.
[McCarthy, 1960]	McCarthy, John P., Part 1, Recursive functions of symbolic expressions and their computation by machine, in <i>Communications of the ACM</i> , Vol. 3, Number 4, (pp. 184-195) April 1960.	[Tesler, 1973]	Tesler, Lawrence, et al., The lisp-70 pattern matcher, in <i>Proceedings of the 3rd International Joint Conference on Artificial Intelligence</i> , Stanford, CA, 1973.
[McCarthy, 1962]	_____ et al., <i>LISP 1.5 Programmer's Manual</i> , Cambridge: MIT Press, 1962.	[Tesler, 1977]	_____ Smalltalk-76 documentation, Xerox Palo Alto Research Center, Learning Research Group Internal Note, 1977.
[Minsky, 1970]	Minsky, Marvin., Form and content in computer science, in <i>The Journal of the Association for Computing Machinery</i> , Vol 17, Number 2, (pp. 197-215), April 1970.	[Tesler, 1981]	_____ 1981, The smalltalk environment, Byte, Vol. 6, Number 8, (p. 90), August, 1981.
[Minsky, 1974]	_____ A framework for representing knowledge, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory Memo No. 306, June 1974. Reprinted in <i>The Psychology of Computer Vision</i> , McGraw-Hill, 1975.	[Thacker, 1972]	Thacker, C.P., A personal computer with microparallel processing, Xerox Palo Alto Research Center, December 1972.
[Newman, 1973]	Newman W.M. and Sproull, R.F., <i>Principles of interactive computer graphics</i> , New York: McGraw-Hill, 1973.	[Thacker 1982]	_____ et al., <i>Alto: a personal computer</i> , in <i>Computer Structures: Principles and Examples</i> , Siewiorek, D. et al. editors, Chapter 33, McGraw-Hill, 1982.
[Nygaard, 1966]	Nygaard, Kristen, and Dahl, Ole-Johan, Simula — an ALGOL-based simulation language, in <i>Communications of the ACM</i> , IX, 9, (pp. 671-678), September 1966.	[Thacker, 1986]	_____ Personal distributed computing: the alto and ethernet hardware, in <i>A History of Personal Workstations</i> , Adele Goldberg, ed., New York: New York, ACM Press, (pp.267-290), 1988, ISBN 020 111 2590.
[Nygaard, 1978]	_____ Early history of simula, in <i>History of Programming Languages</i> , ed. Richard L. Wexelblatt, New York: Academic Press, 1981, ISBN 012 745040 8. This is the proceedings of the ACM Sigplan History of Programming Languages Conference held in Los Angeles, June 1-3, 1978.	[Van Wijngaarden, 1968]	_____ ; Van Wijngaarden, A., ed., Draft report on ALGOL 68, Mathematisch Centrum, MR 93, Amsterdam, The Netherlands, 1968.
[Papert, 1971]	Papert, S., Teaching children to be mathematicians vs. teaching about mathematics, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory Memo 247, LOGO Memo 2, 1971.	[Van Wijngaarden]	_____ Generalized ALGOL, Mathematisch Centrum, Amsterdam, Netherlands.
[Papert, 1971a]	_____ Teaching children to be mathematicians vs. teaching about mathematics, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory Memo 249, LOGO Memo 4, 1971.	[With, 1966]	With, N.K. and Weber, H., EULER: A generalization of ALGOL, and its formal definition: Part I, in <i>Communications of the ACM</i> , Vol 9, No. 1, (pp. 13-25), Jan. 1966
[Papert, 1973]	_____ Uses of technology to enhance education, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory Memo 298, LOGO Memo 8, 1973.	[Winston, 1970]	Winston, Patrick H., Learning structural descriptions from examples, PhD thesis, MIT, January 1970.
		[Zahn, 1974]	Zahn, C.T., Jr., A control statement for natural top-down structured programming, in <i>Proceedings of the Colloque sur la Programmation</i> , April 1974, Paris. A revised version of this paper appears, under the same title, in <i>Programming Symposium</i> , vol. 19 of the lecture notes in Computer Science, Robinet, B., ed., Berlin: Springer Verlag, 1974, (pp. 170-180).

Appendix I: Personal Computer Memo

Smalltalk Program Evolution

From a memo on the "KiddiKomputer"

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

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 and 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 beginning 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" (a 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 1969-70 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 words 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 preserve 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 test bed 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 disply 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 need 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 LD AJ or worse: dumping structured 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*, etc., 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 find 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 accessible by providing a protected way to access and change the relationships of the static and dynamic environment (Fisher 70). 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 metaphysics.

I also decided to ignore the metaphysics of objects even though it was clear that, unlike Simula, in a full blown OOP, 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 did. I 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 didn't want to use any of my precious "half-pages" 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.²²

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 *cj <- 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 *j <- 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 | a | i | c | d * e
```

The message is made up of a literal token "*a*", 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 2 element objects and can be indexed more simply: *c hd, c tl*, and *c hd <- foo*, etc.

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

```
receiver | message
3 | 1+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 | 1+4+5...*, and that the way class number chooses to receive messages should be arranged so that the next subexpression is handled properly. E.g. 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 PROlike 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: This also means that useful elements like lists, atoms, control structures, quote, receivers (such as "receive evalu-

- message receipt would be done by objects in the midst of normal code
- control structures would be handled by objects that could access the context objects
- the context objects (that acted like stack frames, schedulers, etc.) 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 class 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

ated", "is the next token this?", etc.), 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 what a message send actually consisted of. 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, etc.

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

```
I hope this is clear enough. For example,
if c is bound to a cons pair,
c hd <- 3+4
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 (=). If's 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:
(REPLAC 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 evaluating 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. Since 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.

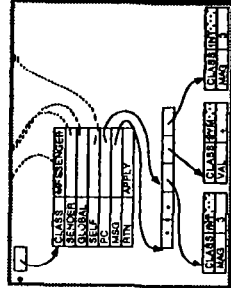
eyeball	looks to see if its message is a literal token in the message stream
eval-bind	evals the next part of message and binds result to its message
unval-bind	picks up next part of message and binds to its message
send-back	returns its value to the sender
quote	overrides any misinterpretation of its message

The above were used in the first interpreter definition. The following were defined when the first "real" implementation was done.

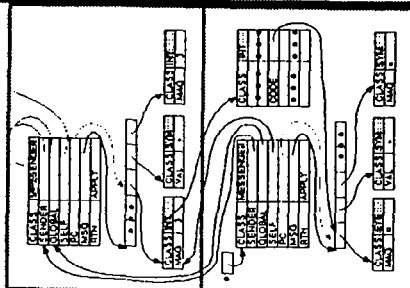
to define	like (code), except can make a class from its message
isnew	is true if a new instance has been created
equals	true only if its receiver and parameter are the same object
then	receiver=true: evals next part of message and this
receiver=false:	skips over the next part of message and continues evaluating
separ	"statement" separator. Quits applying its receiver; starts evaluating its arg

The "One Pager"

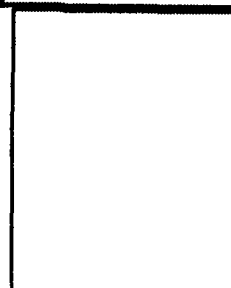
"Before"



"During"



"After"



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

```

eval:
if null(e+MSC) then result <- nil; goto apply;
if escape(e+msg) then goto escape;
if atom(e+MSC) then result <- lookupvalue(e, e+MSC); goto apply;
if notlist(e+MSC) then result <- e+MSC; 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+evlist;
e+PC <- e+PC + 1;
if e+PC > length(MSC) then goto dispatch;
if e+MSC+PC = '
then if result = false
then e+PC <- e+PC + 1; goto evlist;
else e+PC <- e+PC + 1;
    'e <- Table(CLASS, MESSENGER,
        SENDER, e,
        GLOBAL, GLOBAL,
        SELF, SELF,
        PC, 1,
        MSG, e+MSC+PC,
        RTN, FROMTRUE);
        goto eval;

'e <- Table(
    CLASS, MESSENGER,
    SENDER, e,
    GLOBAL, GLOBAL,
    SELF, result,
    PC, 1,
    MSG, result+CLASS+CODE,
    RTN, APPLY);

goto eval;

fromTRUE: 'e <- e+SENDER+SENDER; goto dispatch;
fromEVE: putvalue(e+GLOBAL, e+PC, result); goto apply;

dispatch: select e+RTN
    case APPLY:
        case FROMTRUE:
            case FROMEVE:
                goto apply;
                goto fromTRUE;
                goto fromEVE;
                etc....

escape: select e+MSC+PC+1
    to 0 (metacode) if (e+SENDER+SENDER+SENDER+SENDER+SENDER+SENDER)
        then but(e+SENDER+SENDER+SENDER+SENDER+SENDER+SENDER); result <- TRUE
        else result <- FALSE;
        goto apply;
    to 1 p (p, metacode) if set up a new context and eval sender;
    to 1 p p (metacode) if v <- e+SENDER+SENDER+SENDER+SENDER+SENDER+SENDER;
        if null(e+p) <- e+SENDER+SENDER+SENDER+SENDER+SENDER+SENDER;
        then result <- v;
        else e+p <- result <- v;
        goto apply;
    to ^ b (b, metacode) if return <- e+b; goto apply;
    
```

Appendix III: Acknowledgements

- 1971 Chris Jeffers, + ?
- 1972 Chris Jeffers, John Shoch, Steve Purcell, Bob Shur, Bonny Temmenbaum, Barbara Deutsch
- 1973

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

A C K N O W L E D G E M E N T S
 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 ALPO (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 well-known 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 processes 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 Aita 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 "left-hand calls" prompted SMALLTALK to make "store" intentional --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 intentional. 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 eval 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 preceding acknowledgements it can be seen as a consolidation of good ideas into one simple idea:

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 with his help, virtuosity, and good cheer.

The original design of the "painting editor" was by Alan Kay. It was implemented 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 Schoch.

"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
Dan Ingalls
Ted Kaehler
Dave Robson
Dick Shoup

Students

Tom Horsley
Steve Saunders
David C. Smith

Child Interns

Marian Goldeen (age 13)
Bruce Horn (age 15)
Kathy Mansfield (age 12)
Visitors
Eric Martin

Help From Other Groups At PARC

Dave Boggs
Larry Clark
Peter Deutsch
Bob Flieg
Butler Lampson
Mike Overton
Bob Sproull
Chuck Thacker

Adele Goldberg
Chris Jeffers
Diana Merry
John Schoch
Steve Weyer

Barbara Deutsch
Steve Purcell
Bob Shur
Radia Perlman

Dennis Burke (age 12)
Susan Hammet (age 12)
Lisa Jack (age 12)
Steve Putz (age 15)
Ron Baecker
Bonnie Tenenbaum

Patrick Baudelaire
Bill Bowman
Jim Cucinitti
Bill English
Ralph Kimball
Bob Metcalfe
Alvy Ray Smith
Larry Tesler
Truett Thacher

