The Early History of Smalltalk

Alan C. Kay
Apple Computer
ekay2@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-come 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 power/volume demanded that the sixties be regarded as "almost a new thing" and to finc 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 scops, reduction in complexity, and end-user literacy would require that data and connoisseurs 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 generated by its many predecessors in hardware, language and user interface design. It became the exemplar of the new computing paradigm 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 highy extreme positions which almost forced these new styles to be invented.

Table Of Contents

Introduction .................................................. 2
I. 1960-66—Early oor and other formative ideas of the sixties .......................... 4
   3200 File System
   Sketchboard & Simula
II. 1967-69—The FLEX Machine, an oor-based personal computer .................. 6
   Doug Englebart and NLS
   Tasma Panel, GRAP, LOGO, Dynabook
III. 1970-72—Xerox PARC .................................... 12
   Kildall/Komp
   miniCOM
   Smalltalk-71
   Overlapping Windows
   Font Editing, Pain.sng, Animation, Music
   Byte Codes
   Iconic Programming

IV. 1972-76—Xerox PARC: The First Real Smalltalk (72) ................................ 17
   The two bits: birth of Smalltalk and laterim Dynabook
   Smalltalk-72 Print ples
   The Smalltalk User Interface
   Development of the Smalltalk Applications System

Introduction

I'm writing this introduction in an airplane at 35,300 feet. My lap is a five-pound notebook computer—1992 "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 major systems I use for my current work with children. As a Dynabook (quantitatively), and some ways not quite there yet (qualitatively). All in all, pretty much what I was and 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 average allocation of credit for ideas is intractably difficult. Instead, as Bob Barton liked to quote Goethe, we should "shar 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 and they were transformed in the magnetic field formed by the new personal computing metaphor. It was the climate as well as the great ideas of the pioneers who helped Smalltalk get invented. Many of the people I admire 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 wondrously 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—"outfield" is better than insight—but you 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, logic-based, object-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
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. 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 Pato as an extremely practical idea.

In computer terms, Smalltalk is a recursion on the notion of a computer, instead of dividing "computer stuff" into things each less strong than the whole—like da da da structures, procedures, and functions which are the usual paraphernalia of programming languages—each Smalltalk object is a recursion on the possible manifestations of the computer. Thus its semantics are a bit like having thousands and thousands of computers all hooked together by a very fast network. Question 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 Hammond

"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 of my emotional involvement. I have 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 ideas 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 implementor; 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 include Bob Balzer, Bob Barton, Danny Bobrow, Steve Carr, Wes Clark, Barbara Deutsch, Peter Deutsch, Bill Dublin, Bill Flegel, Laura Gould, Bruce Horn, Butler Lampson, Dave Liddle, William Newman, Bill Paxton, Triggie Reenskaug, Dave Robson, Doug Ross, Paul Rovner, Bob Sproull, Dan Swern, Bert Suberland, Bob Taylor, Warren Teitelman, Bonnie Tennerbaum, Chuck Thacker, and Iohn Warnock. Worse, I have omitted to mention many systems whose whose design I detest, but that generated considerately useful ideas and attitudes in reaction. In other words "histories" of this period are very seriously but considered as "PEEREE CEESTERS OOF" done long after the actors have depatured the stage.

"Thanks to the numerous reviewers for ensuring the many drafts they had to comment on. Special thanks to Mike Mahoney for helping so gently that he hid my suggestions, so well that they greatly improved this essay—and to Joar Santner, an old friend, who quite literally frightened me into finishing it—I did not want to find out what would happen if I were late Sheri McLaughlin and Kim Rose were of great help in getting all the materials together.

L. 1960-66—Early oo and other formative ideas of the sixties

Though oo came from many motivations, two were central. The large scale one was to find a better 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 orinally happened in isolated files and stars.

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 seeing 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.

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 finesse 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 220 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 hit from this machine at this time was not the oop idea, but some insights into HLL translation and evaluation.

[Barton,1962] [Burroughs,1960]

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—by aside from doing a one-dimensional version of a bit-field block transfer (bitblit) 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 its density and cost over many years [Moore,65]. At that time in 1965, standing next to the room-sized neon-bright 12 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 66, "knowing nothing". That is to say, I had never heard of ARPA or its projects, or that Utah's main goal in this community was to solve the "hides line" problem in 3D graphics, until I actually
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."

"The Sketchpad: A man-machine graphical communication system" [Butterfield, 1965]. 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; think was 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 interlace parts. Its data structure was dead-simple—only vaguely familiar concept was the embedding of a portal process and using a process called reverse indexing to jump through them to routines, like the 220 file system [Ross, 1963]. It was the first to have clipping and zooming windows—one "sketched" on a virtual sheet about 1/3 mile square!

II. 1957-69—The FLEX Machine, a first attempt at an ooo-based personal computer

Dave Evans was not a great believer in graduate school as an institution. As with many of the ASRA "contractors" he wanted his students to do "real things", they should move through graduate school as quickly as possible, and their thesis 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 Chudle, a friendly hardware genius at a local aerospace company who was working on a "little machine". It was not the first: personal computer—that was the LINC of Wes Clark—but Ed wanted it for non-computer professionals, in particular. Ed wanted to program it in a higher level language, like BASIC, I said: "What about "JOS'? It's nice!" 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 "JOS" (for any existing language that I knew of) was particularly good at. The machine was too small for "JOS", so that was out. The beauty of "JOS" was the extreme attention of its design to the end-user—in this respect, it has not been surpassed [Ross, 1964, Ross, 1978], "JOS" was too slow for serious computing (but cf. Laptops 65), did not have real procedures, variable scope, and so forth. A language that looked a little like "JOS" but had considerably more potential power was Wirth's "EH" [Wirth, 1966]. This was a generalization of Algol along lines first set forth by van Wijklanden [van Wijklanden 1963] in which types were discarded, different features consolidated, procedures were made into first-class objects, and so forth. Actually kind of Slikke, but without the better insights of USR.

But USR was enough of "an almost new thing" to suggest that the same techniques be applied to simplify Simula. The EHL compiler was a part of its formal definition and made a simple conversion into BSOS-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 EHL compiler however, was tortuously rendered in an "extended precedence" grammar, but actually only required in the system 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 1972] and later used to various top-down schemes, several of them related to Shcerb's META II [Shcerb 1960] 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 EHL. 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
had a "glass teletype", but a Sketchpad-like system seemed far beyond the scope that we could accomplish with the maximum of 16k-16k 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 proponent 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 Vimms system and a remote control for starting and stopping it to show what was going on, whereas we were not used to seeing and following cursors back then. His notion of the NLS dream was that the destiny of NLS (NLS: the "augmentation of human intellect" via an interactive vehicle navigating through "thought vectors in concept space") was the same as the destiny of NLS. What this system could do then—by today's standards—was incredible. Not just hypertext, but graphics, multiple panes, efficient navigation and command input, interactive collaboration, etc. An entire conceptual world and world view (Engelbart 67). The impact of this vision was profound 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.

The middle section of the history of human-computer symbols is the presence of Ed's "little machines". Copernicus's "Law" again came to mind. This time with great impact. For the first time I made the leap of putting the room-sized "X-11" or even a 10 MPF 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 6000 IBM mainframes in the entire world—there were at most a few thousand users trained for each application; these 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 application programmer to anticipate the specific needs of a particular user? Where would the applications and training come from? Why should we expect an application programmer to anticipate the specific needs of a particular user? An extensional system seemed to be called for in which the end-users would do most of the tailoring (or even some of the direct construction) of their tools. We had already figured this out in the context of their early successes in time-sharing.

Their larger metaphor of human-computer symbols helped the community avoid making a religion of their subgoals and kept them focused on the abstruser holy goal 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. This was similar to William Newman's early "Reaction Handler" (Newman 61) work in specifying interfaces by having the end-user or developer construct through table and xalus an iconic regular expression grammar with action procedures at the states (via allowed embeddings via its colorless 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 state that required getting out of before any new kind of interaction could be done. In hierarchical menus or "scissors" 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 everything else—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 NLS machine was too small for a multidisciplinary, 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 BESO 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 (larer we realized that we should put the master pointer in the instance to save space). A different method for handling generalized assignment was used: the BESO used i-values and t-values [Stackey] which work for some cases but couldn't handle more complex objects. For example: if a = 5; then a = 6 will type error. In the FLEX, a new element is generated in the array because a is an operator and a[5] is dereferenced into an i-value before anyone gets to see that the value is the default element, regardless of whether the array is a procedure or a procedure containing an array. What is needed is something like: if a = 5; then a = 6, which can look at all relevant operators before any system is made. In other words, it is not an operator, but a kind of a index that can select a behavior from a complex object. It made me a remarkably long time to realize this, partly I think because one has to invert the traditional notion of operators and functions, etc., to see that objects need not be simply a collection of all of their behaviors: the objects are a kind of mapping whose values are its behaviors. A book on logic by
Another control structure of interest in FLEX was a kind of event-driven "soft interrupt" called when its boolean expression was compiled into a "bureaucratic sort" tree and cached all possible intermediate results. The relevant variables were threaded through all of the sorting trees in all of the where so that any change only had to compute through the necessary parts of the boolean. The efficiency was very high and was similar to the techniques now used for spreadsheets. This was an embarrassment of riches with different styles often encountered in event-driven systems.

Namely, it was a complex task to control the context of just when the wheel 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 third graduate student got invited to the ARPA contract meeting held that year at Altus Air Force Base. Towards the end of the three days, Bob Taylor, who had succeeded Ivan Sutherland as head of Artificial Intelligence, 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 computers-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 main 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 those skills. He didn't have to make any claims about computers and kids, he just wanted to rethink the light of 20th century cognitive psychology and how good thinkers really think. Computers entering as a new representation system were 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 2x2 chart. This included all of the "object structures", the compiler, the bytecode 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 of U of Illinois where I saw a 1" square lamp of glass and neon glass in which individual spots would light up on command—it was the first flip-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.

Four years later I was at RAND I saw a truly beautiful system. This was GRAIL, the graphical follow-on to JOSS. The first table (the famous RAND table) was invented by Tom Ellis [Davis 64] in order to capture human gestures, and Gabe Grinner wrote a program to them [Grinner 76]. Though everything was fastened with bubble gum and the system crashed often, I've never forgotten my first interaction with this system. It was direct manipulation, it was analogical, it was modeless, it was beautiful. I recognized that the FLEX interface was all wrong, but could something like GRAIL be stuffed into such a tiny machine since it required all of a stand-alone 360/44 to run it?

A month later, I finally visited Seymour Papert, Wally Feurzig, Cynthia Solomon and some of the other original researchers who had built LOGO and were using it with children in the Lexington schools. Here were children doing real programming with a specially designed language and environment. As with Simula leading to OOP, this encounter finally hit me with what the cost of personal computing really was going to be. It was 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 in form of image of what a personal computer really should be. I remembered Aldus Menutus 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 LOGO, GRAIL, and LOGO's, but with the power of FLEX. A clear romantic vision has a marvelous ability to focus thought and will reach of Simula and FLEX. A clear romantic vision has a marvelous ability to focus thought and will reach of Simula and FLEX. A clear romantic vision has a marvelous ability to focus thought and will reach of Simula and FLEX.

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). Then I put a keyboard on it as well as a stylus because, even if handwriting and writing were recognized, there was no reason to expect that it would be, there still needed to be a balance perfectly (and there was no reason to expect that it would be), there still needed to be a balance perfectly (and there was no reason to expect that it would be), there still needed to be a balance perfectly (and there was no reason to expect that it would be), there still needed to be a balance perfectly. Since FLEX 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 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 idea. 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 showing among so many familiar ideas among people who, 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 wading eye (the urge to generalize) [ACM 69].
But it was all talk—no one had done anything yet. In the midst of all this, Ned Irons got up and presented his 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 is defined in the more usual way as the kind of code that was emitted. This separated the compiler-extensions 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 direct compiler. Somewhere in all this, I realized that the bridge to an object-based system could be in terms of each object as a syntax directed interpreter for messages sent to it. In one fell swoop this would unify object-oriented semantics with the idea of a complete extensible language. The mental scaffolding was one of separate images required receiving 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 distribution depended entirely on the server's notion of relations with the server. As Libbiza 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 KiddiKompeter than AI. But there were two AI designs that were very intriguing. The first was Carl Hewitt's PLANNER, a programmable logical 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 Winograd's concept formation system, a scheme for building semantic networks with a meta-grammar to form analogies and learning processes [Winograd 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 ADT triples should themselves be modeled as nets. Thus, for example a first order arc called TAPP could be asked a higher order question such as "What is your converse?" and its net could answer: right. 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-TS, a capability-based operating system that seemed very "object-oriented" [Lampson 69]. Unforgivable pointers (ala 5000) 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 object 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 never, no one had perpetrated the mysteries of eval and apply. I could hardly believe how beautiful and wonderful the idea of LISP was [McCarthy 60]. I say it this way because LISP had not only been around enough to get some honest barcodes, 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 component—such as lambda expressions, quotations, and cons—were not functions at all, and instead were called special forms. Landin and others had been able to get quoted and cons in terms of lambda by tricks that were variously clever and useful, but the flaw remained in the gem. In the practical language things were better. There were not just EXPS (which evaluated their arguments), but FIXPS (which did not). My next question was, why on earth call it a functional language? Why not just base everything on FIXPS 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.
that it would take about three years to do a good operating system for the XDS 51604, but that: we could build "our own" PDP-10 in a year. My reaction was "Tell me more." In fact, they pulled it off with considerable panache. PARC was actuated by a microcode emulation of the PDP-10 that used for the first time the novel integrated chip memories (1K bits) instead of core memory. Having practical experience with both of these new technologies was critical for the more radical systems to come.

One little incident of Star 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 in which pointers are manipulated explicitly, and he got into quite a struggle to do the program. In 2 seconds I wrote down:

- `odd(x)`
- `even(x)`

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

- `token odd(x) = if null(x) v null(tail(x)) then x else head(x) & odd(tail(x))
- even(x) = if null(x) v null(tail(x)) then nil else odd(tail(x))`

This characteristic of writing down many solutions in declarative form and having 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. I brought home to me once again that "point of view is worth 50 IQ points." I wasn't smarter but I had a much better intuitive thinking tool to amplify my abilities. This incident and others like it made me realize 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 Endsley 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 nade!" He remained unconverted, and that led to the famous "Pendery Papers for PARC Planning Purposes," a collection of essays on various aspects of the future. Mine proposed a version of the notebook as a "Display Transducer," and Jim Mitchell's was entitled "NLS on a Minicomputer."

Bill English took me under his wing and helped me start my group as I had always been a lone wolf and had no idea how to do it. One of his suggestions was that I should make a budget. I'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 we walked off on a trip to ADEA, "We're almost out of money. Go to 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 and say, "Advance that!" LRG members developed a very close relationship with each other—such Dan Legalis 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, bike-riding, drinking beer, eating Chinese food, and constantly talking about the Dynabook and its potential to amplify human reach and bring new ways of thinking to a faltering civilization that desperately needed it (that kind of goal was common in California in the aftermath of the sixties).

In the summer of '71 I refined the KildiKopp idea into a lighter design called miniCOM. It used a bit-slice approach like the NIX 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 "InfoEuropean god theory" where systems were named Zeus, Odin, and Thorr, 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.
be characterized as universal computers without having to make any exceptions in the central metaphor. What seemed to be needed was a complete consensus 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 71] on the synthesis of control structures. ALCOLO 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 “heurism 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 reflexive design. Putting it together with the IFL 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 B500 and LISP—and a message handling structure—which could be virtual, if the senders and receivers were sympatric—that could be used by all would do the job.

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

- **GLOBAL:** the environment of the parameter values
- **SENDER:** the sender of the message
- **RECEIVER:** the receiver of the message
- **REQUEST STYLE:** waiz, jork, ...?
- **STATUS:** progress of the message
- **REPLY:** eventual result? (if any)
- **OPERATION SELECTION:** relative to the receiver
- **# OF PARAMETERS:**

This is a generalization of a stack frame, such as used by the B500, and very similar to what a good intermodule scheme would require in an operating system such as CAL-TMS—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” meant with reference to object-oriented design. A subjective definition of a beautiful thing is fairly easy but 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 an appeal 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 synony between parsimony, generality, enlightenment, and finesse. For example, the Pythagorean Theorem is expressible in a single line, it is true for all of the institute 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 aesthetics. I mean by this that Nature is wonderful both in elegance and practicality—the cell membrane is partly there to allow useful evolutionary kudges to do their necessary work and still be able act as component by presenting a uniform interface to the world.

One of my continual woes 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 bitmap text 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 Lampion specified an experimental character generator (built by Rogers Bates) for the Polos Sparc OnLine Office System terminals. Gary Stadlerweather had just gotten the first laser printer to work and we ran a coaver over to his lab to feed him some text to print. The “slor machine” (Scanning Laser Output Terminal) was incredible. The only Xerox copier Gary could get to work on was at 1 page a second and could not be slowed down. So Gary just made the laser run at a rate with a resolution of 50 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 write into the font memory. Ben Law’s built a beautiful font editor and he and I spent several months learning about the peculiarities of the human visual system (it is a decided non-linear). I was very interested in high-quality text and graphical presentations because I thought it would be easier to get the Dynaicon to school as a “free entry” by simply replacing school books rather than to try to explain to teachers and school boards what was really good about personal computing.

![Image of a Special Font](image)

Things were generally going well all over the lab until May of 72 when I tried to get resources to build a few micros. 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 E), 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 mainc and this use of resources for personal machines would continue. I was shocked. I crawled away back to the experimental character generator and made a plan to get 4 more made and hooked to work 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 Snoch built a file drawing and gesture recognition system (based on Leder’s [Newman and Spruell 72]) that was integrated with the painting. Bill Duvall of POLOS built a minutiae 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 2/4 D animation system. Along with Ben Law’s font editor, we could give quite aashing demos of what we intended to build for real over the next few years. I remember giving one of these to a Xerox execu
The first painting system—Summer ’72

The First Painting System—Summer ‘72

tive, 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 straight into 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, Buter
Lampson, Peter Deutsch and I worked out a general scheme for emulated HLL machine languages.
I liked the B5000 scheme, but Buter 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 sds. Jef Rulifson was a big fan of Plaget (and
semiotics) and we had many discussions about the “stages” and what iconic thinking might be
about. After reading Plaget and especially Jerome Bruner, I was bothered by the directly symbolic
approach taken by FLEX, LOGO (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 figural, more iconic approach. Rudolph Arnheim [Arnheim 69] had written a classic book about
visual thinking, and so had the eminent art critic Gombrich [Gombrich 71]. It really seemed that
something better needed to be done here. GEAR 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-6 held considerably
more promise [Rovner 68]. It was kind of a visual SNOMOL [Faber 63] and the pattern matching
ideas looked like they would work for the more PLANIFIER-like scheme I was using.

Bill English was still encouraging me to do more reasonable appearing things to get higher credi-
bility, 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 run in small batches that
would involve cop, 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” [At 71]

“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 CS.” Why? They said, “How would you like to build your little machine for you?” I
said, “I’d like it fine. What is it?” Butler said: “I want a 4500 POP-10, Chuck wants a 10 times faster
nowa,” and you want a “kiddiecpop.” 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 and they both 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
set with Bill Viwit 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 arguing with me on. For the next two weeks I got to
PARC every morning at four o’clock and worked on the problem until eight, when Dan joined by
Henry Fuchs, John Shoch, and Steve Porneull showed up to kick my morning’s work.

I had originally made the boast because McCarthy’s self-describing LISP interpreter was written in
itself. It was about a "page", and as far as power goes, LISP was the whole nine-yards for functional
languages. I was quite sure I could do the same for object-oriented languages plus be able to do a rea-
sensible syntax for the code as is 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 pro-
gram 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 “passing” with message reception—the evaluation of pa-
rameters which was handled separately in LISP—required that my object-oriented interpreter n-erator
itself be processor (in fact much more processor 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 severely 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
first version of a little bit of Smalltalk/72 of the form in the first version of a little bit of Smalltalk/72 of the form in the
original 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 val-
ues. 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”
pursuit, 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 likes 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 right. 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 a least 30 major releases of various flavors of
Smalltalk.

In November, I presented these ideas and a demonstration of the interpreter 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 nothing
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 the extensive overviews 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 72] and the surround-
ing 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 spir-
it of the whole culture, Xerox went berserk, forced us to wear badges (over the years many were
printed ch t-shirts), and severely restricted the kinds of publications that could be made. This was particularly disastrous for LGC, since we were the "funnie fringe" (so-called by the other computer scientists) who were planning to go out to the schools, and needed to share our ideas (and programs) with our colleagues such as Seymour Papert and Don Norman.

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

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 McCrea. It had a ~500,000 pixel (106x80) bitmap display, its microcode instruction rate was about 6MHz, it had a grand total of 128k, and the entire machine (exclusive of the memory) was rendered in 160 MSI chips distributed on two boards. It was beautiful [Thackray, 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 lost to interesting events (such as "horizontal race pulse", and "disk sector pulse", etc). Lookaside logic scanned the flags while the current instruction was executing and piced 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 needing 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 to some computer scientists a few months before, but I remember that Wes Clark's TX-2 (the Sketchpad machine) had used the idea first, and I probably talked about it too.

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 $250K was enough to get 15 of the original projects to 30 machines (over the years some 2000 Interim Dynabooks were actually built). True to Schonhauer'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 oo 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 ci, <- drg (with subscripting rendered as "*" and multiplication as "*"), means:

- Everything is an object.
- Objects communicate by sending and receiving messages (in terms of objects).
- Objects have their own memory (in terms of objects).
- Every object is an instance of a class (which must be an object).
- The class holds the shared behavior for its instances (in the form of messages that are pre-defined).
- In every program list, control is passed by the first object and the remainder is treated in its message.
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. It think something similar happened with lisp—that the "reality" of the straightforward and practical syntax you could program in prevailed against the flights of fancy that never quite got built.

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 trace 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. Dana built an early version of a bit field block transfer (bfbt) for displaying variable pitch fonts and generally writing on the display. The first window versions were done as real 2142 druggable 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 2142. Windows were perhaps the most redesigned and reimplmented class in Smalltalk because we didn't quite have enough compute power to just do the continuum. 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 2142. The first practical windows in Smalltalk used the GRIAL conventions of sensitive corners for moving, resizing, 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 LOGO turtle implemented by Ted. This could make many turtle instances that were used both for drawing and as a kind of value for graphics transformations. Dan created a class of "commander" turtles that could control a troop of turtles; soon the turtles were made so they could be clipped by the windows.

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

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

One of the "small program" projects I tried on an adult class in the Spring of '74 was a one-page paragraph editor. It turned out to be too complicated, but the example I did to show them was completely modeless (it was in the air) and became the basis for much of the Smalltalk text work over the next few years. Most of the improvements were made by Dan and Diana Merry. Of course objects mean multi-media documents, you almost get them for free. Early on we realised 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 Plog, Dana Merry, Larry Tesler, Tim Mott, and Trygve Remenkaug.

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 was developed on the nova and 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 generalised input that we had specified for low-speed devices (user for the mouse and keyboard) made it easy to connect 154 move 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 tabulation notation that we devised to make music clear to children [Kay, 1974]. One of the things that was hard to do with sampling was the voltage controlled operator (vco) effects that were popular in the "Unfinished Synthesizer." A summer later, Steve Saunders, one of our bright summer students, was challenged to find a way to accomplish John Chowning's very non-realtime 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 4 real-time voices that were integrated into TWANG [Saunders*].

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

Many nice "computer science" constructs were easy to make in Smalltalk-72. For example, one of the controversies of the day was whether to have goto or not (we didn't!), and if not, how could one quickly move around a program such as multiple exits from a loop—are they specified? Chuck Zahn at XAC proposed an event-artic 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.

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 "message" object, message dictionaries for classes (a step towards real class objects), Diana Merry's bitum (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 OOCZ (Object Oriented Zoned Environment) virtual memory system that served Smalltalk-74, and more importantly, Smalltalk-76 [Ing, 78, Kae]. 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 256-byte model 30 disk was faster and larger than today's hard drives. It was quite similar to the HP direct access disk of the time-greased worm of the B5000 segment swapper. It had not worked as well as I wanted, despite a few good ideas as to how to choose objects when purging. When the gang wanted to adapt this basic scheme, I said: "But I never got it to work well." I remember Ted Kaehler saying: "Don't worry, we'll make it work!"

The basic idea in all 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 (OOCZ) in each disk touch.) Two ideas help a lot. First, Butler's insight in the GE DEN that it was worthwhile to expend a small percentage of time purging dirty objects to make core as clean as possible [Lampson, 1966]. This tends not to hurt as much and there is always clean storage to fetch pages or objects from the disk. 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 when to purge 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 Alto's had a way of just crashing once or twice a day for no discernible good reason. This by the way did not entirely bother anyone as it was fairly easy to come up with some 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 stored there as well to help LRU aging. Purging was done in background by picking a class, positioning the disk to its instances (fall of a particular case was stored together), then running through the LOT to find the dirty ones in storage and stream them out. This was pretty efficient and, true to Butler's insight, fur-
nished a good sized pool of clean storage that could be overwritten. The key to the design (though
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. Ooze swapped objects in just 80K of working storage and could handle
about 65K objects (up to several megabytes worth, more than enough for the entire system, its inter-
face, and its applications).

"Object-oriented" Style

This is probably a good place to comment on the difference between what we thought of as con-
text and the superfluid encapsulation called "abstract data types" that was just starting to be investi-
gated in the academic circles. Our early "tag-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 "offici-
a" computer science world started to regard Simula as a possible vehicle for defining abstract data types
even by one of its inventors[Da1 70]. 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 am-
azed at this, since to us, what Simula had clarified was something much stronger than simply reim-
mplementing a weak and ad hoc idea. What I got from Simula was that you should now replace
bindings and assignment with calls. 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 states of higher
level behaviors more appropriate for use as dynamic components.

Even the way we taught children (cf. Schem) reflected this way of looking at objects. Nor too sur-
prisingly, 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 facer constructs. Many programs are now
loaded with "assignment style" operations done now 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. But the more
likely source of the efficiency comes from a much clearer way to represent a complex system. Here, the constraints are as
useful as the generalities. Four techniques to which we added, persistant state, polymorphism, instan-
tiation, and method-as-goals for the object—account for much of the power. None of these require an
object-oriented language to be employed. ALGOL 68 can almost be turned to this style—an object
merely excuses the designer's mind in a particular fruitful direction. However, doing efficiency
right is a commitment not just to abstraction of state, but to eliminate state-oriented metaphors from
programming.

Perhaps the 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 one object
pretend to be another and not let 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
its strongest kind of goals it can carry out, its superclasses can add up to much more code-function-
ality being invoked than most procedures-on-data-structures. Assignment statements—even abstract
ones—are really very low-level goals, and more of them will be needed to get anything close.

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 it this is that the low-bounding of automatic storage allocation doesn't do anything a pro-
grammer can't do, its presence leads both to simpler and more powerful code. oop is a late binding
strategy for most things and all of them together hold off changes. It is much less fragile 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 1975, let me return to the summer of '73, when we were ready to start experiments with chil-

dren. None of us knew anything about working with children, but we knew that Acele Goldberg and Steve Wise
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 programs to children (or anyone else), the first
experiments Adele did mimicked LOGO turtle graphics, and she got what appeared to be very similar
results. This 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, the content of this new kind of authoring literacy should
be the creation of interactive tools by the children. Procedural turtle graphics just didn't cut 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 Kinsky'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 multiprocess animation. After getting kids to guess what a box might be-
like—they could come surprisingly close—they would be shown:

to box | y size tilt
- (grasp (+ x y) turn tilt, square size.)
- (grasp (+ y x) turn tilt, square size.)

What was so wonderful about this idea was the myriad of children's projects that could spring off the humble
boxes, including some of the earliest were tools! This was when we got really excited. For example, Marion
Goldstein's (12 yrs old) painting system was a full-fledged tool. A few years later, so was Susan Harter's (12 yrs old)
oor 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 Purz's (15 yrs old) circuit design system. Looking back, this could be
called another example in computer science of the "early success" effect. 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
who were chosen from the Palo Alto schools (nearly in average background) and we needed to be much more excited
about the successes than the difficulties. In part, what we were seeing was the "harder phenomenon," that for
any given pursuit, a certain percentage of the population will jump into it naturally, while the rest are
sadly left behind. We learned that it is time to do not find it at all natural.

We had a dim sense of this, but we clung to it. I think I'm not alone in 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 Alto with Acele's Joe book. The problem

Alan G. Kay, The Early History Of Smalltalk
seemed more to be that of design.

It started to hit home in the Spring of '74 after I taught Smalltalk to 20 PARC non-programmer 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 Notelx. They could even get closer 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 corner pondering the board from my talk. Finally, I counted the number of nonobvious ideas in this little program. They came to 15. 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 before one acquires more powerful ideas [Press 76a]

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 work 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 a framework and guide 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 were 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 many 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 live or six page Smalltalk animation tool done by relatively expert adults). Unfortunately,

inheritance—though an incredibly powerful technique—has turned out to be very difficult for novices (and even for 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 didn't apply it long enough. I mean by this that there is now a large accumulation of results from many attempts to teach novices programming [Soloway 88]. 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 the structures that need to be there for design and solution look-ahead.

The problem is not to get the kids to do stuff—they loved to do it even when they are not sure exactly what they are doing. This correlates well with studies of early learning of language, when each 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 conscious skill that 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. Then, 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 the general ability to think well or to take an enlightened stance on human knowledge. If anything, the opposite is true. Expert knowledge often remains rooted in the environments in which it was first learned—and most metaphorical extensions result in misleading analogies. A remarkable number of artists, scientists, and philosophers are quite dull outside of their specialty (and one suspects within it as well). The first sin'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 Julia to 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 aims: an excuse for developing enlighentment, but no tool ever contained it or can dispense it. Centre Parc observed: "to know the
realistically start a Win-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 our current end-user products, others, particularly some of the grad students, really wanted a better Smalltalk that was easier and could be used for bigger problems. I think Dan felt that a better Smalltalk could be the vehicle for the different system we 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 rejected. 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” was that I had a very McLuhanish feeling about media and environments: that once the system’s 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 at 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 ISP 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 gret, 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, simplify everything and start from scratch again.

The NoteTaker was to be a “laptop” that could be built in a few years using the (most available) 16K RAM’s (a vast improvement over the 1K RAM’s the Alto employed). A laptop couldn’t use a mouse (which I hated anyway) and a tablet seemed awkward (not a lot of room and the stylus could flop out of reach when I lost it), so I came up with an embedded pointing device I called a “submouse.” It was a relative pointer and had an pressure 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 systeris 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 bytecode interpreter for a friendlier and simpler system than Smalltalk-77.

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 intentional definition with every piece of code as an intrinsic method. We 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 concession. 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 Smalltalk 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 it to be read unambiguously by both humans and the machine. This allowed a flexable machine-like bytecode compiler and efficient interpreter to be defined that ran up to 180 times
as fast as the previous direct interpreter. The JOSE VM system could be modded to handle the new objects and its capacity was well matched to the Alto’s RAM and disk.

Inheritance

A word about inheritance. Simula-7 had neither classes as objects nor inheritance. Simula-67 added the latter as a generalization to the ALGOL-60 block structure. This was a great advance. But it did have some drawbacks: minor ones like name clashes in multiple threads (lists of common names don’t exist). The minor injury to 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 the Simula’s goals: having to do with various kinds of modeling and interfacing that were of interest in the word of artificial intelligence. For example, the most useful questions couldn’t be answered by following a static chain. Some of them required a kind of "inheriting" or "inferring" through dynamically bound "parts" (i.e. instance variables). Multiple inheritance also lacked important but the corresponding possible clashes between methods of the same name in different superclasses looked impossible to handle, and so forth.

On the other hand, since things can be done with a dynamic language which is 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-72’s smalltalk flexibility. The biggest contributer to these ideas was Larry Tesler who used what is now called "slot inheritance" extensively in his various versions of the early desktop publishing systems. Nowadays, this would be called a "delegation-style" inheritance scheme [Lieberman 84]. Danny Bobrow and Terry Winograd during their period were designing a "frame-based" language called KRL, which was "object-oriented" and I believe was influenced by early Smalltalk. It had a kind of multiple inheritance—called perspective—which permitted an object to play multiple roles in a very clean way. Many of these ideas a few years later went into a very interesting extension of Smalltalk to networks and higher level descriptions by Ira Goldberg and Bobrow [Goldstein & Bobrow 1980].

The time Smalltalk-76 came along. Dan Ingalls had come up with a scheme that was Simula-like with smalltalk 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 isomorphism (which is a kind of inheritance) muddles both pragmatics (such as factoring code to save space) and semantics (used for why way many tasks such as specialization, generalization, specification, 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 Ethernet 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 Pindery Paper

In the 1990’s there will be millions of personal computers. They will be the size of notebooks or index cards, have very-high-resolution flat-screen reflective displays, weigh less than ten pounds, have ten to twenty times the computing and storage capacity of an Alto. Let’s call them Dynabooks. The purchase price will be that of a color television set of the era, although mass of the machines will be given away by manufacturers who will be marketing the computer. The cost rather than the content of personal computing.

... Through the Dynabook will have considerable local storage and do most computing locally. It will have a large percentage of its time hooked to various large, global information utilities which will exist, communication with others of ideas, data, working models, as well as the daily chit-chat, the organization needs in order to function. The communications link will be the telephone and public wires and 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 View

- There need not 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 or lessons drawn from this:

- Material: If the design and development of the hardware computer material is done carefully and completely and Xerox’s development of special light-sensitive alloys, then only one or two computers designs need to be built.... Extreme investment in development here will be vastly repaid by simplifying the manufacturing process and providing lower costs through increased volume.
- Content: Aside from the wonders of generality there are no changes in capability new content from material software has three important characteristics:

- The replication time and cost of a come to a function is zero
- The development time and cost to a content function is zero
- The change time and cost of a content function is zero

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

1. Xerox has the financial size to cover the development costs of a small variety of very powerful computer-types and a large number of software functions.
2. Xerox has the marketing base to sell these functions on a wide enough scale to garner back to use an incredible profit.
3. Xerox has working in 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 fatal decision: not to bring the Alto III to market. This was a huge blow to many of us even I, who had never really really thought of the Alto as any thing but a stepping stone to the "real thing. In 1991, the world market for personal computers and workstations was $900 million—twice as much as the mainframe and mini market, and many times the 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 descendents. A decent history would be as long as this chapter, and none has been written so far. There is a summary of some of the ideas in [Kaye 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 LISP focus was on children. Hence we were thinking about learning as being one of the main effects we wanted to have happen. Early 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 echo of Monieros and Dewey, particularly the former, and got me, or rereading Jerome Bruner, to think beyond the children’s curriculum to a "course of the user interface".

The particular aim of LISP was to find the equivalent of writing—that is: learning and thinking by doing in a medium—the new "pocket universe". For various reasons: had settled on "iconic programming" as the way to achieve this, drawing on the iconic representations used by many art projects in the sixties. My friend Nicholas Negroponte, as architect, was extremely interested in how environments affected people’s 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 ‘had theatre’ hoping to forget; people attend
Paul Rosner showing the iconic "Lincoln Wand" ca. 1968

"good theatre: asking 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 symbol." (Piaget & Bruston; the user is never trapped in a rote (GRAI); the magic is embedded in the familiar (Negropontes); 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 knew it today.

In fact, the UL 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 wandering, the concept almost forced a good design to turn our way: anyway, just about everyone at PARC at this time had opinions about the UL ours and theirs. It is impossible to give detailed credits 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 Smalltalk, the prototype iconic interface for the Xerox Star project (Smith 83).

Meanwhile, I had gotten Doug Fairbairn interested in the Notesproject. 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 bus 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 fun. The system consisted of about 50 classes described in about 180 pages of source code. This included all of the 25 instructions, 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.

Smalltalk-76 was a great improvement on its predecessors! Here are two styles ST-76 classes written by Dan.

Smalltalk-76 User Interface

Class new title: 'DocWindow';

subclassed: 'Window';

fields: 'document scrollbar editMenu';

User events are passed on to the document while the window's active. If the window goes out of the window, scrollbars, and the editMenu are all given a chance to gain control. Event Responses

/* enter (self show, editMenu show, scrollbar show)

leave outside [false]

pendsend (document pending) keyboard (document keyboard)

super menu (super show, self show, Docs (document shown: 3 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 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
In this case, it finds DocWindow method: keyb, which tells the held document to check it out.

In January of 1978 Smalltalk-76 had its first real test. CSL had invited the ten execut-ers of Xerox to PARC for a two day seminar on software, with a special emphasis on computing and what could be done about it. It was agreed to give them a hands-on experience in end-user program-
ning so they could do 'something real' over two 1/2 hour sessions'. We immediately decided not to teach them Smalltalk-76 (my 'burn our dsk packs' point in spades), but to create a two months in Smalltalk-76 a rich system, especially tailored for adult novices (Dan's point in trumps). We took our "Simulpa" job shop simulation model as a starting point and decided to build a user interface for a generalized JCT shop simulation tool that the customer 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 max-
imum 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 Rachell.

There were many interesting problems to be solved. The system itself was straightforward but it had to be complete-
ly 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 errors directly from the lexicon of Smalltalk and translate 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, Work, Jct, Report), and some more surprising as when we tried it on ten non-technical adults of about the same age and found that they couldn't react the screen very well. The small fonts our thirty-something 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 encour-
aged to customize the screen by choosing among different fonts and types 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 so that it would work better. 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 made a "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 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 done easily by closed form mathematics—it revealed a serious flaw in the disposition of workers given

the line's average probability of manufacturing defects.

Another important system done at this time was Alan Binning's Thinfraad [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 en-
ough by changes to values that different methods were based on

on. This was an old idea but Thinfraad showed how the object-oriented definition could be used to automati-
cally limit the contexts for even-driven processing. And we soon discovered that "prototypes" were more hos-
pitable than classes and that multiple inheritance would be less needed if there were cases for methods that knew generally what they were supposed to be about (inspired by Pat Winston's and order models).

Meanwhile, the NoteTaker was getting realer, bigger, and slower. By this time the Western Digital emul-
ated chips I hoped for showed signs of being "diffusion-wafer" and I did not want 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 8080 and 8085), 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 data (networking, etc).

Dan had been interested in the NoteTaker 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 now-like emulated "machine code" from the Alto could be brought over, and it had to fit in memory as well—there would only be 64k of eproms, 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 in here to simplify object access. An experiment in the way it was done was inspired by the machine code was rewritten in Smalltalk and the total machine kernal was reduced to 6K bytes of (the not very strong) 8086 code.

All of the re-engineering had an interesting effect. Though the 8086 was not as good as the Alto (and almost 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 all of the Smalltalk byte-code interpreter would fit into the 8K microcode memory of the Alto). With various kinds of tricks and tuning, graphics display was "largely compen-

Dan Ingalls, the main implementer of Smalltalk, creator of Smalltalk-76, and his implementation plan (below)
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 crashed any lap. It had toppled backward on the desk, and looked suspiciously like micom (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 thought it many sense in engineering success, what had to be done to make them had once again squelched out the real end-user 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 bitters 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) change 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 they come up with the resources to continue to develop it in house by funding a better Notetaker cum Smalltalk.

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 said that it was 'far 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 become 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 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 limbo expressions which, as Peter Deutsch was to observe nine years later: "In retrospect, this proliferation of different kinds of instantiation and looping 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 instantiations of things and classes) and the object-role and instance-of-class-role are quite different views and is easy to deal with relatively small 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 inexcusable 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 too often the hardware has been presented as 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 Salt points out 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 bad ideas have been about finding ways to late-bind, then using campaigns to convince manufacturers to build the ideas into hardware. Early hardware 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 the location of computations—this led to base/bounds registers, segment relocation, paging, machine, migratory processes, and so forth. Time-sharing was held back for years because it was "inefficient"—but the manufacturers wouldn't put MUX's on the machines, universities had to do it themselves!
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: I would be terrible if +A incurred any overhead if A and B were bound, say, ="3" and "4" in a form that could be handled by the ALU. The operation should occur full speed using look-side 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 this objects. This is very similar to indexing—the class 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 overheat that has already been rationalized.

Again, the whole point of OOP is 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 incompatibility—methods—a good discussion of some of the issues is found in [Popel,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 metaobjecet protocol work at Xerox PARC [Kiezales,1993]. 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 class instead of a slot for slots. 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 Simul, FLEX, CML, 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. Assume a point in time will be easier to have it carry even more information about itself—enough to its specifications can be "understood" and its configuration into your mix done by the more subtle matching of code

A look beyond OOP as we know it today can also be done by thinking about late-binding. Prolog's great idea is 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 in when a binding is finally found. Euriiko [Leman**] 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 Euriiko locks in 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—tackles the next five years. Object 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 they do today. Instead, the objects will carry much more information about themselves in a form that permits inferential docking. Some of the ongoing work in specifications can be tuned to this task [Gutt**] [Gosgen**].

Tongue in cheek, I once characterized progress in programming languages as a kind of "sunsput" 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 Euriiko 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 moldered 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 goal. 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 '81 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,1988] 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 boss: 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 Adelles of the '80s and '90s that will take us to the next stage?
Appendix I: Personal Computer Memo

Smalltalk Program Evolution

From a memo on the "KiddiKomputer"
To: Butler Lampson, Chuck Thacker, Bill English, Jerzy, Elkind, George Yake
Subject: "KiddiKomputer"
Date: May 13, 1972

4 January 1972

The Reading Machine1. Another attempt to work on the actual problem of a personal computer. Every part of this gadget (except display) is buildable now but requires some custom chip design and fabrication. This is discussed more completely later on. A meeting was held with all three labs to try to stimulate invention of the display.

B. Utility

I. I think the uses for a personal gadget as an editor, reader, take-home-context, intelligent terminal, etc. are fairly obvious and greatly needed by adults. The idea of having kids use it implies (possibly) a few more constraints having to do with size, weight, cost, and capacity. I have been begging this question under the assumptions that a size and weight that are good for kids will be super acceptable to adults, and that the gadget will almost inevasually have CPU power to burn (more than PDP-10) implies large 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 1959-70 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 audial representation of works in a manner reminiscent of Bloomfield and Moore. This would require a corollary inquiry into why good readers do so much better than average readers. A farther off project in the domain of sight would be an investigation into the nature and topology of kids' internal models for objects and an effort to preserve iconic imagery from being totally replaced by a relational model.

4. 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 surrepticiously.

3. Second Level Projects

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

b. Iconic Programming. If we believe Piaget and Bruner, kids deal neatly with
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 received, incoming, unhandled, etc., it would be possible to handle all these together in one procedure. I think this is analogous to not having COND as a "special form", but instead to have a basic building block in which COND can be defined like any other subroutine.

One way to do this was to use the approach of Dave Fisher, in which no-man's land of control structures may be provided by providing a protected way to access and change the relationships of the static and dynamic environment (Fisher 70). In an object-oriented 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 external interpreter to input to the individual objects that participate in it and dynamically by the language is extended.

I also decided to ignore the metaphysics of objects even though it was clear that, unlike Simula, it would be possible to design classes that existed at run-time as "first-class" objects—indeed, there should be nothing but first-class objects. So there need to be a "class" whose instances were classes, class-dass had to be an instance of itself, there had to be a "class" object that would terminate any subdlassing that might be done, and so forth. All of this is part of the argument concerning what I called "hierarchies" to show what the basic problem was: I wanted to have a much nicher syntax than Lisp and didn't want to use any of my precious "half-page" to write even a simple interpreter. Somehow the idea had to be designed so that syntax got specified as part of the use of the system, not in its basic definition. I wanted the interpretation to go from left to right. In an oop, we can consider the interpreter rule for expressions as meaning: the first element we evaluate is the function that will receive the arguments, and everything that follows will be the arguments. What should express one like a+b and 1< c< d mean? Let's start with (FLX), the second class of expression is a method in object-oriented terms. The c should be bound to an object and all of 1< < d< would be used to evaluate the method of that message to it. Subscribing and multiplicity are implicit in standard mathematical orthography: we need explicit symbols, say "a" and "b". This gives us:

receiver 1 message c
1 ≤ c ≤ 8

The message is made up of a literal token "a", an expression to be evaluated in the senders context
It's a little more complex, let's say we wanted to evaluate the senders context (in the case, I have another literal token, followed by an expression to be evaluated in the senders context (b)). "Lisp" pairs are made from 2 element objects and can be indexed more simply: ch, e, i, ii, ch, c, etc.

The expression 1<4 seemed more troublesome at first. Dic it really make sense to think of it as:

receiver 1 message 1
1 ≤ 4

We are used to thinking of "a" and "b" as operators, function machines. On the other hand, there are many uses of "a" and "b" that go beyond simple APLish generalizations of scalar operators to arrays—for example in matrix and string abstractions. From this standpoint it makes sense to let the objects in question decide what the token "a" means in a particular context. This means that 1<4+5... should be thought of as 1<4+5... and that the way class number choices in receive messages should be arranged so that the next subexpression is handled properly. E.g. 3 check to see if a token (like +, * etc.) 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 these necessary but Prolog-like sequential evaluation is a byproduct.

By this point I had been into analysis and argue the most of the programming that seemed to be required of the eval. To sum up:

This also means that useful elements like lists, atoms, control structures, quote, receivers (such as "receive evaluating") are handled quite differently from what is usual. For example, we would not want to deal with "the next token", 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 of a message being sent a letter, but simply delivering a notice that the letter was to be delivered. It is up to the receiving object to do something about it. In fact, I 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 extend the uniform syntax idea of receive 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 is bound to a code paired,

ch < 4, ch < 4, first etc.

would be dealt with as follows: Control is passed to the object that first tests the token is to see if it is an instance of the receiver method. If it does, the next check is for an "assignment" token (let- or). It's then, last, we want to evaluate the rest of the message (see get 7), bind the value to the internal instance variable, finally return this value to the sender ('that'). So this is like:

(TRANS (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 done "ahead of time" if one's intuition whispers that "you're on the right track". (This one I had 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 1+4 using a recursive evaluator. For code, we use arrays of pointers and expect that some of the pointers will be encoded for literal object (an old Lisp trick). We need good old program counters "PC" that we can bump along the code. The special case of data in the message (message itself and pass arguments at send time) will require us to manipulate both the program counter of the sender and the receiver as the message is read in. One way I worked it out was as a before-after diagram for:

1+4

We start in the middle of a method of some class of objects and we need to evaluate "1+4" using a recursive evaluator. The essential of the eval is that when we are successful we take us into the method of "3" in class integer. Since all methods are only in terms of smalls and all sends are done in a similar manner, this is enough. It is a like an induction proof in which we just assume "n" and show how to get to "n+1".

The cases above were used in the first interpreter definition. The following were defined when the first "real" implementation was done.
Appendix III: Acknowledgements

1971
Chris Jeffers, +7

1972
Chris Jeffers, John Shoch, Steve Purcell, Bob Siour, Boony Tinnenbaum, Barbara Deutsch

1973
A document written by me shortly after Smalltalk-72 started working

ACKNOWLEDGEMENTS

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 711) is a godchild of Wes Clark’s LINC (ca 1962) and a linear descendent of the FLEX machine (ka 67, 68, 69).

The ‘interim Dynabook’ (known as the ALTO 711, MC 711) is the beautiful creation of Chuck Thacker and Ed McCreight of the Computer Science Lab. at PARC.

SML or the language is generally a synthesis of well-known ideas for programming languages and machineries which have appeared in the last 15 years.

The Burroughs B5000 (ca 61) (1969) had many design ideas, a few of which are still in advance of its time (and not generally appreciated): compact ‘addressable’ code; a uniform semantics for names (the PRT); automatic coprocessors; “capability” protection (also by the PRT and descriptors); virtual storage; memory incidental to the ability to call a subroutine from another part of the program, etc. The notion of code as a data structure: names as properties of names (property lists of attribute-value pairs on names); evaluation with respect to arbitrary environments; etc. All of these ideas found in LISP, probably the most simple design for a programming language yet to appear. SML is definitely a “LISP-like” language.

The SIMULA 67 and 68 combined Conway's notions of software coroutines (1968) and hardware version had appeared in the B5000 3 years earlier, ACLGOL-60, and of course, a few 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 the 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 AGLGOLness), moved them from a variable into an object (ala B5000 and EULER), formed a total identification between “concurrent processes” and “data” consolidating notions such as arrays, files, lists, “subroutines” files (ala 528-340), 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-and idea held by many (notably Irons, Leavenworth, etc.)

The Control Definition Language of Dave Fisher (1971) provided many ideas, solutions and approaches to the notion of control. It, with FLEX, is the major source for the semantics of Simon. It is a "combinator" to FLEX: independently invented and independently of the same problems and very frequently arriving at similar answers. 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 K.S. Barton at Alta ski lodge (1968) about computers as communication 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 ISP and JOSA (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" incounatorial—i.e., $b$ means "call a with a message consisting of the token '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 'a', 'b', etc., should also be intensional. This led to an entire philosophy of use (unlike SIMULA '67) to put EVERYTHING in class definitions including the so-called "infix operators". This message idea allows messages to have a wide range of form since all messages can be received incrementally.

"Control of control" allows control structures to be defined. The language SMALLTALK itself thus avoids "primitives" such as 'loop...poop', 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 Shoch. 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 VONL (such as a computer).

This is the basic principle of recursive design. SMALLTALK recasts 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 implemted and tremendously improved by Steve Purcell.

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

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

"Music" was designed and implemented by Alan Kay.

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

We would like to thank CSL and POLOS in general for a great deal of all kinds of help.
Appendix IV: Event Driven Loop Example

First we make a class for events:

```smalltalk
to event | mycode
  (narrow
      mycode <- array1
      mycode[2I] <- 'done.'
  )
  case mycode
      (0...)  
      (1...)  
      (2...)  
      (3...)  
      (not equal) mycode end
  end
```

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

```smalltalk
to until temponame:statement
  (repeat
      (temptoname <- 1)
      temptoname <- event,
      "an indirect start to whatever was in the message"
      event <- (again) done
  )
  (do when statement)
      "the loop body to be ended"
      "pick up an event-case label"
      temptoname end is event
      (on temptoname and event:
          "pick up the corresponding code"
          done)
  repeat (statement end)
```

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

Appendix V: Smalltalk-76 Internal Structures

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