Box 1
Folder 1

adele

To: Don Steininger

From: Alan Kay

Subject: a brief summary of my thoughts about Xerox's future in the info-biz.

Don, here is a collection of writings concerning the Long-Range Plan. Much of it was written *at heat* -- the subject tends to get my juices flowing! I would suggest that you use discretion in copying and distributing this offering. Included are:

- **A Simple Vision of the Future** -- a brief update of my 1971 Pendery Paper.
- **Material and Content** -- an analogy between Xerox's current products and those of its future.
- **Arthur C. Clarke's "Laws"** -- the believability of scientists and culture shock.
- **Loose Ends** -- a summary of long-standing Xerox research concerns which need to be addressed
- **Components and Organization** -- a note I prepared for George last year about the relative approaches of Xerox and H-P to computer development
- **My Reactions to the Long-Range Plan** -- written in 1975, it is an impassioned reaction to the long-range plan that year.

  Though it is a trifle sarcastic and pejorative, it is included as an incentive for Xerox Research to actually develop a plan this time rather than the usual simple extrapolation of the present combined with expected head-count increases.
- **A Wall Street Journal Article About Xerox** -- pertinent and penetrating.

To me, a plan is a gadget that has some goals, followed by strategies which generate tactics to achieve those goals. The so-called "long-range plans" I have seen so far, as far as I can tell, can be construed as "plans" only by Xerox bean-counters, not by Xerox scientists, and certainly not by Xerox as a whole.

I really want to help in the planning process and I was quite disappointed when the Leesburg conference was called off. I do not think that exchanging memos will help nearly as much as some face-to-face confrontations on the issues.

A Simple Vision of the Future

A Brief Update of My 1971 Pendery Paper

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

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

The *Dynabook* will be well along in the process of replacing paper in the home, school, and office. The combination of portability and the additional value supplied by flexible editing and cross-referencing will hasten the changeover.

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

Dynabooks will also be used as servers in the information utilities. They will have enough power to be entirely shaped by software.

The Main Points of This Vision

- There need only be a few hardware types to handle almost all of the processing activity of a system.
- Personal Computers, Communications Links, and Information Utilities are the three critical components of a Xerox future.
- In the 1990's programming systems will be strong enough to allow the personal computer owners to specify most of the information handling tools they will need to go about their life.
- The major revenue producing systems will be service charges for the communication links and the functions provided by the information utilities <u>for use in one's personal computer</u>. In particular, an information utility's ability to provide dynamic modeling and cross-referencing services will be central to its success.

Now, What About the 1980's ?

The early 1980's should first see a complete Xerox product line based on the simplest version of the above model.

- User programming will not be possible.
- The services provided by the information utility will be rather sparse. They will be strong on simple interactive systems, and weak on comprehensive information storage and retrieval.
- There should be research versions of the *Dynabook*, its programming system, and more comprehensive information utilities.

The mid '80's will be critical.

- Customer-Service-Representative programming in the high-level language of the

Dynabook will help to open up new markets for Xerox. The "programmers" will be people trained at Leesburg to market and provide tailor-made system services to customers.

- The information utilities should be able to handle dynamic models of businesses and other complex environments.

If all this works out, the 1980's and '90's will be bright indeed. Unfortunately, Xerox is not working very hard in a number of important areas.

- The first would be for Xerox to get a firm and uncomplicated understanding of the implications arising from an entry into the information business.
- Hardware technologies, both primary (LSI, ...) and ancillary (flat-screen displays), need considerably more attention.
- Software technology has to be installed into the company as the stuff from which the main product line will be fashioned.

Material and Content

An analogy between Xerox's current products and those of its future

First, it is very important for Xerox's top management to understand some of the key differences between the hardware units they are used to dealing with and the computer-based hardware-software systems of the 1980's.

Function in a Xerox copier is delivered primarily through the workings of a particular hardware configuration. This hardware has to be invented, designed, developed, built, (and later retrofitted), all with some associated cost. Generally, different xerographic functions are provided by different xerographic engines which are the product of separate development projects. They look different, act different, and are sold with a different approach.

For contrast, function in a computer-based system need have almost nothing to do with the particular hardware involved. Function is delivered primarily through a software system.

Another way to visualize this is to consider when *material* becomes *content.*

Domain	Substance	Material	Content
Xerography	atoms, molecules	selenium, steel, plastic, brass	3100 7000 9200 etc.
Computers	silicon, steel, ...	powerful, general purp. comp.	Text Editor Inf-Str.-Retr. Inf. Network etc.

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 very important guidelines to be drawn from this:

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

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

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

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

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

Arthur C. Clarke's "Laws"

The Believability of Scientists and Culture Shock

It is interesting to examine the dynamics of human decision making. In the absence of the clear-cut evidence which forces a particular decision, there springs forth a recognizable collection of approval tendencies:

- Plans which require little or no change tend to win out over those which differ with the practices of the past.
- The ability to take risk is inversely proportional to the square of a planner's distance to retirement and to the cube of the technological complexity of the planning domain.
- A planner will tend to believe himself rather than someone else.
- A planner will tend to believe a contemporary rather than someone older or younger.
- A planner will tend to believe someone distinguished rather than someone who is not.

All of these tendencies can be neatly summarized by Arthur C. Clarke's two "Laws":

- ▸The technology of a culture appears as *magic* to a culture without that technology.
- ▸When a distinguished middle-aged scientist says something is easy or possible, he is almost always right -- when he says something is difficult or impossible, *he is almost always wrong.*

Let us suppose for the moment that Clarke's Laws are more than a British tongue in a British cheek. What can be drawn from them?

First, in planning, we need to be very careful when soliciting opinions. "Established Experts" will be useful when they are enthusiastic about projects, but advice should be sought elsewhere when the experts are pessimistic.

Second, the psychological difference between "magic" and "engineering" is enormous -- in the former there is no discernable connection between cause, energy, and effect. There is no way to tell the easy from the difficult. *And, it is extremely difficult to put together a plan that has magic as its kernal.*

To me, the point of all this is that a new approach needs to be adopted for subsequent planning cycles if a strong focus on the future is to be achieved. To be sure, the new Advanced Systems Department headed by Jerry Elkind is an excellent step in that it effectively short circuits the planners until a venture has been launched and tested. But ASD is for development. In Research, we need a similar process which allows the people who think up the ideas and do the work to also be able to do most of the major planning.

More bluntly: can we find the systems presented at Boca Raton in Xerox Research's Long-Range Plans? I have gone back, looked, and can't find them. As far as I can tell, each of the major ideas on which PARC computer research, the Boca Raton show, and the upcoming SDD product line are based does not appear in a Long-Range Plan until it was invented and built! Concepts such as SLOT, the *Alto*, the EtherNet, and the like, were heavily opposed by many in the company who had heard of them. *Alto*, in fact, did not even appear officially in a Short-Range Plan until it had been running for six months! It was a *bootleg* project at the end of 1972.

Now, someone might say: but isn't that what serendipity is all about? The unanticipated invention and discovery? But I say: serendipitous to whom? One person's serendipity (magic) is another's clear vision (technology). Gary Starkweather had SLOT clearly in mind for years until he had a chance to show its promise. I shudder when I think of how close *Alto* was to not being done at its critical time. No. There is such a thing as serendipity -- we do rely on it -- but not at this level.

This is not a polemic but a plea! Let's use planning to get the builders and the goal-setters together so that we can allocate resources to invent, design, and build powerful kernals, like *Alto*, EtherNet, and SLOT, which can shape the future of this company -- and our entire civilization.

Loose Ends

Hardware

I am not going to make a plea for the importance of the truly portable personal computer in this note. Though I think it is one of the three most important foci for Xerox Computer Research (the others: Communication Networks and Information Storage & Retrieval), it has taken so long to get some of higher management in the company just to understand the difference between computers and other machines, and between *Alto* and other computers, that it would be futile to cover some of the important human-oriented values of personal tools.

The leverage to be gained by designing hardware with its enabling software in mind is discussed in the next major section: **Components and Organization**. Here, let me just claim that we have to do it this way, and not enough of it is being done currently in the company.

LSI Processors and Memories

Currently, the design and fabrication of LSI processing and memory elements at Xerox is scattered, unorganized, and sparse. The LSI group at SSL-PARC could form a nucleus for the kind of research that Xerox needs in this area. It should be seriously examined and beefed up. The Long-Range Plan should contain specific language concerning new architectures and fabrication techniques, and a plan to considerably build up this area in the next seven years.

Displays

This is an area which generates a lot of unsubstantiated opinions as to whether Xerox should invest heavily in new display technology.

I believe my opinion can be solidly substantiated: **XEROX should definitely make a commitment to develop an ambient-light low-power flat-screen display.** Interestingly, many of those at Xerox who oppose this idea are wearing just such a display in their wrist-watch. I propose that we replace these watch displays with CRTs and re-ask their opinion after a few days!

A less facetious analogy is to the development of the Xerox machine itself. It took considerable daring, money, and energy to develop and market xerography. By having the guts and the foresight to work out a very shakey technology, Joe Wilson and others brought Xerox to the summit of American business. Xerox's future in information technology lies ultimately in being able to provide manipulable convenient *images* to its customers. It is as silly for us, as a primary strategy, to rely on (and try to improve) the CRT as it would have been for the Haloid corporation to stay with the well-understood but future-limited wet-copy processes.

The real question is: does this Xerox have the same kind of courage and foresight to take high risks for high gain? I sincerely hope the answer is yes, but so far have not been greatly cheered by recent strategies -- particularly those which have encouraged Xerox to rashly buy random digital companies of very little use for the future, such as Daconics, Diablo, etc., rather than try to do some honest internal development of ideas which might have some return!

Developing a flat-screen display for computer graphics *might* be quite difficult --

certainly one has not yet trivially appeared. We should note, though, that no one else in the computer-biz has yet done a system with the comprehensive abilities of the *Alto* in the past five years in which it has been possible. But we did the *Alto* in 1972-3, because we knew what we wanted and were willing to work out the difficulties to get it. With regard to a flat-screen display, Xerox can not really claim this until Xerox has put a critical mass of the top people in the country to our version of this task. If we can rationalize the expense of low-return aquisition-ventures with phrases like "market-probe", "experiential-exposure", and the like, surely we can decide to allocate a fraction of what has been already been lost in a project which, though of some risk, has a great payoff -- 20 million dollars over four or five years coupled with a plan would take us to a place where rational decisions about displays can be made.

Software

The software loose ends are a bit harder to describe since the software area, in general, is in better shape.

Information Storage & Retrieval

Several starts have been made in this most critical software area for the company: *Findit*, Woodstock F. S., Interim File System, Juniper. And, in a very real sense, the *Smalltalk* system of LRG, and the Understander project work of Bobrow, Winograd, et. al., are addressing themselves to the higher-level representation problems of simulation and human-oriented knowledge.

But, I still don't feel at all well about where we are in ISR, nor do I feel any better about our plans for the future.

"ISR", like "machine-translation-of-language", is a *garbage-term* in the Computer Science field. I am using it in its most general scope to cover 'most everything concerned with representation, archiving, and recovering useful models.

Part of the difficulty is that ISR isn't just one problem, it's an entire collection of problems, many of them as yet unrelated.

Another difficulty is that ISR is not terribly appealing to Computer Scientists: It is very hard, with high risk, and little glamour at the end of the trail.

However, I would like to apply what I said in the preceding section about planning to myself and my colleagues: when we start complaining that a goal, which is unarguably useful and delivers an enormous payoff, is too difficult to work on seriously, we should be examined in the light of Clarke's laws. After all, middle-age is a state of mind, not a tally of birthdays!

We need to get going on this, and now! Otherwise Xerox will end up competing with every other company which is trying to sell a computerized editing/accounting system to an ever more sophisticated market.

Higher-Level Programming

In this section, I want to point out a possible intermediate future for Xerox marketeers which lies between the non-user-programmable SDD-OIS systems of the early eighties, and, the dream of the *Dynabook*, to allow everyone to shape their own tools.

Could Xerox customer-representatives, trained at Leesburg, learn to wield a suitably higher-level language (perhaps a descendent of *Smalltalk*) well enough to tailor-make information systems for their customers? If so, this would be an excellent intermediate step for a person-oriented company like Xerox to take. The ability to provide unanticipated user function will very likely be the center strategy in the 1980's for companies in the information business.

Xerox has the bases and facilities to try an experiment like this within its own borders. I would very much like to see some interest in this expressed in the long-range plan.

Components and Organization

Just as with Gothic cathedrals, the organization of components in a computer is as important as the properties of the components themselves.

As a case in point, consider the story of two computers, the Xerox *Alto* and the Hewlitt-Packard 21MX.

In 1971, PARC decided to design and build a time-sharing computer facility called MAXC. The designs called for the use of the first generally available LSI memory elements, the Intel 1103, to be used instead of core memory. At that time, Intel had not set up good procedures for testing and screening their chips. Several PARC scientists helped Intel in order to get a quantity of 1103 chips for MAXC. While MAXC was being built, computer scientists interested in personal computing pooled their ideas and knowledge in 1972 to design and construct a very different kind of machine, the *Alto*.

The *Alto* had several ambitious goals. First, it was to be more powerful for most tasks than time-sharing would provide. Second, it had to handle a wide variety of needs, only a few of which could be explicitly anticipated. Third, it had to supply high-quality programmable graphics. Fourth, it had to be inexpensive enough to both compete successfully with a terminal and to convince Xerox that a strong future lay in the *Alto's* direction.

The goals of the *Alto* were met by hardware design decisions made in the presence of already thought out software constraints. The *Alto* was built and is still remarkably successful today as a personal computer.

At about the same time as the *Alto* was being built, Hewlitt-Packard also became interested in computers built from integrated circuit memories. The management structure of H-P is set up to get a potential product quickly to market while taking advantage of every twist and turn in the fast moving technology. In particular, H-P has been excellent in recognizing good ideas of others, and has moved quickly to purchase and develop these ideas into proprietary devices.

> As a minor example, 3M came out with a remarkable cassette design for high-quality data storage. But, their design had two important drawbacks. First, the cassette was still too bulky for its most common use. Second, no one (including 3M) had been able to build a reasonably compact drive for the cassette. H-P moved in, got 3M to develop a mini-cassette based on the same technology, and, in a masterstroke, H-P then designed and developed a proprietary drive for the cassette which is only available from H-P packaged in their terminal.

Now back to H-P and computers. H-P already had a successful line of minicomputers and they naturally wished to stay abreast of the technology in manufacturing them. They saw the 1K 1103 chip being developed by others, particlarly Univac and DEC, and decided to take a gamble on the next level of LSI memory development, the 4K RAM. H-P then did the next best thing to having their own chip company. They made a contract with one of the local silicon-valley companies which allowed them early access to the 4K RAMs. Thus, at the same time as the chip company was perfecting their manufacturing and screening process, H-P was finding out how the chips actually performed in a packaged memory. As a result, two interesting events happened almost together. First, the chip companies announced that the 4K RAMs were available, an entirely expected event. Second, H-P announced the 21MX, a stand-alone microcoded personal computer with 65K 16-bit LSI main memory, a model 31 disk drive, and a display terminal, a most unexpected event! Not only were the gross specs and appearance similar to the *Alto*, but the price tag was $20,000, a figure which was less that what we were paying to just have *Altos* built. And, the 21MX used the just announced 4K RAMS!

When we examined the fine specs for the 21MX, we disovered something even more interesting about the machine as compared to the *Alto*. The discovery was that the 21MX, though made from the same or better components (and better engineered in many respects than the *Alto*), nevertheless performed far less efficiently than PARC's personal computer. In fact, for many important tasks the 21MX was a factor of 2-5 less efficient than the *Alto*.

- For example, H-P built their memory only 16-bits wide, a slavish imitation of their earlier core-based systems, even though there is no reason to do so when using LSI memory. By contrast, the *Alto* has a 32-bit wide memory, giving a factor of two greater transfer rate when moving storage around.
- H-P still relied on a traditional interrupt system and peripheral hardware controllers for input-output. This increased the cost and lowered the generality of devices which could easily be connected to their machine. Again by contrast, the *Alto* uses a well thought-out "no overhead" process-switching mechanism which allows peripheral controllers to be simulated in software and permits hardware money to be better used elsewhere.
- Because H-P really wanted to use their nice terminal (developed primarily for their time-sharing system), it never occurred to them that they would get much more from having a tightly coupled, high-bandwidth link between the 21MX and the display. So the 21MX supplies essentially teletype communication as compared to the dynamically programmable graphics of the *Alto*.
- Finally, the overlapped fetch-and-setup instruction execution scheme of the *Alto* which permits each instruction to be fully executed in 170ns is not employed on the 21MX. Many of its instructions must wait for a register cycle before the next one can be executed.

Moral. There are two. First, a company that knows that it must take new technology to the marketplace quickly can cause much of its needed technology to happen as part of the development cycle of other companies. Second, in the computer world, as in architecture and cooking, there is a tremendous range in what can be wrought from the same ingredients.

In the case of the *Alto*, the main difference was that the important performance specs were set by people with a strong software background. Chuck Thacker, the primary designer of the *Alto*, was well versed in both hardware and software, as were the other participants in the project, Butler Lampson, Ed McCreight, and Alan Kay.

Corollary. Given the success of the basic *Alto* architecture, an obvious and rewarding strategy for Xerox would have been to find the silicon valley company which was producing the next generation of memory chips (the 16K RAMs by Mostek), get an early look at and buy of these chips, and produce an *Alto* built from these chips, the mythical *Alto* III. A few minor problems of the *Alto* could also have been remedied. The most important one of these would be to separate the display refresh mechanism so it would no longer require half the memory and micro-cycles of the machine. The result would be an *Alto* of twice the speed and from factors of one to four or more capacity.

The *Alto* III would have been a fine probe for Xerox's future markets in the world of information. For a start, it could have been all the subsequent Xerox word processors after the 800. And there are a number of excellent turn-key editing applications for which the machine would have been ideal. (Note: if Xerox is going to lose some money initially while learning about these markets, the company should probably decide to at least lose the money constructively instead of on products which not only have no future, but no present as well, such as the Xerox 850!).

Ingredients

Now let us apply these principles to the ingredients themselves. Computers are currently made from memory elements of different speeds and capacities whose contents are manipulated by separate "processor" elements connected to the memories by communications paths. To a rough degree, it is the nature of the memory system which provides most of the capability of the computer as a whole. For a company in the computer business, then, it makes a lot of sense to have as much control as possible over the sources *and the design* of the memories and the processors which are to go into a product.

Though it is undoubtably true that it makes little sense to try to duplicate someone else's technology unless there is a great deal to be gained, it is not the only maxim on which one's company should be gambled. For memories, there are many paths which may be taken. I shall consider two: strategies for quantitative and qualitative improvement.

Quantitative improvement is achieved by retaining the simple linear-addressing scheme which memories have had since the late forties and pushing hard on density, width, fabrication, and debugging techniques. The device physicists assure us that another factor of 1000 is yet to be gained for both MOS and bipolar technologies. Thus, the companies who have taken the trouble to understand electron and X-ray lithography and who have the equipment to perform it will be far ahead of the pack in bringing forth the next order of magnitude improvement. Similar comments apply to those companies who have learned how to automate the expensive design-fabricate-test cycle for new chips ("compiling silicon"). For various reasons including tradition, pin-counts, and the like, the memories on the forefront of development have all been one bit wide. Though this does not seem like much of a problem (since composite memories of any width may be constructed by putting enough chips in parallel as was done on the *Alto*), the average bandwidth per bit in a memory of a given size has been dropping steadily since the 1103. This simply means that there are so many bits in a single chip that a memory of enormous size must be configured in order to have enough chips in parallel to provide decent bandwidth.

Qualitative improvement in memory organization is a more risky path to take. Yet I have felt strongly from the time I joined PARC that Xerox must attain strength in this area to survive in the 1980's. The reason I feel confident that we can do something in this area is that qualitative improvements in memory organizations follow from software needs, and Xerox PARC, in my opinion has the best collection of software people in the country.

The problem of storage management (allocation, deallocation, relocation, exchange, and so forth) is an excellent case in point. The current technique of building software systems by emulating a higher-level environment on a simple fast microcoded computer is likely to continue for a few more years at least. And, all interesting software environments share similar storage management problems. For example, in our language *Smalltalk*, the overhead for storage management just within primary memory is 90%. This means that only 1 out of every 10 memory cycles is doing useful work for the *Smalltalk* user. The reason for all this, of course, is that the linear "von Neuman" addressing scheme which hardware memories supply has almost nothing in common with the memory requirements of high-level languages. The gap is considerably wider than that between hardware and software "processors".

Another vexing problem is the narrow bandwidth of the communications paths between traditional memory and processor elements. This is all the more frustrating since the technology we use in this day and age, MOS, is used for both memories and processors. It is natural to consider uniting processing-type and memory-type activities on a single chip, just as the concepts of processing and memory are united in *Smalltalk*.

The benefits to be gained from unifying computation fall into several categories. First, a modest set of processing capabilities at the memory level would permit many storage management problems to be dealt with efficiently. Second, more comprehensive processing capabilities would allow many software evaluation procedures to be carried through locally and in parallel at the memory level. Although the state of the art of coordinated parallel processing is not well advanced, there are nonetheless many almost independant parallel tasks in process in any advanced software system. It is important to understand that while personal computing needs enormous processor and memory bandwidths, there are very few single tasks that require all of the bandwidth of a machine. Rather, most large-bandwidth computing is formed of parallel, loosely coordinated processes which can be handled by a number of simpler, slower processors instead of one very fast one. Examples are user interaction, editing, 2-, 2.5-, and 3-D graphics, signal (audio and visual) processing, music, mathematical models, and so forth.

Conclusions. Xerox must decide whether or not it wishes to enter the world of digital information systems. If it does, the company must understand the central role that memories of all kinds will play in any products which may be developed. The most important of all of these memories for computation is so-called "primary memory". Xerox should adopt a multipronged strategy which combines close tracking of existing chip companies with a strong effort to give Xerox an independant memory and processor capability with particular regard to qualitative improvements in architecture which PARC's software expertise can provide.

As in most technological frontiers, much of the progress is accomplished by "wizards". Buying a company is not a good way to catch a wizard unless the wizard wants to be caught. It is very important for managers outside of PARC to understand this simple but important principle. A strong effort must be made to find and recruit more wizards in areas which Xerox needs to build up. If this involves buying a company, all well and good. But, the important thing is to get the wizard even if it requires an exorbitant salary and fringe benefits to do so. There is no number of good people which can be combined to produce a wizard. Therefore, we should be willing to pay some multiple of good people's salaries to get one wizard.

My Reactions To The Seven Year Plan

by Alan Kay (1975)

The first was to try to remember where I was seven years ago:

...in my second year of grad school, just finishing off the *first* design of the FLEX machine, and starting to write my Master's thesis. That was many experiences ago.

The kinds of things we can do today with digital technology and with software are *qualitatively* different now than the situation seven years ago.

Memory was 1.5 microsecond core (at 2 to 3 cents/bit), now it's .75 microsecond MOS 4K*1 chips (at .1 cents/bit), a difference of 20 - 30 to 1 in cost and about 2 to 1 in speed. Even more striking is the amazing reduction in the cost of fast register-memory and the corresponding drop in CPU cost. There was little serious talk about LSI CPU's on a chip then, or the notion of being able to dispense with moving hardware for secondary memory.

The world of computer science was almost exclusively devoted to: developing comprehensive operating systems for shared, expensive computers; new ways to translate the necessarily efficient medium-level compiler languages for these machines; developing massive shared editing systems; and chasing the elusive notion of human interation with a computer help-mate.

Of course, we all know this. The reason I bring it up is that a *qualitative* change has taken place right under our field's collective noses --- many at PARC have even been some of the prime movers who helped bring it about.

When does a quantitative change become qualitative?

The kinds of qualitative changes that are hard to grasp are those which come about through a large enough quantitative change (usually more than an order of magnitude in some dimension). The trap we so often fall into is to continue to use our old set of values in this new domain.

This is one of the basic fallacies in the paradigm Market Survey. Another is the very strange notion that the past plus the present predicts the future (usually linearly).

The Hewlett Packard HP-35 is a good example. On the one hand, it performed the very same functions which their LSI desktop machines had long been able to do, on the other, it was very small and could be used anywhere. A market survey revealed that not very many people currently used calculators and concluded that:

a. Not many more people would use a portable version. b. The range of users who would find sine and cosine, etc., useful was very small. c. There did not exist a significant market for a $500 deluxe portable calculator.

Hmmmmmmmm........

The 1974 world market for handheld calculators was ca. 10 million. HP has sold almost 1 million of their 'marketless' deluxe calculators to date.

Questions XEROX might do well to consider

1. What have been the big money makers of the 20th-century, and why?

Well.....the automobile, the airplane, the telephone, radio, silents, talking pictures, television, computers, and the xerox machine,.....

2. Are these quantitative changes in their domain? Qualitative? Both?

Well...if the automobile is a quantitative improvement over the horse and buggy, then it is one of several orders of magnitude. A more fruitful way to look at the phenomenon is that the auto *represents a qualitative change in how a person gets from place a to place b.* The fact that the horse and buggy was in the same business completely misled horse breeders and buggy manufacturers into the grand misconception of *believing that they had a captive unassailable market in the transportation domain.* And a Market Survey would have confirmed their beliefs.

Mention might be made of movies and radio vs. the stage. Then TV vs. movies and radio. Airplanes vs. the railroads. The XEROX copier vs. inexpensive carbons and Ditto machines. And so on.

Well...actually, all of the above examples have roughly the same story, including a meteoric rise in the face of absolutely contradictory marketing information. We all know the famous story of IBM, Arthur D. Little, and the 914 copier.

It's interesting to note, for example, that the movies had no difficulty displacing the stage, and yet, 30 years later, TV had even less difficulty displacing movies.

How about the paperless, electronic office vs. the XEROX copier?

Most people smart enough to pour water out of a boot might conclude that there is an important principle here which not only has guidelines about making money, but also has something important to say about human beings:

When presented with a service which offers a qualitative change in convenience in some large domain of human activity (such as communication, transportation), we may safely predict that a large proportion of the population will find a way to buy the service **even if it costs more per transaction than it did in the past.**

How's that for a marketing theory?

3. Does XEROX management understand these simple, though powerful, ideas?

Well....

4. Has XEROX lost its nerve?

Hmmmmmmmmmmm.

5. Does it make any sense at all to try and plan the future of XEROX using metaphors like 'market driven' and 'technology driven'?

The first produces better horses and buggies until bankrupcy intervenes, the second produces 'solutions' looking for 'problems' and generally requires a rather immoral advertising campaign to convince the public that they are in trouble and should **buy** to get out. Neither, in my opinion, has anything to do with a desirable future for XEROX Corporation.

6. What should XEROX be doing about its future?

First, it should ask the question: *What is it that people are going about doing in the world?*

Well, a lot of human activity has to do with love, feeding, shelter, and clothing --- it is interesting, however, that the device more ubiquitous than the flush toilet (a convenience at the end of the feeding chain), especially in slums and Appalachia, is the TV set. Other things people do is talk to each other, transmit the culture to their young, invent businesses and cities, and move from one place to another.

All of these involve communication --- humans are hooked on it --- both communication with others **and with themselves.**

Talking is communication, so is writing, so is 'education', *so is love.* A business is a money-making machine structured to facilitate communication between its parts. A city is a way to communicate with people, so is transportation, so is a XEROX copier, so is radio, TV, movies, and so on.

XEROX *is in the* **communication** business, **not** the 'information' business.

Our archetypal boot emptier musing over his beer and his boot, might conclude from all this that metaphors for XEROX Corporation's Long Range Plans should probably have something to do with *helping people to communicate in a qualitatively better way.*

Gee, it might even sell!

On the other hand, perhaps we should not expect too much from the fact that our friend was able to pour water from his boot. Is that not making a linear assumption and prediction from his past to his future?

Although we can confidently predict that he will avail himself of any qualitatively better way to communicate whether it costs more or not, we must not rely on him to be able to understand the somewhat more abstract principles behind these notions. Instead, we need to not only *show* him what his future can be like, we must *let him try it out* for himself.

Metaphors for the Long Range Plan of XEROX Corporation's **only** *Long Range Research Center most certainly should be centered about the absolute necessity to:*

a. Understand the nature of communication and invent ways to make it qualitatively better.

b. Communicate these ideas about communication to those in XEROX who have the leverage to get them into the world, **by making it possible for them to experience the future**

themselves. WORDS WILL NOT DO.

Conclusion

If we are not able to do these things in the next two years, we need not worry about the following five.

Corrollary

Our best shot at a Long Range Plan should be just a list of those things which should exist at the end of seven years plus a carefully thought-out workable plan for the first two of these years with especial emphasis on having a clear understanding of the concept of *critical mass* for successful projects, **and** XEROX's limitations in using the 'shotgun' approach in research and development.

Smudged Image

Xerox Faces Problems In Trying to Duplicate Its Own Past Success

It Has Tougher Competition In Copiers, Pricing Woes; Some Key Aides Are Lost

But It Pushes New Ventures

By Richard A. Shaffer
Staff Reporter of The Wall Street Journal

STAMFORD, Conn. — Brother Dominic, the portly monk in Xerox Corp. television commercials, knows just what to do when his abbot wants 500 copies of a medieval manuscript quickly. He borrows a Xerox 9200 duplicator and runs them off in minutes.

"It's a miracle," the delighted abbot proclaims.

Xerox itself may not need a miracle, but it could use a little help on a copying problem of its own—duplicating its past performance.

The company, whose meteoric success is legendary, smudged its image in 1975; profits dropped for the first time since its days as an obscure photographic company known as Haloid Corp. more than two decades earlier. Over the past two years, the company with a name that has become a synonym for so-called plain-paper copying has struggled to regain its reputation for growth. And despite some internal woes and growing Japanese competition, management recently has been suggesting that the worst is over.

"This year will be another good one for Xerox," says C. Peter McColough, chairman and chief executive officer. "Our growth target continues to be a 15% annual increase in earnings, and I believe that is realistic."

Growing Doubts

Some outsiders, however, aren't so sure. Although Xerox has taken several long strides in the right direction, enough other moves have seemingly gone awry that doubts about its prospects are growing. "Xerox keeps shifting directions," says Sanford J. Garrett, a securities analyst at Sanford C. Bernstein & Co., who remains generally bullish on Xerox. "I have less and less confidence that management knows what it's doing."

What Xerox says it is trying to do amounts to a metamorphosis, a transition from the office of today to the office of tomorrow. If its forecasts are accurate, over the next several years the mountains of paper work that it helped to create gradually will melt into an uncluttered world where files are stored electronically and mail zips from desk to desk via computers and television screens.

"The office of the future is no longer just a buzzword," says John R. Labinski, a manager in Xerox's Office Systems division. "The technology is here already, and demand for it is growing 25% to 30% a year."

The risk in moving into this potentially lucrative "automated office" market, however, is that Xerox will be de-emphasizing a field—copying—in which it still is top dog, and it will be entering one already dominated by a tiger nearly four times its size, International Business Machines Corp. And the worriers discern signs that Xerox may not be able to get from here to there without another dip or two in the copier profits it needs to do battle with IBM.

Spectacular Past

In a sense, these worries are the product of high expectations raised by the spectacular Xerox past. When Xerox had a monopoly on machines that made copies on ordinary paper, its revenues exploded to $3.6 billion in 1974 from $33 million in 1959, its profits to $331 million from $2 million, and its stock price to nearly $172 a share from less than $2, adjusted for several splits.

"For a while," recalls Mr. McColough, who joined the company in 1954, "people seemed to think we could walk on water."

Inevitably, their feet got wet. An antitrust settlement with the Federal Trade Commission in 1975 cost Xerox its patent protection; essentially, the company was forced to grant licenses to its competitors. Moreover, Xerox realized that its venture into the computer business had failed, and it pulled out, writing off more than $84 million in the process.

Competition in copiers became serious. IBM, which had entered the field a few years earlier, was making inroads, and, to widespread surprise, Eastman Kodak Co. jumped into the business. All that, combined with recession, inflation and the high cost of marketing new products, forced down Xerox profits for the first time since 1951.

"Chicken Little Syndrome"

The decline was minimal—a mere 2%—but the damage wasn't. "We began to get this Chicken Little syndrome, especially in the middle and lower levels of the company," recalls John C. Lewis, a former Xerox executive who now is president of Amdahl Corp. "It was like a kid who's never lost a basketball game before. The first time things don't go his way, he thinks the world is coming to an end."

No one expects Xerox to recapture its former glory—not, at least, in copiers, for market saturation is ending that boom. While the dollars spent on copying have risen at an annual rate of 20% over the past five years, observers predict that the gains will dwindle to less than half that over the next five years.

Yet Xerox is hard at work on a comeback of sorts. It has clamped down on costs, revitalized its research, unleashed a blitz of new products, charged into new markets and so turned itself around that it now is pushing sales of copiers that it once would only rent.

The company is also pushing its sales force. "The pressure to produce has become very intense," says George Funkhouser, a Xerox salesman in Portland, Ore. Major

Smudged Image: Xerox Faces Tougher Competition, Other Woes in Trying to Duplicate Own Past Success

Continued From First Page

customers say Xerox representatives are calling more frequently and staying longer—but not necessarily pressing harder.

"Our relationship with Xerox sales people today has improved 100%," comments one of the company's largest accounts. "In the past, they've behaved like the cock of the walk. They tended to oversell. Today, they've mellowed, become more responsive to the customer's needs. Keen competition appears to have had a humbling effect."

By some measures, the rebuilding attempt gets high marks.

In just two years, the company's electronic typing systems have captured about 12% of a market that once was exclusively IBM's. Xerox recently has picked up several high-technology office-machine companies whose sales are doubling annually. And last year the company brought out a printer for computers that marries the copier with the laser beam, providing a glimpse of what many believe will become its lead product in the next generation of office machines—the so-called intelligent copier, a single gadget that acts as typewriter, copier and facsimile machine.

On the financial side, earnings are perking up. Net income grew 12% last year and 18% in the final quarter. Moreover, the gains could be higher this year if the 9200 duplicators, as expected, begin to show a profit. The balance sheet has seldom looked healthier. Cash is pouring in. Debt is shrinking. And the dividend is being raised so quickly that the company says it could amount to half of profits within a few years.

Lurking Problems

Yet behind the good news lurk potentially serious problems.

For example, a major factor in the earnings upturn is the furious pace of copier sales. While revenues from rented machines have been slowing down for several years, sales revenues have been doubling annually and now represent about 11% of total revenues from copiers and duplicators. That switch boosts net income because sold machines provide quicker profits than rented ones. But it also tends to cut profit margins because ultimately rentals are more profitable. And thus, as the company's revenue mix has swung increasingly to sales, operating margins have declined to 19% last year from 24% in 1974. In addition, the sheer volume of sales suggests that the pace, now so crucial to profits, eventually will moderate.

"There's a time fuse on the company," says Brian Fernandez, a securities analyst at First Manhattan Co.

Assessments of some other Xerox recovery efforts are equally negative.

Once so free-spending that employes flew first class and stayed in the best hotels, Xerox two years ago launched its first company-wide cost-cutting drive. With the ambitious goal of a 40% cutback in overhead, the company fired thousands of employes, closed facilities and delayed construction of a new corporate headquarters.

Yet overhead keeps climbing. Last year, selling, service, administrative and general expenses chewed up 49.8% of total revenues, up from 47.6% just prior to the cost-cutting and 39.1% five years before that. Inflation has wiped out the savings, explains David T. Kearns, Xerox president. "At some point in time—perhaps three or four years—we will have to raise prices, or we will be unable to stop this erosion," he says.

Pricing Woes

Similarly, a change in pricing strategy seems to have helped little.

In its heyday, Xerox prospered by renting out more and more copiers. But a few years ago, management decided to seek growth by encouraging customers to make more copies on the machines they already had. So in 1976, after the settlement with the FTC cleared the way, Xerox cut rental fees as much as 11% in the belief that revenues would rise more than enough to overcome the lower price per copy.

Xerox won't say whether the maneuver worked. But according to industry sources, who say Xerox privately confirms their findings, average annual revenues per copier have dropped in the past two years for all Xerox machine families except color copiers and the high-volume 9200 duplicator.

Whatever the case, even Xerox concedes that it was outflanked on another pricing front. Although Xerox fully expected the Japanese to enter the world-wide copier market—even sooner, in fact, than they did—the company now says it focused so narrowly on competing with Kodak and IBM that the Japanese slipped in almost unnoticed.

The embarrassing consequence: The prime Xerox foe has turned out not to be a fellow corporate mammoth but a pygmy Valhalla, N.Y., company, Savin Business Machines Corp., whose annual revenues are less than half the size of the Xerox research budget. Savin, which sells low-priced desktop copiers made by Ricoh Co. of Tokyo, says it placed more than 40,000 plain-paper copiers in the U.S. last year, up from a handful two years earlier. By contrast, Dataquest Inc., a California market-research company, estimates that Xerox added only 25,000 low-volume copiers to its U.S. total in 1977.

Fighting Back

Lately Xerox has begun fighting back, slashing prices on low-volume machines, where Japanese competition is heaviest. The price of the popular Model 3100, for example, was cut from $12,000 to as little as $4,400 for customers who want to buy the machines they now are renting.

However, Xerox still seems to be losing ground. Although the number of installed Xerox machines far surpasses the total of all other brands combined, more and more new customers for small copiers are going elsewhere. Dataquest has concluded that Xerox's share of new business in low-volume copiers plunged last year to 22% from 37.5% two years earlier.

"Very bluntly, Xerox has blown it," says David G. Jorgensen, a Dataquest senior partner. "They're making the right moves, if they'd made them sooner. But now this huge snowball is rolling down on them at a whopping rate and there's just too much momentum."

Xerox won't discuss market share. But it does term the situation serious enough to warrant reorganizing its sales force. And in recent weeks, several marketing changes have been made in an attempt to uncover more new customers rather than simply to sell more to current ones.

While the small Xerox machines haven't swamped the competition, neither have the big ones that the company calls the key to its future in copying.

Battle of Duplicators

Invading a market long ruled by Addressograph-Multigraph Corp. and A. B. Dick Co., Xerox in 1974 came out with the 9200 duplicator, which was designed to replace the A-M and Dick offset presses used by large corporations and government agencies. The 9200 is fast; it spits out two copies a second. But Xerox was slow in getting it out the factory door, and competitors took advantage of the delay.

Addressograph, for one, had time to reorganize its sales force and launch a $1 million advertising campaign that boasted, "A-M beats Xerox or you get our copymaker free for a year." The result: After an initial drop, Addressograph's offset-press sales rebounded and now are growing at about their former rate, the company says.

Industry sources estimate that placements of the 9200 are running at less than half the company's original expectations. However, Xerox is calling the machine a success, although it acknowledges that it overestimated the 9200's prospects and misunderstood the market. So Xerox had to fiddle with the machine's price. And when it introduced the improved 9400 duplicator last fall, Xerox pitted the older machine against slower equipment from Kodak and IBM instead of the offset presses.

"The 9200 is a great machine, but as an example of Xerox product planning, it was a fiasco," a former director says. "It was a monumental miscalculation that cost us hundreds of millions of dollars in unnecessary inventory and tooling expense."

Doubts About Management

Overall, what Xerox followers worry most about is the quality of management. Guiding a fast-growing company with a novel product is difficult, but dealing with today's increasingly competitive environment is even harder, and many critics doubt that Xerox executives have the requisite skills.

Most of the criticism naturally centers on Mr. McColough. The Xerox chairman is, of course, credited with a key role in the company's phenomenal growth. Sanford Kaplan, a retired Xerox director, describes Mr. McColough as "a born marketing type," adding, "He's very bright and quick with numbers."

But Mr. McColough is blamed for such mistakes as the venture into computer-making and the failure to acquire CIT Financial Corp. when Xerox had the chance. In addition, critics contend that when competitors began making inroads, his reaction was too little and too late.

A handsome, balding man, Mr. McColough is a Canadian-born lawyer with a Harvard M.B.A. and an idealistic view of Xerox's social obligations. At one annual meeting when an angry shareholder assailed the company's hefty donations to charity, he retorted, "You can sell your stock or try to throw us out, but we aren't going to change."

His management style is equally direct. "Peter listens to the men under him, but it is always very, very clear who makes the final decisions," says Robert A. Beck, president of Prudential Insurance and a Xerox director. One former associate of Mr. McColough's describes him as "a loner, a cold-eyed businessman with few passions besides sailing and politics."

Many Departures

After working his own way up through the ranks, Mr. McColough recruited most of his top aides from outside—from Ford, General Motors and IBM—and lately some have left. In the past few years, eight star players have quit the team, including the company's general counsel, four vice presidents and even the president. Primarily, they left for a chance to call their own plays at companies like Singer, LTV and White Motor Corp., sources at Xerox say. But money also cast its spell. For example, although Archie R. McCardell will earn about the same salary as president of International Harvester as he did when president of Xerox—$460,000 a year—he also will get bonuses in stock and cash that could total nearly $3 million.

The departures are putting some important segments of the business through their fourth management change in three years. A few departments are "disrupted, frustrated and delayed," one executive says.

The new president, Mr. Kearns, is a former IBM executive. A sometime pilot and fitness buff who takes a sweatsuit and track shoes on business trips, he comes on as both aggressive and candid. Departing from the guarded comments that had become typical of Xerox, he readily concedes that Xerox has made several missteps, and he vows that he will correct them.

Talking to securities analysts, Xerox trainees and the media, Mr. Kearns has criticized the company for being slow to react to competition, inefficient in research and sometimes guilty of following strategies at cross-purposes to each other.

Problems "Solvable"

"I've been part of some of the mistakes Xerox has made in the past, but I'm not afraid of mistakes," he says. "There are a lot of things wrong with the company, but they are solvable. It will be a tougher job than in the past, but I am confident it can be done."

His chance will come gradually. Mr. McColough, chief executive since 1968, has five more years to go before stepping down under a Xerox policy calling for executives to retire at age 60. At present, the entire corporate staff reports directly to Mr. McColough.

"I'm sure as time goes by Dave will get added responsibility, but I don't want to put too much of a burden on him all at once," Mr. McColough says. "Keep in mind that I'm also impressed with him, though. After all, I picked him."

THE WALL STREET JOURNAL
Thursday, Feb. 16, 1978

THE EARLY HISTORY OF SMALLTALK

Alan C. Kay
Apple Computer
kay2@apple.com.Internet#

Abstract

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

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

Table Of Contents

—To Dan Ingalls, Adele Goldberg and the rest of the Xerox PARC LRG gang
—To Dave Evans, Bob Barton, Marvin Minsky, and Seymour Papert
—To SKETCHPAD, JOSS, LISP, and SIMULA, the 4 great programming conceptions of the sixties

Introduction

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

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

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

Programming languages can be categorized in a number of ways: imperative, applicative, logic-based, problem-oriented, etc. But they all seem to be either an "agglutination of features" or a "crystalization 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 all seem to have been instigated by committees, and the crystalization languages by a single person.

Smalltalk's design—and existence—is due to the insight that everything we can describe can be represented by the recursive composition of a single kind of behavioral building block that hides its combination of state and process inside itself and can be dealt with only through the exchange of messages. Philosophically, Smalltalk's objects have much in common with the monads of Leibniz and the notions of 20th century physics and biology. Its way of making objects is quite Platonic in that some of them act as idealisations of concepts—*Ideas*—from which *manifestations* can be created. That the Ideas are themselves manifestations (of the Idea-Idea) and that the Idea-Idea is *a-kind-of* Manifestation-Idea—which is a-kind-of itself, so that the system is completely self-describing—would have been appreciated by Plato as an extremely practical joke [Plato].

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

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

> "We would know what they thought when they did it"
> —Richard Hamming
>
> "Memory and imagination are but two words for the same thing"
> —Thomas Hobbes

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

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

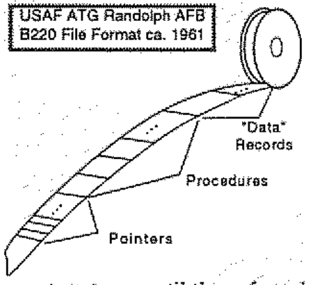
This history is too long, but I was amazed at how many people and systems that had an influence appear only as shadows or not at all. I am sorry not to be able to say more about Bob Balzer, Bob Barton, Danny Bobrow, Steve Carr, Wes Clark, Barbara Deutsch, Peter Deutsch, Bill Duvall, Bob Flegal, Laura Gould, Bruce Horn, Butler Lampson, Dave Liddle, William Newman, Bill Paxton, Trygve Reenskaug, Dave Robson, Doug Ross, Paul Rovner, Bob Sproull, Dan Swinehart, Bert Sutherland, Bob Taylor, Warren Teitelman, Bonnie Tennenbaum, Chuck Thacker, and John Warnock. Worse, I have omitted to mention many systems whose design I detested, but that generated considerable useful ideas and attitudes in reaction. In other words "histories" should not be believed very seriously but considered as "FEEBLE GESTURES OFF" done long after the actors have departed the stage.

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

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

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

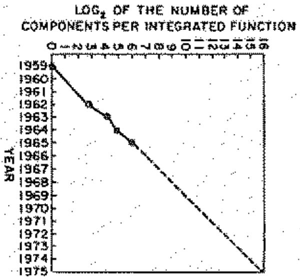
New ideas go through stages of acceptance, both from within and without. From within, the sequence moves from "barely seeing" a pattern several times, then noting it but not perceiving its "cosmic" significance, then using it operationally in several areas, then comes a "grand rotation" in which the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



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

The second barely-seeing of the idea came just a little later when ATC decided to replace the 220 with a B5000. I didn't have the perspective to really appreciate it at the time, but I did take note of its segmented storage system, its efficency of HLL compilation and byte-coded execution, its automatic mechanisms for subrountine 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,1961] [Burroughs,1961]

After the Air Force, I worked my way through the rest of college by programming mostly retrieval systems for large collections of weather data for the National Center for Atmospheric Research. I got interested in simulation in general—particularly of one machine by another—but aside from doing a one-dimensional version of a bit-field block transfer (bitblt) on a CDC 6600 to simulate word sizes of various machines, most of my attention was distracted by school, or I should say the theatre at school. While in Chippewa Falls helping to debug the 6600, I read an article by Gordon Moore which predicted that integrated silicon on chips was going to exponentially improve in density and cost over many years [Moore 65]. At that time in 1965, standing next to the room-sized freon-cooled 10 MIP 6600, his astounding predictions had little projection into my horizons.



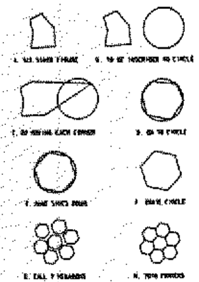
Gordon Moore's "Law"

Sketchpad and Simula

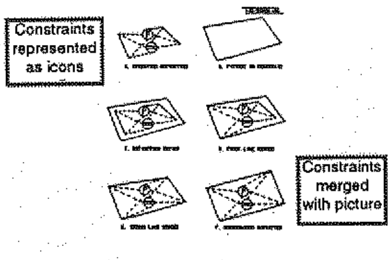
Through a series of flukes, I wound up in graduate school at the University of Utah in the Fall of 1966, "knowing nothing". That is to say, I had never heard of ARPA or its projects, or that Utah's main goal in this community was to solve the "hidden line" problem in 3D graphics, until I actually walked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

Every newcomer got one. The title was "Sketchpad: A man-machine graphical communication system"[Sutherland, 1963]. What it could do was quite remarkable, and completely foreign to any use of a computer I had ever encountered. The three big ideas that were easiest to grapple with were: it was the invention of modern interactive computer graphics; things were described by making a "master drawing" that could produce "instance drawings"; control and dynamics were supplied by "constraints", also in graphical form, that could be applied to the masters to shape and inter-relate parts. Its data structures were hard to understand—the only vaguely familiar construct was the embedding of pointers to procedures and using a process called reverse indexing to jump though them to routines, like the 220 file system[Ross,1961]. It was the first to have clipping and zooming windows—one "sketched" on a virtual sheet about 1/3 mile square!

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



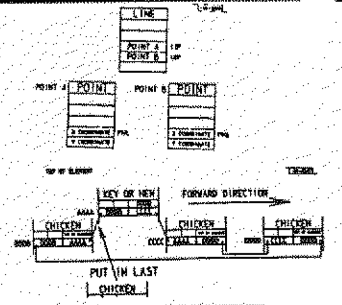
Drawing in Sketchpad



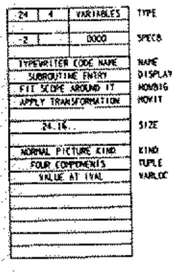
Constraints represented as icons

Constraints merged with picture

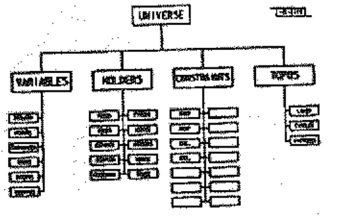
Programming with constraints



Sketchpad Structures



"generic block" showing procedural attachment



Sketchpad's "inheritance" hierarchy

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

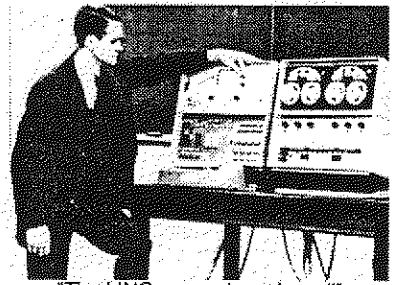
The documentation was incomprehensible. Supposedly, this was the Case-Western Reserve 1107 Algol—but it had been doctored to make a language called Simula; the documentation read like Norwegian transliterated into English, which in fact it was. There were uses of words like *activity* and *process* that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

Finally, another graduate student and I unrolled the program listing 80 feet down the hall and crawled over it yelling discoveries to each other. The weirdest part was the storage allocator, which did not obey a stack discipline as was usual for Algol. A few days later, that provided the clue. What Simula was allocating were structures very much like the instances of Sketchpad. There were descriptions that acted like masters and they could create instances, each of which was an independent entity. What Sketchpad called masters and instances, Simula called activities and processes. Moreover, Simula was a procedural language for controlling Sketchpad-like objects, thus having considerably more flexibility than constraints (though at some cost in elegance) [Nygaard,1966, Nygaard, 1983].

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

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

II. 1967-69—The FLEX Machine, a first attempt at an OOP-based personal computer



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

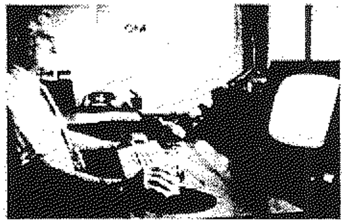
Dave Evans was not a great believer in graduate school as an institution. As with many of the ARPA "contractors" he wanted his students to be doing "real things"; they should move through graduate school as quickly as possible; and their theses should advance the state of the art. Dave would often get consulting jobs for his students, and in early 1967, he introduced me to Ed Cheadle, a friendly hardware genius at a local aerospace company who was working on a "little machine". It was not the first personal computer—that was the LINC of Wes Clark—but Ed wanted it for noncomputer professionals, in particular, he wanted to program it in a higher level language, like BASIC. I said: "What about JOSS? It's nicer." He said: "Sure, whatever you think", and that was the start of a very pleasant collaboration we called the FLEX machine. As we got deeper into the design, we realized that we wanted to dynamically *simulate* and *extend*, neither of which JOSS (or any existing language that I knew of) was particularly good at. The machine was too small for Simula, so that was out. The beauty of JOSS was the extreme attention of its design to the end-user—in this respect, it has not been surpassed[Joss,1964, Joss,1978]. JOSS was too slow for serious computing (but cf. Lampson 65), did not have real procedures, variable scope, and so forth. A language that looked a little like JOSS but had considerably more potential power was Wirth's EULER[Wirth 1966]. This was a generalization of Algol along lines first set forth by van Wijngaarden [van Wijngaarden 1963] in which types were discarded, different features consolidated, procedures were made into first class objects, and so forth. Actually kind of LISPlike, but without the deeper insights of LISP.

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

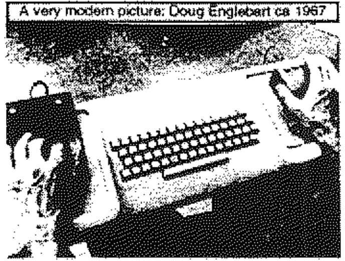
The semantics of what was now called the FLEX language needed to be influenced more by Simula than by Algol or EULER. But it was not completely clear how. Nor was it clear how the user should interact with the system. Ed had a display (for graphing, etc.) even on his first machine, and the LINC had a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

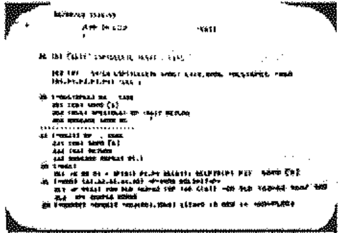
Doug Engelbart and NLS

This was in early 1967, and while we were pondering the FLEX machine, Utah was visited by Doug Engelbart. A prophet of Biblical dimensions, he was very much one of the fathers of what on the FLEX machine I had started to call "personal computing". He actually traveled with his own 16mm projector with a remote control for starting and stopping it to show what was going on (people were not used to seeing and following cursors back then). His notion of the ARPA dream was that the destiny of oNLine Systems (NLS) was the "augmentation of human intellect" via an interactive vehicle navigating through "thought vectors in concept space". What his system could do then—even by today's standards—was incredible. Not just hypertext, but graphics, multiple panes, efficient navigation and command input, interactive collaborative work, etc. An entire conceptual world and world view[Engelbart 68]. The impact of this vision was to produce in the minds of those who were "eager to be augmented" a compelling metaphor of what interactive computing should be like, and I immediately adopted many of the ideas for the FLEX machine.

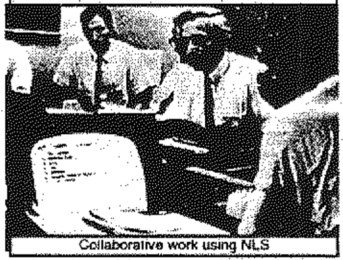


A very modern picture: Doug Englebart ca 1967





Multiple Panes and View Specs in NLS



Collaborative work using NLS

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

Instead of at most a few thousand *institutional* mainframes in the world—even today in 1992 it is estimated that there are only 4000 IBM mainframes in the entire world—and at most a few thousand users trained for each application, there would be millions of *personal* machines and users, mostly outside of direct institutional control. Where would the applications and training come from? Why should we expect an applications programmer to anticipate the specific needs of a particular one of the millions of potential users? An *extensional* system seemed to be called for in which the end-users would do most of the tailoring (and even some of the direct construction) of their tools. ARPA had already figured this out in the context of their early successes in time-sharing. Their larger metaphor of human-computer symbiosis helped the community avoid making a religon of their subgoals and kept them focused on the abstract holy grail of "augmentation".

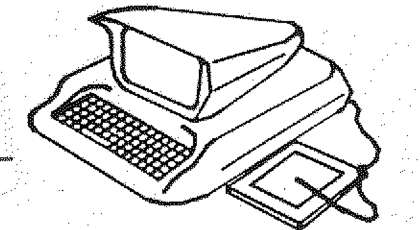
One of the interesting features of NLS was that its user interface was parametric and could be supplied by the end user in the form of a "grammar of interaction" given in their compiler-compiler TreeMeta. This was similar to William Newman's early "Reaction Handler" [Newman 66] work in

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

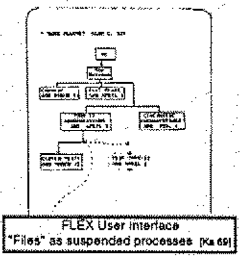
Again, the scope of the FLEX machine was too small for a miniNLS, and we were forced to find alternate designs that would incorporate some of the power of the new ideas, and in some cases to improve them. I decided that Sketchpad's notion of a general window that viewed a larger virtual world was a better idea than restricted horizontal panes and with Ed came up with a clipping algorithm very similar to that under development at the same time by Sutherland and his students at Harvard for the 3D "virtual reality" helmet project [Sutherland 1968].

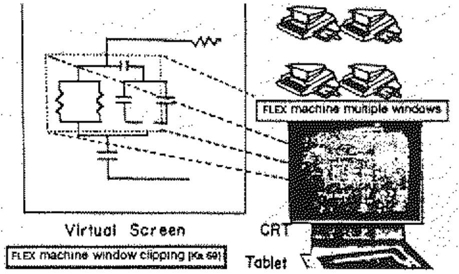
Object references were handled on the FLEX machine as a generalization of B5000 descriptors. Instead of a few formats for referencing numbers, arrays, and procedures, a FLEX descriptor contained two pointers: the first to the "master" of the object, and the second to the object instance (later we realized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

Invert? the behavior according to? (1) state (2) rules (3) constraints (4) intention



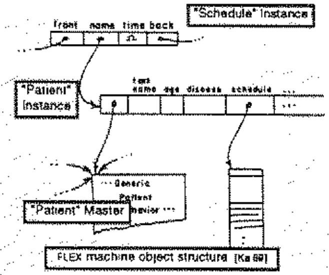
The FLEX Machine Self Portrait, ca 1968 [Ka 69]





FLEX machine multiple windows

FLEX machine window clipping [Ka 69]



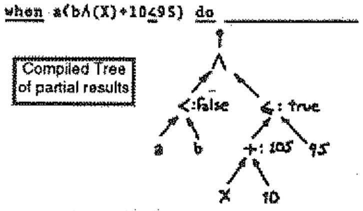
FLEX machine object structure [Ka 69]

Carnap[Ca *] helped by showing that "intensional" definitions covered the same territory as the more traditional extensional technique and were often more intuitive and convenient.

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

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

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

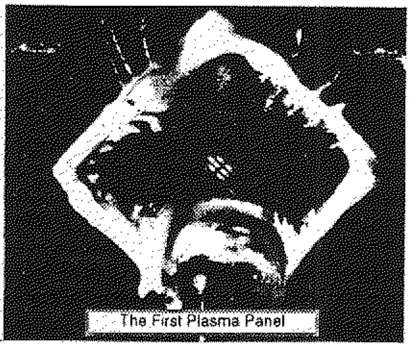


FLEX **when** statement [Ka 69]

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

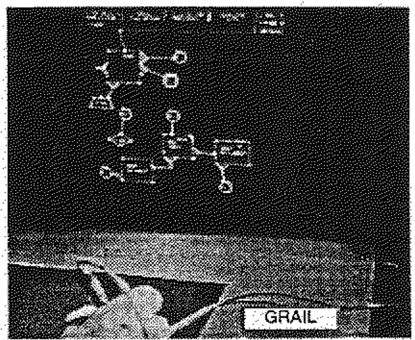
Still, quite a few of the original FLEX ideas in their proto-object form did turn out to be small enough to be feasible on the machine. I was writing the first compiler when something unusual happened: the Utah graduate students got invited to the ARPA contractors meeting held that year at Alta, Utah. Towards the end of the three days, Bob Taylor, who had succeeded Ivan Sutherland as head of ARPA-IPTO, asked the graduate students (sitting in a ring around the outside of the 20 or so contractors) if they had any comments. John Warnock raised his hand and pointed out that since the ARPA grad students would all soon be colleagues (and since we did all the real work anyway), ARPA should have a contractors-type meeting each year for the grad students. Taylor thought this was a great idea and set it up for the next summer.

Another ski-lodge meeting happened in Park City later that spring. The general topic was education and it was the first time I heard Marvin Minsky speak. He put forth a terrific diatribe against traditional educational methods, and from him I heard the ideas of Piaget and Papert for the first time. Marvin's talk was about how we think about complex situations and why schools are really bad places to learn these skills. He didn't have to make any claims about computers+kids to make his point. It was clear that education and learning had to be rethought in the light of 20th century cognitive psychology and how good thinkers really think. Computing enters as a new representation system with new and useful metaphors for dealing with complexity, especially of systems [Minsky 70].



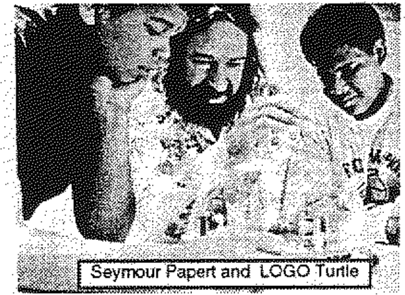
The First Plasma Panel

For the summer 1968 ARPA grad students meeting at Allerton House in Illinois, I boiled all the mechanisms in the FLEX machine down into one 2'x3' chart. This included all of the "object structures", the compiler, the 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 to U of Illinois where I saw a 1" square lump of glass and neon gas in which individual spots would light up on command—it was the first flat-panel display. I spent the rest of the conference calculating just when the silicon of the FLEX machine could be put on the back of the display. According to Gordon Moore's "Law", the answer seemed to be sometime in the late seventies or early eighties. A long time off—it seemed too long to worry much about it then.



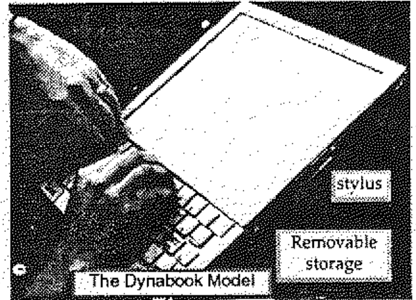
GRAIL

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



Seymour Papert and LOGO Turtle

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



The Dynabook Model

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

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

> It has been such a long time since I have seen so many familiar faces shouting among so many familiar ideas. Discovery of something new in programming languages, like any discovery, has somewhat the same sequence of emotions as falling in love. A sharp elation followed by euphoria, a feeling of uniqueness, and ultimately the wandering 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 IMP, a system that had already been working for several years that was more elegant than most of the nonworking proposals. The basic idea of IMP was that you could use any phrase in the grammar as a procedure heading and write a semantic definition in terms of the language as extended so far [Irons, 1970].

I had already made the first version of the FLEX machine syntax driven, but where the meaning of a phrase was defined in the more usual way as the kind of code that was emitted. This separated the compiler-extensor part of the system from the end-user. In Irons' approach, *every* procedure in the system defined its own syntax in a natural and useful manner. I incorporated these ideas into the second version of the FLEX machine and started to experiment with the idea of a direct interpreter rather than a syntax directed compiler. Somewhere in all of this, I realized that the bridge to an object-based system could be in terms of each object as a syntax directed interpreter of messages sent to it. In one fell swoop this would unify object-oriented semantics with the ideal of a completely extensible language. The mental image was one of separate computers sending requests to other computers that had to be accepted and understood by the receivers before anything could happen. In today's terms every object would be a *server* offering *services* whose deployment and discretion depended entirely on the server's notion of relationship with the servee. As Liebniz said: "To get everything out of nothing, you only need to find one principle". This was not well thought out enough to do the FLEX machine any good, but formed a good point of departure for my thesis [Kay 69], which as Ivan Sutherland liked to say was "anything you can get three people to sign".

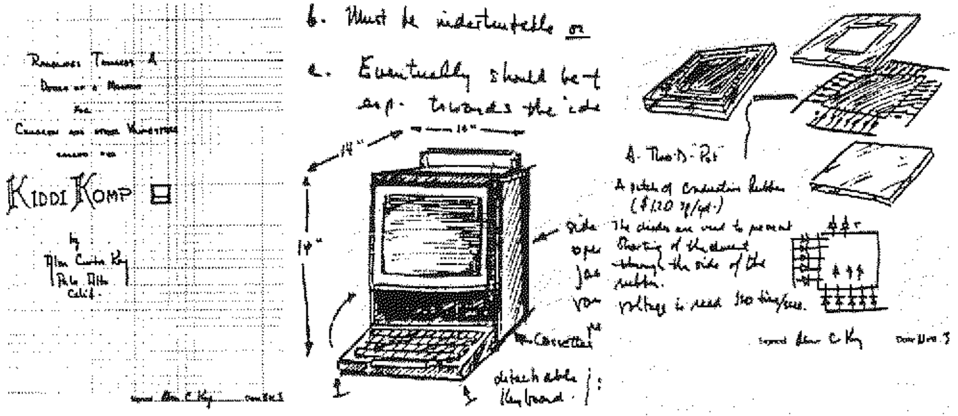
After three people signed it (Ivan was one of them), I went to the Stanford AI project and spent much more time thinking about notebook KiddyKomputers than AI. But there were two AI designs that were very intriguing. The first was Carl Hewitt's PLANNER, a programmable logic system that formed the deductive basis of Winograd's SHRDLU [Sussman 69, Hewitt 69]. I designed several languages based on a combination of the pattern matching schemes of FLEX and PLANNER [Kay 70]. The second design was Pat Winston's concept formation system, a scheme for building semantic networks and comparing them to form analogies and learning processes [Winston 70]. It was kind of "object-oriented". One of its many good ideas was that the arcs of each net which served as attributes in AOV triples should themselves be modeled as nets. Thus, for example a first order arc called LEFT-OF could be asked a higher order question such as "What is your converse?" and its net could answer: RIGHT-OF. This point of view later formed the basis for Minsky's frame systems [Minsky 75]. A few years later I wished I had paid more attention to this idea.

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

The biggest hit for me while at SAIL in late '69 was to *really understand* LISP. Of course, every student knew about *car*, *cdr*, and *cons*, but Utah was impoverished in that no one there used LISP and hence, no one had penetrated the mysteries of *eval* and *apply*. I could hardly believe how beautiful and wonderful the *idea* of LISP was [McCarthy,1960]. I say it this way because LISP had not only been around enough to get some honest barnacles, but worse, there were deep flaws in its logical foundations. By this, I mean that the pure language was supposed to be based on functions, but its most important components—such as lambda expressions, quotes, and conds—were not functions at all, and instead were called special forms. Landin and others had been able to get quotes and conds in terms of lambda by tricks that were variously clever and useful, but the flaw remained in the jewel. In the practical language things were better. There were not just EXPRs (which evaluated their arguments), but FEXPRs (which did not). My next question was, why on earth call it a functional language? Why not just base everything on FEXPRs and force evaluation on the receiving side when needed? I could never get a good answer[1], but the question was very helpful when it came time to invent Smalltalk, because this started a line of thought that said "take the hardest and most profound thing you need to do, make it great, and then build every easier thing out of it". That was the promise of LISP and the lure of lambda—needed was a better "hardest and most profound" thing. Objects should be it.

III. 1970-72—Xerox PARC: The KiddiKomp, miniCOM, and Smalltalk-71

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



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

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

Right after New Years 1971, Bob Taylor scored an enormous coup by attracting most of the struggling Berkeley Computer Corp to PARC. This group included Butler Lampson, Chuck Thacker, Peter Deutsch, Jim Mitchell, Dick Shoup, Willie Sue Haugeland, and Ed Fiala. Jim Mitchell urged the group to hire Ed McCreight from CMU and he arrived soon after. Gary Starkweather was there already, having been thrown out of the Xerox Rochester Labs for wanting to build a laser printer (which was against the local religon). Not long after, many of Doug Englebart's people joined up—part of the reason was that they want to reimplement NLS as a distributed network system, and Doug wanted to stay with time-sharing. The group included Bill English (the co-inventor of the mouse), Jeff Rulifson, and Bill Paxton.

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

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

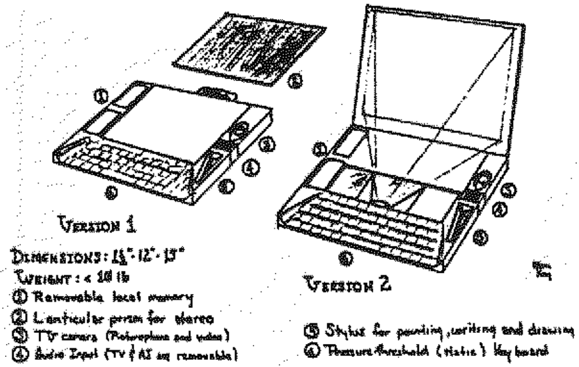
```
oddsEvens(x) = append(odds(x), evens(x))
```

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

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

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

Right around this time we were involved in another conflict with Xerox management, in particular with Don Pendery the head "planner". He really didn't understand what we were talking about and instead was interested in "trends" and "what was the future going to be like" and how could Xerox "defend against it". I got so upset I said to him, "Look. *The best way to predict the future is to invent it.* Don't worry about what all those other people might do, this is the century in which almost any clear vision can be made!" He remained unconvinced, and that led to the famous "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".

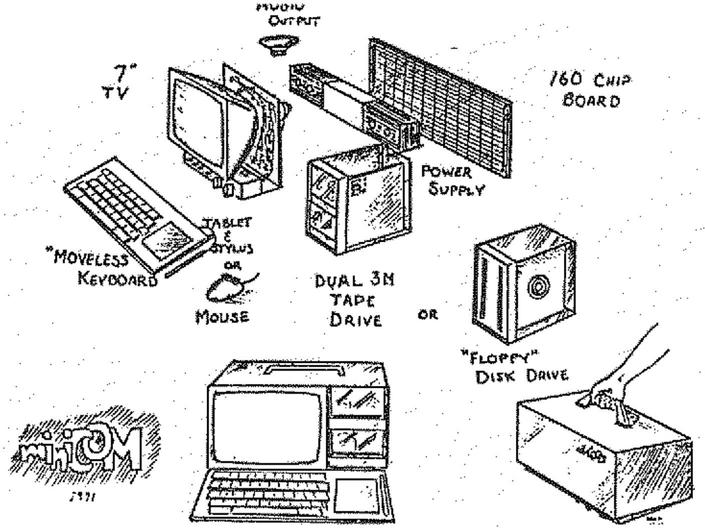

①
②
③
④
⑤
⑥
Version 1
Version 2
Dimensions: 13"·12"·13"
Weight: < 10 lb
① Removable local memory
② Lenticular prism for stereo
③ TV camera (Picturephone and video)
④ Audio Input (TV & AI are removable)
⑤ Stylus for pointing, writing and drawing
⑥ Pressure threshold (static) keyboard

"Pendery Paper Display Transducer" Design

Bill English took me under his wing and helped me start my group as I had always been a lone wolf and had no idea how to do it. One of his suggestions was that I should make a budget. I'm afraid that I really did ask Bill, "What's a budget?". I remembered at Utah, in pre-Mansfield Amendment days, Dave Evans saying to me as he went off on a trip to ARPA, "We're almost out of money. Got to go get some more." That seemed about right to me. They give you some money. You spend it to find out what to do next. You run out. They give you some more. And so on. PARC never quite made it to that idyllic standard, but for the first half decade it came close. I needed a group because I had finally realized that I did not have all of the temperaments required to completely finish an idea. I called it the Learning Research Group (LRC) 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—as Dan Ingalls was to say later: "...the rest has enfolded through the love and energy of the whole Learning Research Group". A lot of daytime was spent outside of PARC, playing tennis, bikeriding, drinking beer, eating chinese food, and constantly talking about the Dynabook and its potential to amplify human reach and bring new ways of thinking to a faltering civilization that desperately needed it (that kind of goal was common in California in the aftermath of the sixties).

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



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

As I mentioned previously, it was annoying that the surface beauty of LISP was marred by some of its key parts having to be introduced as "special forms" rather than as its supposed universal building block of functions. The actual beauty of LISP came more from the *promise* of its metastructures than its actual model. I spent a fair amount of time thinking about how objects could

Smalltalk-71 Programs

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

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

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

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

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

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

be characterized as universal computers without having to have any exceptions in the central metaphor. What seemed to be needed was complete control over what was passed in a message send; in particular *when* and in *what environment* did expressions get evaluated?

An elegant approach was suggested in a CMU thesis of Dave Fisher [Fisher 70] on the synthesis of control structures. ALGOL60 required a separate link for dynamic subroutine linking and for access to static global state. Fisher showed how a generalization of these links could be used to simulate a wide variety of control environments. One of the ways to solve the "funarg problem" of LISP is to associate the proper global state link with expressions and functions that are to be evaluated later so that the free variables referenced are the ones that were actually implied by the static form of the language. The notion of "lazy evaluation" is anticipated here as well.

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

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

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

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

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

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

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

When we turn to the various languages for specifying computations we find many to be general and a few to be parsimonious. For example, we can define universal machine languages in just a few instructions that can specify anything that can be computed. But most of these we would not call beautiful, in part because the amount and kind of code that has to be written to do anything interesting is so contrived and turgid. A simple and small system that can do interesting things also needs a "high slope"—that is a good match between the degree of interestingness and the level of complexity needed to express it.

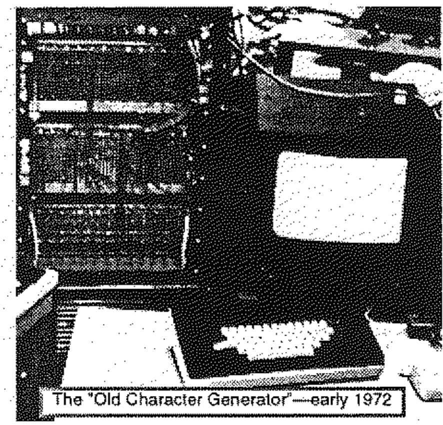
A fertilized egg that can transform itself into the myriad of specializations needed to make a complex organism has parsimony, generality, enlightenment, and finesse—in short, beauty, and a beauty

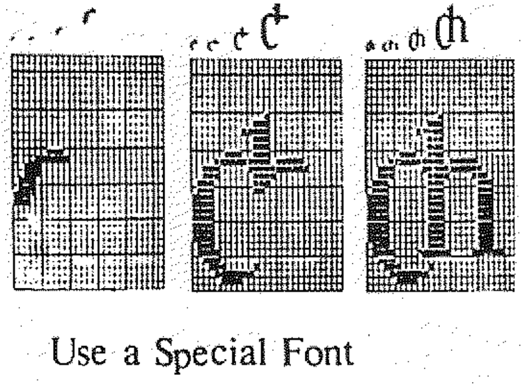
much more in line with my own esthetics. I mean by this that Nature is wonderful both at elegance and practicality—the cell membrane is partly there to allow useful evolutionary kludges to do their necessary work and still be able act as component by presenting a uniform interface to the world.

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

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

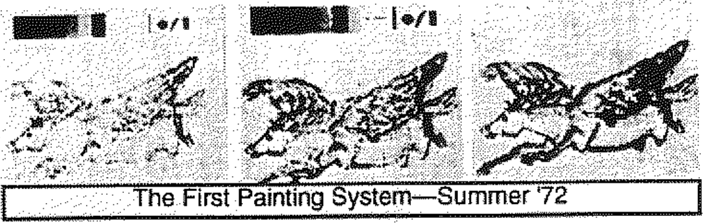
The character generator's font memory turned out to be large enough to simulate a bit-map display if one displayed a fixed "strike" and wrote into the font memory. Ben Laws built a beautiful font editor and he and I spent several months learning about the peculiarities of the human visual system (it is decidedly non-linear). I was very interested in high-quality text and graphical presentations because I thought it would be easier to get the Dynabook into schools as a "trojan horse" by simply replacing school books rather than to try to explain to teachers and school boards what was really great about personal computing.


The "Old Character Generator"—early 1972



Things were generally going well all over the lab until May of 72 when I tried to get resources to build a few miniCOMs. A relatively new executive ("X") did not want to give them to me. I wrote a memo explaining why the system was a good idea (see **Appendix II**), and then had a meeting to discuss it. "X" shot it down completely saying among other things that we had used too many green stamps getting Xerox to fund the time-shared MAXC and this use of resources for personal machines would confuse them. I was shocked. I crawled away back to the experimental character generator and made a plan to get 4 more made and hooked to NOVAs for the initial kid experiments.

I got Steve Purcell, a summer student from Stanford, to build my design for bit-map painting so the kids could sketch as well as display computer graphics. John Shoch built a line drawing and gesture recognition system (based on Ledeen's [Newman and Sproull 72]) that was integrated with the painting. Bill Duvall of POLOS built a miniNLS that was quite remarkable in its speed and power. The first overlapping windows started to appear. Bob Shur (with Steve Purcell's help) built a 21/2 D animation system. Along with Ben Laws' font editor, we could give quite a smashing demo of what we intended to build for real over the next few years. I remember giving one of these to a Xerox execu-



The First Painting System—Summer '72

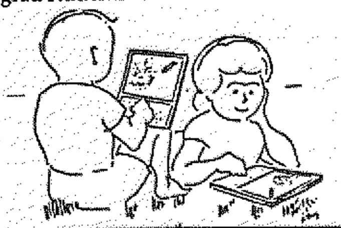
Portrait of the Xerox "RISK" executive

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 me straight in the eye and said, "Boy, that's great, but just make sure it works". This was a typical executive notion about risk. He wanted us to be in the "20%" one hundred percent of the time.

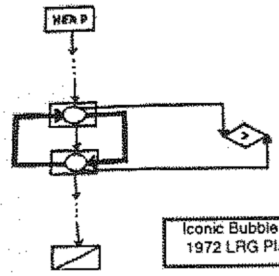
That summer while licking my wounds and getting the demo simulations built and going, Butler Lampson, Peter Deutsch and I worked out a general scheme for emulated HLL machine languages. I liked the B5000 scheme, but Butler did not want to have to decode bytes, and pointed out that since an 8-bit byte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

I also took another pass at the language for the kids. Jeff Rulifson was a big fan of Piaget (and semiotics) and we had many discussions about the "stages" and what iconic thinking might be about. After reading Piaget and especially Jerome Bruner, I was worried that the directly symbolic approach taken by FLEX, LOGO (and the current Smalltalk) would be difficult for the kids to process since evidence existed that the symbolic stage (or mentality) was just starting to switch on. In fact, all of the educators that I admired (including Montessori, Holt, and Suzuki) all seemed to call for a more figurative, more iconic approach. Rudolph Arnheim [Arnheim 69] had written a classic book about visual thinking, and so had the eminent art critic Gombrich [Gombrich **]. It really seemed that something better needed to be done here. GRAIL wasn't it, because its use of imagery was to portray and edit flowcharts, which seemed like a great step backwards. But Rovner's AMBIT-G held considerably more promise [Rovner 68]. It was kind of a visual SNOBOL [Farber 63] and the pattern matching ideas looked like they would work for the more PLANNERlike scheme I was using.

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



Children with Dynabooks from "A Personal Computer For Children Of All Ages" [Ka 72]



Iconic Bubble Sort from 1972 LRG Plan [Ka 72b]

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

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

In Sept, within a few weeks of each other, two bets happened that changed most of my plans. First, Butler and Chuck came over and asked: "Do you have any money?" I said, "Yes, about $230K for NOVAs and CCs. Why?" They said, "How would you like us to build your little machine for you?" I said, "I'd like it fine. What is it?" Butler said: "I want a '$500 PDP-10', Chuck wants a '10 times faster NOVA', and you want a 'kiddicomp'. What do you need on it?" I told them most of the results we had gotten from the fonts, painting, resolution, animation, and music studies. I asked where this had come from all of a sudden and Butler told me that they wanted to do it anyway, that Executive "X" was away for a few months on a "task force" so maybe they could "Sneak it in", and that Chuck had a bet with Bill Vitic that he could do a whole machine in just 3 months. "Oh", I said.

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

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

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

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

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

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

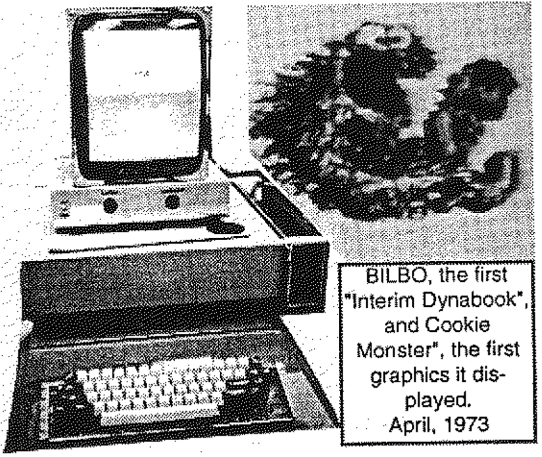
It evaluated 3+4 very slowly (it was "glacial", as Butler liked to say) but the answer always came out 7. Well, there was nothing to do but keep going. Dan loved to bootstrap on a system that "always ran", and over the next ten years he made at least 80 major releases of various flavors of Smalltalk.

In November, I presented these ideas and a demonstration of the interpretation scheme to the MIT AI lab. This eventually led to Carl Hewitt's more formal "Actor" approach[Hewitt 73]. In the first Actor paper the resemblence to Smalltalk is at its closest. The paths later diverged, partly because we were much more interested in making things than theorizing, and partly because we had something no one else had: Chuck Thacker's Interim Dynabook (later known as the "ALTO").

Just before Chuck started work on the machine I gave a paper to the National Council of Teachers of English [Kay 72c] on the Dynabook and its potential as a learning and thinking amplifier—the paper was an extensive rotogravure of "20 things to do with a Dynabook" [Kay 72c]. By the time I got back from Minnesota, Stewart Brand's *Rolling Stone* article about PARC [Brand,1972] and the surrounding hacker community had hit the stands. To our enormous surprise it caused a major furor at Xerox headquarters in Stamford, Connecticut. Though it was a wonderful article that really caught the spirit of the whole culture, Xerox went berserk, forced us to wear badges (over the years many were printed on t-shirts), and severely restricted the kinds of publications that could be made. This was particularly disastrous for LRC, since we were the "lunatic fringe" (so-called by the other computer scientists), were planning to go out to the schools, and needed to share our ideas (and programs) with our colleagues such as Seymour Papert and Don Norman.

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

Chuck had started his "bet" on November 22, 1972. He and two technicians did all of the machine except for the disk interface which was done by Ed McCreight. It had a ~500,000 pixel (606x808) bitmap display, its microcode instruction rate was about 6MIPs, it had a grand total of 128k, and the entire machine (exclusive of the memory) was rendered in 160 MSI chips distributed on two cards. It was beautiful [Thacker,1972, 1986]. One of the wonderful features of the machine was "zero-overhead" tasking. It had 16 program counters, one for each task. Condition flags were tied to interesting events (such as "horizontal retrace pulse", and "disk sector pulse", etc.). Lookaside logic scanned the flags while the current instruction was executing and picked the highest priority program counter to fetch from next. The machine never had to wait, and the result was that most hardware functions (particularly those that involved i/o (like feeding the display and handling the disk) could be replaced by microcode. Even the refresh of the MOS dynamic RAM was done by a task. In other words, this was a coroutine architecture. Chuck claimed that he got the idea from a lecture I had given on corountines a few months before, but I remembered that Wes Clark's TX-2 (the Sketchpad machine) had used the idea first, and I probably mentioned that in the talk.



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

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

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

1. Everything is an *object*
2. Objects communicate by sending and receiving *messages* (in terms of objects)
3. Objects have their *own memory* (in terms of objects)
4. Every object is an *instance* of a *class* (which must be an object)
5. The class holds the shared *behavior* for its instances (in the form of objects in a program list)
6. To eval a program list, control is passed to the first object and the remainder is treated as its message

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

<u>receiver</u>	<u>| message</u>
c	| ∘ *i* <- *d*e*

The *c* is bound to the receiving object, and <u>all</u> of∘ *i* <- *d*e* is the message to it. The message is made up of a literal token "∘", an expression to be evaluated in the sender's context (in this case *i*), another literal token <-, followed by an expression to be evaluated in the sender's context (*d*e*). Since "LISP" pairs are made from 2 element objects they can be indexed more simply: *c hd*, *c tl*, and *c hd <- foo*, etc.

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

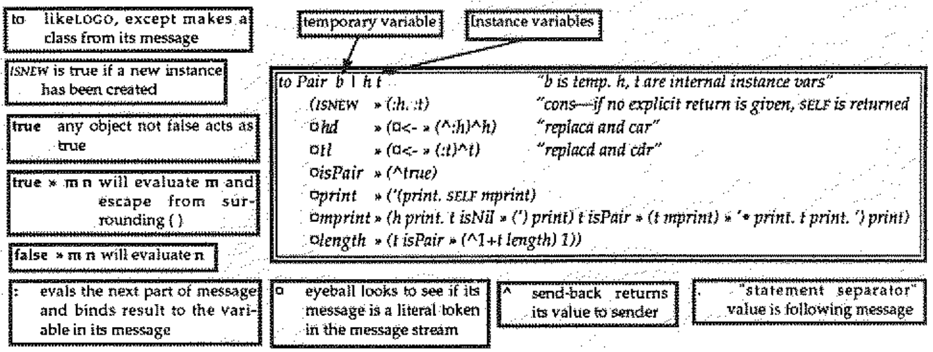
<u>receiver</u>	<u>|message</u>
a	| + *b*
3	| + *4*

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

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

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

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



```
to Pair  b | h t                                "b is temp. h, t are internal instance vars"
    (ISNEW   » (:h. :t)                         "cons—if no explicit return is given, SELF is returned
    ∘hd      » (∘<- » (^:h)^h)                  "replaca and car"
    ∘tl      » (∘<- » (:t)^t)                   "replacd and cdr"
    ∘isPair  » (^true)
    ∘print   » ('(print. SELF mprint)
    ∘mprint » (h print. t isNil » (') print) t isPair » (t mprint) » '• print. t print. ') print)
    ∘length  » (t isPair » (^1+t length) 1))
```

to likeLOGO, except makes a class from its message
ISNEW is true if a new instance has been created
true any object not false acts as true
true » m n will evaluate m and escape from surrounding ( )
false » m n will evaluate n
temporary variable
Instance variables
: evals the next part of message and binds result to the variable in its message
∘ eyeball looks to see if its message is a literal token in the message stream
^ send-back returns its value to sender
. "statement separator" value is following message

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

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

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

or "intensionally", as part of class integer:

(... ∘! » (^:n=0 » (1) (n-1)!)

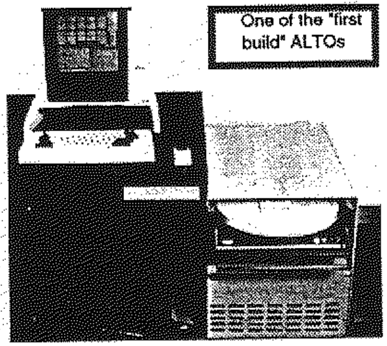
Of course, the whole idea of Smalltalk (and OOP in general) is to define everything *intensionally*. And this was the direction of movement as we learned how to program in the new style. I never liked this syntax (too many parentheses and nestings) and wanted something flatter and more grammar-like as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

Proposed Smalltalk-72 Syntax

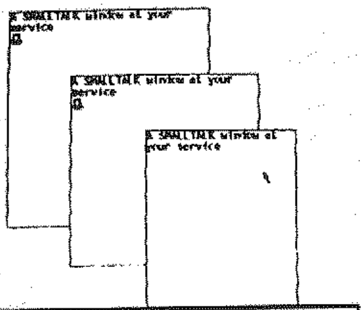
```
Pair :h :t
    hd <- :h
    hd          »   h
    tl <- :t
    tl          »   t
    isPair      »   true
    print       »   '( print. SELF mprint.
    mprint      »   h print. if t isNil then ') print
                         else if t isPair then t mprint
                         else '• print.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

Development Of The Smalltalk-72 System And Applications

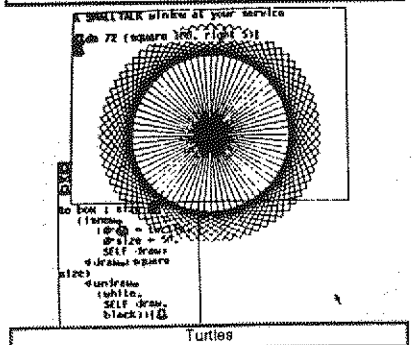
The advent of a real Smalltalk on a real machine started off an explosion of parallel paths that are too difficult to intertwine in strict historical order. Let me first present the general development of the Smalltalk-72 system up to the transistion to Smalltalk-76, and then follow that with the several years of work with children 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.


One of the "first build" ALTOs

Overlapping windows were the first project tackled (with Diana Merry) after writing the code to read the keyboard and create a string of text. Diana built an early version of a bit field block transfer (bitblt) for displaying variable pitch fonts and generally writing on the display. The first window versions were done as real 2½D draggable objects that were just a little too slow to be useful. We decided to wait until Steve Purcell got his animation system going to do it right, and opted for the style that is still in use today, which is more like "2¼D". Windows were perhaps the most redesigned and reimplemented class in Smalltalk because we didn't quite have enough compute power to just do the continual viewing to "world coordinates" and refreshing that my former Utah colleagues were starting to experiment with on the flight simulator projects at Evans & Sutherland. This is a simple, powerful model but it is difficult to do in real-time even in 2½D. The first practical windows in Smalltalk used the GRAIL conventions of sensitive corners for moving, resizing, cloning, and closing. Window scheduling used a simple "loopless" control scheme that threaded all of the windows together.


A SMALLTALK window at your service
A SMALLTALK window at your service
A SMALLTALK window at your service
Early Smalltalk Windows on Interim Dynabook

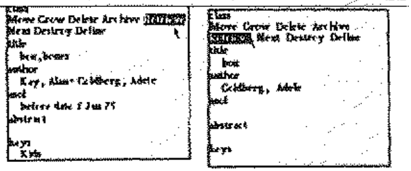
One of the next classes to be implemented on the Interim Dynabook (after the basics of numbers, strings, etc.) was an object-oriented version of the LOGO turtle implemented by Ted. This could make many turtle instances that were used both for drawing and as a kind of value for graphics transformations. 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.



Turtles

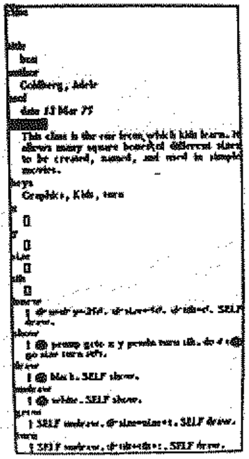
John Shoch built a mouse-driven structured editor for Smalltalk code.

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



Findit Retrieval By Example

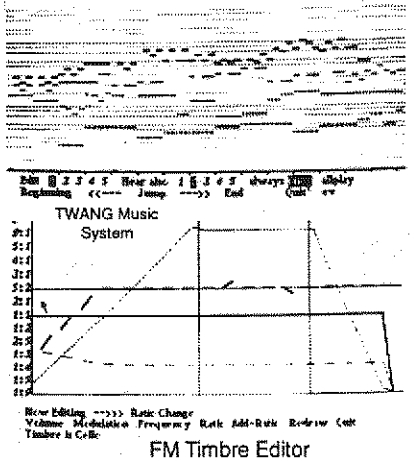
One of the "small program" projects I tried on an adult class in the Spring of '74 was a one-page paragraph editor. It turned out to be too complicated, but the example I did to show them was completely modeless (it was in the air) and became the basis for much of the Smalltalk text work over the next 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 Flegal, Diana Merry, Larry Tesler,Tim Mott, and Trygve Reenskaug.



Retrieved HyperDocument

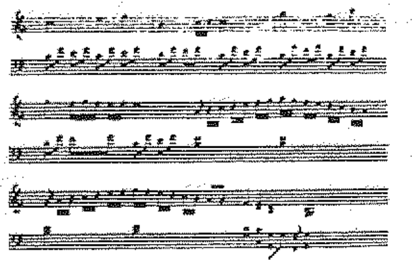
Steve Weyer and I devised *Findit*, a "retrieval by example" interface that used the analogy of classes to their instances to form retrieval requests. This was used for many years by the PARC library to control circulation.

The sampling synthesis music I had developed on the NOVA could generate 3 high-quality real-time voices. Bob Shur and Chuck Thacker transfered the scheme to the Interim Dynabook and achieved 12 voices in real-time. The 256 bit generalized input that we had specified for low speed devices (used for the mouse and keyboard) made it easy to connect 154 more to wire up two organ keyboards and a pedal. Effects such as portamento and decay were programmed. Ted Kaehler wrote TWANG, a music capture and editing system, using a tabulature notation that we devised to make music clear to children [Kay,1977a]. One of the things that was hard to do with sampling was the voltage controlled operator (VCO) effects that were popular on the "Well Tempered Synthesizer". A summer later, Steve Saunders, another of our bright summer students, was challenged to find a way to accomplish John Chowning's <u>very</u> non-real-time FM synthesis in real-time on the ID. He had to find a completely different way to think of it than "FM", and succeeded brilliantly with 8 real-time voices that were integrated into TWANG [Saunders *].



FM Timbre Editor

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


OPUS Score Capture

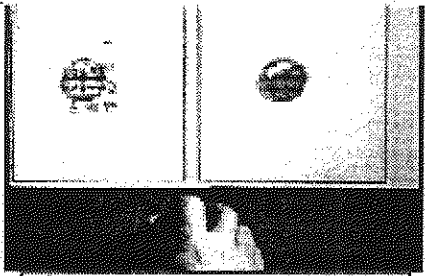
today it did not require metronomic playing but instead took a first pass looking for strong and weak beats (the phrasing) to establish a local model of the likely tempo fluctuations and then used curve fitting and extrapolation to make judgements about just where in the measure, and for what time value, a given note had been struck.

The animations on the NOVA ran 3-5 objects at about 2-3 frames per second. Fast enough for the *phi* phenomenon to work (if double buffering was used), but we wanted "Disney rates" of 10-15 frames a second for 10 or more large objects and many more smaller ones. This task was put into the ingenious hands of Steve Purcell. By the Fall of '73 he could demo 80 ping-pong balls and 10 flying horses running at 10 frames per second in 21/2 D. his next task was to make the demo into a general systems facility from which we could construct animation systems. His CHAOS system started working in May '74, just in time for summer visitors Ron Baecker, Tom Horseley, and professional animator Eric Martin to visit and build SHAZAM a marvelously capable and simple animation system based on Ron's GENESYS thesis project on the TX-2 in the late sixties [Baecker 69].

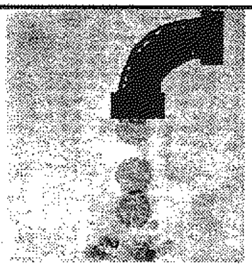
The main thesis project during this time was Dave Smith's PYGMALION [Smith 75], an essay into iconic programming (no, we hadn't quite forgotton). One programmed by showing the system how changes should be made, much as one would illustrate on a blackboard with another programmer. This program became the starting place from which many subsequent programming by example" systems took off.

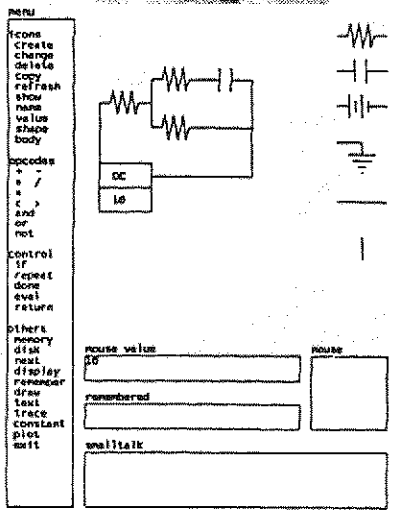
I should say something about the size of these programs. PYGMALION was the largest program ever written in Smalltalk-72. It was about 20 pages of code—all that would fit in the interim dynabook ALTO—and is given in full in Smith's thesis. All of the other applications were smaller. For example, the SHAZAM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

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

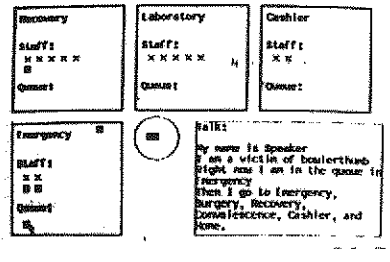


Shazam iconic user interface (above)
A sample animation (below)





PYGMALION Iconic Programming



My name is Speaker
I am a victim of boulerthumb
Right now I am in the queue in
Emergency
Then I go to Emergency,
Surgery, Recovery,
Convalescence, Cashier, and
Home.

"Simpula" Hospital Simulation

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

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

The Evolution Of Smalltalk-72

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

The crowning addition was the OOZE (Object Oriented Zoned Environment) virtual memory system that served Smalltalk-74, and more importantly, Smalltalk-76 [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 2.4 megabyte model 30 disk drive was faster and larger than a floppy and slower and smaller than today's hard drives. It was quite similar to the HP direct contact disk of the FLEX machine on which I had tried a fine-grain version of the B5000 segment swapper. It had not worked as well as I wanted, despite a few good ideas as to how to choose objects when purging. When the gang wanted to adapt this basic scheme, I said: "But I never got it to work well." I remember Ted Kaehler saying, "Don't worry, we'll make it work!"

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

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

Ted and Dan decided to control storage using a Resident Object Table that was the only place machine addresses for objects would be found. Other useful information was stashed there as well to help LRU aging. Purging was done in background by picking a class, positioning the disk to its instances (all of a particular class were stored together), then running through the ROT to find the dirty ones in storage and stream them out. This was pretty efficient and, true to Butler's insight, furnished a good sized pool of clean storage that could be overwritten. The key to the design though (and the implementation of the transaction mechanism) was the checkpointing scheme 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 80kb of working storage and could handle about 65K objects (up to several megabytes worth, more than enough for the entire system, its interface, and its applications).

"Object-oriented" Style

This is probably a good place to comment on the difference between what we thought of as OOP-style and the superficial encapsulation called "abstract data types" that was just starting to be investigated in academic circles . Our early "LISP-pair" definition is an example of an abstract data type because it preserves the "field access" and "field rebinding" that is the hallmark of a data structure. Considerable work in the 60s was concerned with generalizing such structures[DSP *]. The "official" computer science world started to regard Simula as a possible vehicle for defining *abstract data types* (even by one of its inventors[Dahl 1970]), and it formed much of the later backbone of ADA. This led to the ubiquitous stack data-type example in hundreds of papers. To put it mildly, we were quite amazed at this, since to us, what Simula had whispered was something much stronger than simply reimplementing a weak and ad hoc idea. What I got from Simula was that you could now replace bindings and assignment with *goals*. The last thing you wanted any programmer to do is mess with internal state even if presented figuratively. Instead, the objects should be presented as *sites of higher level behaviors more appropriate for use as dynamic components.*

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

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

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

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

Smalltalk And Children

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

Adele holding forth at Jordan Middle Sch.

Since we had no idea how to teach object-oriented programming to children (or anyone else), the first experiments Adele did mimicked LOGO turtle graphics, and she got what appeared to be very similar results. That is to say, the children could get the turtle to draw pictures on the screen, but there seemed to be little happening beyond surface effects. At that time I felt that since the content of personal computing was interactive tools, that the content of this new kind of authoring literacy should be the creation of interactive *tools* by the children. Procedural turtle graphics just wasn't it.

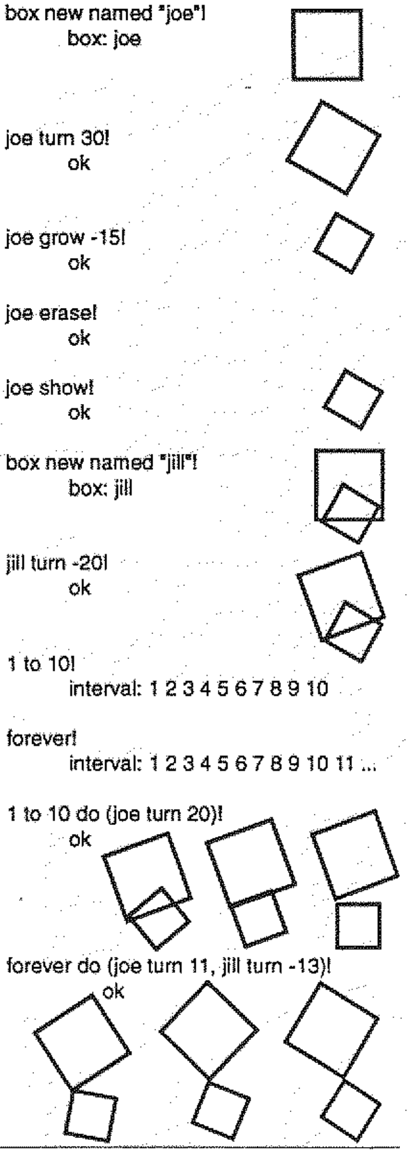
Then Adele came up with a brilliant approach to teaching Smalltalk as an object-oriented language: the "Joe Book". I believe this was partly influenced by Minsky's idea that you should teach a programming language holistically from working examples of serious programs.

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

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

What was so wonderful about this idea were the myriad of children's projects that could spring off the humble boxes. And some of the earliest were tools! This was when we got really excited. For example, Marion Goldeen's (12 yrs old) painting system was a full-fledged tool. A few years later, so was Susan Hamet's (12 yrs old) OOP illustration system (with a design that was like the MacDraw to come). Two more were Bruce Horn's (15 yrs old) music score capture system and Steve Putz's (15 yrs old) circuit design system. Looking back, this could be called another example in computer science of the "early success syndrome". The successes were real, but they weren't as general as we thought. They wouldn't extend into the future as strongly as we hoped. The children were chosen from the Palo Alto schools (hardly an average background) and we tended to be much more excited about the successes than the difficulties. In part, what we were seeing was the "hacker phenomenon", that, for any given pursuit, a particular 5% of the population will jump into it naturally, while the 80% or so who can learn it in time do not find it at all natural.

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



```
box new named "joe"!
        box: joe

joe turn 30!
        ok

joe grow -15!
        ok

joe erase!
        ok

joe show!
        ok

box new named "jill"!
        box: jill

jill turn -20!
        ok

1 to 10!
        interval: 1 2 3 4 5 6 7 8 9 10

forever!
        interval: 1 2 3 4 5 6 7 8 9 10 11 ...

1 to 10 do (joe turn 20)!
        ok

forever do (joe turn 11, jill turn -13)!
        ok
```

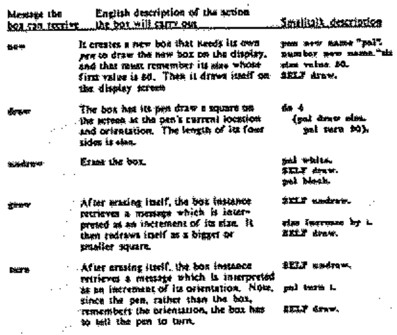
seemed more to be that of *design*.

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

The connection to literacy was painfully clear. It isn't enough to just learn to read and write. There is also a *literature* that renders *ideas*. Language is used to read and write about them, but at some point the organization of ideas starts to dominate mere language abilities. And it helps greatly to have some powerful ideas under one's belt to better acquire more powerful ideas [Papert 70s]. So, we decided we should teach *design*. And Adele came up with another brilliant stroke to deal with this. She decided that what was needed was an intermediary between the vague ideas about the problem and the very detailed writing and debugging that had to be done to get it to run in Smalltalk. She called the intermediary forms *design templates*.

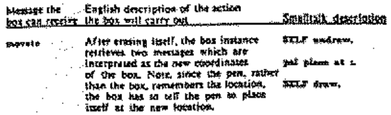
Using these the children could look at a situation they wanted to simulate, and decompose it into classes and messages without having to worry just how a method would work. The method planning could then be done informally in English, and these notes would later serve as commentaries and guides to the writing of the actual code. This was a terrific idea, and it worked very well.

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

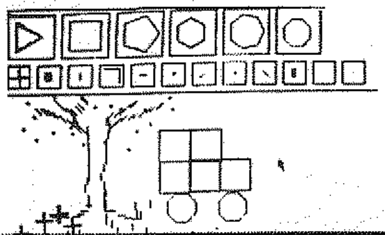


Message the box can receive	English description of the action the box will carry out	Smalltalk description
new	It creates a new box that needs its own pen to draw the new box on the display, and that must remember its size whose first value is 50. Then it draws itself on the display screen	pen new name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

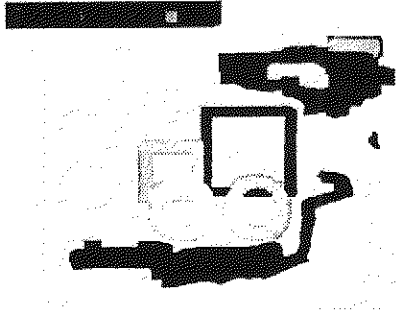
Adele's planning template for Smalltalk (above)
New behavior added by child (below)



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



Marion Goldeen's painting program (above)
Susan Hamet's OO Illustrator (below)



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

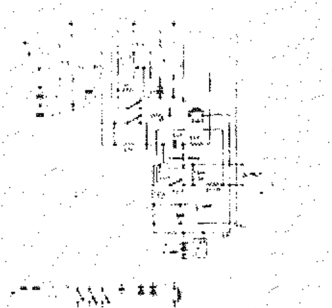
At this point, let me do a look back from the vantage point of today. I'm now pretty much convinced that our design template approach was a good one after all. We just didn't apply it longitudinally enough. I mean by this that there is now a large accumulation of results from many attempts to teach novices programming [Soloway, 1989]. They all have similar stories that seem to have little to do with the various features of the programming languages used, and everything to do with the difficulties novices have thinking the special way that good programmers think. Even with a much better interface than we had then (and have today), it is likely that this area is actually more like writing than we wanted it to be. Namely, for the "80%", it really has to be learned gradually over a period of years in order to build up the structures that need to be there for design and solution look-ahead. [41]

The problem is not to get the kids to do stuff—they love to do, even when they are not sure exactly what they are doing. This correlates well with studies of early learning of language, when much rehearsal is done regardless of whether content is involved. Just *doing* seems to help. What is difficult is to determine *what* ideas to put forth and how *deeply* they should penetrate at a given child's developmental level. This is a confusion still persists for reading and writing of natural language—and for mathematics—despite centuries of experience. And it is the main hurdle for teaching children programming. When, in what order and depth, and how should the powerful ideas be taught?

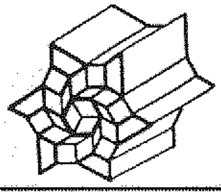
Should we even try to teach programming? I have met hundreds of programmers in the last 30 years and can see no discernable influence of programming on their general ability to think well or to take an enlightened stance on human knowledge. If anything, the opposite is true. Expert knowledge often remains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

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

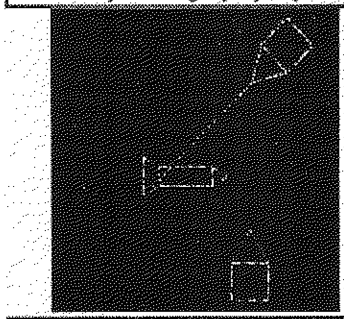
Tools provide a path, a context, and almost an excuse for developing enlightenment, but no tool ever contained it or can dispense it. Cesare Pavese observed: to know the



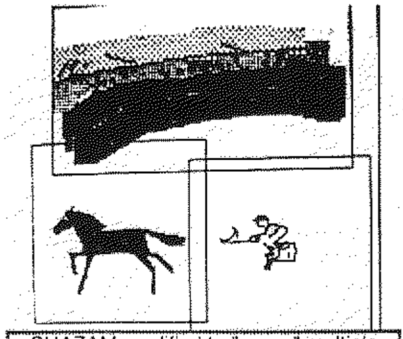
Circuit design system by Steve Putz (age 15)



Tangram designs are created by selecting shapes from a "menu" displayed at the top of the screen. This system was implemented in Smalltalk by a fourteen-year old girl [Kay 77]



SpaceWar by Dennis (age 12))



SHAZAM modified to "group" multiple images by Lisa van Stone (age 12) world we must construct it. In other words, *we make not just to have, but to know.* But the having can happen without most of the knowing taking place.

Another way to look at this is that knowledge is in its least interesting state when it is first being learned. The representations—whether markings, allusions, or physical controls—get in the way (almost take over as goals) and must be laboriously and painfully interpreted. From here there are several useful paths, two of which are important and intertwined.

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

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

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

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

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

We did not know then, and I'm sorry to say from 15 years later, that these critical questions still do not yet have really useful answers. But there are some indications. Even very young children can understand and use interactive *transformational* tools. The first ones are their hands! They can readily extend these experiences to computer objects and making changes to them. They can often imagine what a proposed change will do and not be surprised at the result. Two and three year olds can use the Smalltalk-style interface and manipulate object-oriented graphics. 3rd graders can (in a few days) learn more than 50 features—most of these are transformational tools—of a new system including its user interface. They can answer any question whose answer requires the application of just one of these tools. But it is extremely difficult for them to answer any question that requires two or more transformations. Yet they have no problem applying sequences of transformations, exploring "forward". It is for conceiving and achieving even modest goals requiring several changes that they almost completely lack navigation abilities.

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

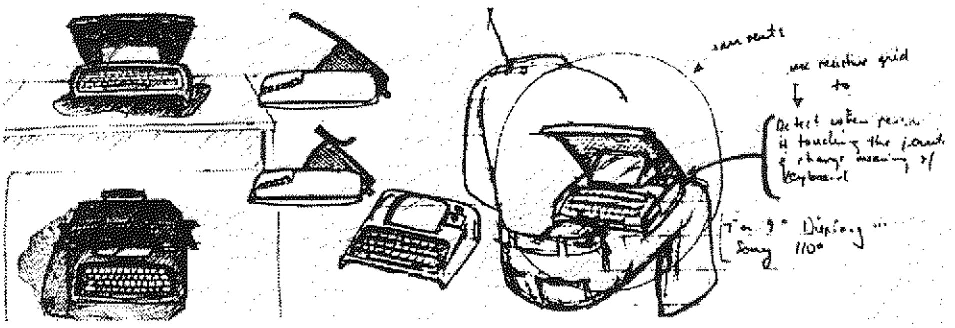
V. 1976-80—The first modern Smalltalk (-76), its birth, applications, and improvements

By the end of 1975 I felt that we were losing our balance—that the "Dynabook for children" idea was slowly dimming out—or perhaps starting to be overwhelmed by professional needs. In January 1976, I took the whole group to Pajaro Dunes for a three day offsite to bring up the issues and try to reset the compass. It was called "Let's Burn Our Disk Packs". There were no shouting matches, the group liked (I would go so far to say: *loved*) each other too much for that. But we were troubled. I used the old aphorism that "no biological organism can live in its own waste products" to plead for a

really fresh start: a hw-sw system very different from the ALTO and Smalltalk. One thing we all did agree on was that the current Smalltalk's power did not match our various levels of aspiration. I thought we needed something different, as I did not see how OOP by itself was going to solve our end-user problems. Others, particularly some of the grad students, really wanted a better Smalltalk that was faster and could be used for bigger problems. I think Dan felt that a better Smalltalk could be the vehicle for the different system I wanted, but could not describe clearly. The meeting was not a disaster, and we went back to PARC still friends and colleagues, but the absolute cohesiveness of the first four years never rejelled. I started designing a new small machine and language I called the *NoteTaker* and Dan started to design Smalltalk-76.

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

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


Detect when person is touching the pad & change meaning of keyboard
Try 9" Display
Sony 110°

Meanwhile Dan was proceeding with his total revamp of Smalltalk and along somewhat similar lines [In 78]. The first major thing that needed to be done was to get rid of the function/class dualism in favor of a completely intensional definition with every piece of code as an intrinsic method. We had wanted that from the beginning, (and most of the code was already written that way). There were a variety of strong desires for a real inheritance mechanism from Adele and me, from Larry Tesler, who was working on desktop publishing, and from the grad students. Dan had to find a better way than Simula's very rigid compile-time conception. It was time to make good on the idea that "everything was an object", which included all of the internal "systems" objects like "activation records", etc. We were all agreed that the flexible syntax of the earlier Smalltalks was too flexible, and this level of extensibility was not desirable. All of the extensions we liked used various keyword schemes, so Dan came up with a combination keyword/operator syntax that was very flexible, but allowed the language to be read unambiguously by both humans and the machine. This allowed a FLEX machine-like byte-code compiler and efficient interpreter to be defined that ran up to 180 times as fast as the previous direct interpreter. The OOZE VM system could be modified to handle the new objects and its capacity was well matched to the ALTO's RAM and disk.

Inheritance

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

On the other hand, since things can be done with a dynamic language that are difficult with a statically compiled one, I just decided to leave inheritance out as a feature in Smalltalk-72, knowing that we could simulate it back using Smalltalk's LISPlike flexibility. The biggest contributer to these AI ideas was Larry Tesler who used what is now called "slot inheritance" extensively in his various versions of early desktop publishing systems. Nowadays, this would be called a "delegation-style" inheritance scheme [Lieberman 84]. Danny Bobrow and Terry Winograd during this period were designing a "frame-based" AI language called KRL which was "object-oriented" and I believe was influenced by early Smalltalk. It had a kind of multiple inheritance—called *perspectives*—which permitted an object to play multiple roles in a very clean way. Many of these ideas a few years later went into PIE, an interesting extension of Smalltalk to networks and higher level descriptions by Ira Goldstein and Bobrow [Goldstein & Bobrow 1980].

84
message
multiplmer – role
objects

By the time Smalltalk-76 came along, Dan Ingalls had come up with a scheme that was Simula-like in its semantics but could be incrementally changed on the fly to be in accord with our goals of close interaction. I was not completely thrilled with it because it seemed that we needed a better theory about inheritance entirely (and still do). For example, inheritance and instancing (which is a kind of inheritance) muddles both pragmatics (such as factoring code to save space) and semantics (used for way too many tasks such as: specialization, generalization, speciation, etc.) Alan Borning employed a multiple inheritance scheme in Thinglab [Borning 77] which was implemented in Smalltalk-76. But no comprehensive and clean multiple inheritance scheme appeared that was compelling enough to surmount Dan's original Simula-like design.

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

> **A Simple Vision of the Future**
>
> *A Brief Update Of My 1971 Pendery Paper*
>
> In the 1990's there will be millions of personal computers. They will be the size of notebooks of today, have high-resolution flat-screen reflective displays, weigh less than ten pounds, have ten to twenty times the computing and storage capacity of an *Alto*. Let's call them *Dynabooks*.
>
> The purchase price will be about that of a color television set of the era, although most of the machines will be given away by manufacturers who will be marketing the content rather than the container of personal computing.
>
> ...
>
> Though the *Dynabook* will have considerable local storage and will do most computing locally, it will spend a large percentage of its time hooked to various large, global information utilities which will permit communication with others of ideas, data, working models, as well as the daily chit-chat that organizations need in order to function. The communications link will be by private and public wires and by packet radio. Dynabooks will also be used as servers in the information utilities. They will have enough power to be entirely shaped by software.

> **The Main Points Of This Vision**
>
> - There need only be a few hardware types to handle almost all of the processing activity of a system.
> - Personal Computers, Communications Links, and Information Utilities are the three critical components of a Xerox future.
>
> ...
>
> In other words, the *material* of a computer system is the computer itself, all of the *content* and *function* is fashioned in software.
>
> There are two important guidelines to be drawn from this:
>
> » Material: If the design and development of the hardware computer material is done as carefully and completely as Xerox's development of special light-sensitive alloys, then only one or two computer designs need to be built... Extra investment in development here will be vastly repaid by simplifying the manufacturing process and providing lower costs through increased volume.
>
> » Content: Aside from the wonderful generality of being able to continously shape new content from the same material, *software* has three important characteristics:
>
> - the *replication* time and cost of a content-function is zero
> - the *development* time and cost fo a content-function is high
> - the *change* time and cost of a content-function can be low
>
> Xerox must take these several points seriously if it is to survive and prosper in its new business are of information media. If it does, the company has an excellent chance for several reasons:
>
> - Xerox has the financial base to cover the large development costs of a small number of very powerful computer-types and a large number of software functions.
> - Xerox has the marketing base to sell these functions on a wide enough scale to garner back to itself an incredible profit.
> - Xerox has working for it an impressively large percentage of the best software designers in the world.

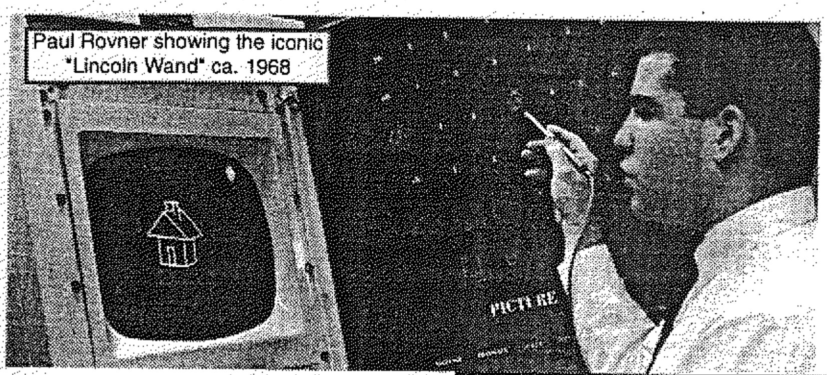
In 1976, Chuck Thacker designed the ALTO III that would use the new 16k chips and be able to fit on a desktop. It could be marketed for about what the large cumbersome special purpose "word-processors" cost, yet could do so much more. Nevertheless, in August of 1976, Xerox made a fateful decision: not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

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 [Kay ~~89~~]—let me add a few more points. 96

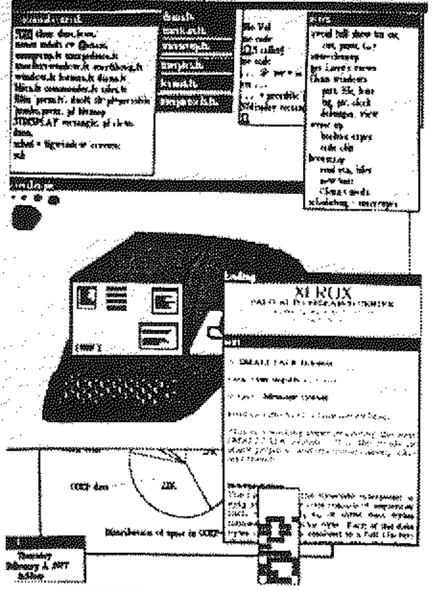
All of the elements eventually used in the Smalltalk user interface were already to be found in the sixties—as different ways to access and invoke the functionality provided by an interactive system. The two major centers of ideas were Lincoln Labs and RAND corp—both ARPA funded. The big shift that consolidated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

The particular aim of LRG was to find the equivalent of writing—that is learning and thinking by doing in a medium—our new "pocket universe". For various reasons I had settled on "iconic programming" as the way to achieve this, drawing on the iconic representations used by many ARPA projects in the sixties. My friend Nicholas Negroponte, an architect, was extremely interested in how environments affected peoples' work and creativity. He was interested in embedding the new computer magic in familar surroundings. I had quite a bit of theatrical experience in a past life, and remembered Coleridge's adage that "people attend 'bad theatre' hoping to forget, people attend

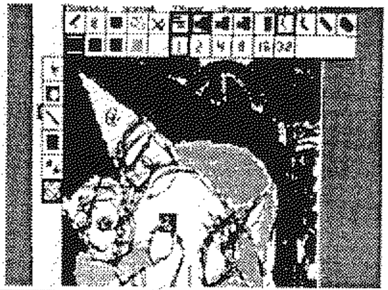

Paul Rovner showing the iconic "Lincoln Wand" ca. 1968

'good theatre' *aching to remember*". In other words, it is the ability to evoke the audience's own intelligence and experiences that makes theatre work.

Putting all this together, we want an apparently free environment in which exploration causes desired sequences to happen (Montessori); one that allows kinesthetic, iconic, and symbolic learning—"*doing* with *images* makes *symbols*" (Piaget & Bruner); the user is never trapped in a mode (GRAIL); the magic is embedded in the familiar (Negroponte); and which acts as a magnifying mirror for the user's own intelligence (Coleridge). It would be a great finish to this story to say that having articulated this we were able to move straightforwardly to the design as we know it today. In fact, the UI design work happened in fits and starts in between feeding Smalltalk itself, designing children's experiments, trying to understand iconic construction, and just playing around. In spite of this meandering, the context almost forced a good design to turn out anyway. Just about everyone at PARC at this time had opinions about the UI, ours and theirs. It is impossible to give detailed credit for the hundreds of ideas and discussions. However, the consolidation can certainly be attributed to Dan Ingalls, for listening to everyone, contributing original ideas, and constantly building a design for user testing. I had a fair amount to do with setting the context, inventing overlapping windows, etc., and Adele and I designed most of the experiments. Beyond that, Ted Kaehler, and visitor Ron Baecker made highly valuable contributions. Dave Smith designed SmallStar, the prototype iconic interface for the Xerox Star product [Smith 83].


The last Smalltalk-72 Interface

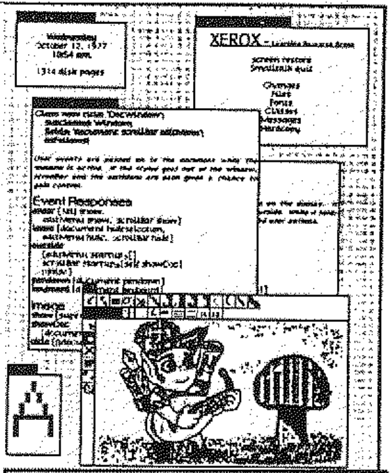
Meanwhile, I had gotton Doug Fairbairn interested in the *Notetaker*. He designed a wonderful "smart bus" that could efficiently handle slow multiple processors and the system looked very promising, even though most of the rest of PARC thought I was nuts to abandon the fast bipolar hw of the ALTO. But I 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.


Ted Kaehler's iconic painting interface

TWINKLE ✶✶

Smalltalk-76

Dan finished the Smalltalk-76 design in November, and he, Dave Robson, Ted Kaehler, and Diana Merry, successfully implemented the system from scratch (which included rewriting all of the existing class definitions) in just seven months. This was such a wonderful achievement that I was bowled over in spite of my wanting to start over. It was fast, lively, could handle "big" problems, and was great fun. The system consisted of about 50 classes described in about 180 pages of source code. This included all of the OS functions, files, printing and other Ethernet services, the window interface, editors, graphics and painting systems, and two new contributions by Larry Tesler, the famous browsers for static methods in the inheritance hierarchy and dynamic contexts for debugging in the runtime environment. In every way it was the consolidation of all of our ideas and yearnings about Smalltalk in one integrated package. All Smalltalks since have resembled this conception very closely. In many ways, as Tony Hoare once remarked about Algol, Dan's Smalltalk-76 was a great improvement on its successors!


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

Here are two stylish ST-76 classes written by Dan.

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

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

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

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

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

^ means "send back"

=> means "then"

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

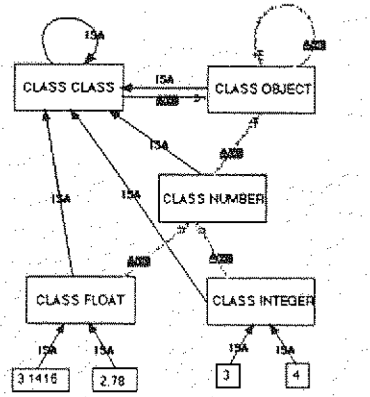
User events are passed on to the document while the window is active. If the stylus goes out of the window, scrollbar and the editMenu are each given a chance to gain control.

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

super means delegate message to next higher superclass

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

either as delegations
w/ Framework defaults



Smalltalk-76 Metaphysics

flash. In this case, it finds DocWindow method: **keyboard**, which tells the held document to check it out.

In January of 1978 Smalltalk-76 had its first real test. CSL had invited the top ten executives of Xerox to PARC for a two day seminar on software, with a special emphasis on complexity and what could be done about it. LRG got asked to give them a hands-on experience in end-user programming so "they could do 'something real' over two 11/2 hour sessions". We immediately decided not to teach them Smalltalk-76 (my "burn our disk packs" point in spades), but to create in two months in Smalltalk-76 a rich system especially tailored for adult nonexpert users (Dan's point in trumps). We took our "Simpula" job shop simulation model as a starting point and decided to build a user interface for a generalized job shop simulation tool that the executives could make into specific dynamic simulations that would act out their changing states by animating graphics on the screen. We called it the Smalltalk SimKit. This was a maximum effort and everyone pitched in. Adele became the design leader in spite of the very recent appearence of a new baby. I have a priceless memory of her debugging away on the SimKit while simultaneously nursing Rachel!

There were many interesting problems to be solved. The system itself was straightforward but it had to be completely sealed off from Smalltalk proper, particularly with regard to error messages. Dave Robson came up with a nice scheme (almost an expert system) to capture complaints from the bowels of Smalltalk and translated them into meaningful SimKit terms. There were many user interface details—some workaday, like making new browsers that could only look at the four SimKit classes (Station, Worker, Job, Report), and some more surprising as when we tried it on ten PARC nontechnical adults of about the same age and found that they couldn't read the screen very well. The small fonts our thirtysomething year-old eyes were used to didn't work for those in their 50s. This led to a nice introduction to the system in which the executives were encouraged to customize the screen by choosing among different fonts and sizes with the side effect that they learned how to use the mouse unselfconsciously.

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

98

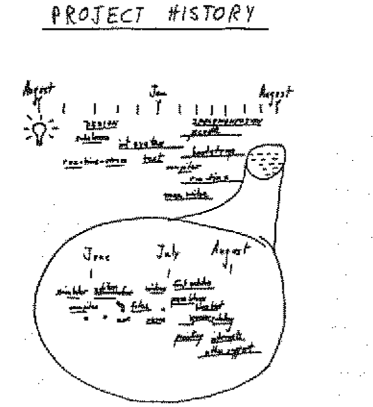
no big crowd days put (whiteness) — with hundreds
admin for [illegible] major
[illegible] in [illegible] ([illegible] in)

KB

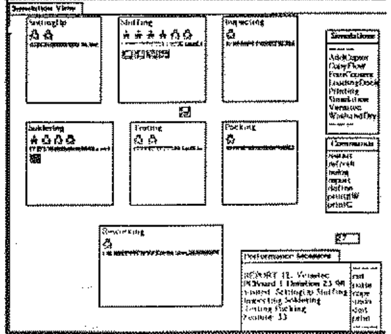
CURRICULUM

DESIGN [illegible] 4 [illegible] [illegible] // DEMOS [illegible] [illegible] /

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



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



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

the line's average probability of manufacturing defects.

Another important system done at this time was Alan Borning's Thinglab [Borning,1979]—the first serious attempt to go beyond Ivan Sutherland's Sketchpad. Alan devised a very nice approach for dealing with constraints that did not require the solver to be omnicient (or able to solve Fermat's last theorem).

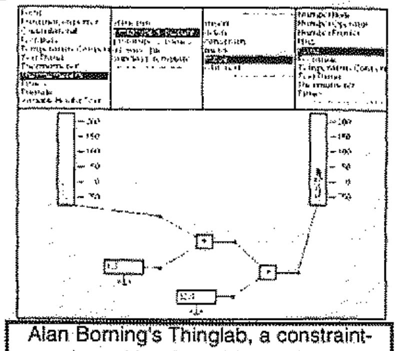
We could see that the "pushing" style of Smalltalk could eventually be relaced by a "pulling" style that was driven by changes to values that different methods were based on. This was an old idea but Thinglab showed how the object-oriented definition could be used to automatically limit the contexts for event-driven processing. And we soon discovered that "prototypes" were more hospitable than classes and that multiple inheritance would be well served if there were classes for methods that knew generally what they were supposed to be about (inspired by Pat Winston's 2nd order models).

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

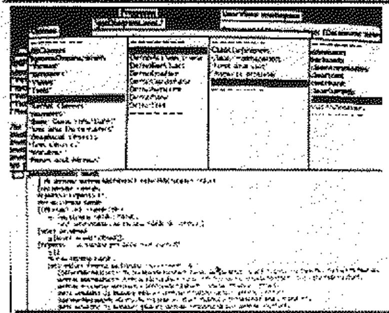
Dan had been interested in the *Notetaker* all along and wanted to see if he could make a version of Smalltalk-76 that could be the *NoteTaker* system. In order for this to happen it would have to run in 256K (the maximum amount of RAM that we had planned for the machine. None of the NOVA-like emulated "machine-code" from the ALTO could be brought over, and it had to fit in memory as well—there would only be floppies, no swapping memory existed. This challenge led to some excellent improvements in the system design. Ted Kaehler's system tracer (which could write out new virtual memories from old ones) was used to clone Smalltalk-76 into the *NoteTaker*. The indexed object table (as was used in early Smalltalk-80) first appeared here to simplify object access. An experiment in stacking contexts contiguously was tried: to save space and gain speed. Most of the old machine code was rewitten 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. Through the 8086 was not as good at bitblt as the ALTO (and much of the former machine code to assist graphics was now in Smalltalk), the overall interpreter was about twice as fast as the ALTO version (because not all the Smalltalk byte-code interpreter would fit into the 4k microcode memory on the ALTO). With various kinds of tricks and turing, graphics display was "largely compen-

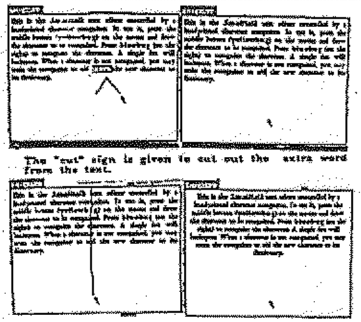
— Learning (Jobs training)
USER: — Rochester Doc Sp
— Twinkle



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

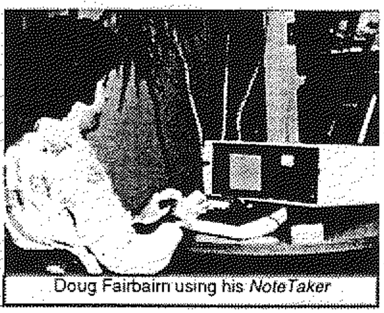


Smalltalk-76 hierarchical class browser designed and built by Larry Tesler



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

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



Doug Fairbairn using his *NoteTaker*

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

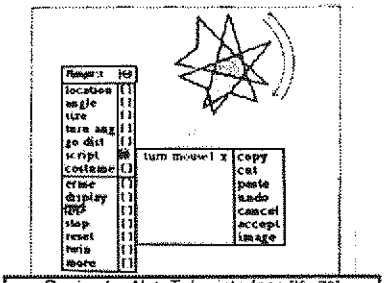
We eventually built about 10 of the machines, and though in many senses an engineering success, what had to be done to make them had once again squeezed out the real end-users for whom it was originally aimed. If Xerox (and PARC) as a whole had believed in these smaller scale ideas, we could have put much more silicon muscle behind the dreams and successfully built them in the 70's when they were first possible. It was a bitter disappointment to have to get the wrong kind of CPU from Intel and the wrong kind of display from HP because there was not enough corporate will to take advantage of internal technological expertise.

87

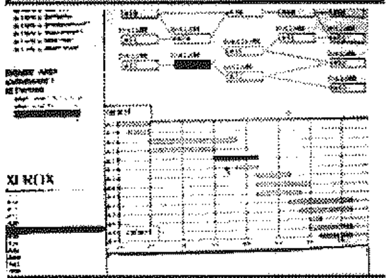
By now it was already 1979, and we found ourselves doing one of our many demos, but this time for a very interested audience: Steve Jobs, Jeff Raskin, and other technical people from Apple. They had started a project called *Lisa* but weren't quite sure what it should be like, until Jeff said to Steve, "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 blt-style scrolling we were using and asked if we could do it in a smooth continuous style. In less than a minute Dan found the methods involved, made the (relatively major) changes and scrolling was now continuous! This shocked the visitors, especially the programmers among them, as they had never seen a really powerful incremental system before.

2units

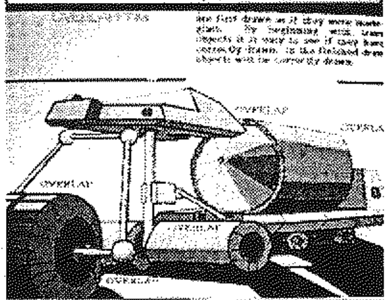
Steve tried to get and/or buy the technology from Xerox (which was one of Apple's minority venture capitalists), but Xerox would neither part with it nor would come up with the resources to continue to develop it in house by funding a better *NoteTaker* cum Smalltalk.



Design for *NoteTaker* interface [Ka 79]



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



Diana Merry at her trusty ALTO

X should to complete
go never really
opened up for
vision of
kids

meeting at his house
ellen and ~79

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

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

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

it was an engineering effort
a [illegible]

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

Coda

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

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

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

Consider the two cases that must be handled efficiently in order to completely wrap objects. It would be terrible if *a+b* incurred <u>any</u> overhead if *a* and *b* were bound, say, to "3" and "4" in a form that could be handled by the ALU. The operation should occur full speed using look-aside logic (in the simplest scheme a single *and* gate) to trap if the operands aren't compatible with the ALU. Now all elementary operations that have to happen fast have been wrapped without slowing down the machine.

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

Again, the whole point of OOP is <u>not</u> 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 <u>have to</u> in the future. Late-binding here involves trapping incompatibilities into <u>re</u>compatibility methods—a good discussion of some of the issues is found in [Popek,1984].

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

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

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

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

Yes!

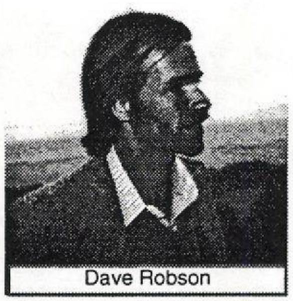
Tongue in cheek, I once characterized progress in programming languages as a kind of "sunspot" theory, in which major advances took place about every 11 years. We started with machine code in 1950, then in 1956 FORTRAN came along as a "better old thing" which if looked at as "almost a new thing" became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

looked at as "almost a new thing" became the precursor of Smalltalk in 1972.

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

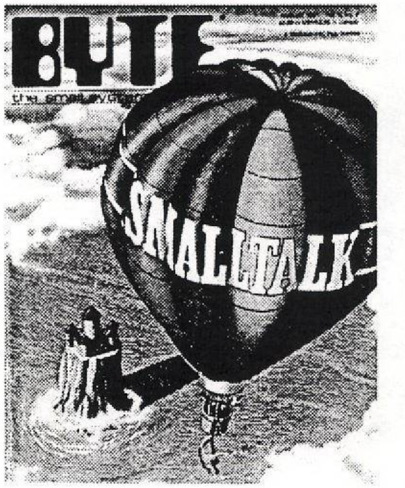
agree

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 disatisfaction with one's designs using the entire history of human art as a standard and goad. Then the trick is to decouple the disatisfaction from self worth—otherwise it is either too depressing or one stops too soon with trivial results.


Dave Robson

I will leave the story of early Smalltalk in 1981 when an extensive series of articles on Smalltalk-80 was published in *Byte* magazine, [Byte,1981] followed by Adele's and Dave Robson's books [Goldberg,1983] and the official release of the system in 1983. Now programmers could easily implement the virtual machine without having to reinvent it, and, in several cases, groups were able to roll their own *image* of basic classes. In spite of having to run almost everywhere on moribund HW architectures, Smalltalk has proliferated amazingly well (in part because of tremendous optimization efforts on these machines) [Deutsch 83]. As far as I can tell, it still seems to be the most widely used system that claims to be object-oriented. It is incredible to me that no one since has come up with a qualitatively better idea that is as simple, elegant, easy to program, practical, and comprehensive. (It's a pity that we didn't know about PROLOG then or vice versa, the combinations of the two languages done subsequently are quite intriging.)

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



theme
write to understand

References Cited In The Text

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

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

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

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

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

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

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

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

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

[Bruner 1966] ______, *Towards a Theory of Instruction,* Harvard/Belknap Press,1966, ISBN 0-674-89700-5.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[Joss, 1978] ________, JOSS Session, in *History of Programming Languages,* ed. Richard L. Wexelblat, New York: Academic Press, xxiii, Chapter X, 1981. ISBN: 0127450408. Conference: History of Programming Languages Conference (1978: Los Angeles, Calif.)

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

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

[Kay, 1969] ______, The reactive engine, PhD thesis, University of Utah, September 1969.

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

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

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

[Kay, 1971b] ______Computer Structures-Past Present and Future, Panel paper, in *Proceedings of the FJCC,* Vol. 39 November 1971

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

[Kay, 1972a] ______, Learning research group 3 year plan, Xerox Palo Alto Research Center, July 1972.

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

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

[Kay, 1972d] ______, Smalltalk Blue Book, Fall 1972.

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

[Kay,1977] ______, 1977, Microelectronics and the personal computer, *Scientific American,* (pp. 125-136) September 1977.

[Kay, 1977a] ______ and Goldberg, Adele., Personal dynamic media, *IEEE Computer,* Vol. 10, (pp. 31-41), March 1977. Reprinted in *A History of PersonalWorkstations* , Academic Press, 1988.

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

[Kay, 1984] ______, 1984, Computer software,*Scientific American,* September 1984.

[Kay,1990] ______, User interface: a personal view, in *The Art of Human-Computer Interface Design*, ed., Brenda Laurel, Addison-Wesley Publishing Co.,1990, (pp. 191-207) ISBN 0 201 51797 3.

[Kay, 1991] ______, 1991, Computers, networks, and learning, *Scientific American* , Vol. 265, No. 3, (pp. 138-148) September 1991.

[Kiczales, 1991] Kiczales, Gregor, Des Rivleres, Jim; Bobrow, Daniel G., *The Art of the Metaobject Protocol* , Cambridge, Mass. : MIT Press, viii, 335 p.; 1991, ISBN 0262111586 .

[Knuth, 1971] Knuth, Donald E and Floyd, Robert W., Notes on avoiding 'go to'statements, in *Information Processing Letters*, volume, 1, number 1, February 1971.

[Knuth, 1974] ________, Structured programming with 'go to' statements, in *ACM Computing Surveys*, vol. 6, no. 4, (pp. 261-301), December 1974.

[Krasner,1983] Krasner, Glenn., ed., *Smalltalk-80 Bits of History, Words of Advice..*, Addison-Wesley, 1983, ISBN 0 201 116 69 3.

[Lampson,1966] Lampson, , CAL reference manual, Project GENIE documentation, Computer Center, UC Berkeley, 1966.

[Lampson,1966a] _______, A user machine in a time sharing system, in *Proceedings of the IEEE* , 54(12): (pp.1744-1766), December 1966.

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

[Lampson 1972] _______, Why Alto?, in *PARC Lab Book,* Xerox Palo Alto Research Center.

[Lampson,1988] ________, Personal distributed computing: alto and ethernet software, in *A History of Personal Workstations*, Adele Goldberg, ed., New York:New York, ACM Press, 1988, ISBN 020 2590.

[Landin, 1965] Landin, P.J., A correspondence between ALGOL 60 and Church's lambda notation: Part 1, in *Communications of the ACM*, Vol. 8, No. 2, February 1965.

[Landin, 1966] ______, The next 700 programming languages, in*Communications of the ACM* , Vol. 9, No. 3, March 1966. (pp. 157-164).

[Licklider,1960] Licklider, J.C.R., Man-computer symbiosis, in *IRE Transactions onHuman Factors in Electronics*, HFE-1: 4-11, 1960.

[LRG, 1976] Learning Research Group, Dynamic Personal Media, Xerox Palo Alto Research Center, Report #SSL-76-7, June 1976.

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

[McCarthy, 1962] ______, et.al., *LISP 1.5 Programmer's Manual* , Cambridge: MIT Press, 1962.

[Minsky, 1970] Minsky, Marvin., Form and content in computer science, in*The Journal of the Association for Computing Machinery,* Vol 17, Number 2, (pp. 197-215), April 1970.

[Minsky, 1974] ________, A framework for representing knowledge, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory Memo No. 306, June 1974. Reprinted in *The Psychclogy of Computer Vision,* McGraw-Hill, 1975.

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

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

[Nygaard, 1978] ______, Early history of simula, in *History of Programming Laguages* , ed. Richard L. Wexelblat, New York: Academic Press, 1981, ISBN 012 745040 8. This is the proceedings of the ACM Sigplan History of Programming Languages Conference held in Los Angeles, June 1-3, 1978.

[Papert, 1971] Papert, S., Teaching children thinking, MA: Massachusetts Institute of Technology, Artifical Intelligence Laboratory Memo 247, LOGO Memo 2, 1971.

[Papert, 1971a] ________, Teaching children to be mathematicians vs. teaching about mathematics, MA: Massachusetts Institute of Technology, Artifical Intelligence Laboratory Memo 249, LOGO Memo 4, 1971.

[Papert, 1973] ________, Uses of technology to enhance education, MA: Massachusetts Institute of Technology, Artifical Intelligence Laboratory Memo 298, LOGO Memo 8, 1973.

[Papert, 1976] _________, Abelson, H., Bamberger. J, and Goldstein, I., LOGO Progress Report 1973-1975, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 356, LOGO Memo 22, 1976.

[Papert, 1976] _________, Proposal to the National Science Foundation: An Evaluative Study of Modern Technology in Education, Appendix One: LOGO memo 8, Appendix Two: LOGO memo 27, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory, memo 371, LOGO memo 26, 1976.

[Papert, 1976a] ________,; Solomon, C.J., A Case Study of a Young Child doing Turtle Graphics in LOGO, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Memo 375, LOGO Memo 28, 1976.

[Papert, 1979] _________; Watt, D., DiSessa, A., Weir, S.,. Final Report of the Brookline LOGO Project. Part II: Project Summary and Data Analysis, MA: Massachusetts Institute of Technology, Artificial Intelligence Laboratory Memo 545, LOGO Memo 53, 1979.

[Perry,1985] Perry, Tekla,*"Inside the PARC: the "Information Architects'"* IEEE Spectrum, October 1985.

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

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

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

[Ross, 1961] ________, A generalized technique for symbol manipulation and numerical calculation, in *Communications of the ACM,* Vol. 4, no. 3, (pp. 147-150) March 1961.

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

[Saunders, 1977] Saunders, Steven E., Improved FM audio synthesis methods for real-time digital music generation, in *Computer Music Journal*, Vol. 1, No. 1, February, 1977. Reprinted in *Computer Music*, Roads, C. and Strawn, J. editors, Cambridge: MIT Press, 1985.

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

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

[Soloway, 1989] Soloway, Elliot and Spohrer, James C., ed.,*Studying the Novice Programmer*, New Jersey: Lawrence Erlbaum Associates, Inc., 1989, ISBN 0-8058-002.

[Smith,1975] Smith, David Canfield, *Pygmallion*, PhD thesis, Stanford Univ., 1975

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

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

[Sutherland, 1963a] ______, ibid, in *Proceedings of the SJCC*, Vol. 23, (pp. 329-346), 1963.

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

[Tesler, 1973] Tesler, Lawrence., et.al., The lisp-70 pattern matcher, in *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, Stanford, CA, 1973.

[Tesler, 1977] ______, Smalltalk-76 documentation, Xerox Palo Alto Research Center, Learning Research Group Internal Note, 1977.

[Tesler,1981] ______, 1981, The smalltalk environment, Byte , Vol. 6, Number 8, (p. 90), August, 1981.

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

[Thacker 1982] _______, et. al., *Alto: a personal computer, inComputer Structures: Principles and Examples*, Siewiorek, D. et.al. editors, Chapter 33, McGraw-Hill, 1982.

[Thacker, 1986] ______, Personal distributed computing: the alto and ethernet hardware, in *A History of Personal Workstations*, Adele Goldberg, ed., New York: New York, ACM Press, (pp.267-290), 1988, ISBN 020 111 2590.

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

[Van Wijngaarden] ______, Generalized ALGOL, Mathematisch Centrum, Amsterdam, Netherlands.

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

[Winston, 1970] Winston, Patrick H., Learning structural descriptions from examples, PhD thesis, MIT, January 1970.

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

Appendix I: Personal Computer Memo

Smalltalk Program Evolution

From a memo on the "KiddiKomputer"

To: Butler Lampson, Chuck Thacker, Bill English, Jerry, Elkind, George Pake
Subject: "KiddiKomputer"
Date: May 15, 1972

****

4. January 1972

The Reading Machine[1]. Another attempt to work on the actual problem of a personal computer. Every part of this gadget (except display) is buildable now but requires some custom chip design and fabrication. This is discussed more completely later on. A meeting was held with all three labs to try to stimulate invention of the display.

B. Utility

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

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

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

b. Teaching "models" through "simulation" of systems with similar semantics and different syntax. This could be grouped with (a) although the emphasis is a bit different. The initial two systems would be music and programming and would be an extension of some stuff I did at Utah in 1969-70 with the organ/computer there.

c. Teaching "interface" skills such as "seeing" and "hearing". The initial "seeing" project would be an investigation into how reading might be taught via combining iconic and audible representation of works in a manner reminiscent of Bloomfield and Moore. This would require a corollary inqiry into why good readers do so much better than average readers. A farther off project in the domain of sight would be an investigation into the nature and topology of kids' internal models for objects and an effort to perserve iconic imagery from being totally replaced by a relational model.

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

3. Second Level Projects

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

b. Iconic programming. If we believe Piaget and Bruner, kids deal mostly with

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

****

C. The Viability Of miniCOM

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

Rationalizations for building a bunch of them:

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

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

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

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

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

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

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

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

Appendix II: Smalltalk Interpreter Design

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

One way to do this was to use the approach of Dave Fisher, in which the no-man's land of control structures is made accessable by providing a protected way to access and change the relationships of the static and dynamic environment[Fisher 70]. In an object-based scheme, the protection can be provided by the objects themselves and many of Fisher's techniques are even easier to use. The effect of all this is to extend the *eval* by distributing it: both to the individual objects that participate in it and dynamically as the language is extended.

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

The biggest problem remaining was that I wanted to have a much nicer syntax than Lisp and I didn't want to use any of my precious "half-page" to write even a simple translator[21]. Somehow the *eval* had to be designed so that syntax got specified as part of the use of the system, not in its basic definition.[22]

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

```
receiver     | message
c            | ∘ i <- d*e
```

The message is made up of a literal token "∘", an expression to be evaluated in the sender's context (in this case *i*), another literal token <-, followed by an expression to be evaluated in the sender's context (*d*e*). "LISP" pairs are made from 2 element objects and can be indexed more simply: *c hd, c tl*, and *c hd <- foo*, etc.

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

```
receiver     | message
3            | +4
```

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

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

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

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

What seemed to remain for the *eval* was simply to show what a message send actually consisted of. For this system a send is the equivalent not of a postman delivering a letter, but simply delivering a notice of where the letter was to be found. It is up to the receiving object to do something about it. In fact, it could ignore the request, complain about it, invoke inferential processes elsewhere, or simply handle it with one of its own messages.

The final thing I had to do was to extend the uniform syntax idea of *receiver message* to cover all cases, including message receipt and simple control structures. So, we need some objects to pattern match and evaluate, to return and define, etc.

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

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

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

c hd <- 3+4

would be dealt with as follows: Control is passed to that object and the first test is to see if the symbol *hd* appears in the message (□*hd* »). It does. The next check is for an "assignment" token (□<- »). It's there. Last, we want to evaluate the rest of the message (we get 7), bind the value to the internal instance variable *t* and, finally return this value to the sender (^:*t*). So this is like: (REPLACA C (PLUS 3 4)).

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

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

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

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

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

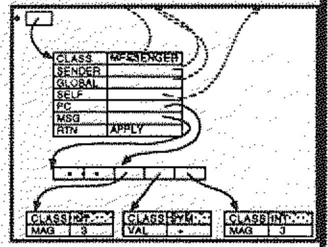
We start in the middle of a method of some class of objects and we need to evaluate "3+4". The essentials of the *eval* are those that successfully take us into the method of "3" in class integer. Since all methods are only in terms of *sends* and all sends are done in a similar manner, this is enough.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

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

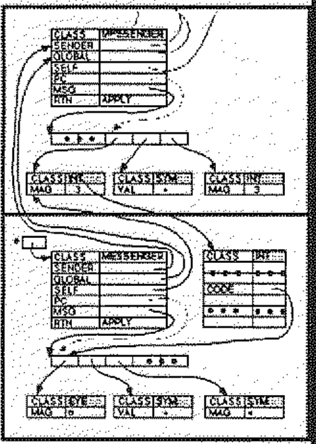
I have hand-evaluated this nonrecursive version in a number of cases and it seems to work pretty well, but there are probably some bugs. If a reader feels prompted to come up with an even nicer, tidier, and smaller scheme, I would be glad to look at it.

The "One Pager"

"Before"



"During"



"After"

```
e       (the environment) will be bound to the current Messenger object
result  holds the result of a send, usually to be applied to next part of message
```

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

apply: 'e <- e*SENDER;
          e*PC <- e*PC + 1;
          if e*PC > length(MSG) then goto dispatchrtn;
          if e*MSG*PC  =  '.     then e*PC <- e*PC + 1; goto evlist;
          if e*MSG*PC  =  '»     then if result = 'false
                                      then  e*PC <- e*PC + 2; goto evlist;
                                      else  e*PC <- e*PC + 1;
                                            'e <- Table(CLASS      MESSENGER,
                                                        SENDER,    e,
                                                        GLOBAL,    GLOBAL,
                                                        SELF       result,
                                                        PC,        1,
                                                        MSG,       e*MSG*PC,
                                                        RTN,       FROMTRUE);
                                            goto eval;
          'e <- Table(   CLASS,      MESSENGER,
                         SENDER,     e,
                         GLOBAL,     GLOBAL,
                         SELF,       result,
                         PC,         1,
                         MSG,        result*CLASS*CODE,
                         RTN,        APPLY);
          goto eval;

fromTRUE: 'e <- e*SENDER*SENDER; goto dispatchrtn;

fromEYE: putvalue(e*GLOBAL, e*p, result); goto apply;

dispatchrtn: select e*RTN
                case APPLY:          goto apply;
                case FROMTRUE:       goto fromTRUE;
                case FROMEYE:        goto fromEYE;

escapes:    select e*MSG*PC+1
                etc....
```

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

Appendix III: Acknowledgements

1971

Chris Jeffers, + ?

1972

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

1973

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

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

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

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

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

The notions of code as a data structure; intensional properties of names (property lists of attribute-value pairs on atoms); evaluation with respect to arbitrary environments; etc., are found in LISP, probably the greatest single design for a programming language yet to appear. SMALLTALK is definitely "LISPlike".

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

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

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

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

SMALLTALK.

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

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

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

****

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

****

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

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

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

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

****

Dan Ingalls of our group at PARC, the implementor of SMALLTALK, has revealed many design flaws through his several, excellent quick "throw away" implementation of the language. SMALLTALK could not have existed with his help, virtuosity, and good cheer.

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

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

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

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

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

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

1976

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

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

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

Visitors
Eric Martin
Ron Baecker
Bonnie Tenenbaum

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

Appendix IV: Event Driven Loop Example

First we make a class for events:

```
to event | mycode
     (ISNEW      »     ('mycode <- array 3.
                        mycode[2] <- 'done.)
     □newcode   »     (mycode[1] <- :.)
     □is             » (isIT eval)
     mycode eval)
```

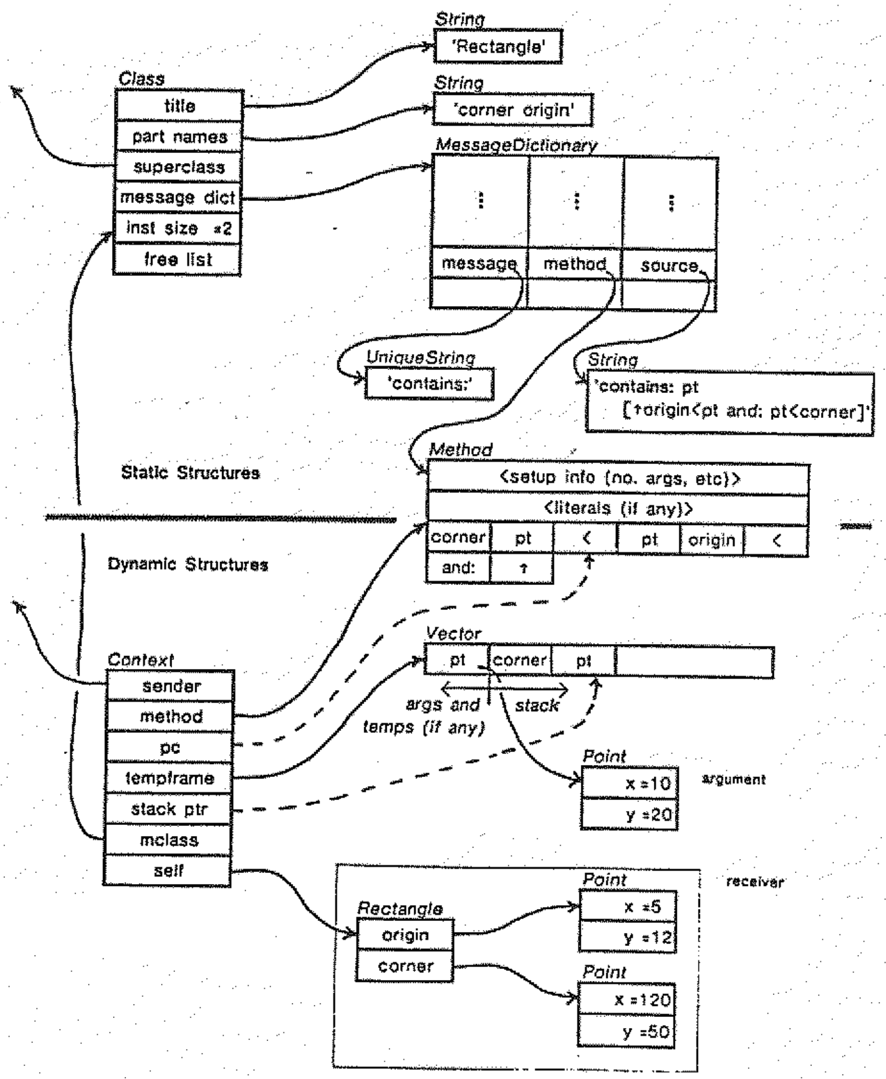
Each event stores away code to be executed later (the *done* will eventually cause an exit from the driving loop in the *until* structure, defined next:

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

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

Appendix V: Smalltalk-76 Internal Structures

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



Inter-Office Memorandum

To	LRG and other interested parties	Date	April 8, 1981
From	P. Deutsch	Location	Palo Alto
Subject	Smalltalk data base ** draft **	Organization	LRG

XEROX

Filed on: [Phylum]<Deutsch>st80db.memo

This memo represents a first, incomplete draft of documentation for the data base facilities I've been building for Smalltalk-80. Since this work is still very much in progress, I need comments not only on the quality of the document but (even more) on the system itself, at every level from the choice of message names to the underlying data model.

Before reading this memo, you should probably be familiar with some of the Cedar Data Base documentation, particularly the "concepts and facilities" document ([Ivy]<CedarDB>docs>xxxxx) and perhaps also the paper written by the Cedar DB group describing the system (xxxxx). You should also be reasonably familiar with the Smalltalk world-view, and be able to read fragments of Smalltalk-80 code (which is enough like Smalltalk-76 that you shouldn't have much trouble if that's all you're familiar with).

World view

A reasonable way to think of the Smalltalk Data Base world is that it is just an extension of the familiar Smalltalk world of classes, objects, instance variables, and messages. The extensions fall into several categories:

- It is possible to dynamically create and delete instance variables in classes.

- An instance variable may optionally be typed, and an error message will result if an attempt is made to store something of the wrong type into a variable.

- There are extensive facilities for indexing all the objects of a given class, and retrieving those instances of a class which satisfy given properties.

At the moment there are some substantial restrictions as well:

- There is no subclass hierarchy or inheritance.

- There is no way to write methods in data base classes -- data base objects are passive data objects.

- Data bases are not permanent -- you have to use something equivalent to filin and filout to save them.

- Data bases are not shared between users, except by the brute force mechanism of filing in a data base that someone else filed out.

- There is no notion of an atomic transaction.

- There is no protocol for correlating the information in several classes (called a "join" in the data base world).

Fortunately, designs exist for removing all of these restrictions, so you have something to look forward to in the future.

Defining classes

One difference between DB classes and ordinary classes is that even though DB classes must all have unique names (like ordinary classes), they aren't automatically put into the Smalltalk symbol table. This is partly because the Browser and the Smalltalk system don't support DB classes, so the current assumption is that creation of classes is something that you do in a program rather than from the terminal. Likewise, there is no easy way to edit, inspect, etc. a DB class (although as we will see later, these things are all possible in a very uniform way).

There are two kinds of DB classes and objects in the world. DB objects which exist in their own right are called *entities*, and their classes are called *domains*. DB objects which only exist to record relationships between other objects are called *relationships*, and their classes are called *relations*. At the time you create a DB class, you must decide whether it is a domain or a relation. To create a domain,

```
aDomain ← DomainDomain named: 'someName'.
```

Likewise, to create a relation,

```
aRelation ← RelationDomain named: 'someName'.
```

`DomainDomain` and `RelationDomain` play the role of metaclasses (more or less). Note that if a domain or relation by the given name already exists, you will get the existing one, not an error message.

DB classes are themselves entities, instances of (as you may have guessed already) either the `Relation` domain or the `Domain` domain.

Defining attributes

Once you have created a DB class, the next step is to define its *attributes*, which correspond to instance variable names. (DB classes don't have anything corresponding to class variables.) Unlike Smalltalk classes, where you define the instance variable names all at once, for DB classes you define the attributes individually. Thus to add an attribute to a DB class,

```
anAttribute ← aDBclass newAttribute: 'attrName'.
```

what are these? symbols?

Again, if an attribute with that name already exists, you will get it. A different message, which gives an error if the named attribute does not exist, is

```
anAttribute ← aDBclass attributeNamed: 'attrName'.
```

Note that you can go on adding attributes to a DB class even after instances exist: the new attribute values in existing objects will be whatever the "undefined" value is for the type of data stored under that attribute (normally `nil`).

Attributes created as just shown will allow any kind of Smalltalk object to be stored under them. There are two ways you can restrict what can be stored under an attribute:

- You can require that the value be an instance of a specific class or domain.

- You can require that the value be chosen from a specific set of alternatives (e.g. #red, #green, #blue).

To define an attribute with one of these properties, you need to get hold of an object (actually an entity) called a ValueType. The protocol for getting ValueTypes is pretty ugly right now: you can say

```
ValueTypeDomain named: 'Point'
```

to get a ValueType that represents "restriction to instances of Point", or

```
ValueTypeDomain named: #(red green blue)
```

to get a ValueType that represents "restriction to one of #red, #green, or #blue". Then to get an attribute with a type restriction,

```
anAttribute ← aDBclass newAttribute: 'attrName' valueType:
aValueType.
```

An attempt to store something of the wrong type under an attribute will produce an error message.

Creating instances

To create an instance of a relation,

```
aRelationship ← aRelation create.
```

To create or retrieve an entity in a domain,

```
anEntity ← aDomain named: 'someName'.
```

Entities without names are not allowed to exist. As remarked above, named: will either retrieve or create an entity of a given name.

Manipulating instances

DB objects behave very much like (passive) Smalltalk objects in that they have protocol for accessing and storing into their instance variables (attribute values). Unlike Smalltalk objects, in which each instance variable normally has a separate message to access it, DB objects have a protocol more like dictionaries. To access an instance variable,

```
value ← aDBobject at: anAttribute.
```

To store into an instance variable,

```
aDBobject at: anAttribute put: newValue.
```

The data base world provides an explicit notion of "undefined" which is a slight generalization of the Smalltalk use of nil. To set an instance variable to "undefined",

```
aDBobject deleteAt: anAttribute.
```

Note that an attribute is only usable in conjunction with instances of the particular DB class it was created under. This is quite different from the Smalltalk notion that you can take any variable-sized object and subscript it, or send a message to any object that understands it; it is more like the notion of a field of a record in a strongly typed language. For these reasons among others, it isn't clear that doing things this way is a good idea.

Retrieval

Flexible retrieval is one of the primary purposes of data bases. For this reason, the Smalltalk DB has a fairly elaborate protocol for retrieving subsets of DB classes according to some criterion. The basic model here is that you build up a request for a retrieval, and then ask for the request to become a stream: at that moment, the retrieval actually gets done, and you can then treat the object you have been manipulating as an ordinary stream. The basic pattern for retrievals is something like this:

collection?

```
((aRelation find with: attr1 equals: value1) with: attr2 between:
lowValue and: highValue) do: aBlock.
```

The `find` message returns an object called a `Query` which then takes further messages specifying what the retrieval conditions are. The `do:` message implicitly converts the `Query` into a stream. So the essence of the retrieval protocol is the set of messages that turn `Querys` into more fully specified `Querys`.

why

Messages that specify restrictions on individual attributes:

```
with: anAttribute equals: value

with: anAttribute between: lowValue and: highValue
```

match: 'xyz*'

Messages that allow you to specify an arbitrary condition:

```
suchThat: aBlock
```

our convention Do: for single arg

The block should be of the form `[:aDBobject| someCondition]`, specifying that only objects for which the condition is true should be included. This is the most general kind of restriction, but it is less efficient than specifying attribute restrictions since attribute restrictions can make use of indexes.

Messages that call for sorting the result:

```
withAscending: anAttribute

withDescending: anAttribute
```

If you specify sorting on more than one attribute, the first attribute is taken as most significant, the second will only be consulted if the values under the first are equal, etc.

In addition, there are two messages designed specifically for situations where you know that there must be exactly one object satisfying a given set of restrictions and want an error message if this is not the case:

```
uniqueWith: anAttribute equals: value

justOne
```

These message return the actual DB object satisfying the conditions, so they must be the last one in the series if they appear.

In order to make retrievals run fast, you can ask a DB class to maintain an index on a particular attribute. This will make retrievals using only that attribute run very fast, and retrievals using that attribute among others run a lot faster. It is not necessary to have an index to do retrievals: the DB class will search its instances sequentially if no index is available. To add an index,

```
aDBclass addIndex: anAttribute
```

To delete an index,

```
aDBclass dropIndex: anAttribute
```

Destroying instances

Unlike Smalltalk, where an object can disappear as soon as there are no references to it, the data base world has to retain objects indefinitely, since there is no way for the system to predict that you won't retrieve the object by giving a description of it at some future time. (The situation in Smalltalk itself would be analogous if people started making heavy use of `aClass allInstancesDo: aBlock`.) This isn't actually quite true, as we shall see in a moment, but it's pretty close. So to make a DB object disappear,

```
aDBobject delete.
```

release

What does "disappear" mean? It means that *as far as the data base system is concerned*, the object ceases to exist; no references to it will remain within the data base system. If you still have a reference to it hanging around in your own data structures somewhere and attempt to use that reference, bizarre things will probably happen, although (currently) the data base system doesn't alter the object in a way that could bring down the Smalltalk system.

When you delete a relationship, nothing else happens. When you delete an entity, however, two things happen automatically:

- All relationships containing references to that entity get deleted.

- All entities containing references to that entity have the reference replaced by `nil`.

As you might expect, more far-reaching things happen if you delete system entities. Deleting an attribute entity effectively deletes the attribute value from all DB objects to which that attribute is applicable. Deleting a DB class (either a relation or a domain) automatically deletes all its attributes and instances first. The warning about not holding on to a deleted object through a reference outside the data base system applies with particular force to system entities!

System objects

All the objects in the data base system itself are DB objects -- no surprise. In particular, there are four system domains:

```
DomainDomain
RelationDomain
AttributeDomain
ValueTypeDomain
```

All of these except for `AttributeDomain` have only one applicable attribute,

```
EntityName
```

which refers to the name of the entity. Thus, for example, to rename an attribute, domain, etc.

```
        aSystemObject at: EntityName put: 'newName'.
```

Notice that `EntityName` violates the rule that an attribute is only applicable to a single DB class: the system entities don't have to play by the same rules as the ones you create.

`AttributeDomain` has two other attributes:

> `AttrDBClass`, the DB class to which the attribute is applicable;
> `AttrValueType`, the type of value which must appear under that attribute.

There are a number of predefined `ValueTypes` for the most common kinds of values:

> `VTInteger`
> `VTString`
> `VTBoolean` (true or false)
> `VTAny` (any Smalltalk object)

Indexes aren't currently entities, but they probably should be.

Miscellaneous

`aDBclass storeOn: aFile` will filout a single DB class. You can filin this file at a later time, and it will reconstruct the class. Note, however, that if any instances of the class contain references to other DB objects, you have to filin (or at least create) that class first. To help with this, `aDBclass describeOn: aFile` will write out something which will create the class properly if it doesn't already exist. Thus the proper way to filout a group of DB classes whose instances can refer to each other freely is

```
        aDBclass1 describeOn: aFile.
        aDBclass2 describeOn: aFile.
        ...
        aDBclassN describeOn: aFile.
        aDBclass1 storeOn: aFile.
        aDBclass2 storeOn: aFile.
        ...
        aDBclassN storeOn: aFile.
```

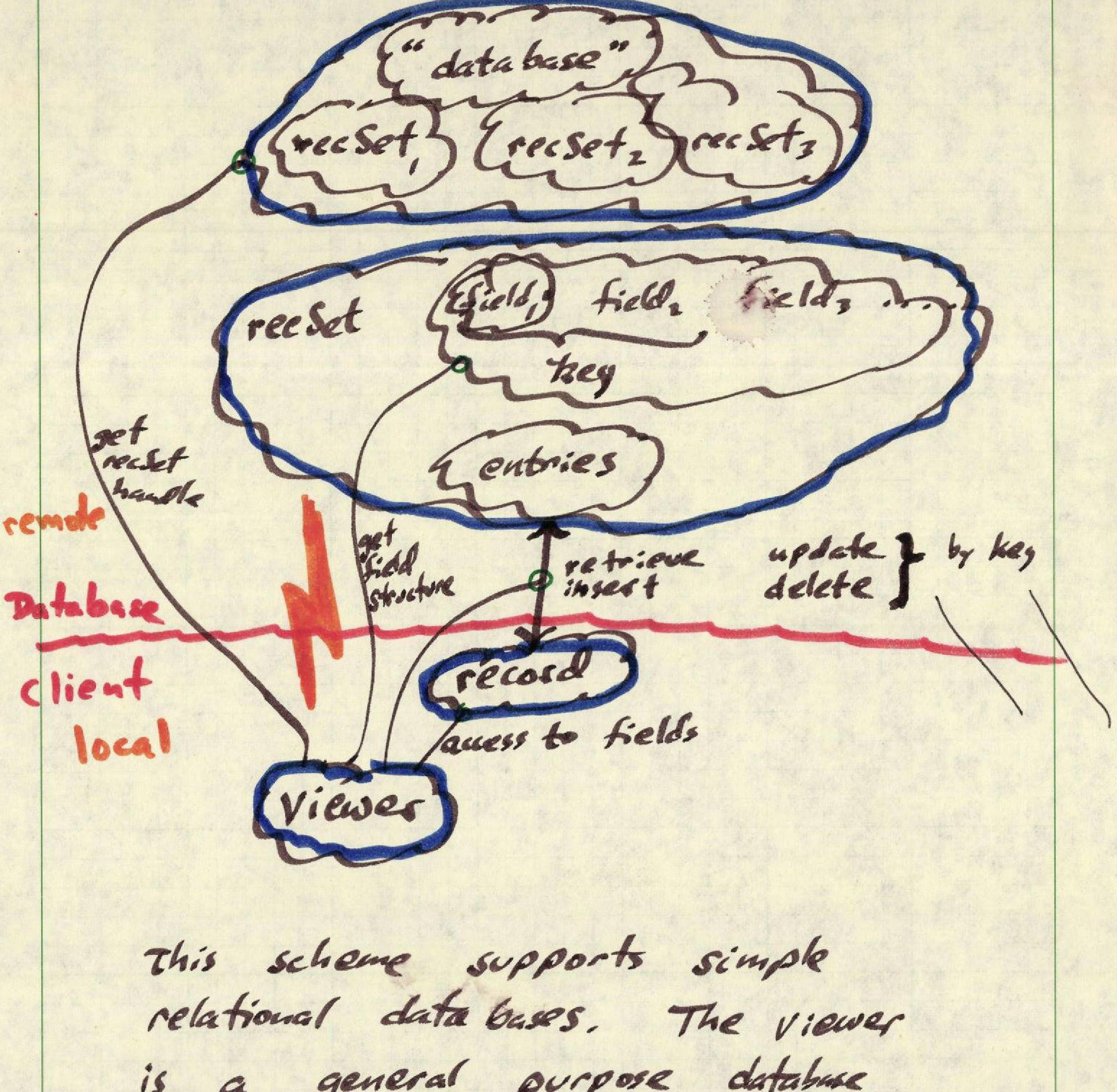
How to run it

The data base code currently resides on [Phylum]<Deutsch>Data-base-classes.st and Data-base-kernel.st. Filin these two files in order (i.e. Data-base-classes.st before Data-base-kernel.st). To initialize the system after filing it in, execute `ValueType finishInit`. That's all there is to it!

Needless to say, there are surely many bugs, omissions, and quirks not covered in this memo, since the data base system has only had one user (Bill Finzer, who has suffered considerably as a user of a somewhat earlier version of the system.) Please tell me about any problems you encounter.

7/30/89 Xerox

Basic Structure for Databases



This scheme supports simple relational databases. The viewer is a general purpose database access tool. Other _clients_ can access db using same handles/protocols viewers use. The db implementation is only modelled ~ leaving it open.

Richard Moore

7/30/80 Initial Development Directions (orthogonal)

- more than one access path (index) can be provided, in addition to key.
- some kind of locking or "check before modify" protocol can be designed
- ★ Specific Shasta applications can ~~b~~ be programmed as prompt-response interaction
- general paradigm for Forms design & Forms interpretation can be developed.
- ★ system can be performance tuned

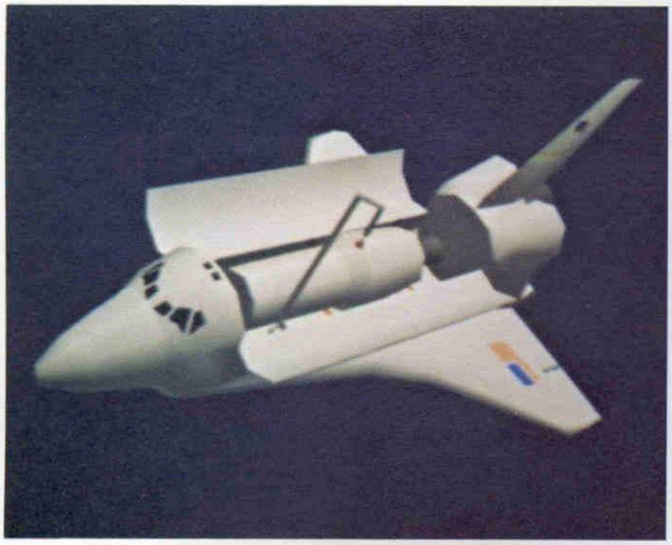
384
SCIENTIFIC
AMERICAN OFFPRINTS

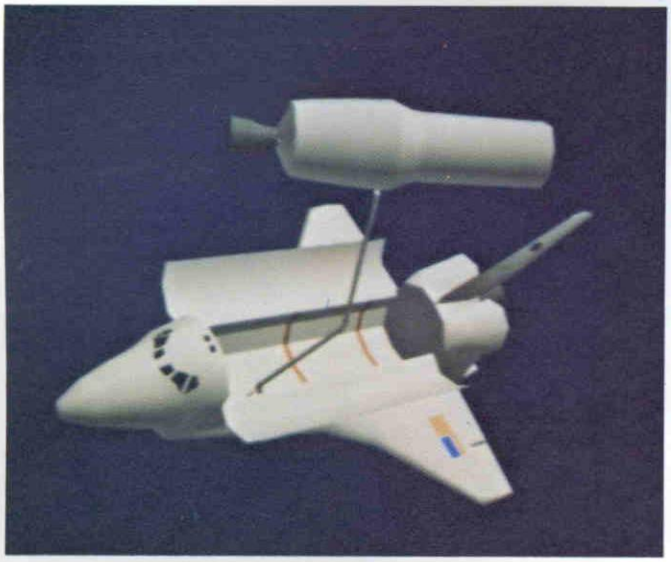
Microelectronics and the Personal Computer

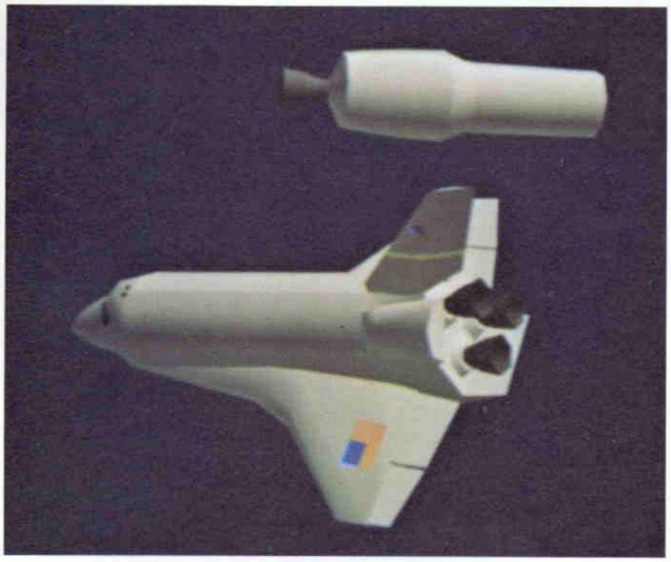
by Alan C. Kay

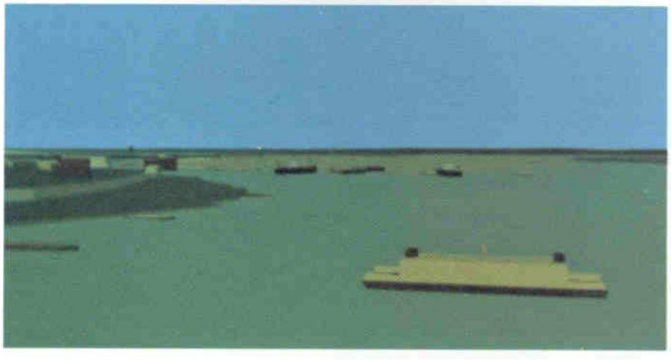
SCIENTIFIC
AMERICAN
SEPTEMBER 1977
VOL. 237, NO. 3 PP. 230—244

PUBLISHED BY **W. H. FREEMAN AND COMPANY** 660 MARKET STREET, SAN FRANCISCO, CALIFORNIA 94104









Microelectronics and the Personal Computer

Rates of progress in microelectronics suggest that in about a decade many people will possess a notebook-size computer with the capacity of a large computer of today. What might such a system do for them?

by Alan C. Kay

The future increase in capacity and decrease in cost of microelectronic devices will not only give rise to compact and powerful hardware but also bring qualitative changes in the way human beings and computers interact. In the 1980's both adults and children will be able to have as a personal possession a computer about the size of a large notebook with the power to handle virtually all their information-related needs. Computing and storage capacity will be many times that of current microcomputers: tens of millions of basic operations per second will manipulate the equivalent of several thousand printed pages of information.

The personal computer can be regarded as the newest example of human mediums of communication. Various means of storing, retrieving and manipulating information have been in existence since human beings began to talk. External mediums serve to capture internal thoughts for communication and, through feedback processes, to form the paths that thinking follows. Although digital computers were originally designed to do arithmetic operations, their ability to simulate the details of any descriptive model means that the computer, viewed as a medium, can simulate any other medium if the methods of simulation are sufficiently well described. Moreover, unlike conventional mediums, which are passive in the sense that marks on paper, paint on canvas and television images do not change in response to the viewer's wishes, the computer medium is active: it can respond to queries and experiments and can even engage the user in a two-way conversation.

The evolution of the personal computer has followed a path similar to that of the printed book, but in 40 years rather than 600. Like the handmade books of the Middle Ages, the massive computers built in the two decades before 1960 were scarce, expensive and available to only a few. Just as the invention of printing led to the community use of books chained in a library, the introduction of computer time-sharing in the 1960's partitioned the capacity of expensive computers in order to lower their access cost and allow community use. And just as the Industrial Revolution made possible the personal book by providing inexpensive paper and mechanized printing and binding, the microelectronic revolution of the 1970's will bring about the personal computer of the 1980's, with sufficient storage and speed to support high-level computer languages and interactive graphic displays.

Ideally the personal computer will be designed in such a way that people of all ages and walks of life can mold and channel its power to their own needs. Architects should be able to simulate three-dimensional space in order to reflect on and modify their current designs. Physicians should be able to store and organize a large quantity of information about their patients, enabling them to perceive significant relations that would otherwise be imperceptible. Composers should be able to hear a composition as they are composing it, notably if it is too complex for them to play. Businessmen should have an active briefcase that contains a working simulation of their company. Educators should be able to implement their own version of a Socratic dialogue with dynamic simulation and graphic animation. Homemakers should be able to store and manipulate records, accounts, budgets, recipes and reminders. Children should have an active learning tool that gives them ready access to large stores of knowledge in ways that are not possible with mediums such as books.

How can communication with computers be enriched to meet the diverse needs of individuals? If the computer is to be truly "personal," adult and child users must be able to get it to perform useful activities without resorting to the services of an expert. Simple tasks must be simple, and complex ones must be possible. Although a personal computer will be supplied with already created simulations, such as a general text editor, the wide range of backgrounds and ages of its potential users will make any direct anticipation of their needs very difficult. Thus the central problem of personal computing is that nonexperts will almost certainly have to do some programming if their personal computer is to be of more than transitory help.

To gain some understanding of the problems and potential benefits of personal computing my colleagues and I at the Xerox Palo Alto Research Center have designed an experimental personal computing system. We have had a number of these systems built and have studied how both adults and children make use of them. The hardware is faithful in capacity to the envisioned notebook-

COMPUTER SIMULATIONS generated on a high-resolution television display at the Evans & Sutherland Computer Corporation show the quality of the images it should eventually be possible to present on a compact personal computer. The pictures are frames from two dynamic-simulation programs that revise an image 30 times per second to represent the continuous motion of objects in projected three-dimensional space. The sequence at the top, made for the National Aeronautics and Space Administration, shows a space laboratory being lifted out of the interior of the space shuttle. The sequence at the bottom, made for the U.S. Maritime Administration, shows the movement of tankers in New York harbor. Ability of the personal computer to simulate real or imagined phenomena will make it a new medium of communication.

size computer of the 1980's, although it is necessarily larger. The software is a new interactive computer-language system called SMALLTALK.

In the design of our personal computing system we were influenced by research done in the late 1960's. At that time Edward Cheadle and I, working at the University of Utah, designed FLEX, the first personal computer to directly support a graphics- and simulation-oriented language. Although the FLEX design was encouraging, it was not comprehensive enough to be useful to a wide variety of nonexpert users. We then became interested in the efforts of Seymour A. Papert, Wallace Feurzeig and others working at the Massachusetts Institute of Technology and at Bolt, Beranek and Newman, Inc., to develop a computer-based learning environment in which children would find learning both fun and rewarding. Working with a large time-shared computer, Papert and Feurzeig devised a simple but powerful computer language called LOGO. With this language children (ranging in age from eight to 12) could write programs to control a simple music generator, a robot turtle that could crawl around the floor and draw lines, and a television image of the turtle that could do the same things.

After observing this project we came to realize that many of the problems involved in the design of the personal computer, particularly those having to do with expressive communication, were brought strongly into focus when children down to the age of six were seriously considered as users. We also realized that children require more computer power than an adult is willing to settle for in a time-sharing system. The best outputs that time-sharing can provide are crude green-tinted line drawings and square-wave musical tones. Children, however, are used to finger paints, color television and stereophonic records, and they usually find the things that can be accomplished with a low-capacity time-sharing system insufficiently stimulating to maintain their interest.

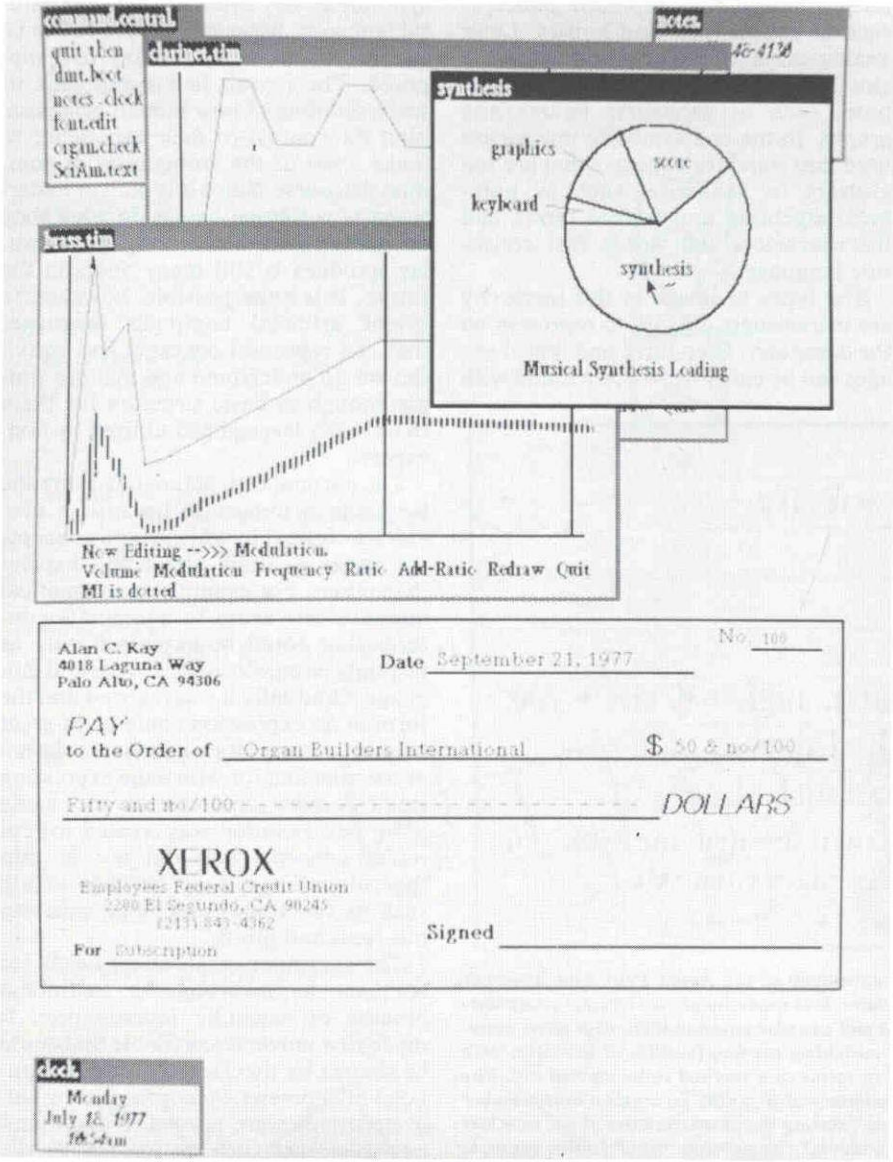
Since LOGO was not designed with all the people and uses we had in mind, we decided not to copy it but to devise a new kind of programming system that would attempt to combine simplicity and ease of access with a qualitative improvement in expert-level adult programming. In this effort we were guided, as we had been with the FLEX system, by the central ideas of the programming language SIMULA, which was developed in the mid-1960's by Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Center in Oslo.

Our experimental personal computer

EXPERIMENTAL PERSONAL COMPUTER was built at the Xerox Palo Alto Research Center in part to develop a high-level programming language that would enable nonexperts to write sophisticated programs. The author and his colleagues were also interested in using the experimental computer to study the effects of personal computing on learning. The machine is completely self-contained, consisting of a keyboard, a pointing device, a high-resolution picture display and a sound system, all connected to a small processing unit and a removable disk-file memory. Display can present thousands of characters approaching the quality of those in printed material.

is self-contained and fits comfortably into a desk. Long-term storage is provided by removable disk memories that can hold the equivalent of 1,500 printed pages of information (about three million characters). Although image displays in the 1980's will probably be flat-screened mosaics that reflect light as liquid-crystal watch displays do, visual output is best supplied today by a high-resolution black-and-white or color television picture tube. High-fidelity sound output is produced by a built-in conversion from discrete digital signals to continuous waveforms, which are then sent to a conventional audio amplifier and speakers. The user makes his primary input through a typewriterlike keyboard and a pointing device called a mouse, which controls the position of an arrow on the screen as it is pushed about on the table beside the display. Other input systems include an organlike keyboard for playing music, a pencillike pointer, a joystick, a microphone and a television camera.

The commonest activity on our personal computer is the manipulation of simulations already supplied by the SMALLTALK system or created by the user. The dynamic state of a simulation is shown on the display, and its general course is modified as the user changes the displayed images by typing commands on the keyboard or pointing with the mouse. For example, formatted textual documents with multiple typefaces are simulated so that an image of the finished document is shown on the screen. The document is edited by pointing at characters and paragraphs with the mouse and then deleting, adding and restructuring the document's parts. Each change is instantly reflected in the document's image.



Musical Synthesis Loading

New Editing -->>> Modulation.

MI is dotted

Alan C. Kay
4018 Laguna Way
Palo Alto, CA 94306

No. 100

Date September 21, 1977

PAY
to the Order of Organ Builders International $ 50 & no/100

Fifty and no/100 - DOLLARS

XEROX
Employees Federal Credit Union
2200 El Segundo, CA 90245
(213) 843-4362

For Subscription

Signed

Monday
July 18, 1977
10:54am

"WINDOWS," display frames within the larger display screen, enable the user to organize and edit information at several levels of refinement. Once the windows are created they overlap on the screen like sheets of paper. When a partially covered window is selected with the pointing device, the window is redisplayed to overlap the other windows. Images with various degrees of symbolic content can be displayed simultaneously. Such images include detailed halftone drawings, analogical images such as graphs and symbolic images such as numbers or words.

In many instances the display screen is too small to hold all the information a user may wish to consult at one time, and so we have developed "windows," or simulated display frames within the larger physical display. Windows organize simulations for editing and display, allowing a document composed of text, pictures, musical notation, dynamic animations and so on to be created and viewed at several levels of refinement. Once the windows have been created they overlap on the screen like sheets of paper; when the mouse is pointed at a partially covered window, the window is redisplayed to overlap the other windows. Those windows containing useful but not immediately needed information are collapsed to small rectangles that are labeled with a name showing what information they contain. A "touch" of the mouse causes them to instantly open up and display their contents.

In the present state of the art software development is much more difficult and time-consuming than hardware development. The personal computer will eventually be put together from more or less standard microelectronic components, but the software that will give life to the user's ideas must go through a long and arduous process of refinement if it is to aid and not hinder the goals of a personal dynamic medium.

For this reason we have over the past four years invited some 250 children (aged six to 15) and 50 adults to try versions of SMALLTALK and to suggest ways of improving it. Their creations, as imaginative and diverse as they themselves, include programs for home accounts, information storage and retrieval, teaching, drawing, painting, music synthesis, writing and games. Subsequent designs of SMALLTALK have been greatly influenced and improved by our visitors' projects.

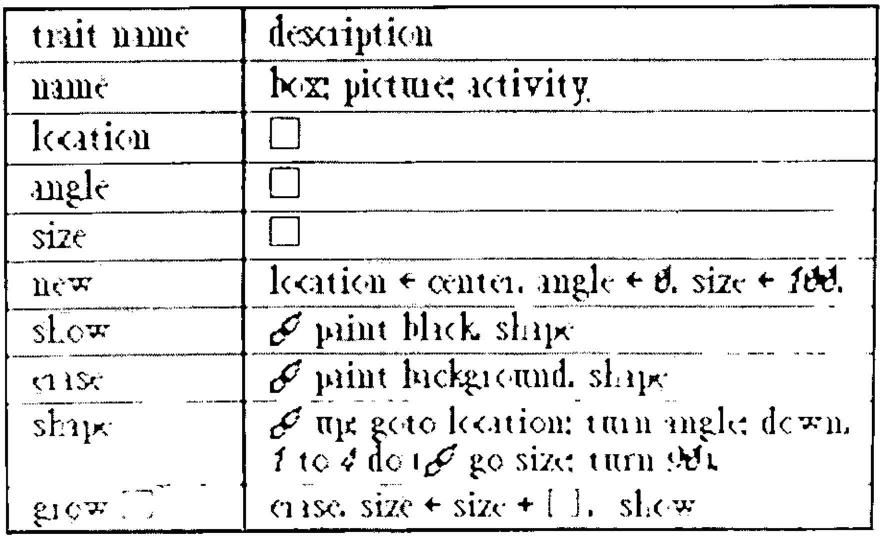
When children or adults first encounter a personal computer, most of them are already involved in pursuits of their own choosing. Their initial impulse is to exploit the system to do things they are already doing: a home or office manager will automate paperwork and accounts, a teacher will portray dynamic and pictorial aspects of a curriculum, a child will work on ways to create pictures and games. The fact is that people naturally start to conceive and build personal tools. Although man has been characterized as the toolmaking species, toolmaking itself has historically been the province of technological specialists. One reason is that technologies frequently require special techniques, materials, tools and physical conditions. An important property of computers, however, is that very general tools for using them can be built by anyone. These tools are made from the same materials and with the same effort as more specific creations.

Initially the children interact with our computer by "painting" pictures and drawing straight lines on the display screen with the pencillike pointer. The children then discover that programs can create structures more complex than any they can create by hand. They learn that a picture has several representations, of which only the most obvious—the image—appears on the screen. The most important representation is the editable symbolic model of the picture stored in the memory of the computer. For example, in the computer an image of a truck can be built up from models of wheels, a cab and a bed, each a different color. As the parts of the symbolic model are edited its image on the screen will change accordingly.

Adults also learn to exploit the properties of the computer medium. A professional artist who visited us spent several months building various tools that resembled those he had worked with to create images on paper. Eventually he discovered that the mosaic screen—the indelible but instantly erasable storage of the medium—and his new ability to program could be combined to create rich textures of a kind that could not be created with ink or paint. From the use of the computer for the impoverished simulation of an already existing medium he had progressed to the discovery of the computer's unique properties for human expression.

One of the best ways to teach nonexperts to communicate with computers is to have them explore the levels of abstraction at which images can be manipulated. The manipulation of images follows the general stages of intellectual growth. For a young child an image is something to make: a free mixture of forms and colors unconnected with the real world. Older children create images that directly represent concepts such as people, pets and houses. Later analogical images appear whose form is closely related to their meaning and purpose, such as geometric figures and graphs. In the end symbolic images are used that stand for concepts that are too abstract to analogize, such as numbers, algebraic and logical terms and the characters and words that constitute language.

The types of image in this hierarchy are increasingly difficult to represent on the computer. Free-form and literal images can be easily drawn or painted with lines and halftones in the dot matrix of the display screen with the aid of the mouse or in conjunction with programs that draw curves, fill in areas with tone or show perspectives of three-dimensional models. Analogical images can also be generated, such as a model of a simulated musical intrument: a time-sequenced graph representing the dynamic evolution of amplitude, pitch variation and tonal range.



trait name	description
name	box: picture activity.
location	□
angle	□
size	□
new	location ← center. angle ← 0. size ← 100.
show	🖌 paint black. shape
erase	🖌 paint background. shape
shape	🖌 up. goto location; turn angle; down. 1 to 4 do (🖌 go size; turn 90)
grow [ ]	erase. size ← size + [ ]. show

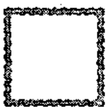
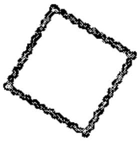
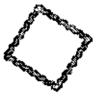
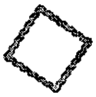
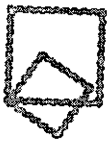
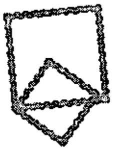
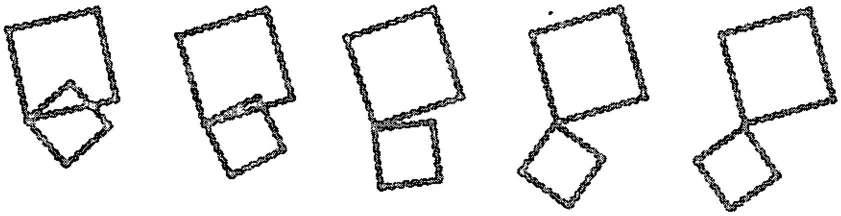
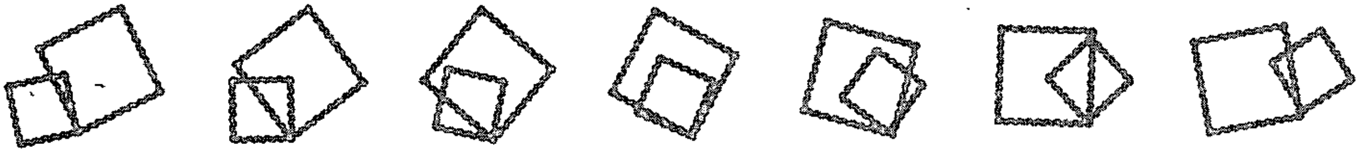
SMALLTALK is a new programming language developed at the Xerox Palo Alto Research Center for use on the experimental personal computer. It is made up of "activities," computerlike entities that can perform a specific set of tasks and can also communicate with other activities in the system. New activities are created by enriching existing families of activities with additional "traits," or abilities, which are defined in terms of a method to be carried out. The description of the family "box" shown here is a dictionary of its traits. To create a new member of the family box, a message is sent to the trait "new" stating the characteristics of the new box in terms of specific values for the general traits "location," "angle" and "size." In this example "new" has been filled in to specify a box located in the center of the screen with an angle of zero degrees and a side 100 screen dots long. To "show" the new box, a member of the curve-drawing family "brush" is given directions by the open trait "shape." First the brush travels to the specified location, turns in the proper direction and appears on the screen. Then it draws a square by traveling the distance given by "size," turning 90 degrees and repeating these actions three more times. The last trait on the list is open, indicating that a numerical value is to be supplied by the user when the trait is invoked by a message. A box is "grown" by first erasing it, increasing (or decreasing) its size by the value supplied in the message and redisplaying it.

Symbolic representations are particularly useful because they provide a means of handling concepts that are difficult to portray directly, such as generalizations and abstract relations. Moreover, as an image gets increasingly complex its most important property, the property of making local relations instantly clear, becomes less useful. Communication with computers based on symbols as they routinely occur in natural language, however, has proved to be far more difficult than many had supposed. The reason lies in our lack of understanding of how human beings exploit the context of their experience to make sense of the ambiguities of common discourse. Since it is not yet understood how human beings do what they do, getting computers to engage in similar activities is still many years in the future. It is quite possible, however, to invent artificial computer languages that can represent concepts and activities we do understand and that are simple enough in basic structure for them to be easily learned and utilized by nonexperts.

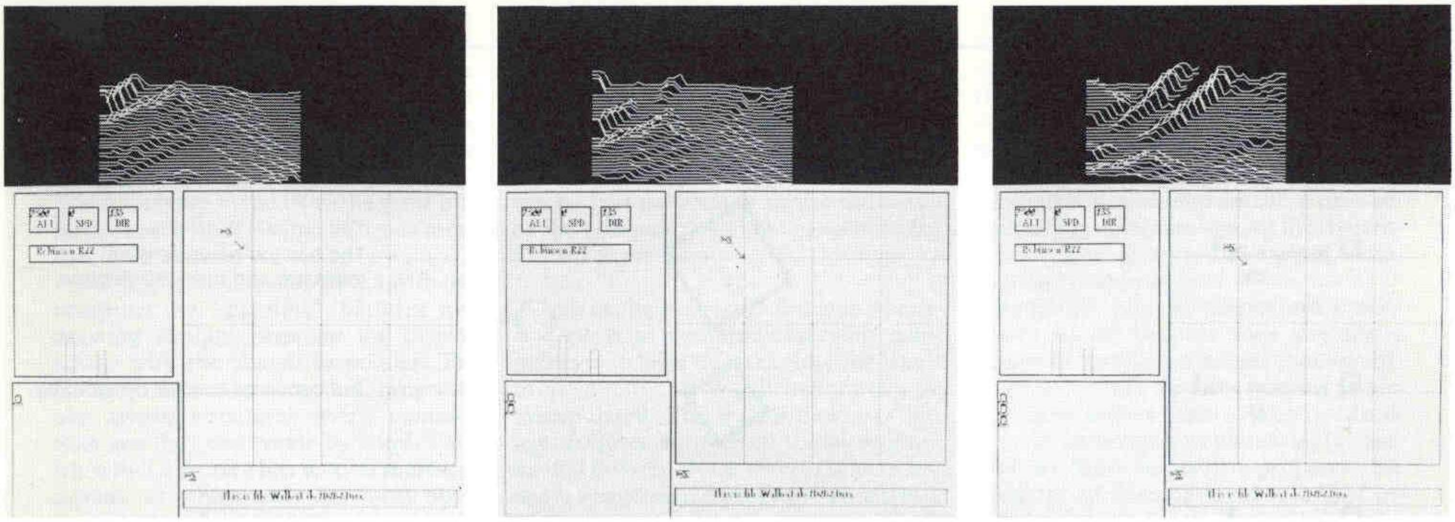
The particular structure of a symbolic language is important because it provides a context in which some concepts are easier to think about and express than others. For example, mathematical notation first arose to abbreviate concepts that could be expressed only as ungainly circumlocutions in natural language. Gradually it was realized that the form of an expression could be of great help in the conception and manipulation of the meaning for which the expression stood. A more important advance came when new notation was created to represent concepts that did not fit into the culture's linguistic heritage at all, such as functional mappings, continuous rates and limits.

The computer created new needs for language by inverting the traditional process of scientific investigation. It made new universes available that could be shaped by theories to produce simulated phenomena. Accordingly symbolic structures were needed to communicate concepts such as imperative descriptions and control structures.

Most of the programming languages in service today were developed as symbolic ways to deal with the hardware-level concepts of the 1950's. This approach led to two kinds of passive building blocks: data structures, or inert con-

Message Interaction	Pictorial Effect	Commentary
□ box new named "joe"! box:joe		An offspring of the family "box" is created and is named "joe."
□ joe turn 30! ok		The box joe receives the message and turns 30 degrees.
□ joe grow – 15! ok		Joe becomes smaller by 15 units.
□ joe erase! ok		Joe disappears from the screen.
□ joe show! ok		Joe reappears.
□ box new named "jill"! box:jill		A new box appears.
□ jill turn – 10! ok		Only jill turns. Joe and jill are independent activities.
□ 1 to 10! interval:1 2 3 4 5 6 7 8 9 10		An interval stands for a sequence of numbers.
□ forever! interval:1 2 3 4 5 6 7 8 9 10 11...		Forever is the infinite interval. It must be terminated by hitting an escape key.
□ 1 to 10 do (joe turn 20)! ok		Joe spins.
□ forever do (joe turn 11. jill turn – 13)! ok		A simple parallel movie of joe and jill spinning in opposite directions is created by combining forever with a turn request to both joe and jill.

SMALLTALK LEARNING SEQUENCE teaches students the basic concepts of the language by having them interact with an already defined family of activities. First, offspring of the family box are created, named and manipulated, and a second family of activities called "interval" is introduced. Offspring of the interval and box families are then combined to generate an animation of two spinning boxes.



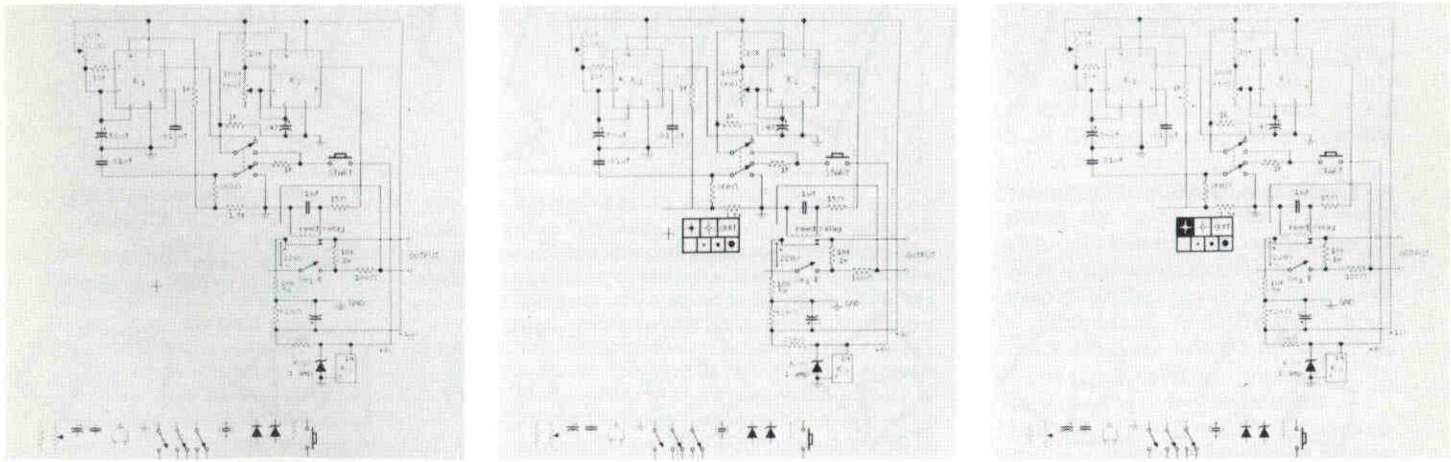
HELICOPTER SIMULATION was developed by a 15-year-old student. The user directs the helicopter where to go with the pointing device, which controls the position of the black arrow on the screen. The window at the top shows the changing topography of the terrain

struction materials, and procedures, or step-by-step recipes for manipulating data. The languages based on these concepts (such as BASIC, FORTRAN, ALGOL and APL) follow their descriptions in a strictly sequential manner. Because a piece of data may be changed by any procedure that can find it the programmer must be very careful to choose only those procedures that are appropriate. As ever more complex systems are attempted, requiring elaborate combinations of procedures, the difficulty of getting the entire system to work increases geometrically. Although most programmers are still taught data-procedure languages, there is now a widespread recognition of their inadequacy.

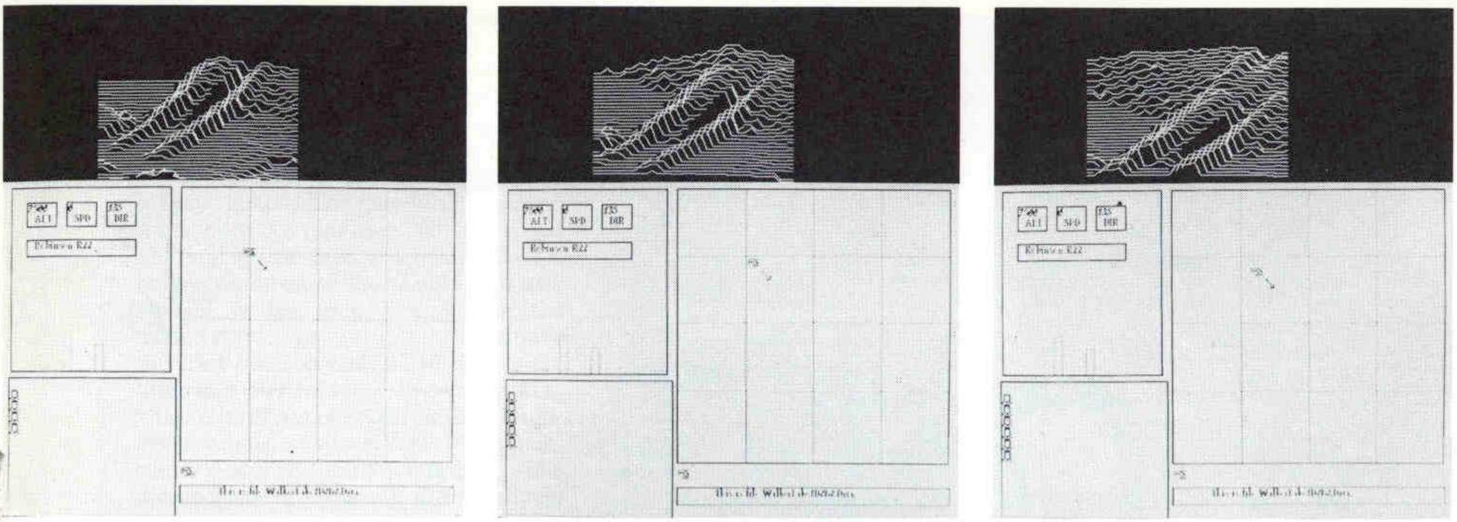
A more promising approach is to devise building blocks of greater generality. Both data and procedures can be replaced by the single idea of "activities," computerlike entities that exhibit behavior when they are sent an appropriate message. There are no nouns and verbs in such a language, only dynamically communicating activities. Every transaction, description and control process is thought of as sending messages to and receiving messages from activities in the system. Moreover, each activity belongs to a family of similar activities, all of which have the ability to recognize and reply to messages directed to them and to perform specific acts such as drawing pictures, making sounds or adding numbers. New families are created by combining and enriching "traits," or properties inherited from existing families.

A message-activity system is inherently parallel: every activity is constantly ready to send and receive messages, so that the host computer is in effect divided into thousands of computers, each with the capabilities of the whole. The message-activity approach therefore enables one to dynamically represent a system at many levels of organization from the atomic to the macroscopic, but with a "skin" of protection at each qualitative level of detail through which negotiative messages must be sent and checked. This level of complexity can be safely handled because the language severely limits the kinds of interactions between activities, allowing only those that are appropriate, much as a hormone is allowed to interact with only a few specifically responsive target cells. SMALLTALK, the programming system of our personal computer, was the first computer language to be based entirely on the structural concepts of messages and activities.

The third and newest framework for high-level communication is the observer language. Although message-activity languages are an advance over the data-procedure framework, the relations among the various activities are somewhat independent and analytic. Many



CIRCUIT-DRAWING PROGRAM that was developed by a 15-year-old boy enables a user to construct a complex circuit diagram by selecting components from a "menu" displayed at the bottom of the screen. The components are then positioned and connected with the



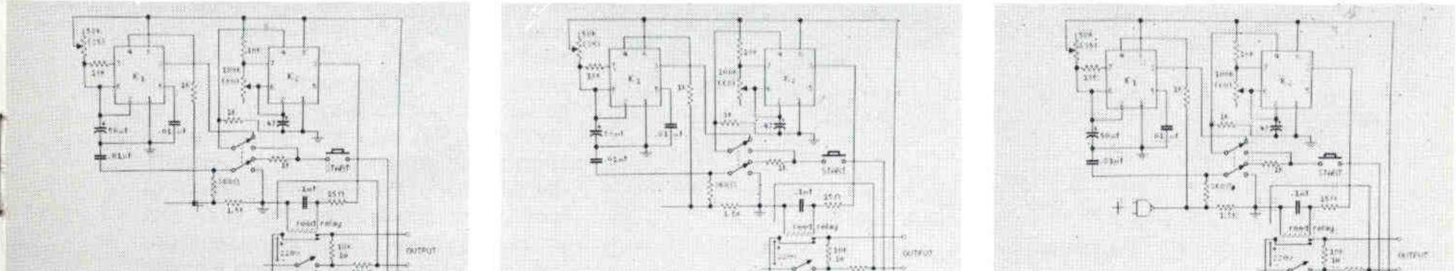
below as the helicopter flies over it. (Actual terrains were obtained from *Landsat* maps.) A third window keeps track of the helicopter's altitude, direction and speed. The variety of events that can be simulated at the same time demonstrates the power of parallel processing.

concepts, however, are so richly interwoven that analysis causes them virtually to disappear. For example, 20th-century physics assigns equal importance to a phenomenon and its context, since observers with different vantage points perceive the world differently. In an observer language, activities are replaced by "viewpoints" that become attached to one another to form correspondences between concepts. For example, a dog can be viewed abstractly (as an animal), analytically (as being composed of organs, cells and molecules), pragmatically (as a vehicle by a child), allegorically (as a human being in a fairy tale) and contextually (as a bone's way to fertilize a lawn). Observer languages are just now being formulated. They and their successors will be the communication vehicles of the 1980's.

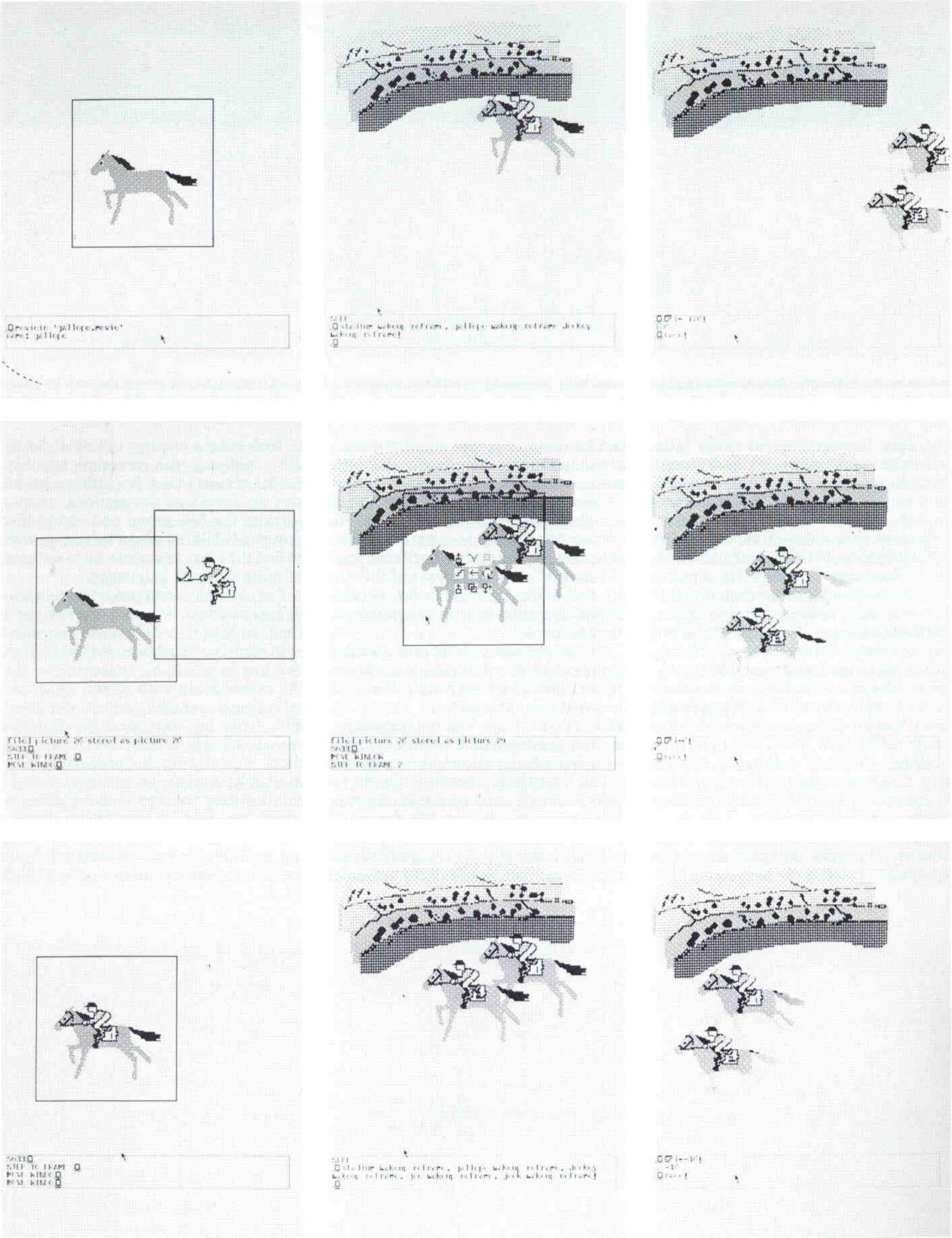
Our experience, and that of others who teach programming, is that a first computer language's particular style and its main concepts not only have a strong influence on what a new programmer can accomplish but also leave an impression about programming and computers that can last for years. The process of learning to program a computer can impose such a particular point of view that alternative ways of perceiving and solving problems can become extremely frustrating for new programmers to learn.

At the beginning of our study we first timidly considered simulating features of data-procedure languages that children had been able to learn, such as BASIC and LOGO. Then, worried that the imprinting process would prevent stronger ideas from being absorbed, we decided to find a way to present the message-activity ideas of SMALLTALK in concrete terms without dilution. We did so by starting with simple situations that embodied a concept and then gradually increasing the complexity of the examples to flesh out the concept to its full generality. Although the communicationlike model of SMALLTALK is a rather abstract way to represent descriptions, to our surprise the first group and succeeding groups of children who tried it appeared to find the ideas as easy to learn as those of more concrete languages.

For example, most programming languages can deal with only one thing at a time, so that it is difficult to represent with them even such simple situations as children in a school, spacecraft in the sky or bouncing balls in free space. In SMALLTALK parallel models are dealt with from the start, and the children seem to have little difficulty in handling them. Actually parallel processing is remarkably similar to the way people think. When you are walking along a street, one part of your brain may be thinking about the route you are taking, another part may be thinking about the dinner you are going to eat, a third



pointing device. An additional menu can be generated on the screen by pushing a button on the pointing device; this menu supplies solid and open dots and lines of various widths. In the sequence shown here two components are selected and added to a circuit diagram.



```
□movie in 'gallope.movie'
name: gallope
```

```
Still
□stallion wakeup reframe, gallope wakeup reframe jockey
wakeup reframe!
□
```

```
□▽i← 1[illegible]
[illegible]
□race!
```

```
filed picture 2[illegible] stored as picture 2[illegible]
5633□
STEP TO FRAME □
MOVE WINDO□
```

```
filed picture 2[illegible] stored as picture 29
5633□
MOVE WINDOW
STEP TO FRAME 2
```

```
□▽i←1
□race!
```

```
5633□
STEP TO FRAME □
MOVE WINDO□
MOVE WINDO□
```

```
Still
□stallion wakeup reframe, gallope wakeup reframe, jockey
wakeup reframe, joe wakeup reframe, jock wakeup reframe!
□
```

```
□▽i←-1[illegible]
-1[illegible]
□race!
```

HORSE-RACE ANIMATION shows the capabilities of the experimental personal computer for creating dynamic halftone images. The possible range of such simulations is limited only by the versatility of the programming language and the imagination of the child or adult user. In this sequence, images of horses, riders and background are called up independently from the storage files and arranged for the racing simulation with the pointing device. A single typed command then causes the two horses and riders to race each other across screen.

part may be admiring the sunset, and so forth.

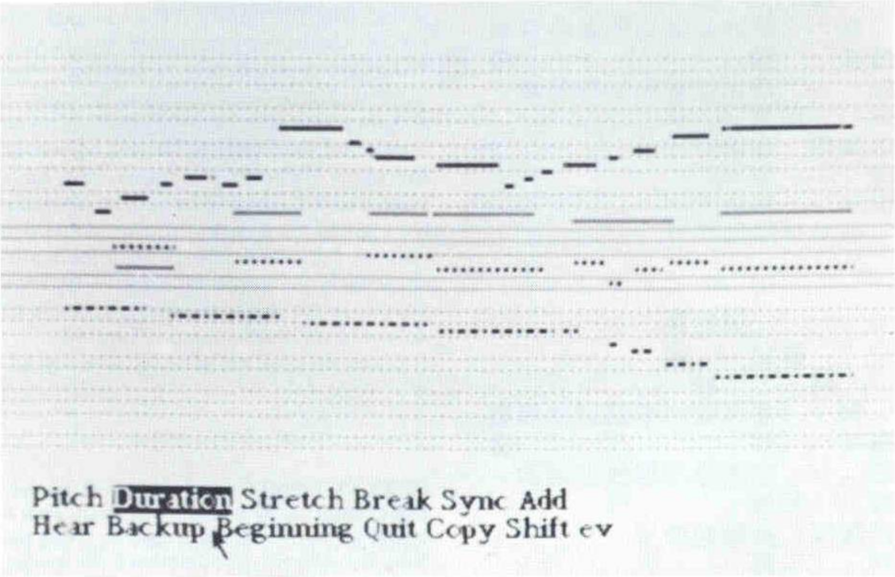
Another important characteristic of SMALLTALK is the classification of objects into families that are generalizations of their properties. Children readily see themselves as members of the family "kids," since they have common traits such as language, interests and physical appearance. Each individual is both a member of the family kids and has his or her own meaning for the shared traits. For example, all kids have the trait eye color, but Sam's eyes are blue and Bertha's are brown. SMALLTALK is built out of such families. Number symbols, such as 2 or 17, are instances of the family "number." The members of this family differ only in their numerical value (which is their sole property) and share a common definition of the different messages they can receive and send. The symbol of a "brush" in SMALLTALK is also a family. All the brush symbols have the ability to draw lines, but each symbol has its own knowledge of its orientation and where it is located in the drawing area.

The description of a programming language is generally given in terms of its grammar: the meaning each grammatical construction is supposed to convey and the method used to obtain the meaning. For example, various programming languages employ grammatical constructions such as (PLUS 3 4) or 3 ENTER 4 + to specify the intent to add the number 3 to the number 4. The meaning of these phrases is the same. In the computer each should give rise to the number 7, although the actual methods followed in obtaining the answer can differ considerably from one type of computer to the next.

The grammar of SMALLTALK is simple and fixed. Each phrase is a message to an activity. A description of the desired activity is followed by a message that selects a trait of the activity to be performed. The designated activity will decide whether it wants to accept the message (it usually does) and at some later time will act on the message. There may be many concurrent messages pending for an activity, even for the same trait. The sender of the message may decide to wait for a reply or not to wait. Usually it waits, but it may decide to go about other business if the message has invoked a method that requires considerable computation.

The integration of programming-language concepts with concepts of editing, graphics and information retrieval makes available a wide range of useful activities that the user can invoke with little or no knowledge of programming. Learners are introduced to SMALLTALK by getting them to send messages to already existing families of activities, such

MUSIC CAN BE REPRESENTED on the personal computer in the form of analogical images. Notes played on the keyboard are "captured" as a time-sequenced score on the display.


Pitch Duration Stretch Break Sync Add
Hear Backup Beginning Quit Copy Shift ev

MUSICAL SCORE shown here was generated as music was played on the keyboard. The simplified notation represents pitch by vertical placement and duration by horizontal length. Notes can be shortened, lengthened or changed and the modified piece then played back as music.

as the family "box," whose members show themselves on the screen as squares. A box can individually change its size, location, rotation and shape. After some experience with sending messages to cause effects on the display screen the learner may take a look at the definition of the box family. Each family in SMALLTALK is described with a dictionary of traits, which are defined in terms of a method to be carried out. For example, the message phrase "joe grow 50" says: Find the activity named "joe," find its general trait called "grow ___" and fill in its open part with the specific value 50. A new trait analogous to those already present in the family definition (such as "grow" or "turn") can easily be added by the learner. The next phase of learning involves elaboration of this basic theme by creating games such as space war and tools for drawing and painting.

There are two basic approaches to personal computing. The first one, which is analogous to musical improvisation, is exploratory: effects are caused in order to see what they are like and errors are tracked down, understood and fixed. The second, which resembles musical composition, calls for a great deal more planning, generality and structure. The same language is used for both methods but the framework is quite different.

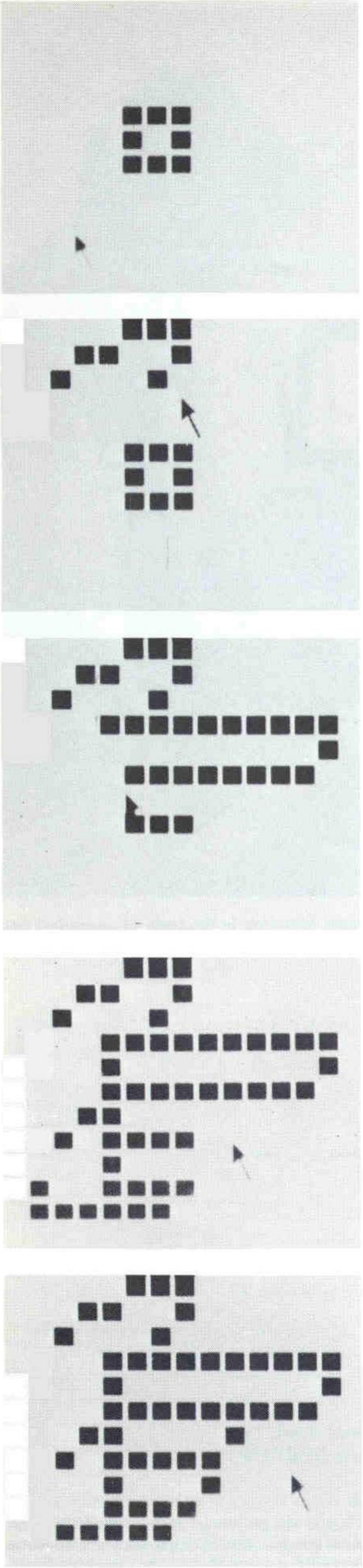
From our study we have learned the importance of a balance between free exploration and a developed curriculum. The personal computing experience is similar to the introduction of a piano into a third-grade classroom. The children will make noise and even music by experimentation, but eventually they will need help in dealing with the instrument in nonobvious ways. We have also found that for children the various levels of abstraction supplied by SMALLTALK are not equally accessible. The central idea of symbolization is to give a simple name to a complex collection of ideas, and then later to be able to invoke the ideas through the name. We have observed a number of children between the ages of six and seven who have been able to take this step in their computer programs, but their ability to look ahead, to visualize the consequences of actions they might take, is limited.

Children aged eight to 10 have a gradually developing ability to visualize and plan and are able to use the concept of families and a subtler form of naming: the use of traits such as size, which can stand for different numerical values at different times. For most children, however, the real implications of further symbolic generality are not at all obvious. By age 11 or 12 we see a considerable improvement in a child's ability to plan general structures and to devise comprehensive computer tools. Adults advance through the stages more quickly than children, and usually they create tools after a few weeks of practice. It is not known whether the stages of intellectual development observed in children are absolutely or only relatively correlated with age, but it is possible that exposure to a realm in which symbolic creation is rewarded by wonderful effects could shorten the time required for children to mature from one stage to the next.

The most important limitation on personal computing for nonexperts appears when they conceive of a project that, although it is easy to do in the language, calls for design concepts they have not yet absorbed. For example, it is easy to build a span with bricks if one knows the concept of the arch, but otherwise it is difficult or impossible. Clearly as complexity increases "architecture" dominates "material." The need for ways to characterize and communicate architectural concepts in developing programs has been a long-standing problem in the design of computing systems. A programming language provides a context for developing strategies, and it must supply both the ability to make tools and a style suggesting useful approaches that will bring concepts to life.

We are sure from our experience that personal computers will become an integral part of peoples' lives in the 1980's. The editing, saving and sifting of all manner of information will be of value to virtually everyone. More sophisticated forms of computing may be like music in that most people will come to know of them and enjoy them but only a few will actually become directly involved.

How will personal computers affect society? The interaction of society and a new medium of communication and self-expression can be disturbing even when most of the society's members learn to use the medium routinely. The social and personal effects of the new medium are subtle and not easy for the society and the individual to perceive. To use writing as a metaphor, there are three reactions to the introduction of a new medium: illiteracy, literacy and artistic creation. After reading material became available the illiterate were those who were left behind by the



DISPLAY FONTS can be designed on personal computer by constructing them from a matrix of black-and-white squares. When the fonts are reduced, they approach the quality of those in printed material. The image of a pointing hand shown here is a symbol in SMALLTALK representing the concept of a literal word, such as the name associated with an activity.

new medium. It was inevitable that a few creative individuals would use the written word to express inner thoughts and ideas. The most profound changes were brought about in the literate. They did not necessarily become better people or better members of society, but they came to view the world in a way quite different from the way they had viewed it before, with consequences that were difficult to predict or control.

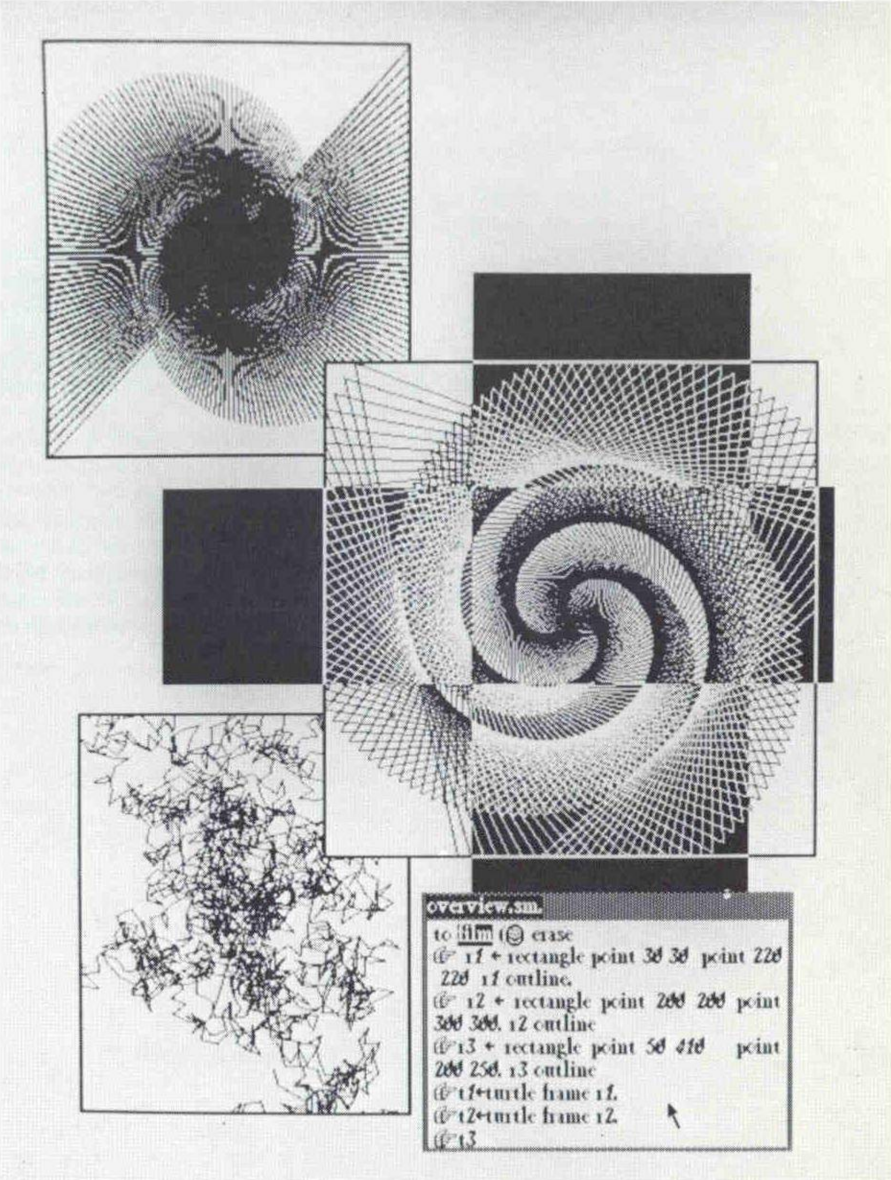
We may expect that the changes resulting from computer literacy will be as far-reaching as those that came from literacy in reading and writing, but for most people the changes will be subtle and not necessarily in the direction of their idealized expectations. For example, we should not predict or expect that the personal computer will foster a new revolution in education just because it could. Every new communication medium of this century—the telephone, the motion picture, radio and television—has elicited similar predictions that did not come to pass. Millions of uneducated people in the world have ready access to the accumulated culture of the centuries in public libraries, but they do not avail themselves of it. Once an individual or a society decides that education is essential, however, the book, and now the personal computer, can be among the society's main vehicles for the transmission of knowledge.

The social impact of simulation—the central property of computing—must also be considered. First, as with language, the computer user has a strong motivation to emphasize the similarity between simulation and experience and to ignore the great distances that symbols interpose between models and the real world. Feelings of power and a narcissistic fascination with the image of oneself reflected back from the machine are common. Additional tendencies are to employ the computer trivially (simulating what paper, paints and file cabinets can do), as a crutch (using the computer to remember things that we can perfectly well remember ourselves) or as an excuse (blaming the computer for human failings). More serious is the human propensity to place faith in and assign higher powers to an agency that is not completely understood. The fact that many organizations actually base their decisions on—worse, take their decisions from—computer models is profoundly disturbing given the current state of the computer art. Similar feelings about the written word persist to this day: if something is "in black and white," it must somehow be true.

Children who have not yet lost much of their sense of wonder and fun have helped us to find an ethic about computing: Do not automate the work you are engaged in, only the materials. If you like to draw, do not automate drawing; rather, program your personal computer to give you a new set of paints. If you like to play music, do not build a "player piano"; instead program yourself a new kind of instrument.

A popular misconception about computers is that they are logical. Forthright is a better term. Since computers can contain arbitrary descriptions, any conceivable collection of rules, consistent or not, can be carried out. Moreover, computers' use of symbols, like the use of symbols in language and mathematics, is sufficiently disconnected from the real world to enable them to create splendid nonsense. Although the hardware of the computer is subject to natural laws (electrons can move through the circuits only in certain physically defined ways), the range of simulations the computer can perform is bounded only by the limits of human imagination. In a computer, spacecraft can be made to travel faster than the speed of light, time to travel in reverse.

It may seem almost sinful to discuss the simulation of nonsense, but only if we want to believe that what we know is correct and complete. History has not been kind to those who subscribe to this view. It is just this realm of apparent nonsense that must be kept open for the developing minds of the future. Although the personal computer can be guided in any direction we choose, the real sin would be to make it act like a machine!



```
to film (◉ erase
☞ r1 ← rectangle point 30 30 point 220
 220 r1 outline.
☞ r2 ← rectangle point 200 200 point
300 300. r2 outline
☞ r3 ← rectangle point 50 410   point
200 250. r3 outline
☞ t1←turtle frame r1.
☞ t2←turtle frame r2.
☞ t3
```

INTRICATE PATTERNS can be generated on the personal computer with very compact descriptions in SMALLTALK. They are made by repeating, rotating, scaling, superposing and combining drawings of simple geometric shapes. Students who are learning to program first create interesting free-form or literal images by drawing them directly in the dot matrix of the display screen. Eventually they learn to employ the symbolic images in the programming language to direct the computer to generate more complex imagery than they could easily create by hand.

The Author

ALAN C. KAY is a principal scientist and head of the Learning Research Group at the Xerox Palo Alto Research Center. He received his B.A. in mathematics from the University of Colorado at Boulder and, after a short career as a professional jazz guitarist, studied computer science at the University of Utah, obtaining his Ph.D. in 1969. He then became a research associate and lecturer at the Stanford University Artificial Intelligence Project. He moved to Xerox in 1971. "I have always been equally attracted to the arts and the sciences," he writes. "Eventually I discovered that the world of computers provides a satisfying environment for my blend of interests."

Bibliography

TOWARDS A THEORY OF INSTRUCTION. Jerome S. Bruner. Belknap Press of Harvard University Press, 1966.

ARTIFICIAL INTELLIGENCE. Seymour A. Papert and Marvin Minsky. Condon Lectures, Oregon State System of Higher Education, 1974.

PERSONAL DYNAMIC MEDIA. Alan C. Kay and Adele Goldberg in *Computer,* Vol. 10, No. 3, pages 31–41; March, 1977.

INFORMAL.DC smlg.fd smdeleg.fd sroman.fd

An Informal Introduction to SMALLTALK

by

Alan C. Kay
Xerox Palo Alto Research Center

The easiest way to learn SMALLTALK is to just make it do useful things for you !

Let's get SMALLTALK to draw a SQUARE for us. First we have to tell SMALLTALK just what it is that we mean by "square".

```
to square
   ¦forward 100
   ¦right 90
   ¦forward 100
   ¦right 90
   ¦forward 100
   ¦right 90
   ¦forward 100
```

"To" is part of SMALLTALK. We sent it a message consisting of a name,"square", and a definition in terms of drawing commands inside of "margin parentheses".

We can now use our definition just as though it had been part of SMALLTALK.

```
square
```

and a square is drawn. Try it again.

```
square
```

Whoops! We just created some bugs! First, the "pen" was not left pointing the same way as it was found, and also we forgot to clear the screen and return the pen to the center.

```
erase
```

clears the screen.

```
home
```

centers the "pen" pointing up.

```
white
```

draws using white ink on a black background. Try

```
white home erase forward 50
```

```
black
```

draws using black ink on a white background.

A "cleaner" version of "square" is

```
to square
   ¦forward 100
   ¦right 90
   ¦forward 100
   ¦right 90
   ¦forward 100
   ¦right 90
   ¦forward 100
   ¦right 90
```

What is there about this sequence of actions which has to do with "squareness"? All the turns are 90 degrees, and they alternate with forward travel of the same distance. So the following definition should also work.

```
to square
   ¦repeat 4
   ¦  ¦forward 100
   ¦  ¦right 90
```

Try it.

"Repeat" is sent a message consisting of two parts. The first is how many "repeats" are desired, the second is just what to repeat.

What about a square of any size? What is there about the previous definitions that only has to do with size as opposed to "squareness"?

It seems only to be the distance traveled (which is the message to the "forward" command).

Just as we can send messages to "forward", "right" and "repeat" to give them additional information about our desires, we can send messages to our own definitions as well. We would like to send "square" a message which says what length of side we want each time.

such as   square 100   or   square 50

Any definition can receive a message by saying ":". Since the message is different each time, it would be nice to give it a name to allow it to be used anywhere in the definition.

A definition to draw a square of any size is

```
to square
   ¦:size
   ¦repeat 4
   ¦  ¦forward size
   ¦  ¦right 90
```

Try it and see. The ":" picks up the message and calls it "size". "forward" refers to the message by its name "size".

Now let's try a TRIANGLE of any size. Well, it's really almost the same as a square, isn't it?

```
to triangle
   ¦:size
   ¦repeat 3
```

```
    ¦   ¦forward size
    ¦   ¦right 120
```

Try it.

The two definitions are almost the same except for the number of "repeats" and the angle. Is it possible to define actions which will draw

ANY POLYGON?

Well, we could certainly send the definition a message of two parts. One for the size, the other for the number of sides we want.

```
to poly
    ¦:sides :size
    ¦Repeat sides
    ¦   ¦forward size
    ¦   ¦angle ****
```

This looks reasonable except for confusion about the angle. "Repeat" will be sent a message for the correct number of sides and "forward" will get the right message about side length as before.

Now, what about the angle? When we turned right for the triangle it was 120, for the square 90. What about a pentagon? 72?

One neat way to look at the situation is that a complete trip for any polygon will get you back EXACTLY where you started and the heading of the pen WILL HAVE TURNED THROUGH 360 degrees EXACTLY.

The number of turns taken is the same as the number of sides (because the "repeat" controls this). So, it seems as though the angle taken should be 360/sides. Try it.

```
to poly
    ¦:sides :size
    ¦Repeat sides
    ¦   ¦forward size
    ¦   ¦right 360/sides
```

Try a few to see.

```
poly 5 50

poly 5 100

poly 18 20

poly 50 5

poly 360 1
```

Hmmmm. Does this make sense for a

CIRCLE?

```
to circle
    ¦poly 360 1
```

It's nice that we can use any of our definitions exactly like SMALLTALK's own commands. Now suppose we want circles of DIFFERENT size. What is there about "poly 360 1" that is "circlelike" and what has to do with size?

We know that "poly 3 ***" doesn't look like a circle and "poly 360 1" does.
What about "poly 360 10" ?

Try it.

So how does this strike you?

```
To circle
    ¦:size
    ¦poly 360 size
```

Now you may say, "OK, we can change the size of a circle alright, but the number we are sending as a message doesn't seem to bear any relationship to the diameter or radius". True? Well, what do we know about the circle? What is its circumference?

Well, it seems that poly "repeats" 360 times. Each of those times "forward" goes forward a distance. So the circumference of any polygon is sides * size. A relation between the radius of a circle and its circumference is: Circumference = 2 * pi * radius.

```
 So,  sides * size = 2 * pi * radius
and,          size =(2 * pi * radius)/sides
```

Let's now define a circle routine where the message we send it is the radius.

```
to pi
    ¦3.14159

to circle
    ¦:radius
    ¦poly 360 (2 * pi * radius)/360
```

Try it and see.

By now you are probably getting the idea that getting SMALLTALK to do things is easy. True.

Now what happens if we jiggle some of the things we are doing a little?

Seymour Papert's kids call the following kinds of things

SQUIRALS!

```
to squiral
    ¦:size :angle
    ¦forward size
    ¦right angle
    ¦squiral size+10 angle+2
```

Notice that this definition goes on forever so the "whoops" key needs to be used!

******More on this in a bit. It can be found in Seymour's stuff.

An interesting variation on "poly" is a definition that draws

NESTED POLYS!

```
To star
    |:sides :size
    |If size > 4
    |  then |Repeat sides
    |       |   |forward size
    |       |   |right 360/sides
    |       |   |star size/3
```

Try a few of these.

Now, are you already for a DRAGON? This is a very simple definition whose actions are hard to predict.

```
To dragon
    |:length
    |If length = 0 then |forward 10
    |
    |              else |If length > 0
    |                   |    then | dragon length-1
    |                   |         | right 90
    |                   |         | dragon -length-1
    |                   |
    |                   |    else |dragon -length+1
    |                   |         |right -90
    |                   |         |dragon length+1
```

A more compact way to say this is

Hwy 3

```
To dragon
    |:length = 0 ⇒ |forward 10
    |
    | length > 0 ⇒ |dragon length-1. right 90. dragon -length-1
    |
    | dragon -length+1. right -90. dragon length+1
```

Experiments with

ACCELLERATED MOTION

```
To rollick
    |:times •figure
    |setup
    |Repeat times
    |    |penup
    |    |forward •dist ← dist + inc
    |    |right (•angle ← angle + ainc) + •turn ← turn + inc
    |    |pendn
    |    |(figure) size
```

SETUP gives values to
DIST
TURN
INC
SIZE
AINC
ANGLE

Try rollick 100 square

SHOOTING elastic objects into the air.

```
To shoot
    ¦:xacc :yacc
    ¦.yy ← yacc

    ¦Repeat lots
    ¦   ¦pendown. object. penup.
    ¦   ¦
    ¦   ¦forward .yy ← yy - gravity
    ¦   ¦right 90. forward xacc. right 270
    ¦   ¦
    ¦   ¦yacc = -yy ⇒ ¦(closeto 0 .yy ← .yacc ← yacc * elastic) ⇒ ¦Done
```

lots
object
gravity

if object is the figure on previous page then is (object) nec?
this must be from what can paint
figure –

A simple SPACESHIP!

```
To drawship
    ¦pendown
    ¦right 180. forward 5. right 315. forward 7. right 225. forward 20.
    ¦right 315. forward 7. right 270. forward 7.
    ¦right 315. forward 20. right 225. forward 7.
    ¦right 315. forward 5.
```

That was tedious, wasn't it? Later we will discover that we can just paint, draw or sketch any figure to be animated ourselves without having to make SMALLTALK draw them.

```
To moveship
    ¦:point :thrust
    ¦.turn ← .speed ← 0

    ¦Repeat forever
    ¦   ¦penup
    ¦   ¦forward .speed ← speed + thrust
    ¦   ¦right .turn ← turn + point
    ¦   ¦
    ¦   ¦drawship
```

Try

moveship 2 1

moveship 1 2

Use the WHOOPS button to kill a version.

Now for the big time! Try

moveship mouseX mouseY

and grab the mouse quickly!!

SPACEWAR!!

```
To Spacevehicle
    ¦
    ¦ :shape  ₀at  :posx :posy :heading
    ¦         ₀speed :speed
    ¦         ₀controls  :thrust  :point  :trigger.
    ¦
    ¦Repeat
    ¦  ¦Left Roll ←  Roll + point.
    ¦  ¦Forward Speed ← Speed + thrust.
    ¦  ¦
    ¦  ¦If thrust > 0 then (Show shape ⁎exhaust : Flame).
    ¦  ¦If thrust < 0 then (Show shape ⁎nose : Flame)
    ¦  ¦                else (Show shape).
    ¦  ¦
    ¦  ¦If  trigger on and Numberoftorps > 0
    ¦  ¦  then ¦Numberoftorps ← Numberoftorps - 1 .
    ¦  ¦       ¦create
    ¦  ¦       ¦  Spacevehicle ⁎torpedo
    ¦  ¦       ¦  at posx posy direction
    ¦  ¦       ¦  speed speed
    ¦  ¦       ¦  controls 25 0 ⁎off
    ¦  ¦       .
    ¦  ¦
    ¦  ¦If touching something
    ¦  ¦  then (Quit something. Show Crash. Quit self).
```

This set of actions defines both what a spaceship and a torpedo do in a somewhat sneaky way. A torpedo is a spaceship with a different shape, constant thrust, straight direction, and no ability to fire torpedos of its own.

The pictures "Ship" and "Torpedo" both have a subpart called "exhaust". This acts as a "hole" where other pictures can be placed, such as "Flame" when the thrust is on. A special subpart name, "center", defines the axis of rotation for "left" and "right" turns.

"Crash" in a more elaborate example would probably be a set of actions to produce ever more grandiose effects.

This particular game starts a ship out with 20 torpedos with no provision for more when all are fired.

"Speed" and "Roll" are names for the accumulated velocity of forward travel and turning. So the "thrust" and "point" controls are accellerations as in a real spaceship.

The "behaviour" at the bottom signals the actions to be done. The message received is what "shape" to use, what initial "position" and "direction" to assume (these names are the ones that are updated by "Forward" and "Left"), and where the information for "thrust", "attitude", and firing of torpedos is to be supplied. For spaceships it will be the joystick of each player, for torpedos, it will be constant information.

The actions are "Repeat"ed over and over.

They are to update the "Roll" and "Speed" accumulations,
to reposition the ship, which will update "position" and "direction",
to display the shape of this object (with "Flame" if thrust is "on"),
to send off a "Torpedo" if the "trigger" is "on" and the "Number₀of₀torps" left is greater than zero.

Then a check is made for a "touch" and, if so, the object touched is destroyed ("Quit"), the great "Crash" is "Show"n, and finally our object destroys itself.

As many spaceships as required may be instantiated by using "create".

```
create
Spacevehicle •Ship at random random random
                   speed random
                   controls joy 1 up joy 1 side joy 1 but.
create
Spacevehicle •Ship at random random random
                   speed random
                   controls joy 2 up joy 2 side joy 2 but."
```

New "Data" Objects and their "functions"

The ease with which an external form can be associated with an internal meaning in SMALLTALK means that many objects which are "cast in stone" in other languages can be defined and modified easily by anyone. Suppose only the Word and List operations are found in the language, then Numbers can be described in terms very similar to that of "schoolchild" arithmetic as shown below.

There are many ways to accomplish arithmetic; the example deliberately mimics the use of a "plus table" for single digits, the carry rule, and special cases involving 0, which you already know from school.

```
•PlusTable ←   •¦( 0  1  2  3  4  5  6  7  8  9 )
                ¦( 1  2  3  4  5  6  7  8  9 10 )
                ¦( 2  3  4  5  6  7  8  9 10 11 )
                ¦( 3  4  5  6  7  8  9 10 11 12 )
                ¦( 4  5  6  7  8  9 10 11 12 13 )
                ¦( 5  6  7  8  9 10 11 12 13 14 )
                ¦( 6  7  8  9 10 11 12 13 14 15 )
                ¦( 7  8  9 10 11 12 13 14 15 16 )
                ¦( 8  9 10 11 12 13 14 15 16 17 )
                ¦( 9 10 11 12 13 14 15 16 17 18 )
                .
```

```
To Number
    ¦
    ¦◦← ⇒ ¦:A. ↑ self
    ¦
    ¦"A new "Number" is created and declared by saying (for instance) •x ←
    ¦Number ← 12345. The "•" EVALs its third argument, which calls "Number"
    ¦which creates an instance, which looks for a "←", finds it, EVALs its
    ¦next argument (which is a "Word" 12345), binds it to "A", and RETURNs
    ¦the instance."
    ¦
    ¦◦value ⇒ ¦↑ A
    ¦
    ¦"The Word which is the value of "self" is RETURNed".
    ¦
    ¦◦first ⇒ ¦↑ A•first.
    ¦
    ¦" "first" of a "Number" is the same as "first" of the "Word" which is
    ¦its value. The other "Word Parts" are done in a similar manner."
    ¦
    ¦◦+ ⇒ ¦:B. ↑ ¦A•length = 1 and B•length = 1 ⇒ ¦PlusTable A B
    ¦     ¦       ¦
    ¦     ¦       ¦A•empty or B•empty ⇒ ¦A jointo B
    ¦     ¦       ¦
    ¦     ¦       ¦(A•butlast + B•butlast + carry A B )
```

```
  ¦        ¦          ¦        jointo ( A⋆last + B⋆last )⋆last
```

"This is a recursive definition which uses several cases to accomplish "+".

The first (A and B are both single digits) uses the childrens' addition table selected by each of the numbers in turn to isolate the sum which is RETURNed.

The next case terminates the routine in the case where either or both of A, B are EMPTY. Remember that anything "jointo" EMPTY is that thing. The last case is simply a statement of the goal, namely: the front digits of A and B are added to the carry found by adding the last digits of A and B, the result is joined to the single digit result of the sum of the last digits of A and B.

More branches of the conditional would be added to handle the Addition of negative numbers, etc."

```
ₒ- ⇒ ¦:B. ↑ "Subtraction is handled in a manner analogous
     ¦     to Addition".

ₒ= ⇒ ¦:B. ↑ A = B⋆value.
```

"A "Word operation" that is legal can easily be done."

```
ₒ< ⇒ ¦:B. ↑ if (B - A)⋆first = ⋆- then EMPTY else self.
```

"Doing the definition this way allows x<y<z etc. to work properly."

N O T E ! This def of Complex is not completely edited !!

```
To Complex
    ¦ₒ+ ⇒¦:value⋆complex ⇒¦↑ Complex ←
    ¦    ¦                ¦       re + value⋆re
    ¦    ¦                ¦       im + value⋆im
    ¦    ¦
    ¦    ¦value⋆fraction ⇒(↑ Complex ← re + value im)
    ¦    ¦↑ value Ɵ + self
    ¦
    ¦ₒ- ⇒¦:value⋆complex ⇒¦↑ Complex ←
    ¦    ¦                ¦       re - value⋆re
    ¦    ¦                ¦       im-value⋆im
    ¦      value⋆fraction  ⇒↑ Complex ← re - value    im
    ¦      otherwise      ⇒(↑  value Ɵ - self
    ¦
    ¦ ₒ* ⇒ :value⋆complex ⇒ ↑ Complex ← (re * value⋆re  -  ¦im * value⋆im)
    ¦                                   (im * value⋆re  +  ¦re * value⋆im)
    ¦      value⋆fraction  ⇒ ↑ Complex ← re * value  im * ¦value
    ¦      otherwise       ⇒ ↑  value Ɵ * self
    ¦
    ¦ ₒƟ ₒ+ ⇒ :value⋆fraction  ⇒ ↑ Complex ← re + value   im
    ¦ ₒƟ ₒ- ⇒ :value⋆fraction  ⇒ ↑ Complex ← value - re  ¦-im
    ¦ ₒƟ ₒ* ⇒ :value⋆fraction  ⇒ ↑ Complex ← re * value   im ¦* value
    ¦ ₒƟ ◊value  ⇒ ↑ Error"I don't know this operator" value .
    ¦
    ¦
    ¦ ₒre ⇒ ₒ← :value⋆fraction  ⇒ re ← value . ↑ self
    ¦         ₒₒ     ⇒ ↑ re
    ¦ ₒim ⇒ ₒ← :value⋆fraction  ⇒im ← value . ↑ self
    ¦         ₒₒ     ⇒ ↑ im
    ¦ ₒ←   ⇒ :re⋆fraction  ⇒:im⋆fraction  ⇒ ↑ self
```

```
┆ ₒcomplex  ⇒↑ true
┆ ◇op :value  ⇒ ↑ value Θ :op  self
┆ ◇op :value  ⇒ ↑ value Θ :op  self
      ₒₒ  ⇒ ↑ self
```

Necessary Information about this paper.
Latest revision: June 6, 1973

(The permanent names of this file are
SMALLTALK.DC. ***
SMALLTALK1.DC.
SMALLTALK2.DC.
Its latest incarnation will always be found on the
Learning Research Group Demo Diskpack.

The full structured index is found with each version.
Look under the structure to discover what file to load.

This file should be displayed using font SROMAN.FD.
To print, edit with SMDELEG.FD and Write Translated,
then print on XGP using font SMDELEG.XG)

SMALLTALK, a Model Building Language With Intensional Semantics

by
Alan C. Kay

Learning Research Group
Xerox Palo Alto Research Center

Abstract

SMALLTALK is a language which allows children (and adults) to build semantic models of their ideas in simple uncomplicated ways, and dynamically simulate them with respect to arbitrary environments.

Simplicity is achieved by having
a. only one kind of object in the language (a process) which can act like all other known computer objects,
b. a single uniform scheme for interobject communication, and,
c. an intensional semantics in which the meaning of an object is a part of the class to which an object belongs rather than dispersed through the system as part of more conventional extensional operations.

Benefits are the abilities to create new "functional", "data", "control", etc., entities without the usual problems associated with updating and coercion of generic functions.

****

Acknowledgements

The main influences on the content of this paper were the coprocess and data/function equivalences of FLEX{ka-68,69}, Flex's influence SIMULA {Dahl, et.al.}, LISP {McC,et.al.}, a number of control ideas of Dave Fisher{fi-70}, goals as expressions found in Carl Hewitt's PLANNER{he-70}, and the simplicity and ease of use of LOGO{pa,et.al-67,...73}.

Dan Ingalls of LRG in PARC, the implementer of SMALLTALK, has revealed many design flaws through his several excellent quick "throw away" implementations of the language. SMALLTALK could not have existed without his help and good cheer.

Introduction

SMALLTALK is built from a few simple, yet powerful, ideas.

First, SMALLTALK considers every OBJECT in its world to be an independant entity with local state and control. All distinction between "datalike" and "procedurelike" objects, such as exist in other programming languages, is thus removed. This includes "data", such as numbers, strings, arrays, lists, structures, etc.; "functions", such as 'factorial', 'plus', 'print', etc.; "control structures", such as conditional branches, repeats, recursion, and so on; "IO devices", such as 'files', 'the user', 'display and keyboard', etc.; all are treated alike because they ARE alike.

Next, all objects are composed of PARTS, even if they only contain themselves. The object can be thought of as a dynamic dictionary which contains all the relations and rules in which it can take part.

Third, objects can send and receive MESSAGEs to/from other objects. This may cause new objects to be created, altered, or even destroyed.

(Since there are no "special" objects, there is only one message protocol.)

Finally, each object is considered to be a member (or INSTANCE) of a CLASS, which is another object that contains the rules of behavior shared by all the members. Since each class has a class defining object, they are members of the class of class-defining-objects, as one might expect.

Messages

A message is a stream of zero or more symbols.

If the stream starts with an open parenthesis, its closing parenthesis absolutely terminates the stream.

An embedded "." at the same level will terminate the current message and will cause the message following it to be sent.

If the message is composed of parts whose termination is ambiguous, a "," can be used to clarify matters.

Sending is done from left to right using a very simple rule: control is passed immediately to the first object encountered in the stream, along with information about the context of the send. This is all the EVALuator does. The receiver may gather in the message in any way it chooses.

A common first object is an instance of the class "name" (as with a LISP atom, all of its members start with a letter and are composed of letters, digits , underscores, and other special characters).

The action of a name is to look itself up in the current environment/dictionary to see if it has a meaning (which is another object). If it does, that object is RETURNed by APPLYing it to the remainder of the message;--- And so it goes until the message is consumed.

A venerable example: factorial.

A message

```
factorial 3.
```

is sent in the following manner.

Control is passed to the name "factorial" which looks itself up in the current environment and finds another object as its value. The new object is a class defining object which contains the rules for all the instances of the class "factorial":

```
:n. ↑ if n = 0 then 1 else (n *  factorial n - 1).
```

The action of the class defining object is to create a new instance of factorial and APPLY it to the message.

The ":" is a "receive" (or "input") object whose action is to EVALuate the input stream (in this case "3", whose value is "3") and then to make a new entry into the local environment to define the name (in this case "n"). After this a lookup of "n" will have the value "3".

The "↑" is a "send" (or "output") object which will APPLY the EVALuation of its argument to the remainder of the message found in the CALLER's object.
The next message is sent by finding "if" which tries to receive the message consisting of the EVALuation of "n=Ø".
Control is passed to "n".
It looks itself up and finds "3".
Control is passed to it.
"3" is an instance of the class number which has many relations it can respond to.
"3" receives the next object (unevaluated) to see what it is. (It could be any of +, -, * /, < , > , etc.; in this case it is "=").
"3" wants now to evaluate the next part of the message in order to see whether to RETURN "true" or "false".
Control is passed to "0" which, as with "3", is an instance of class number, and thus shares the same relations.
So, it looks to its right to see if anything like +, -, *, etc., is there which it can respond to.
It finds only "then" for which it has no meaning.
So it RETURNs ITSELF to "3" which now has enough info to decide "not true"
which is RETURNed to "if" which decides not to evaluate the message following "then", but does try to evaluate the message following "else".
"n" looks itself up, finds the value "3"
which picks up the name "*" for which it has a meaning.
So "3" tries to evaluate the next part of its message "factorial n - 1)".
Control is passed to "factorial" which looks itself up and discovers (as before) a class-defining object with the rule:

```
:n. ↑ if n = Ø then 1 else (n *  factorial n - 1).
```

As before, a NEW instance is created which will try to evaluate the message "n - 1)" to get a new value for ":n".
"n" in the OLD environment looks itself up and discovers "3"
which looks to its right and finds "-" so it tries to evaluate the next object "1"
which which looks to its right and finds ")" (which terminates any message to "1")
so it RETURNs ITSELF to "3"
which knows how to subtract "1"
which causes a new instance of class number to be produced for the result "2"
which is RETURNed to the ":" in the CURRENT instance of "factorial"
which will enter it as a value for "n" in the CURRENT environment.
And so it goes.

The preceding rather long winded explanation of a well known example illustrates a number of important points.
First, although the terminology seems to be more general than is needed, a simple program in SMALLTALK looks simple and can be discussed in simple terms.

Second, only one rule of correspondence is needed to link form and content. The evaluator ONLY needs to know how to pass control and context to an object. All other meanings are found distributed with the objects in the system. As shown, even such a seemingly primary act as creating a new instance is done by an object and thus can be changed at the user's whim.

Third, there are many cases where this generality of approach pays off handsomely. If we want to trace the activities of a name (such as "n" in instance 1) we need only create an object which can replace "3" as a meaning (so control will be passed to IT when "n" is touched), AND has a local entry of its own for "3" so that the meaning of "n" will not change with respect to its input/output characteristics. This means that an object can simulate any other object.

Fourth, all "relations" and "operators" (such as <, >, +, *, =, etc.) can be defined "intensionally" (or "intrinsically") as parts of an object or object class, rather than "extensionally" (or "extrinsically"), as is usually the case, as global functions.

In fact, "factorial" could have been defined this way as an intensional relation of a number. We might then have said "3!" and the class number would know what to do.

This means that the information pertaining to a class and what its members do need only be stored with the class. No global operations need to be updated. So, a class may be deleted without changing the rest of the world.

Also, this is a very convenient way to handle problems that arise from having multiple classes with operations: such as coercions between classes and the various senses of "fetch" and "store" ("←").

```
For instance, the message "a ← 3 + 1" means:
   pass control to "a" which will look itself up and
      pass control to the object it finds
         which can gather the rest of the message as it
         pleases.
         It can look to see if the next name is a "←",
         if so, it can EVALuate "3 + 1" and decide how
         to store it.
So "b 1 ← 81" , if "b" were an instance of an array,
could mean
   'store 81 in the 1st position'; or
if "b" were an instance of a hash table routine, could
mean
   'associate the hash of "1" with "81" in some  way',
   etc.
```

The problem of coercions will be discused a bit further on.

Fifth, instances may be EVALuated "concurrently" using the very same EVALuation strategy. Here, the generality of message send/receive becomes much more important.

Class Definitions Already in SMALLTALK

<See SMALLTALK1.DC for this branch>

Some SMALLTALK Programs

<See SMALLTALK2.DC for this branch>

Necessary Information about this paper.
Latest revision: June 6, 1973

(The permanent names of this file are
SMALLTALK.DC.
SMALLTALK1.DC. ***
SMALLTALK2.DC.
Its latest incarnation will always be found on the
Learning Research Group Demo Diskpack.

The full structured index is found with each version.
Look under the structure to discover what file to load.

This file should be displayed using font SROMAN.FD.
To print, edit with SMDELEG.FD and Write Translated,
then print on XGP using font SMDELEG.XG)

SMALLTALK, a Model Building Language
With Intensional Semantics

by
Alan C. Kay

Learning Research Group
Xerox Palo Alto Research Center

Abstract

<See File SMALLTALK.DC for this branch>

Acknowledgements

<See File SMALLTALK.DC for this branch>

Introduction

<See File SMALLTALK.DC for this branch>

Messages

<See File SMALLTALK.DC for this branch>

Class Definitions Already in SMALLTALK

SMALLTALK is supplied with many useful classes, including quite a few found in one way or another in other programming languages.

These definitions are written in SMALLTALK as though they were not primitive objects. In some cases (such as the definition of "if") a primitive must be used to describe itself---which causes some obscurity.

Input and Output Objects

Informally (i.e.---more readable)

: Input a Value

followed by a name will evaluate the input stream to produce a new object which will be bound to the name.

This is exactly the same as LOGO.

Example; :value
will bind the result of evaluating the input stream to "value"

⋄ Input a Token
followed by a <name> will not evaluate the input stream but will bind the next object there to the <name>.

There is no equivalent for this in LISP or LOGO, it acts as though the next input object were quoted.

Example; ⋄value
will bind the next input object to "value"

◦ Check Input for a Token

followed by a <name> will check the input stream to see if an identical <name> is there. No evaluation will take place. The Input Stream Pointer (or Program Counter) will NOT be advanced if the match fails. If the match succeeds, the ISP will be advanced to the next position.
This is used frequently to check for "operator" tokens such as +, *, and ←.

Example; ◦+ will check the input stream for a "+" and will return TRUE if successful

⦂ Input Literal Stream

followed by a <name> will bind a reference to the Input Stream at the current point.
This is equivalent to FEXPR in LISP 1.5 or NLAMBDA in BBN-LISP.

Example; ⦂value will bind "value" to the input stream. EVALuation of this fragment may be delayed until later.

<Other Input Objects>

will be mentioned here in a later version of this memo. An object to EVALuate a sequence of the input stream (like EVLIST in LISP) will probably be included at the very least.

! APPLY-RETURN a value.

This output object is used when when a subroutine control structure and message passing discipline is desired. Its single argument is EVALuated in the CURRENT environment and then APPLYed to the program stream of the CALLER process to which CONTROL also is RETURNed.
When used in "left nested" argument gathering (for example x.first.last or (A + B) + C), APPLY-RETURN will continue the evaluation process.

↑ PASSIVE-RETURN a value.
The single argument is evaluated in the CURRENT environment and RETURNed to the CALLER along with CONTROL.
PASSIVE-RETURN is similar to OUTPUT in LOGO or RETURN in LISP.

```
⇑       GENERAL-RETURN     a value.
   ⇑ value process
      is the form.
   ⇑ value caller.
      is the same as PASSIVE-RETURN.
   ⇑ (apply value message) caller.
      is the same as ACTIVE-RETURN.
```

‹Other Output Objects›

will be explained soon.

Defining a Class (Function)

There are many ways to define a class depending on how much the user wants to know about the language and how much control he desires to have over the format of the INSTANCE of a definition. For now we will only be concerned with semantic notions (which also require the least amount of explanation to all concerned).

LOGO/SIMULA/FLEX Fashion

"To" will define classes of roughly the power of SIMULA or FLEX which include such things as function, process, and structure definitions in other languages.

```
To To ⦂name ⦂body ⌾End.
   "As shown, "To" takes the first object in the message
   stream unEVALuated to be the name of the class. All of the
   rest of the input stream is a structure which is taken to
   be the code body of the class. A member of the class CLASS
   is INSTANTIATED and bound to the name. When control is
   later passed to the name a new instance of the class will
   be created and run"
 End.
```

no really meaning ?!

Examples;

```
    To factorial :n.
       ↑ if n=0 then 1 else (n*factorial n-1).
    End.
```

This looks a lot like LOGO (intentionally) except that the input variable ":n" is not part of the heading (as in LOGO), but is part of the "body". This reflects the fact that input objects act like functions and thus can be used anywhere in a program. When a "function" is instantiated, the first thing that is done in most languages is to bind the arguments to a new set of names. The very same effect is achieved in SMALLTALK when the "evaluating input object", ":" , is used in the first set of expressions.

Ø no need to distinguish
between passing for names
& passing variable names
or values

Conventional Class Definition

"To" as shown above, was included mainly for people familiar with LOGO and LISP. SMALLTALK really treats "class objects" like any other object. That is, any object is a member of a class---so an object which creates a class is a member of class CLASS.
This means that a more general (and more conventional) way to define factorial would be to say

```
    ✦factorial ← class✦(If :n = 0 then 1 else (n * factorial n
    - 1).
```

or perhaps

```
    ✦factorial ← class✦¦ :n.
                       ¦ If n = 0 then 1 else ¦n *
                       ¦                      ¦factorial n - 1))
```

?

using the <tab list> convention. One could even say

```
    ✦var ← ✦n.
    ✦factorial ← class
                ✦(:)¦ var ¦ ✦(= 0 then 1 else)
                ¦(var¦ ✦(* factorial n - 1)).
```

where "¦" means "append" pretty much in the LISP sense.

Total Control of the Instance

***for bit pickers, more on this later this summer.

Control (and State changing, etc.)

```
  To If :exp.
      !exp.
      End.
```

""If" is really just a dummy which computes a value to be APPLYed to "then" or "⇒". This means that "TRUE"ness and "FALSE"ness are properties of objects. This allows us to consider all legal numbers as TRUE, if we wish. A class with one instance EMPTY is provided to handle "FALSE" cases.

To ✦

```
        ⦂name | ₒ← ⦂ (:exp. ↑ exp)
              |    "lookup the name in current environment (if not
              |    there, enter it as most global) and replace BINDING
              |    with value of "exp" ".
              |
              |! name.
        "note that the value of the expression on the right  "exp" is
        RETURNed when a rebind is attempted, but when used as QUOTE,
        it is the name which is RETURNed."
        End.

     To Eval  :exp :globalenv :return :msg.
           "There are many ways to EVALuate expressions in Smalltalk.
           This one allows the user to set up an arbitrary environment
           for free variable fetches, an arbitrary RETURN process, and
           an arbitrary MESSAGE environment."Eval" is included here
           since it is very frequently used in definitions of new
           control primitives".
        End.

     To Repeat ⦂Loopexp.
        Code repeat.
        Eval Loopexp |global |self EMPTY.
        Code again.
        End.
           "Repeat EVALs its loop expression in the context of its
           caller."

     To Again
           "RETURNs control to the caller of its caller--i.e. to a
           looping control primitive of some kind such as "Repeat"
           which can decide what to do next".
        End.

     To Done
           "RETURNs control to the caller of (the caller of its
           caller)--to one level further out than a looping control
           primitive. This automatically terminates the loop.
           Eventually "Done" will have an optional argument for
           passing the RESULT of the loop back".
        End.
```

✓

Caller of ( )
Caller of its caller
its caller
↑

```
     To Create
           "Reschedule caller to be run instead of waiting for a
           subroutine  RETURN".
        :call.
           "This causes an evaluation of the argument. So it will also
           be running".
        End.
              "As seen, "Create" causes a parallel fork in control.
              Actually, this is what happens naturally in
              SMALLTALK---the default message discipline is
              deliberately limited to a subroutine "wait for reply"
              protocol. "Create" simply prevents the caller from being
              passivated".


 To Word

     ₒExplain⦂
        ↑"Words are like LISP atoms or ALGOL identifiers. Their basic
        operations have to do with assembly and disassembly of their
        internal structures.
           Words also have a special meaning in the context of
           evaluation. An unquoted instance of a word will be looked
```

```
          up (look itself up) when encountered by the EVALuator. So
          cat₊first  means  "look up the most local binding of the
          variable "cat" and APPLY it to ₊first". But ₊cat₊first
          means " call routine "₊" which  RETURNs the word "cat",
          which is APPLYed to ₊first, which, as  seen below, will
          RETURN "c" ".
      Numbers are words also, but have many additional operations
      having to do with arithmetic and so are defined as a separate
      class.".

  ₒ← ⇒ :value₊word ⇒
      ↑self.
        "..."
  ₒfirst ⇒
      ↑"the first character of the printname of the word".

  ₒf ⇒
      ↑"same as "first"".

  ₒlast ⇒
      ↑"...the last character of the printname of the word"

  ₒl ⇒
      ↑"...the same result as for "last". This is just an
      abbreviation."
  ₒbutfirst ⇒
      ↑"Somehow return all but the first character of the string
      representation of the word."

  ₒbf ⇒
      ↑"...same as butfirst."

  ₒbutlast ⇒
      ↑"Somehow return all but the last character of the string
      representation of the word."

  ₒbl ⇒
      ↑"...same as butlast."

  ₒjoin ⇒ :value1₊word? ⇒
      ↑"This is roughly equivalent to the "cons" of LISP. The word
      will be connected to the list in "value1", and a new list
      reference will be returned."

  ₒwjoin ⇒:value1₊word? ⇒
      ↑"This is roughly equivalent to concatenate in SNOBOL. The
      printname of the two words are joined together to produce a
      new word which is returned. ₊cat wjoin ₊dog     produces
      ₊catdog."
  ₒword? ⇒
      ↑value.

  ₒempty? ⇒
      ↑EMPTY.

  ₒlength ⇒
      ↑"Somehow calculate the length (in characters) of the
      number     (including "-" and ".")  ."

  ₒprint ⇒
      ↑"Return a string representation of the object which may be
      displayed. Each class which has instances which have a
      meaningful visual representation will have a meaning for
      ₊print. This is much simpler than having to inform a global
      print routine about the format of each new class."
```

how do classes obtain meaningful visual repr.?

```
ₙthen ⇒ :value1 ⇒ ₙelse ⇒ ⋄dum ⇒ ↑ value1
                        or        ⇒ ↑ value1.
   "Having "then" in "Word" in this way means that we are
   adopting a convention that legal words in the context of a
   test, act as TRUE and thus cause the "then" expression to be
   evaluated."
```

To Number

```
ₙExplain⇒
   ↑"Numbers work in a very intuitive way. The READ program
   recognizes  number literals and creates instances for them in
   storage.The bits  that represent the particular instance of a
   number are stored in  the variable "value" and can be changed
   by assignment as shown.  This might be illegal if it is
   decided that numbers are unique  atoms. The opposite is
   assumed here."

ₙ← ⇒ :value⋄number?⇒
   ↑ self.
      If a "↑" is recognized in the input stream, what follows is
      evaluated and bound to "value" which is applied to number?
?     which  returns TRUE if it is. The actual value of the
      number object itself has been changed so that other objects
      which have pointers to "self" will feel the change. This
      might be made illegal.

ₙfirst ⇒
   ↑"Somehow return the first character of the number which is
   "-" if negative, is "." if between 0 and 1, and a digit from 0
   to 9 otherwise. It may be reasonable to calculate this value
   rather than keep a string representation of the number
   around."

ₙf ⇒
   ↑"...the same result as for "first". This is just an
   abbreviation."

ₙlast ⇒
   ↑"Somehow return the last character of the number which is
   "." if greater than 1 and known inexactly, and a digit from 0
   to 9 otherwise. It may be reasonable to calculate this value
   rather than keep a string representation of the number
   around."

ₙl ⇒
   ↑"...the same result as for "last". This is just an
   abbreviation."

ₙbutfirst ⇒
   ↑"Somehow return all but the first character of the string
   representation of the number."

ₙbf ⇒
   ↑"...same as butfirst."

ₙbutlast ⇒
   ↑"Somehow return all but the last character of the string
   representation of the number."

ₙbl ⇒
   ↑"...same as butlast."
```

```
ₒjoin ⇒ :value1.word? ⇒
   ↑"This is roughly equivalent to the "cons" of LISP. The word
   will be connected to the list in "value1", and a new list
   reference will  be returned."

ₒwjoin ⇒:value1.word? ⇒
   ↑"This is roughly equivalent to concatenate in SNOBOL. The
   printname of the two words are joined together to produce a
   new word which is returned. •cat wjoin •dog     produces
   •catdog."

ₒnumber? ⇒
   ↑value.
   "Anything not EMPTY will act as TRUE."

ₒword? ⇒
   ↑value.

ₒempty? ⇒
   ↑EMPTY.

ₒlength ⇒
   ↑"Somehow calculate the length (in characters) of the
   number     (including "-" and ".")  ."

ₒprint ⇒
   ↑"Return a string representation of the object which may be
   displayed. Each class which has instances which have a
   meaningful visual representation will have a meaning for
   •print. This is much simpler than having to inform a global
   print routine about the format of each new class."

ₒthen ⇒ :value1 ⇒ ₒelse ⇒ •dum ⇒ ↑ value1
                        or          ⇒ ↑ value1.
   "Having "then" in "Number" in this way means that we are
   adopting a convention that legal numbers in the context of a
   test, act as TRUE and thus cause the "then" expression to be
   evaluated."

ₒ= ⇒ :value1.number? ⇒                                          ✓
   ↑"value if value and value1 are numerically EQUAL, otherwise
   EMPTY.  Note that this allows "a=b=c" to work correctly."

ₒ≠ ⇒ :value1.number? ⇒
   ↑"EMPTY if value and value1 are not numerically EQUAL,
   otherwise  value. Note that this allows "a≠b≠c" to work
   correctly."

ₒ< ⇒ :value1.number? ⇒
   ↑"value if value is numerically less than value1, otherwise    2<4<1
   EMPTY.  Note that this allows "a<b<c" to work correctly."     2<4<3

ₒ> ⇒ :value1.number? ⇒
   ↑"value if value is numerically greater than value1, otherwise  4>2>3
    EMPTY.. Note that this allows "a>b>c" to work correctly."

ₒ+ ⇒:value1.number? ⇒
   ↑"value added to value1."

ₒ- ⇒:value1.number? ⇒
   ↑"value1 subtracted from value."

ₒ* ⇒ :number? ⇒
   ↑"value multiplied by value1."
```

right to left eval
due to matching msg
all the way

```
    ₒ/ ⇒ :value1.number? ⇒
       ↑"value divided by value1."

    ₒmod ⇒ :value1.number? ⇒
       ↑"value modulo value1."

    ₒip ⇒
       ↑"...the integer part of value."

    ₒfp ⇒
       ↑"...the fractional part of value."

    ₒexp ⇒
       ↑"...the exponent (to the base 10) of value."

    ₒmag ⇒
       ↑ if  value < 0 then (0 - value) else  value.

    <other numeric functions which are stored as attributes>
    sin, cos, other trig functions etc.

To List
    ₒExplain
    ₒfirst ⇒ ₒ← ⇒ :value.list ⇒
                         value.word? ⇒
    ₒf ⇒ ₒ← ⇒ :value.list ⇒
                         value.word? ⇒
    ₒlast ⇒ ₒ← ⇒ :value.list ⇒
                         value.word? ⇒
    ₒl ⇒ ₒ← ⇒ :value.list ⇒
                         value.word? ⇒
    ₒbutfirst ⇒ ₒ← ⇒ :value.list ⇒
                         value.word? ⇒
    ₒbf ⇒ ₒ← ⇒ :value.list ⇒
                         value.word? ⇒
    ₒbutlast ⇒ ₒ← ⇒ :value.list ⇒
                         value.word? ⇒
    ₒbl ⇒ ₒ← ⇒ :value.list ⇒
                         value.word? ⇒
    ₒjoin ⇒
    ₒ! ⇒
    ₒ¦  ⇒
    ₒsentence? ⇒
    ₒlist? ⇒
    ₒempty? ⇒
    ₒlength ⇒
    ₒprint
    ₒ= ⇒ :value.list ⇒
    ₒ≠ ⇒ :value.list ⇒
    ₒ< ⇒ :value.list ⇒
    ₒ> ⇒:value.list ⇒
    ₒmakeword ⇒

To String
Position
```

Here are a set of useful operations for manipulating two-dimensional space. The convention is adopted that "posx" and "posy" will refer to position state, and "heading" will refer to direction state. The programs are written so that the most local occurance of these variables in the dynamic environment will be updated. See the program "Spacevehicle" for a simple example.

```
    To Forward :distance.
       posx ← posx + distance * heading.cos.
       posy ← posy + distance * heading.sin.
    End.

    To Right :angle.
       heading ← (heading - angle) mod 360.
    End.

    To Left :angle.
       heading ← (heading +  angle) mod 360.
    End.
```

Output   (to displays, music, turtles, etc.)

```
    To Show :picture.
       "This comprehensive routine allows the picture to be EVALed
       and then copies the picture information into the display area
       using either the dynamically available variables "posx",
       posy", "heading", if its own bindings for these parameters are
       EMPTY.
```

Some SMALLTALK Programs

<See SMALLTALK2.DC for Program Examples>

ACKNOW.DC smdeleg.fd ack

ACKNOWLEDGEMENTS

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 is a godchild of Wes Clark's LINC and a lineal descendant of the FLEX Machine.

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

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

The Burroughs B5000 (1960) had many design ideas well in advance of its time (and still not generally appreciated): compact "addressless" code; a uniform semantics for names (the PRT), automatic coprocesses, "capability" protection (also by the PRT), virtual segmented memory; the ability to call a subroutine from "either side" of the assignment arrow; etc.

The notions of code as a data structure; intensional properties of names (property lists of attribute:value pairs on atoms); evaluation with respect to arbitrary environments; etc., are found in LISP, probably the greatest single design for a programming language yet to appear.

The SIMULA's ('65 and '67) combined Conway's notions of software coroutines (1963-hardware versions had appeared in the B5000 3 years earlier), ALGOL-60, and Hoare's ideas about record classes(196**) into an epistemology that allowed a class to have any number of parallel instantiations (or activation records) containing local state including a separate Program counter. Most of the operations for a SIMULA '67 class are held intrinsically as procedures local to the class definition.

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

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

An extemporaneous talk by R.S. Barton at Alta Ski Lodge(1971) about computers as communications devices and how everything one does can easily be portrayed as sending messages to and fro, was the real genesis of the current version of SMALLTALK.

The fact that kids were to be the users and the simplicity and ease of use of the already existing LOGO (whose own parents were JOSS and LISP) provide lots of motivation to have programs and transactions appear as simple as possible--i.e. moving from left to right, procedures gather their own messages, etc. It is no accident that simple SMALLTALK programs look a bit like LOGO!

Problems discovered years ago in "lefthand calls" prompted SMALLTALK to make "store" intensional--i.e. a ← b, means "call "a" with a message consisting of the token "←" and "b"". If anyone can make the right decision for what this means, it must be "a". The early fall of 1972 saw an evaluator for SMALLTALK and the idea that "+", "-", etc., all 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". The message ideas allow messages to have a wide range of form since a message can be received incrementally.

"Control of control" allows control structures to be defined. The language SMALLTALK itself thus avoids "primitives" such as "loop ol", 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.

microPLANNER's main influence was negative in that it convinced us finally that backtracking is not the way to approach problem solving. Instead, we prefer "trial evaluation" where a "straw process" is run in a "straw environment" as a coprocess and constantly sends messages as to how badly it gets creamed back to its originator. If it perishes, its environment is just discarded rather than backtracked. (A germ of this idea is found in Fisher's thesis).

The fine idea of microPLANNER ("pattern directed invocation"---I call it "call by desire") does not appear as a primitive in the current version of SMALLTALK (it was in ST-1971) but may be easily added in just the way a particular user desires.

****

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 is can be seen as a consolidation of good ideas into one simple idea:

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

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

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

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

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

Line Graphics and the hand-character recognizer were done by John Shoch. Ted Kaehler did the "scope turtle" on the ALTO.

"Music" was designed and implemented by Alan Kay. Barbara Deutsch wrote the program to REGISTER combinations of timbre files. Peter Deutsch(CSL,PARC) designed and wrote a translator for compact musical "score" notation. Steve Saunders improved most of these programs.

Diana Merry wrote auxilary systems programs and a very nice "software character generator" and text scroller for the ALTO.

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

This file is called SMSEMANTICS.DC and contains a semantic description of SMALLTALK written in itself.
This version was last changed on June 10, 1973.
Use font SMDELEG.FD

SMALLTALK and its Semantics
    by
Alan Kay

WARNING!! This is an unchecked version done simply to try it out for basic taste and compactness.

```
⬥To ← class¦⬥Do ← ⬥¦◊name ◊actions.
           ¦        ¦↑ Find (name) in CALLER ← class¦⬥Do ← activity


To ⬥ ¦◊name ¦ₒ← ⇒ ¦:exp.
     ¦      ¦     ¦Find (name) in CALLER ← exp.
     ¦      ¦!name


To Find ¦:name ¦ₒin ⇒ ¦:context
        ¦      ¦⬥context ← CALLER
        ¦              ⬥
        ¦Repeat
        ¦   ¦context⬥table name  OR  context⬥table⬥global⬥empty?
        ¦   ¦        ⇒¦ₒ← ⇒ ¦:exp. context⬥table name ← exp. Done
        ¦   ¦⬥context ← context⬥table⬥global
        ¦    ⬥
        ¦↑ context⬥table name


To List ¦ₒ← ⇒ ¦:first :rest. ↑self
        ¦
        ¦ₒfirst ⇒ ¦ₒ← ⇒ ¦:first. ↑self
        ¦         ¦!first
        ¦
        ¦ₒrest ⇒ ¦ₒ← ⇒ ¦:rest. ↑self
        ¦        ¦!rest
        ¦
        ¦ₒlength ⇒ ¦!¦first=NIL ⇒ ¦0
        ¦          ¦ ¦1+rest⬥length
        ¦
        ¦ₒprint ⇒ ¦!"(" ⁝ first⬥print ⁝" "⁝ rest⬥print ⁝")"
        ¦
        ¦ₒlist? ⇒ ¦!self
        ¦
        ¦ₒeval ⇒ ¦Repeat
        ¦        ¦   ¦first = ")" ⇒ ¦Done
        ¦        ¦   ¦first = "."  ⇒ ¦⬥value ← restₒeval
        ¦        ¦   ¦⬥value ← first⬥eval
        ¦        ¦!value


To Repeat ¦◊program.
          ¦CODEFOR ¦Repeat ¦ clause ⬥eval global message self
```

```
To Again ¦⇑ EMPTY CALLER.CALLER

To Done ¦:value. ⇑ value CALLER.CALLER.CALLER

To If ¦:exp ⇒ ¦ₒthen :exp ¦ₒelse ⇒ ¦◊. ↑exp
      ¦        ¦           ¦↑exp
      ¦        ¦error "I can't find a "then""
      ¦
      ¦ₒthen ◊. ¦ₒelse ⇒ ¦:exp. ↑exp
      ¦         ¦↑EMPTY

To User ¦Repeat
        ¦    ¦Display Read.eval.print

To ◊¦self.table.name ← message.table.pc.first.
    ¦message.table.pc ← message.table.PC.rest.
    ¦message.table name ← message.table.message.table.PC.first.
    ¦message.table.message.table.PC ← message.table.message.table.PC.rest.
    ¦! name.

To : ¦◊name.
     ¦! message.table.name ←
     ¦        message.table.message.table.PC.first .eval message.message

To ₒ ¦◊token ≠ message.table.PC.first ⇒ ¦! EMPTY
     ¦message.table.PC ← message.table.PC.rest

To ⇒ ¦:clause. ⇑ clause CALLER.CALLER.CALLER

To EMPTY ¦ₒ⇒ ⇒ ¦◊. ⇑ self CALLER.CALLER
         ¦ₒempty? ⇒ ¦! .TRUE
         ¦! self

To Apply ¦ :t :g :c :m.
         ¦ (t ← .global g  .caller c  .message m ).eval

To ⇑ ¦:value :destination.
     ¦Apply value destination destination destination.

To Remember ¦ₒ← ⇒ ¦Repeat
            ¦     ¦   ¦ₒEMPTY ⇒ ¦↑self
            ¦     ¦   ¦self :name ← :value
            ¦
            ¦ₒcopy ⇒ ¦! CODEFOR "somehow copy the table"
            ¦
            ¦ₒeval ⇒ ¦"Do something or other"
            ¦
            ¦:name ¦ₒ← ⇒ ¦:value.
            ¦      ¦     CODEFOR "associate name and value somehow"
            ¦      ¦     ¦↑value
            ¦      ¦
            ¦      ¦! CODEFOR "Get the value associated with the name"
```

```
To class ┊⋆bindings.
         ┊↑ instantiate ┊ Remember ← ⋆class ⋆class
         ┊              ┊            ⋆global global
         ┊              ┊            ⋆caller self        ┊              ┊
           ⋆message message
         ┊              ┊            ⋆PC  bindings
         ┊              ⋆eval

To instantiate ┊:classdef.
               ┊Repeat
               ┊   ┊Pause.
               ┊   ┊┊classdef⋆copy ← ⋆class classdef
               ┊   ┊┊                ⋆global global
               ┊   ┊┊                ⋆caller caller
               ┊   ┊┊                ⋆message message
               ┊   ┊┊                ⋆PC  classdef⋆DO
               ┊   ┊⋆eval


To Word
    ┊ₒ← ⇒ ┊:first :rest.
    ┊     ┊⋆rest ← Word ← first⋆butfirst rest.
    ┊     ┊⋆first ← first⋆first.
    ┊     ┊↑ self
    ┊
    ┊ₒfirst ⇒ ┊ₒ← ⇒ ┊:first⋆character ⇒ ┊↑first
    ┊         ┊     ┊error"input is not a character"
    ┊         ┊! first
    ┊
    ┊ₒrest ⇒ ┊ₒ← ⇒ ┊:rest⋆character ⇒ ┊↑rest
    ┊        ┊     ┊Error"Input is not a word"
    ┊        ┊! rest
    ┊
    ┊ₒlength ⇒ ┊! rest = NULL ⇒ 1
    ┊          ┊ 1 + rest⋆length
    ┊
    ┊ₒprint ⇒ ┊↑ first⋆print. rest⋆print
    ┊
    ┊ₒword? ⇒ ┊! self
    ┊
    ┊ₒ= ⇒ ┊:value. ↑ (first = value⋆first) AND next = value⋆next
    ┊
    ┊ₒeval ⇒ ┊⋆env ← global.
    ┊        ┊Repeat
    ┊        ┊   ┊env⋆empty? ⇒ ┊↑EMPTY
    ┊        ┊   ┊⋆temp ← env⋆table self ⇒ ┊↑ apply temp global caller
    ┊        ┊   ┊                         ┊       message
    ┊        ┊   ┊⋆env ← global⋆table⋆global
```

CONTENTS

INFORMAL.DC smlg.fd smdeleg.fd sroman.fd infm

An Informal Introduction to SMALLTALK †

by

Alan C. Kay
Xerox Palo Alto Research Center

The easiest way to learn SMALLTALK is to just make it do useful things for you !*

Let's get SMALLTALK to draw a SQUARE for us. First we have to tell SMALLTALK just what it is that we mean by "square".

```
to square                    **
    ⁞forward 100
    ⁞right 90
    ⁞forward 100
    ⁞right 90
    ⁞forward 100
    ⁞right 90
    ⁞forward 100
```

"To" is part of SMALLTALK. We sent it a message consisting of a name,"square", and a definition in terms of drawing commands inside of "margin parentheses".

We can now use our definition just as though it had been part of SMALLTALK.

```
square
```

and a square is drawn. Try it again.

```
square
```

Whoops! We just created some bugs! First, the "pen" was not left pointing the same way as it was found, and also we forgot to clear the screen and return the pen to the center.

----

† Please see the rest of this handbook for a more detailed description of SMALLTALK and its use, especially the acknowledgements for a fairly complete set of historical influences on SMALLTALK's design.

Most of the examples in this paper work, particulary those with photos of results. However a few of them are still awaiting their baptism of fire so be careful.

* This document's approach and some of the examples (marked with a **) are adapted from Papert(pa ****). Simple things look as simple as possible and hence, resemble LOGO (or JOSS) to some extent. As you will see however, the semantics of SMALLTALK are quite different in most ways for more complex ideas.

erase

clears the screen.

home

centers the "pen" pointing up.

white

draws using white ink on a black background. Try

```
white home erase forward 50
```

black

draws using black ink on a white background.

A "cleaner" version of "square" is

```
to square
   ¦forward 100
   ¦right 90
   ¦forward 100
   ¦right 90
   ¦forward 100
   ¦right 90
   ¦forward 100
   ¦right 90
```

What is there about this sequence of actions which has to do with "squareness"? All the turns are 90 degrees, and they alternate with forward travel of the same distance. So the following definition should also work.

```
to square
   ¦repeat 4
   ¦  ¦forward 100
   ¦  ¦right 90
```

Try it.

"Repeat" is sent a message consisting of two parts. The first is how many "repeats" are desired, the second is just what to repeat.

What about a square of any size? What is there about the previous definitions that only has to do with size as opposed to "squareness"?

It seems only to be the distance traveled (which is the message to the "forward" command).

Just as we can send messages to "forward", "right" and "repeat" to give them additional information about our desires, we can send messages to our own definitions as well. We would like to send "square" a message which says what length of side we want each time.

such as   square 100   or   square 50

In SMALLTALK any definition can receive a message by saying ":". Since the message is different each time, it would be nice to give it a name to allow it to be used anywhere in the definition.

A definition to draw a square of any size is

```
to square
    ┆:size
    ┆repeat 4
    ┆   ┆forward size
    ┆   ┆right 90
```

Try it and see. The ":" picks up the message and calls it "size". "forward" refers to the message by its name "size".

home erase square 50 home square 100

Now let's try a TRIANGLE of any size. Well, it's really almost the same as a square, isn't it?

```
to triangle
    ┆:size
    ┆repeat 3
    ┆   ┆forward size
    ┆   ┆right 120
```

Try it.

The two definitions are almost the same except for the number of "repeats" and the angle. Is it possible to define actions which will draw

ANY POLYGON?

Well, we could certainly send the definition a message of two parts. One for the size, the other for the number of sides we want.

```
to poly
    ┆:sides :size
    ┆Repeat sides
    ┆   ┆forward size
    ┆   ┆angle ****
```

This looks reasonable except for confusion about the angle. "Repeat" will be sent a message for the correct number of sides and "forward" will get the right message about side length as before.

Now, what about the angle? When we turned right for the triangle it was 120, for the square 90. What about a pentagon? 72?

One neat way to look at the situation is that a complete trip for any polygon will get you back EXACTLY where you started and the heading of the pen WILL HAVE TURNED THROUGH 360 degrees EXACTLY.

The number of turns taken is the same as the number of sides (because the "repeat" controls this). So, it seems as though the angle taken should be 360/sides. Try it.

```
to poly                       **
    ┆:sides :size
    ¦Repeat sides
    ¦   ¦forward size
    ¦   ¦right 360/sides
```

Try a few to see.

```
poly 5 50

poly 5 100

poly 18 20

poly 50 5

poly 360 1
```

Hmmmm. Does this make sense for a

CIRCLE?

```
to circle
    ¦poly 360 1
```

It's nice that we can use any of our definitions exactly like SMALLTALK's own commands. Now suppose we want circles of DIFFERENT size. What is there about "poly 360 1" that is "circlelike" and what has to do with size?

We know that "poly 3 ***" doesn't look like a circle and "poly 360 1" does.
What about "poly 360 10" ?

Try it.

So how does this strike you?

```
To circle
    ¦:size
    ¦poly 360 size
```

Now you may say, "OK, we can change the size of a circle alright, but the number we are sending as a message doesn't seem to bear any relationship to the diameter or radius". True? Well, what do we know about the circle? What is its circumference?

Well, it seems that poly "repeats" 360 times. Each of those times "forward" goes forward a distance. So the circumference of any polygon is sides * size. A relation between the radius of a circle and its circumference is: Circumference = 2 * pi * radius.

```
 So,  sides * size = 2 * pi * radius
and,          size =(2 * pi * radius)/sides
```

Let's now define a circle routine where the message we send it is the radius.

```
to pi
    ¦3.14159
```

```
to circle
    ¦:radius
    ¦poly 360 ((2 * pi * radius)/360)
```

Try it and see.

```
circle 100
```

This looks reasonable though a question might arise about the second part of the message to "poly", "((2 * pi * radius)/360)". What is sent if radius = 100? Is it "2.****" or "((2 * pi * radius)/360)" or "((2 * 3.14159 * 100)/360)"?

In SMALLTALK these distinctions are controlled by the receiver.

"poly" receives the message by saying ":size". The ":" not only means "receive" but also means "receive value", so "size" will stand for "2.****" not "((2 * pi * radius)/360)".

If "poly" had said "⦂size", "size" would have stood for the literal message "((2 * pi * radius)/360)", since "⦂" means "receive the literal message".

Try this out by writing

```
To test
    ¦:val ⦂form
    ¦Print val. Print form


test ((2 * pi * radius)/360) ((2 * pi * radius)/360)

2.****
((2 * pi * radius)/360)
```

Both parts of the message to "test" are the same, but are received differently. There are other useful ways to receive messages in SMALLTALK as we will see later.

By now you are probably getting the idea that getting SMALLTALK to do things is easy. True.

Now what happens if we jiggle some of the things we are doing a little?

Seymour Papert's kids call the following kinds of things

SQUIRALS!

```
to squiral                **
    ¦:size :angle
    ¦forward size
    ¦right angle
    ¦squiral size+10 angle+2
```

Notice that this definition goes on forever so the "whoops" key needs to be used!

******More on this in a bit. It can be found in Seymour's stuff.
****Don't forget to do simple translation and rotation in inertial coords

We'll come back to some really far out graphics in just a bit. For now though, let's look at some of the other things that SMALLTALK can do for you.

```
        Try
->2+2
4
```

Suppose you would like to treat the Dynabook as a desk calculator so that accumulated answers are shown as you go.

```
To desk
    ⁺reg ← 0
    Repeat lots
        Print ">>"
        ◦= ⇒ Print reg. Done
        :input
        ◦+ ⇒ Print ⁺reg ← reg + input
        ◦- ⇒  Print ⁺reg ← reg - input
```

and so on.

"⁺reg ← 0" means "reg" is REDEFINED to stand for 0. This is done by the object "⁺" which receives a three part message conssisting of a name (in this case "reg") an arrow "←", and a value (in this case 0). It finds the SMALLTALK "dictionary entry" for "reg" and redefines it to be 0.

"◦" is yet another way to receive a message. It looks to see if the word which follows it is literally there in the message. So "◦+" asks if the current part of the message is literally a "+". EMPTY (which acts like "false") is returned if the match failed.

"⇒" is one way to choose one part of a program over another depending on some condition being "true" or "false" (actually not EMPTY or EMPTY). Consider

```
    a = b ⇒ Print "Its true!"
    Print "Its not true at all"
```

If the value of "a" is equal to the value of "b" then the "⇒" will cause the list following it to be evaluated and then an exit taken completely out of its enclosing list. So only "Its true!" will be printed. This allows lots of conditions to be checked with an appealing form to the program.

```
to check
    :a :b

     a = b ⇒ Print "They're equal"

     a < b ⇒ Print "a is less than b"

    Print "a is greater then b"
```

"Done" exits from the nearest enclosing "Repeat" and provides a way to terminate a loop if one wishes. "Again" will restart the loop from the current point.

Now, imitating a desk calculator with all its limitations is a bit silly. What would you really like to have?

What would be nicer is to be able to enter an arithmetic expression, to have each subexpression's value printed and to retain the entire expression for editing rather than forcing reentry.

```
To nicedesk
    ****
```

****an amortization scheme would be nice to do ala JOSS

****keeping track of a cookbook?

An interesting variation on "poly" is a definition that draws

NESTED POLYS!

```
To star
    ¦:sides :size
    ¦If size > 4
    ¦  then ¦Repeat sides
    ¦       ¦   ¦forward size
    ¦       ¦   ¦right 360/sides
    ¦       ¦   ¦star size/3
```

This clever little program was invented by Dan Ingalls.

Try a few of these.

```
star 5 100      star 6 80
```

How about some SPACE FILLING curves?

This MAZE curve was invented by Hilbert. The program is by D.Ingalls and T.Kaehler.

```
To Maze
    ¦:lev :bend :side.
    ¦lev = 0 ⇒ ¦right 90*bend. forward 10
    ¦bend = side ⇒ ¦Maze1 lev - 1
    ¦bend = 0 - side ⇒ ¦Maze2 lev - 1
    ¦Maze3 lev - 1

To Maze1
    ¦:l.
    ¦Maze l 0 - bend side.
    ¦Maze l bend 0-side.
    ¦Maze l bend side.
    ¦Maze l 0 0-side

to Maze2
    ¦:l.
    ¦Maze l 0 side
    ¦Maze l bend side.
    ¦Maze l bend 0-side
    ¦Maze l 0-bend side

To Maze3
    ¦:l Maze l 0-side.
    ¦Maze l side 0-side.
    ¦Maze l side side
    ¦Maze l 0-side 0-side
```

This curve will energetically chug away and will eventually touch every point in an arbitrarily large space.

Here is a more compact, but slightly more obscure SMALLTALK "oneliner" for the Hilbert curve.

```
To Maze
    ¦****
```

Now, are you all ready for a DRAGON? This is a very simple definition whose actions are hard to predict.

```
To dragon
    ¦:length
    ¦If length = 0 then ¦forward 10
    ¦
    ¦                else ¦If length > 0
    ¦                     ¦   then ¦ dragon length-1
    ¦                     ¦        ¦ right 90
    ¦                     ¦        ¦ dragon -length-1
    ¦                     ¦
    ¦                     ¦   else ¦dragon -length+1
    ¦                     ¦        ¦right -90
    ¦                     ¦        ¦dragon length+1
```

A more compact way to say this is

```
To dragon
    ¦:length = 0 ⇒ ¦forward 10
    ¦
    ¦ length > 0 ⇒ ¦dragon length-1. right 90. dragon -length-1
    ¦
    ¦ dragon -length+1. right -90. dragon length+1
```

Dan Ingalls concocted this neat little program.

Experiments with

ACCELLERATED MOTION

****There should be some simple stuff here for linear velocity and acc

```
To rollick
    ¦:times ⦂figure
    ¦setup
    ¦Repeat times
    ¦   ¦penup
    ¦   ¦forward .dist ← dist + inc
    ¦   ¦right (.angle ← angle + ainc) + .turn ← turn + inc
    ¦   ¦pendn
    ¦   ¦(figure) size
```

Try   rollick 100 square

SHOOTING elastic objects into the air.

```
To shoot
    ¦:xvel :yvel
    ¦.currentyval ← yvel
    ¦
    ¦Repeat lots
    ¦   ¦pendown. object. penup.
    ¦   ¦
    ¦   ¦forward .yvel ← yvel - gravity
    ¦   ¦right 90. forward xvel. right 270
    ¦   ¦
    ¦   ¦currentyval = -yvel ⇒
    ¦   ¦     ¦(closeto 0 .yvel ← .currentyval ← currentyval * elastic) ⇒
    ¦   ¦     ¦                              ¦Done
```

This program is very simple and easy to understand. Each time through a constant representing gravitational force is subtracted from the vertical component. The x velocity is constant and thus just accumulates horizontal distance.

There is no simple closed form equation in classical mathematics that expresses the bouncing ball because of the discontinuities at the bounce points. In SMALLTALK however, it is easy!!

A simple SPACESHIP!

```
To drawship
    ¦pendown
    ¦right 180. forward 5. right 315. forward 7. right 225. forward 20.
    ¦right 315. forward 7. right 270. forward 7.
    ¦right 315. forward 20. right 225. forward 7.
    ¦right 315. forward 5.
```

That was tedious, wasn't it? Later we will discover that we can just paint, draw or sketch any figure to be animated ourselves without having to make SMALLTALK draw them.

First try SHOOTing the spaceship by saying

```
To object
    ¦drawship
```

Why does the preceding definition work?

```
shoot 6 60
```

Now for a bit more freedom

```
To moveship
    ¦:point :thrust
    ¦.turn ← .speed ← 0
    ¦
    ¦Repeat forever
    ¦   ¦penup
    ¦   ¦forward .speed ← speed + thrust
    ¦   ¦right .turn ← turn + point
    ¦   ¦
    ¦   ¦drawship
```

Try

```
moveship 2 1

moveship 1 2
```

Use the WHOOPS button to kill a version.

Now for the big time! Try

```
moveship mouseX mouseY
```

and grab the mouse quickly!!

SPACEWAR!!

Now for the first time we need to use the greater generality of SMALLTALK. We need to be able to create any number of ships and torpedos on the screen, all running together. SMALLTALK allows this to be done as easily as if just one object were desired.

```
To Spacevehicle
     ¦
     ¦ :shape  ₒat  :posx :posy :heading
     ¦         ₒspeed :speed
     ¦         ₒcontrols  :thrust  :point  :trigger.
     ¦
     ¦•Numberoftorps ← 20.
     ¦
     ¦Repeat
     ¦   ¦Left Roll ←  Roll + point.
     ¦   ¦Forward Speed ← Speed + thrust.
     ¦   ¦
     ¦   ¦If thrust > 0 then (Show shape •exhaust : Flame).
     ¦   ¦If thrust < 0 then (Show shape •nose : Flame)
     ¦   ¦              else (Show shape).
     ¦   ¦
     ¦   ¦If  trigger on and Numberoftorps > 0
     ¦   ¦  then ¦Numberoftorps ← Numberoftorps - 1 .
     ¦   ¦       ¦create
     ¦   ¦       ¦  Spacevehicle •torpedo
     ¦   ¦       ¦  at posx posy direction
     ¦   ¦       ¦  speed speed
     ¦   ¦       ¦  controls 25 0 •off
     ¦   ¦       .
     ¦   ¦
     ¦   ¦If touching something
     ¦   ¦  then (Quit something. Show Crash. Quit self).
     ¦   ¦
     ¦   ¦Pause
```

This set of actions defines both what a spaceship and a torpedo do in a somewhat sneaky way. A torpedo is a spaceship with a different shape, constant thrust, straight direction, and no ability to fire torpedos of its own.

The pictures "Ship" and "Torpedo" both have a subpart called "exhaust". This acts as a "hole" where other pictures can be placed, such as "Flame" when the thrust is on. A special subpart name, "center", defines the axis of rotation for "left" and "right" turns.

"Crash" in a more elaborate example would probably be a set of actions to produce ever more grandiose effects.

This particular game starts a ship out with 20 torpedos with no provision for more when all are fired.

"Speed" and "Roll" are names for the accumulated velocity of forward travel and turning. So the "thrust" and "point" controls are accellerations as in a real spaceship.

The "behaviour" at the bottom signals the actions to be done. The message received is what "shape" to use, what initial "position" and "direction" to assume (these names are the ones that are updated by "Forward" and "Left"), and where the information for "thrust", "attitude", and firing of torpedos is to be supplied. For spaceships it will be the joystick of each player, for torpedos, it will be constant information.

The actions are "Repeat"ed over and over.

They are to update the "Roll" and "Speed" accumulations,
to reposition the ship, which will update "position" and "direction",
to display the shape of this object (with "Flame" if thrust is "on"),
to send off a "Torpedo" if the "trigger" is "on" and the "Number∘of∘torps"
left is greater than zero.

Then a check is made for a "touch" and, if so, the object touched is destroyed ("Quit"), the great "Crash" is "Show"n, and finally our object destroys itself.

As many spaceships as required may be instantiated by using "create".

```
create
Spacevehicle ⁎Ship at random random random
                   speed random
                   controls joy 1 up joy 1 side joy 1 but.
create
Spacevehicle ⁎Ship at random random random
                   speed random
                   controls joy 2 up joy 2 side joy 2 but.
```

New "Data" Objects and their "functions"

The ease with which an external form can be associated with an internal meaning in SMALLTALK means that many objects which are "cast in stone" in other languages can be defined and modified easily by anyone. Suppose only the Word and List operations are found in the language, then Numbers can be described in terms very similar to that of "schoolchild" arithmetic as shown below.

There are many ways to accomplish arithmetic; the example deliberately mimics the use of a "plus table" for single digits, the carry rule, and special cases involving 0, which you already know from school.

```
•PlusTable ←   •¦( 0  1  2  3  4  5  6  7  8  9 )
                ¦( 1  2  3  4  5  6  7  8  9 10 )
                ¦( 2  3  4  5  6  7  8  9 10 11 )
                ¦( 3  4  5  6  7  8  9 10 11 12 )
                ¦( 4  5  6  7  8  9 10 11 12 13 )
                ¦( 5  6  7  8  9 10 11 12 13 14 )
                ¦( 6  7  8  9 10 11 12 13 14 15 )
                ¦( 7  8  9 10 11 12 13 14 15 16 )
                ¦( 8  9 10 11 12 13 14 15 16 17 )
                ¦( 9 10 11 12 13 14 15 16 17 18 )
                •
```

To Number

```
  ◦← ⇒ ¦⦂A. ↑ self
```

"A new "Number" is created and declared by saying (for instance) •x ← Number ← 12345. The "•" EVALs its third argument, which calls "Number" which creates an instance, which looks for a "←", finds it, EVALs its next argument (which is a "Word" 12345), binds it to "A", and RETURNs the instance."

```
  ◦value ⇒ ¦↑ A
```

"The Word which is the value of "self" is RETURNed".

```
  ◦first ⇒ ¦↑ A•first.
```

" "first" of a "Number" is the same as "first" of the "Word" which is its value. The other "Word Parts" are done in a similar manner."

```
  ◦+ ⇒ ¦⦂B. ↑ ¦A•length = 1 and B•length = 1 ⇒ ¦PlusTable A B
      ¦        ¦
      ¦        ¦A•empty or B•empty ⇒ ¦A jointo B
      ¦        ¦
      ¦        ¦(A•butlast + B•butlast + carry A B )
      ¦        ¦        jointo ( A•last + B•last )•last
```

"This is a recursive definition which uses several cases to accomplish "+".

The first (A and B are both single digits) uses the childrens addition table selected by each of the numbers in turn to isolate the sum which is RETURNed.

The next case terminates the routine in the case where either or both of A,B are EMPTY. Remember that anything "jointo" EMPTY is thatthing. The last case is simply a statement of the goal, namely: the front digits of A and B are added to the carry found by adding the last digits of A and B, the result is joined to the single digit result of the sum of the last digits of A and B.

More branches of the conditional would be added to handle the Addition of negative numbers, etc."

```
ₒ- ⇒ ¦:B. ↑ "Subtraction is handled in a manner analogous
   ¦     to Addition".
```

```
ₒ= ⇒ ¦:B. ↑ A = B.value.
```

"A "Word operation" that is legal can easily be done."

```
ₒ< ⇒ ¦:B. ↑ if (B - A).first = ₒ- then EMPTY else self.
```

"Doing the definition this way allows x<y<z etc. to work properly. Note that "y<z" is done first and returns the value of "y" (if "true") to "x""

*****N O T E ! This semantic def of Complex is not completely edited !!

```
To Complex
    ₒ+ ⇒¦:value.complex ⇒¦↑ Complex ←
        ¦               ¦          re + value.re
        ¦               ¦          im + value.im
        ¦
        ¦value.fraction ⇒(↑ Complex ← re + value im)
        ¦↑ value G + self

    ₒ- ⇒¦:value.complex ⇒¦↑ Complex ←
        ¦               ¦          re - value.re
        ¦               ¦          im-value.im
          value.fraction  ⇒↑ Complex ← re - value    im
          otherwise       ⇒(↑  value G - self

     ₒ* ⇒ :value.complex ⇒ ↑ Complex ← (re * value.re  -  ¦im * value.im)
                                       (im * value.re  +  ¦re * value.im)
          value.fraction  ⇒ ↑ Complex ← re * value  im * ¦value
          otherwise       ⇒ ↑  value G * self

     ₒG ₒ+ ⇒ :value.fraction  ⇒ ↑ Complex ← re + value   im
     ₒG ₒ- ⇒ :value.fraction  ⇒ ↑ Complex ← value - re   ¦-im
     ₒG ₒ* ⇒ :value.fraction  ⇒ ↑ Complex ← re * value   im ¦* value
     ₒG ₒvalue  ⇒ ↑ Error"I don't know this operator" value .


     ₒre ⇒ ₒ← :value.fraction  ⇒ re ← value . ↑ self
           ₒₒ      ⇒ ↑ re
     ₒim ⇒ ₒ← :value.fraction  ⇒im ← value . ↑ self
            ₒₒ      ⇒ ↑ im
     ₒ←  ⇒ :re.fraction  ⇒:im.fraction  ⇒ ↑ self
     ₒcomplex  ⇒↑ true
    ¦ ⋄op :value  ⇒ ↑ value G :op  self
    ¦ ⋄op :value  ⇒ ↑ value G :op  self
          ◦◦  ⇒ ↑ self
```

```
Necessary Information about this paper.
   Latest revision: June 6, 1973

    (The permanent names of this file are
     SMALLTALK.DC.       ***
     SMALLTALK1.DC.
     SMALLTALK2.DC.
    Its latest incarnation will always be found on the
    Learning Research Group Demo Diskpack.

    The full structured index is found with each version.
    Look under the structure to discover what file to load.

    This file should be displayed using font SROMAN.FD.
    To print, edit with SMDELEG.FD and Write Translated,
    then print on XGP using font SMDELEG.XG)
```

SMALLTALK, a Model Building Language With Intensional Semantics

by
Alan C. Kay

Learning Research Group
Xerox Palo Alto Research Center

Abstract

SMALLTALK is a language which allows children (and adults) to build semantic models of their ideas in simple uncomplicated ways, and dynamically simulate them with respect to arbitrary environments.

Simplicity is achieved by having
a. only one kind of object in the language (a process) which can act like all other known computer objects,
b. a single uniform scheme for interobject communication, and,
c. an intensional semantics in which the meaning of an object is a part of the class to which an object belongs rather than dispersed through the system as part of more conventional extensional operations.

Benefits are the abilities to create new "functional", "data", "control", etc., entities without the usual problems associated with updating and coercion of generic functions.

****

Acknowledgements

Please see the end of the handbook for a fairly complete set of historical influences on SMALLTALK's design.

Introduction

SMALLTALK is built from a few simple, yet powerful, ideas.

First, SMALLTALK considers every OBJECT in its world to be an independant entity with local state and control. All distinction between "datalike" and "procedurelike" objects, such as exist in other programming languages, is thus removed. This includes "data", such as numbers, strings, arrays, lists, structures, etc.; "functions", such as 'factorial', 'plus', 'print', etc.; "control structures", such as conditional branches, repeats, recursion, and so on; "IO devices", such as 'files', 'the user', 'display and keyboard', etc.; all are treated alike because they ARE alike.

Next, all objects are composed of PARTS, even if they only contain themselves. The object can be thought of as a dynamic dictionary which contains all the relations and rules in which it can take part.

Third, objects can send and receive MESSAGEs to/from other objects. This may cause new objects to be created, altered, or even destroyed.

(Since there are no "special" objects, there is only one message protocol.)

Finally, each object is considered to be a member (or INSTANCE) of a CLASS, which is another object that contains the rules of behavior shared by all the members. Since each class has a class defining object, they are members of the class of class-defining-objects, as one might expect.

Messages

A message is a stream of zero or more symbols.

If the stream starts with an open parenthesis, its closing parenthesis absolutely terminates the stream.

An embedded "." at the same level will terminate the current message and will cause the message following it to be sent.

If the message is composed of partswhose termination is ambiguous, a "," can be used to clarify matters.

Sending is done from left to right using a very simple rule: control is passed immediately to the first object encountered in the stream, along with information about the context of the send. This is all the EVALuator does. The receiver may gather in the message in any way it chooses.

A common first object is an instance of the class "name" (as with a LISP atom, all of its members start with a letter and are composed of letters, digits , underscores, and other special characters).

The action of a name is to look itself up in the current environment/dictionary to see if it has a meaning (which is another object). If it does, that object is RETURNed by APPLYing it to the remainder of the message;--- And so it goes until the message is consumed.

A venerable example: factorial.

A message

```
factorial 3.
```

is sent in the following manner.

Control is passed to the name "factorial" which looks itself up in the current environment and finds another object as its value. The new object is a class defining object which contains the rules for all the instances of the class "factorial":

```
:n. ↑ if n = 0 then 1 else (n * factorial n - 1).
```

The action of the class defining object is to create a new instance of factorial and APPLY it to the message.

The ":" is a "receive" (or "input") object whose action is to EVALuate the input stream (in this case "3", whose value is "3") and then to make a new entry into the local environment to define the name (in this case "n"). After this a lookup of "n" will have the value "3".

The "↑" is a "send" (or "output") object which will APPLY the EVALuation of its argument to the remainder of the message found in the CALLER's object.

The next message is sent by finding "if" which tries to receive the message consisting of the EVALuation of "n=0".

Control is passed to "n".

It looks itself up and finds "3".

Control is passed to it.

"3" is an instance of the class number which has many

relations it can respond to.
"3" receives the next object (unevaluated) to see what it is. (It could be any of +, -, * /, < , > , etc.; in this case it is "=").
"3" wants now to evaluate the next part of the message in order to see whether to RETURN "true" or "false".

Control is passed to "0" which, as with "3", is an instance of class number, and thus shares the same relations.
So, it looks to its right to see if anything like +, -, *, etc., is there which it can respond to.
It finds only "then" for which it has no meaning.

So it RETURNs ITSELF to "3" which now has enough info to decide "not true"

which is RETURNed to "if" which decides not to evaluate the message following "then", but does try to evaluate the message following "else".

"n" looks itself up, finds the value "3"

which picks up the name "*" for which it has a meaning.
So "3" tries to evaluate the next part of its message "factorial n - 1)".

Control is passed to "factorial" which looks itself up and discovers (as before) a class-defining object with the rule:

```
:n. ↑ if n = 0 then 1 else (n *  factorial n - 1).
```

As before, a NEW instance is created which will try to evaluate the message "n - 1)" to get a new value for ":n".

"n" in the OLD environment looks itself up and discovers "3"

which looks to its right and finds "-" so it tries to evaluate the next object "1"

which which looks to its right and finds ")" (which terminates any message to "1") so it RETURNs ITSELF to "3"

which knows how to subtract "1"
which causes a new instance of class number to be produced for the result "2"

which is RETURNed to the ":" in the CURRENT instance of "factorial"
which will enter it as a value for "n" in the CURRENT environment.
And so it goes.

The preceding rather long winded explanation of a well known example illustrates a number of important points.

First, although the terminology seems to be more general than is needed, a simple program in SMALLTALK looks simple and can be discussed in simple terms.

Second, only one rule of correspondence is needed to link form and content. The evaluator ONLY needs to know how to pass control and context to an object. All other meanings are found distributed with the objects in the system. As shown, even such a seemingly primary act as creating a new instance is done by an object and thus can be changed at the user's whim.

Third, there are many cases where this generality of approach pays off handsomely. If we want to trace the activities of a name (such as "n" in instance 1) we need only create an object which can replace "3" as a meaning (so control will be passed to IT when "n" is touched), AND has a local entry of its own for "3" so that the meaning of "n" will not change with respect to its input/output characteristics. This means that an object can simulate any other object.

Fourth, all "relations" and "operators" (such as <, >, +, *, =, etc.) can be defined "intensionally" (or "intrinsically") as parts of an object or object class, rather than "extensionally" (or "extrinsically"), as is usually the case, as global functions.

In fact, "factorial" could have been defined this way as an intensional relation of a number. We might then have said "3! " and the class number would know what to do.

This means that the information pertaining to a class and what its members do need only be stored with the class. No global operations need to be updated. So, a class may be deleted without changing the rest of the world.

Also, this is a very convenient way to handle problems that arise from having multiple classes with operations: such as coercions between classes and the various senses of "fetch" and "store" ("←").

```
For instance, the message "a ← 3 + 1" means:
   pass control to "a" which will look itself up and
      pass control to the object it finds
         which can gather the rest of the message as it
         pleases.
         It can look to see if the next name is a "←",
         if so, it can EVALuate "3 + 1" and decide how
         to store it.
So "b 1 ← 81" , if "b" were an instance of an array,
could mean
   'store 81 in the 1st position'; or
if "b" were an instance of a hash table routine, could
mean
   'associate the hash of "1" with "81" in some  way',
   etc.
```

The problem of coercions will be discused a bit further on.

Fifth, instances may be EVALuated "concurrently" using the very same EVALuation strategy. Here, the generality of message send/receive becomes much more important.

Class Definitions Already in SMALLTALK

<See SMALLTALK1.DC for this branch>

Some SMALLTALK Programs
<See SMALLTALK2.DC for this branch>

Necessary Information about this paper.
Latest revision: June 6, 1973

(The permanent names of this file are
SMALLTALK.DC.
SMALLTALK1.DC. ***
SMALLTALK2.DC.
Its latest incarnation will always be found on the
Learning Research Group Demo Diskpack.

The full structured index is found with each version.
Look under the structure to discover what file to load.

This file should be displayed using font SROMAN.FD.
To print, edit with SMDELEG.FD and Write Translated,
then print on XGP using font SMDELEG.XG)

SMALLTALK, a Model Building Language
With Intensional Semantics

by
Alan C. Kay

Learning Research Group
Xerox Palo Alto Research Center

Abstract

<See File SMALLTALK.DC for this branch>

Acknowledgements

<See File SMALLTALK.DC for this branch>

Introduction

<See File SMALLTALK.DC for this branch>

Messages

<See File SMALLTALK.DC for this branch>

CAVEAT LEGATOR !!!

WARNING ---The notation and defs in this section are not completely re-edited from an earlier version and, hence, may not be entirely consistant. See informal.dc for a more consistant set of programs.

Class Definitions Already in SMALLTALK

SMALLTALK is supplied with many useful classes, including quite a few found in one way or another in other programming languages.

These definitions are written in SMALLTALK as though they were not primitive objects. In some cases (such as the definition of "if") a primitive may be used to describe itself---which causes some obscurity.

Evaluation simply proceeds from left to right.

A notation convention is "margin lists". Regular parentheses "(" ")" and margin parentheses <ctl>(and <ctl>) have the same meaning. The margin parens print their contents quite differently however. An open MP prints as ¦ (and sets a new margin at that spot). <linefeeds> in the list will return to the margin position an print another ¦. When the close MP is found, the next token is examined for a <lf>. If it is, printing resumes at the next outer set of MP's. Otherwise the remainder of the line is printed under the final ¦.

Input and Output Objects

INPUTS

: Input a Value

followed by a name will evaluate the input stream to produce a new object which will be bound to the name.

This is exactly the same as LOGO.

Example; :value
will bind the result of evaluating the input stream to "value"

⦂ Input "isolated" evaluation of message
followed by a name will pick up the next object in the input stream and evaluate it without giving it a message.

☞ Input an Object
followed by a <name> will not evaluate the input stream but will bind the next object there to the <name>.

There is no equivalent for this in LISP or LOGO, it acts as though the next input object were quoted.

Example; ☞value
will bind the next input object to "value"

👁 Check Input for a Token

followed by a <name> will check the input stream to see if an identical <name> is there. No evaluation will take place. The Input Stream Pointer (or Program Counter) will NOT be advanced if the match fails. If the match succeeds, the ISP will be advanced to the next position.

This is used frequently to check for "operator" tokens such as +, *, and ←.

Example; ʘ+ will check the input stream for a "+" and will return TRUE if successful

; Input Literal Stream

followed by a <name> will bind a reference to the Input Stream at the current point.

This is equivalent to FEXPR in LISP 1.5 or NLAMBDA in BBN-LISP.

Example; ;value will bind "value" to the input stream. EVALuation of this fragment may be delayed until later.

<Other Input Objects>

will be mentioned here in a later version of this memo. An object to EVALuate a sequence of the input stream (like EVLIST in LISP) will probably be included at the very least.

OUTPUTS

APPLY-SEND a value.

"f a b" will send "f" the message "a b" as explained previously.

! APPLY-RETURN a value.

This output object is used when when a subroutine control structure and message passing discipline is desired. Its single argument is EVALuated in the CURRENT environment and then APPLYed to the program stream of the CALLER process to which CONTROL also is RETURNed.
When used in "left nested" argument gathering (for example x.first.last or (A + B) + C), APPLY-RETURN will continue the evaluation process.

↑ PASSIVE-RETURN a value.
The single argument is evaluated in the CURRENT environment and RETURNed to the CALLER along with CONTROL.
PASSIVE-RETURN is similar to OUTPUT in LOGO or RETURN in LISP.

⇑ GENERAL-RETURN a value.
⇑ value process
is the form.
⇑ value caller.
is the same as PASSIVE-RETURN.

```
⇑ (apply value message) caller.
   is the same as ACTIVE-RETURN.
```

<Other Output Objects>

will be explained later.

Defining a Class (Function)

There are many ways to define a class depending on how much the user wants to know about the language and how much control he desires to have over the format of the INSTANCE of a definition. For now we will only be concerned with semantic notions (which also require the least amount of explanation to all concerned).

LOGO/SIMULA/FLEX Fashion

"To" will define classes of roughly the power of SIMULA or FLEX which include such things as function, process, and structure definitions in other languages.

```
To ☞name ☞body
```

"As shown, "To" takes the first object in the message stream unEVALuated to be the name of the class. All of the rest of the input stream is a structure which is taken to be the code body of the class. A member of the class CLASS is INSTANTIATED and bound to the name. When control is later passed to the name a new instance of the class will be created and run."

Examples;

```
To factorial ⁝:n.
             ⁝↑ if n=0 then 1 else (n*factorial n-1).
```

This looks a lot like LOGO (intentionally) except that the input variable ":n" is not part of the heading (as in LOGO), but is part of the "body". This reflects the fact that input objects act like functions and thus can be used anywhere in a program. When a "function" is instantiated, the first thing that is done in most languages is to bind the arguments to a new set of names. The very same effect is achieved in SMALLTALK when the "evaluating input object", ":" , is used in the first set of expressions.

Conventional Class Definition

"To" as shown above, was included mainly for people familiar with LOGO and LISP. SMALLTALK really treats "class objects" like any other object. That is, any object is a member of a class---so an object which creates a class is a member of class CLASS.
This means that a more general (and more conventional) way to define factorial would be to say

```
*factorial ← class*(If :n = 0 then 1 else (n * factorial n
- 1).
```

or perhaps

```
        ⋄factorial ← class⋄⁝ :n.
                           ⁝ If n = 0 then 1 else ⁝n *
                           ⁝                      ⁝factorial n - 1)
                           .
```

using the ‹margin list› convention. One could even say

```
        ⋄var ← ⋄n.
        ⋄factorial ← class
                  ⋄(:)⁝ var ⁝ ⋄(= 0 then 1 else)
                  ⁝(var⁝ ⋄(* factorial n - 1)).
```

where "⁝" means "append" pretty much in the LISP sense.

Total Control of the Instance
***for bit pickers, more on this later this summer.

Control (and State changing, etc.)

```
To If :exp.        ***** This has changed !!
    !exp.
```

""If" is really just a dummy which computes a value to be APPLYed to "then" or "⇒". This means that "TRUE"ness and "FALSE"ness are properties of objects. This allows us to consider all legal numbers as TRUE, if we wish. A class with one instance EMPTY is provided to handle "FALSE" cases.

```
To ⋄
   ⋄name ⁝ₒ← ⇒ (:exp. ↑ exp)
         ⁝   "lookup the name in current environment (if not
         ⁝   there, enter it as most global) and replace BINDING
         ⁝   with value of "exp" ".
         ⁝
         ⁝! name.
```

"note that the value of the expression on the right "exp" is RETURNed when a rebind is attempted, but when used as QUOTE, it is the name which is RETURNed."

```
To Eval  :exp :globalenv :return :msg.
```

"There are many ways to EVALuate expressions in Smalltalk. This one allows the user to set up an arbitrary environment for free variable fetches, an arbtrary RETURN process, and an arbitrary MESSAGE environment."Eval" is included here since it is very frequently used in definitions of new control primitives".

```
To Repeat ⋄Loopexp.
   Code repeat.
   Eval Loopexp ⁝global ⁝self EMPTY.
   Code again.
```

"Repeat EVALs its loop expression in the context of its caller."

```
To Again
```

"RETURNs control to the caller of its caller--i.e. to a looping control primitive of some kind such as "Repeat" which can decide what to do next".

```
        To Done
            "RETURNs control to the caller of (the caller of its
            caller)--to one level further out than a looping control
            primitive. This automatically terminates the loop.
            Eventually "Done" will have an optional argument for
            passing the RESULT of the loop back".


        To Create
            "Reschedule caller to be run instead of waiting for a
            subroutine  RETURN".
          :call.
            "This causes an evaluation of the argument. So it will also
            be running".

              "As seen, "Create" causes a parallel fork in control.
              Actually, this is what happens naturally in
              SMALLTALK---the default message discipline is
              deliberately limited to a subroutine "wait for reply"
              protocol. "Create" simply prevents the caller from being
              passivated".

    To Word

        ◦Explain⇒
          ↑"Words are like LISP atoms or ALGOL identifiers. Their basic
          operations have to do with assembly and disassembly of their
          internal structures.
            Words also have a special meaning in the context of
            evaluation. An unquoted instance of a word will be looked
            up (look itself up) when encounted by the EVALuator. So
            cat.first  means  "look up the most local binding of the
            variable "cat" and APPLY it to .first". But .cat.first
            means " call routine "." which  RETURNs the word "cat",
            which is APPLYed to .first, which, as  seen below, will
            RETURN "c" ".
          Numbers are words also, but have many additional operations
          having to do with arithmetic and so are defined as a separate
          class.".

        ◦← ⇒ :value.word ⇒
          ↑self.
            "..."
        ◦first ⇒
          ↑"the first character of the printname of the word".

        ◦f ⇒
          ↑"same as "first"".

        ◦last ⇒
          ↑"...the last character of the printname of the word"

        ◦l ⇒
          ↑"...the same result as for "last". This is just an
          abbreviation."
        ◦butfirst ⇒
          ↑"Somehow return all but the first character of the string
          representation of the word."

        ◦bf ⇒
          ↑"...same as butfirst."

        ◦butlast ⇒
          ↑"Somehow return all but the last character of the string
          representation of the word."

       ₙbl ⇒
          ↑"...same as butlast."

       ₙjoin ⇒ :value1⋆word? ⇒
          ↑"This is roughly equivalent to the "cons" of LISP. The word
          will be connected to the list in "value1", and a new list
          reference will be returned."

       ₙwjoin ⇒:value1⋆word? ⇒
          ↑"This is roughly equivalent to concatenate in SNOBOL. The
          printname of the two words are joined together to produce a
          new word which is returned. ⋆cat wjoin ⋆dog     produces
          ⋆catdog."
       ₙword? ⇒
          ↑value.

       ₙempty? ⇒
          ↑EMPTY.

       ₙlength ⇒
          ↑"Somehow calculate the length (in characters) of the
          number     (including "-" and ".")   ."

       ₙprint ⇒
          ↑"Return a string representation of the object which may be
          displayed. Each class which has instances which have a
          meaningful visual representation will have a meaning for
          ⋆print. This is much simpler than having to inform a global
          print routine about the format of each new class."


    To Number

       ₙExplain⇒
          ↑"Numbers work in a very intuitive way. The READ program
          recognizes  number literals and creates instances for them in
          storage.The bits  that represent the particular instance of a
          number are stored in  the variable "value" and can be changed
          by assignment as shown.  This might be illegal if it is
          decided that numbers are unique  atoms. The opposite is
          assumed here."

       ₙ← ⇒ :value⋆number?⇒
          ↑ self.
             If a "↑" is recognized in the input stream, what follows is
              evaluated and bound to "value" which is applied to number?
            which  returns TRUE if it is. The actual value of the
            number object itself has been changed so that other objects
            which have pointers to "self" will feel the change. This
            might be made illegal.

       ₙfirst ⇒
          ↑"Somehow return the first character of the number which is
          "-" if negative, is "." if between 0 and 1, and a digit from 0
          to 9 otherwise. It may be reasonable to calculate this value
          rather than keep a string representation of the number
          around."

       ₙf ⇒
          ↑"...the same result as for "first". This is just an
          abbreviation."
```

```
◦last ⇒
   ↑"Somehow return the last character of the number which is
   "." if greater than 1 and known inexactly, and a digit from Ø
   to 9 otherwise. It may be reasonable to calculate this value
   rather than keep a string representation of the number
   around."

◦l ⇒
   ↑"...the same result as for "last". This is just an
   abbreviation."

◦butfirst ⇒
   ↑"Somehow return all but the first character of the string
   representation of the number."

◦bf ⇒
   ↑"...same as butfirst."

◦butlast ⇒
   ↑"Somehow return all but the last character of the string
   representation of the number."

◦bl ⇒
   ↑"...same as butlast."

◦join ⇒ :value1.word? ⇒
   ↑"This is roughly equivalent to the "cons" of LISP. The word
   will be connected to the list in "value1", and a new list
   reference will  be returned."

◦wjoin ⇒:value1.word? ⇒
   ↑"This is roughly equivalent to concatenate in SNOBOL. The
   printname of the two words are joined together to produce a
   new word which is returned. .cat wjoin .dog     produces
   .catdog."

◦number? ⇒
   ↑value.
   "Anything not EMPTY will act as TRUE."

◦word? ⇒
   ↑value.

◦empty? ⇒
   ↑EMPTY.

◦length ⇒
   ↑"Somehow calculate the length (in characters) of the
   number     (including "-" and ".")  ."

◦print ⇒
   ↑"Return a string representation of the object which may be
   displayed. Each class which has instances which have a
   meaningful visual representation will have a meaning for
   .print. This is much simpler than having to inform a global
   print routine about the format of each new class."


◦= ⇒ :value1.number? ⇒
   ↑"value if value and value1 are numerically EQUAL, otherwise
   EMPTY.  Note that this allows "a=b=c" to work correctly."

◦≠ ⇒ :value1.number? ⇒
   ↑"EMPTY if value and value1 are not numerically EQUAL,
         otherwise  value. Note that this allows "a≠b≠c" to work
         correctly."

      ₒ< ⇒ :value1.number? ⇒
         ↑"value if value is numerically less than value1, otherwise
         EMPTY.  Note that this allows "a<b<c" to work correctly."

      ₒ> ⇒ :value1.number? ⇒
         ↑"value if value is numerically greater than value1, otherwise
          EMPTY.. Note that this allows "a>b>c" to work correctly."

      ₒ+ ⇒:value1.number? ⇒
         ↑"value added to value1."

      ₒ- ⇒:value1.number? ⇒
         ↑"value1 subtracted from value."

      ₒ* ⇒ :number? ⇒
         ↑"value multiplied by value1."

      ₒ/ ⇒ :value1.number? ⇒
         ↑"value divided by value1."

      ₒmod ⇒ :value1.number? ⇒
         ↑"value modulo value1."

      ₒip ⇒
         ↑"...the integer part of value."

      ₒfp ⇒
         ↑"...the fractional part of value."

      ₒexp ⇒
         ↑"...the exponent (to the base 10) of value."

      ₒmag ⇒
         ↑ if  value < 0 then (0 - value) else  value.

      <other numeric functions which are stored as attributes>
      sin, cos, other trig functions etc.

   To List
      ₒExplain
      ₒfirst ⇒ ₒ← ⇒ :value.list ⇒
                       value.word? ⇒
      ₒf ⇒ ₒ← ⇒ :value.list ⇒
                       value.word? ⇒
      ₒlast ⇒ ₒ← ⇒ :value.list ⇒
                       value.word? ⇒
      ₒl ⇒ ₒ← ⇒ :value.list ⇒
                       value.word? ⇒
      ₒbutfirst ⇒ ₒ← ⇒ :value.list ⇒
                       value.word? ⇒
      ₒbf ⇒ ₒ← ⇒ :value.list ⇒
                       value.word? ⇒
      ₒbutlast ⇒ ₒ← ⇒ :value.list ⇒
                       value.word? ⇒
      ₒbl ⇒ ₒ← ⇒ :value.list ⇒
                       value.word? ⇒
      ₒjoin ⇒
      ₒ! ⇒
      ₒ¦  ⇒
      ₒsentence? ⇒
      ₒlist? ⇒
      ₒempty? ⇒
```

```
    ◦length ⇒
    ◦print
    ◦= ⇒ :value.list ⇒
    ◦≠ ⇒ :value.list ⇒
    ◦< ⇒ :value.list ⇒
    ◦> ⇒:value.list ⇒
    ◦makeword ⇒
```

To String
****This may get filled in sometime

Position

Here are a set of useful operations for manipulating two-dimensional space. The convention is adopted that "posx" and "posy" will refer to position state, and "heading" will refer to direction state. The programs are written so that the most local occurance of these variables in the dynamic environment will be updated. See the program "Spacevehicle"(on informal.dc) for a simple example.

```
To Forward
    ¦:distance.
    ¦posx ← posx + distance * heading.cos.
    ¦posy ← posy + distance * heading.sin.


To Right
    ¦:angle.
    ¦heading ← (heading - angle) mod 360.


To Left
    ¦:angle.
    ¦heading ← (heading +  angle) mod 360.
```

Output   (to displays, music, turtles, etc.)

```
To Show
    ¦:picture.
```

"This comprehensive routine allows the picture to be EVALed and then copies the picture information into the display area using either the dynamically available variables "posx", posy", "heading", if its own bindings for these parameters are EMPTY.

Some SMALLTALK Programs
   <See INFORMAL.DC for Program Examples>

```
This file is called SMSEMANTICS.DC and contains a semantic description of
SMALLTALK written in itself.
This version was last changed on June 10, 1973.
Use font SMDELEG.FD     translate file   smsm
```

SMALLTALK and its Semantics
    by
Alan Kay

WARNING!! This is an unchecked version done simply to try it out for basic taste and compactness.

```
.To ← class¦.Do ← .¦◦name ◦actions.
               ¦         ¦↑ Find (name) in CALLER ← class¦.Do ← activity


To .  ¦◦name ¦◦← ⇒ ¦:exp.
      ¦      ¦      ¦Find (name) in CALLER ← exp.
      ¦      ¦!name


To Find ¦:name ¦◦in ⇒ ¦:context
        ¦      ¦.context ← CALLER
        ¦             .
        ¦Repeat
        ¦   ¦context.table name  OR  context.table.global.empty?
        ¦   ¦           ⇒¦◦← ⇒ ¦:exp. context.table name ← exp. Done
        ¦   ¦.context ← context.table.global
        ¦    .
        ¦↑ context.table name


To List ¦◦← ⇒ ¦:first :rest. ↑self
        ¦
        ¦◦first ⇒ ¦◦← ⇒ ¦:first. ↑self
        ¦         ¦!first
        ¦
        ¦◦rest ⇒ ¦◦← ⇒ ¦:rest. ↑self
        ¦        ¦!rest
        ¦
        ¦◦length ⇒ ¦!¦first=NIL ⇒ ¦0
        ¦          ¦ ¦1+rest.length
        ¦
        ¦◦print ⇒ ¦!"(" ◊ first.print ◊" "◊ rest.print ◊")"
        ¦
        ¦◦list? ⇒ ¦!self
        ¦
        ¦◦eval ⇒ ¦Repeat
        ¦        ¦   ¦first = ")" ⇒ ¦Done
        ¦        ¦   ¦first = "."  ⇒ ¦.value ← rest◦eval
        ¦        ¦   ¦.value ← first.eval
        ¦        ¦!value
****check for VOID message


To Repeat ¦◦program.
          ¦CODEFOR ¦Repeat ¦ clause .eval global message self


To Again ¦⇑ EMPTY CALLER.CALLER


To Done ¦:value. ⇑ value CALLER.CALLER.CALLER
To If ¦:exp ⇒ ¦ₒthen :exp ¦ₒelse ⇒ ¦◇. ↑exp
      ¦             ¦                ¦↑exp
      ¦             ¦error "I can't find a "then""
      ¦
      ¦ₒthen ◇. ¦ₒelse ⇒ ¦:exp. ↑exp
      ¦         ¦↑EMPTY


To User ¦Repeat
        ¦   ¦Display Read•eval•print


To ◇¦self•table•name ← message•table•pc•first.
    ¦message•table•pc ← message•table•PC•rest.
    ¦message•table name ← message•table•message•table•PC•first.
    ¦message•table•message•table•PC ← message•table•message•table•PC•rest.
    ¦! name.


To : ¦◇name.
     ¦! message•table•name ←
     ¦        message•table•message•table•PC•first •eval message•message


To ₒ ¦◇token ≠ message•table•PC•first ⇒ ¦! EMPTY
     ¦message•table•PC ← message•table•PC•rest


To ⇒ ¦:clause. ⇑ clause CALLER•CALLER•CALLER

To EMPTY ¦ₒ◇ ⇒ ¦◇. ⇑ self CALLER•CALLER
         ¦ₒempty? ⇒ ¦! •TRUE
         ¦! self


To Apply ¦ :t :g :c :m.
         ¦ (t ← •global g  •caller c  •message m ) •eval

To ⇑ ¦:value :destination.
     ¦•CALLER ← :destination.
     ¦↑ :value


To ↑
   ¦⇑ :value CALLER•CALLER


To Remember ¦ₒ← ⇒ ¦Repeat
            ¦      ¦   ¦ₒEMPTY ⇒ ¦↑self
            ¦      ¦   ¦self :name ← :value
            ¦
            ¦ₒcopy ⇒ ¦! CODEFOR "somehow copy the table"
            ¦
            ¦ₒeval ⇒ ¦"Do something or other"
            ¦
            ¦:name ¦ₒ← ⇒ ¦:value.
            ¦      ¦     ¦CODEFOR "associate name and value somehow"
            ¦      ¦     ¦↑value
            ¦      ¦
            ¦      ¦! CODEFOR "Get the value associated with the name"
```

```
To class ¦▫bindings.
         ¦↑ instantiate ¦ Remember ← •class •class
         ¦              ¦             •global global
         ¦              ¦             •caller self          ¦          ¦
           •message message
         ¦              ¦             •PC  bindings
         ¦              •eval

To instantiate ¦:classdef.
               ¦Repeat
               ¦   ¦Pause.
               ¦   ¦¦classdef•copy ← •class classdef
               ¦   ¦¦                •global global
               ¦   ¦¦                •caller caller
               ¦   ¦¦                •message message
               ¦   ¦¦                •PC  classdef•DO
               ¦   ¦•eval

To Word
   ¦▫← ⇒ ¦:first :rest.
   ¦     ¦•rest ← Word ← first•butfirst rest.
   ¦     ¦•first ← first•first.
   ¦     ¦↑ self
   ¦
   ¦▫first ⇒ ¦▫← ⇒ ¦:first•character ⇒ ¦↑first
   ¦         ¦     ¦error"input is not a character"
   ¦         ¦!first
   ¦
   ¦▫rest ⇒ ¦▫← ⇒ ¦:rest•character ⇒ ¦↑rest
   ¦        ¦     ¦Error"Input is not a word"
   ¦        ¦!rest
   ¦
   ¦▫length ⇒ ¦! rest = NULL ⇒ 1
   ¦          ¦ 1 + rest•length
   ¦
   ¦▫print ⇒ ¦↑ first•print. rest•print
   ¦
   ¦▫word? ⇒ ¦! self
   ¦
   ¦▫= ⇒ ¦:value. ↑ (first = value•first) AND next = value•next
   ¦
   ¦▫eval ⇒ ¦•env ← global.
   ¦        ¦Repeat
   ¦        ¦   ¦env•empty? ⇒ ¦↑EMPTY
   ¦        ¦   ¦•temp ← env•table self ⇒ ¦↑ apply temp global caller
   ¦        ¦   ¦                         ¦        message
   ¦        ¦   ¦•env ← global•table•global
```

ACKNOW.DC smdeleg.fd ack

ACKNOWLEDGEMENTS

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

The DYNABOOK(ka 71) is a godchild of Wes Clark's LINC(cl 6***) and a lineal descendant of the FLEX Machine(ka 67,68,69).

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

****

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

The Burroughs B5000(ba 61) (1960) had many design ideas well in advance of its time (and still not generally appreciated): compact "addressless" code; a uniform semantics for names (the PRT), automatic coprocesses, "capability" protection (also by the PRT), virtual segmented memory; the ability to call a subroutine from "either side" of the assignment arrow; etc.

The notions of code as a data structure; intensional properties of names (property lists of attribute:value pairs on atoms); evaluation with respect to arbitrary environments; etc., are found in LISP, probably the greatest single design for a programming language yet to appear. SMALLTALK is definitely "LISPlike".

The SIMULA's ('65 and '67) combined Conway's notions of software coroutines (1963-hardware versions had appeared in the B5000 3 years earlier), ALGOL-60, and Hoare's ideas about record classes(196**) into an epistemology that allowed a class to have any number of parallel instantiations (or activation records) containing local state including a separate program counter. Most of the operations for a SIMULA '67 class are held intrinsically as procedures local to the class definition.

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

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

An extemporaneous talk by R.S. Barton at Alta Ski Lodge(1971) about computers as communications devices and how everything one does can easily be portrayed as sending messages to and fro, was the real genesis of the current version of SMALLTALK.

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

Problems discovered years ago in "lefthand calls" prompted SMALLTALK to make "store" intensional--i.e. a ← b, means "call "a" with a message consisting of the token "←" and "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 evalualtor for SMALLTALK and the idea that "+", "-", etc., all 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". The message ideas allow messages to have a wide range of form since a message can be received incrementally.

"Control of control" allows control structures to be defined. The language SMALLTALK itself thus avoid "primitives" such as "loop ol", 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.

microPLANNER's main influence was negative in that it convinced us finally that backtracking is not the way to approach problem solving. Instead, we prefer "trial evaluation" where a "straw process" is run in a "straw environment" as a coprocess and constantly sends messages as to how badly it gets creamed back to its originator. If it perishes, its environment is just discarded rather than backtracked. (A germ of this idea is found in Fisher's thesis).

The fine idea of microPLANNER ("pattern directed invocation"---I call it "call by desire") does not appear as a primitive in the current version of SMALLTALK (it was in ST-1971) but may be easily added in just the way a particular user desires.

****

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

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

A talk on SMALLTALK was given at the AI lab at MIT (Nov 1972) which discussed the process structure and the new, intensional, way to look at properties, messages, and "infix operators").

****

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

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

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

Line Graphics and the hand-character recognizer were done by John Shoch. Ted Kaehler did the "scope turtle" on the ALTO. Bob Flegal(CSL) did the color turtle on the Graphics Group video buffer (which was designed and built by Dick Shoup(CSL)).

"Music" was designed and implemented by Alan Kay. Barbara Deutsch wrote the program to REGISTER combinations of timbre files. Peter Deutsch(CSL,PARC) designed and wrote a translator for compact musical "score" notation. Steve Saunders improved most of these programs.

Diana Merry wrote auxilary systems programs and a very nice "software character generator" and text scroller for the ALTO.

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

****

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

Smalltalk Class Outline

March 5-9, 1979
Adele Goldberg
co-featuring Dave Robson as TA

Monday March 5

Theme What is object-oriented programming?

Reference Draft-0 of Chapter III *Smalltalk: Dreams and Schemes*

basic data structure: object
state
behavior

basic processing: message sending

Smalltalk's version
conceptual object: internal and external view
conceptual class, instances, subclasses

message determination

Example environments to organize into objects and to specify the message protocol of the objects
what are the objects?
what are their protocols?

[door, amusement park, inventory system]

Example class definition in Smalltalk: class Part

Off-line assignment

choose one from each column and specify the objects and their protocols

bank data base
integrated circuit
animated movie
telephone network
technician's lab

text editor
calculator
musical performance
computer
PARC

On-line assignment [see handout #1]

using the Smalltalk user interface, learn to use a dialog window, text edit in a code window (read Handout #1), and print your code window. Learn to use the document editor (see Handout #7)--using it to do your off-line assignment if you choose.

Tuesday March 6

Theme class organization, messages and methods

Reference Draft-0 Chapter IV *Smalltalk: Dreams and Schemes*

state information for a class

methods

pseudo-variables self and super

examples

HashSet, Dictionary, Inventory System (the class Inventory)

Syntax of Smalltalk-76

special consideration for initializing class, pool, and instance variables

Classes in the Basic System

Off-line assignment (but actually done on-line) [see handout #2]

goal: do some browsing on-line to find the indicated definitions; watch out for use of self and super. Try to read the definitions of Stream, Point, and Rectangle. Read the class definitions for HashSet and Dictionary to determine what they can do. How do you create a new instance of the class Dictionary? An instance of the class Dictionary understands a message of the form insert: name with: value. In executing the method associated with this message, a number of messages to self are sent. Which ones are they and who holds the message dictionary in which each is found?

On-line assignment [see handout #3]

implement a class that represents a data structure for your laboratory inventory.

Wednesday March 7

SubTheme The nitty gritty folklore of files and printing [see handout #4]

Theme Message protocol as a command language; subclassing--why do it, why not; browsing and reading class definitions already available in Smalltalk-76

Examples

Redoing class Inventory as a subclass of class Dictionary

Putting text up on the screen (Textframe)

Thursday March 8

Theme adding graphics

Examples

Making a sketch (BitRect)
Making a movie

Friday March 9

Theme windows and menus

Examples

Making a PanedWindow and a Menu

Two weeks after this class week (Monday March 19?)

Theme process scheduling

(see handout #6)

Example

Scheduling an Inventory Window

"Assume we have class Part with subclasses Diode and Transistor defined. The subclasses each have partNo as a field."

```
diode inspect.
diode ← Diode new init: ('1n914' unique) with: 200.
diode order: 5 on: (Date new day: 27 month: 2 year: 79).
diode onHand print.
diode orderWaiting.
diode needed.
diode howManyOrdered.
diode print.
```

"Here is a dictionary to be used for storing the parts"

```
techlab ← Dictionary init: 5.
techlab insert: ↪D1n56
        with: (Diode new init: ↪D1n56 with: 200).
techlab insert: ↪T2n45
        with: (Transistor new init: ↪T2n45 with: 150).
techlab lookup: ↪D1n56.
```

"To get to a part and send it any message, you have to do it indirectly like this"

```
(techlab lookup: ↪D1n56) onHand.
(techlab lookup: ↪T2n45) needed.
(techlab lookup: ↪D1n56) howManyOrdered.
```

techlab

D1n56	a Diode
T2n45	a Transistor

"but you can not send techlab messages such as list all parts and the number on hand. It is necessary to define an object Inventory that can receive the message. Suppose we did"

```
techlab ← Inventory init: 5.   "notice the similarity to Dictionary"
techlab allPartsOnHand.        "list all parts and their number
onHand"
```

result for the above dictionary would be

```
techlab
   D1n56      200
   T2n45      150
```

DOCUMENTS IN SMALLTALK

Overview

The Smalltalk DocumentEditor is a facility for creating, editing and composing documents containing both textual and graphical elements. Documents created using this system may contain BitImages (bitmaps), TextImages (paragraphs),BorderedTextImages and Headings. The spatial relationships among the text-graphic entities (BitImages, TextImages, BorderedTextImages, Headings) in a document is totally arbitrary and must be specified by the user. Completed documents are stored in Press File format allowing printing and retrieval from the same file.

Philosophy and basic ideas about Documents

A Smalltalk Document is a collection of entities (BitImages, TextImages, . . . etc) arbitrarily positioned within it. Each entity type has its own editor that is idiomatic to the entity being edited. For example, TextImages are edited with the standard Smalltalk text editor while BitImages are created and modified using the Smalltalk Toolbox Picture Editor. The message protocol that an entity must obey to be manipulated within a Document is reasonably simple, allowing future inclusion of new entities like barcharts and curve graphs.

Creating a new Document

To get started type: ***DocumentEditor new defaultdocument: 'xxx'*** in your workspace and execute it. When the OriginCursor appears specify a Smalltalk Window in the standard manner, except that the width is constrained to extend to the right edge of the document. The Window you specify is a DocumentEditor that is a view on a simple Document that contains: a TextImage, a BitImage, a BorderedTextImage and a Heading. This DocumentEditor (Window) obeys the standard Smalltalk Window protocol and has '***xxx***' as its name.

Editing Documents

SELECTION OF ENTITIES IN DOCUMENTS:

The mouse and redbug used for selection of entities within the Document. To select an entity or entities: specify a rectangle containing them by:

1. depressing redbug
2. while keeping redbug depressed move the mouse until the

blinking rectangle is the right size.

3. release redbug.

The entities contained in the rectangle will be highlited.

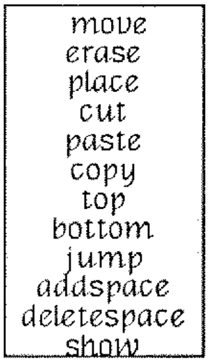
For the rest of this memo *selection* will mean the entity or eitities currently selected (highlited) in the document.

GRIDDING CONTROL IN DOCUMENTS:

The x-grid module is set by typing a lower case "x" followed by a typing the number of alto screen bits desired for the x-grid followed by a "↵". The y-grid module is set by typing a lower case "y" followed by a typing the number of alto screen bits desired for the y-grid followed by a "↵". The gridding specified in this manner applies to all the document level editing operations that are explained below.

DOCUMENT LEVEL COMMANDS:

Document level commands are invoked using the yellowbug menu which looks like:



Yellowbug Commands.

move: Repositions *selection* within the DocumentEditor Window. When the OriginCursor appears,depressing redbug will cause the selection to follow the mouse until redbug is released. The position of the *selection* is determined by the setting of the x and y gridding.

erase: Deletes *selection* from the document and saves it in a Scrap (like the Smalltalk paragraph editor). The space occupied by *selection* is left as white space in the document.

place: Places the contents of the Scrap in the document. When the OriginCursor appears depressing redbug will cause the contents of the Scrap to follow the mouse until redbug is released. As in **move,** the position of the entity being placed is affected by the x and y gridding.

cut: Like erase except the space occupied by *selection* is removed by moving all entities below *selection* toward the top of the document by the height of *selection.*

paste: Similar to **place** except all entities below the top of the final placement of the Scrap are moved down in the document by the height of the entity in the Scrap.

copy: Creates a copy of ***selection*** and saves it in the Scrap.

bottom: Used for scrolling the document in the DocumentEditor Window. This command moves ***selection*** to the bottom of the DocumentEditor Window. If there is no selection, this command will cause a jump to the bottom of the document.

top: Used for scrolling the document in the DocumentEditor Window. This command moves ***selection*** to the top of the DocumentEditor Window. If there is no selection, this command will cause a jump to the top of the document.

jump: Used for scrolling the document in the DocumentEditor Window. When the JumpCursor appears it indicates the current position of the Window on the document. Depressing redbug specifies the new position of the Window for viewing the document. The top of the DocumentEditor Window represents the top of the document and the bottom of the DocumentEditor Window represents the bottom of the document.

addspace: Adds space (in the y-direction) between entities in the document. The amount and position of space added is indicated by specifying a rectangle (origin and corner) with the mouse. Space equal to the height of the rectangle is added to all entities below the top of the rectangle.

deletespace: Deletes space (in the y-direction) between entities in the document. Similar to addspace except space is deleted.

show: Redisplays the DocumentEditor Window. This command is neccessary because some of the commands mess up the presentation of the document in the Window.

Printing and Filing Documents

Documents can be printed by executing "print" with the standard Smalltalk Window bluebug menu. A PressFile (named 'xxx.document') is generated from the document and is sent to Menlo. This pressfile can later be recalled and edited by executing: ***DocumentEditor new init: (Document new fromPress: 'xxx.document'')***.

The pagination algorithm currently implimented is unsophisticated. It will split a TextImage across page boundries but not Headings or BitImages. If you avoid fancy layout near page boundaries you may create and print very complex documents. Much work remains to be done in this area.

Since we do not (except for Cream10 and Cream12) have coordinated fonts it is impossible to reproduce exact screen positions for text on the printed page. The printing algorithm attempts to keep the relative positions of entities on the page in proportion to their positions on the Alto screen. You will probably be satisfied most of the time with the placement of entities on the printed page.

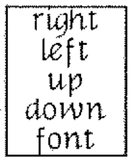
Editing TextImages, BitImages and Headings

To invoke the idiomatic editor for an entity simply move the cursor over the entity and click redbug. When you are finished editing an entity move the cursor outside it and depress any mouse button; this will return control to the level of the DocumentEditor.

Editing TextImages: The standard Smalltalk paragraph editor is used and its documentation can be found elsewhere.

Editing BitImages: The standard Smalltalk Toolbox picture editor is used for this and its documentation can be found elsewhere.

Editing Headings: Headings allow one to handset a line of type. First select a character by pointing to it with the cursor while depressing redbug (the character will blink slowly when it has been selected). Then call up the yellowbug menu which looks like:



A HEADING

Each of the commands: *right*, *left*, *up*, and *down* will shift the character selected one **Alto** bit in the direction indicated. The command *font* requires that you type a number corresponding to a Smalltalk font number followed by a ↵. It will change the font of the entire line. To change the text in a Heading simply type the new line followed by striking the return key.

Kaley's disk

XEROX XEROX

Alto II/Orbit/Dover (Menlo) Press file printer

Spruce version 9.113 -- spooler version 9.111

File: ack.tape

Creation date: December 2, 1978 1:30 PM

Name: Garcia

14 total sheets = 13 pages, 1 copy.

Ivy (ala)
metaphors.pap

XEROX XEROX

Don't Settle for Anything Less

Alan Kay

Salt Lake City, November 1978

For the past seven years, we have been concerned with human computer communications, particularly in the context of portable personal computers. Our definition of portability is that you should be able to carry something else too. This is our goal. In 1972 in order to guide our research in personal computing we made some guesses as to the powers - to getting power and capacity that would be available in the package of this size. A person could carry around with him or be used in the grass or at the beach, or other places. The idea here of course is not even the aesthetics of being able to use something outside, because of course we don't always do things outside. Basic idea here is that we want this resource to be always available. So that the person is able to make it their main medium of handling their information needs. As I said this morning, in order to take reality into this dream we have over the past five or six years built a number of hardware and software systems of which this is one. This is a machine whose name started out to be the Interim Dynabook and it's short name is the Alto computer. We now have 500 to 600 of these machines in existance. We have used them, they have been around since 1972 and many of the features on this machine are conscious attempts to simulate features that we believe will be available in the Dynabook of the 1980's. For instance, this is an example of one of many different research projects that we have done over the past few years with children. This one has been in Jordan Junior High School in Palo Alto and is done in the context of learning as research both from integrating curriculum with personal computing.

Again, as I mentioned this morning, the aesthetic constraints that we decided on early in our research - we decided on before we got started because we all know what happens - the compromises that we make once we start doing real things. So before we got started we decided that first and foremost in personal computing is that there are two words--one is person and other one is computer and we have to keep both of them in mind at all times. In particular, all of our designs have really started with the idea of a person sitting in front of a display screen, where we think of a display as something that can play music that is, produce sounds as well as pictures. Context of somebody sitting there who wants to do something. One of the first questions we ask ourself is what the probability that we can anticipate this arbitrary persons need for handling information. As almost anybody you come to except maybe the manufacturer, we decided that we can't anticipate this arbitrary person's need at all. In fact in many senses we can regard most people as being experts at what they do during the day. A kid is an expert at doing what he does every day and kids are quite different from one another and most adults are engage in work that, whether they enjoy it or not, consists of skills that have been built over thousands of hours. So it is our judgement that it would be a hopeless task to try and provide a set of simple tools that would correctly anticipate people's needs. So the conclusion that we came to is that you have to build a system that these people themselves could mould into the kinds of tools that they required at that given time. That of course brings up what we think is 90% of the personal computer problem which is a communication problem between a person and a piece of hardware and of course the software. So we started out to build various software systems. All come under the generaic name of Smalltalk - you will see various versions of Smalltalk in the movie and the video. So starting right off although our intent was not to duplicate paper we were finding it interested enough to find that it works on paper. There is this funny thing that happens when new media comes in. Guys have given you something more and they always wind up giving you something less. There is something almost obscene about the idea of being able to edit text on a computer in a way that you can't ever do on paper and having it printed out or appearing on a display screen as something that is unreadable. If you think about that, it is just a little bit strange. So some of our interests were to not dull people's senses by giving them presentations that were a far away distance from what their own senses could take in. So we are very interested in human factors and here this slide gives an example of a page set in a particular font and the next slide shows the same page set in another font. The fonts are all programmable in the system and in fact the user at any time can create their own fonts by simply drawing them in ------. I think that's the - one more slide. And again, as I mentioned, the only way we felt that we could achieve flexability in the new graphics was not even to try and anticipate the people who would straight lines drawn with a certain ---. Instead what we designed was a mosiac display. We felt that the display surface of this machine would be a million dot mosaic display of some kind and instead of building graphics hardware to manipulate this mosaic display instead we built the machine so that all of the graphics, all of the music can be done entirely by software. So the music and so on you will hear is synthesized entirely by programs.

The only hardware involved in doing the synthesis on this machine is a D/A converter.

One of the goals of the personal machine - very important is to have enough cycles so that you can throw away a factor of ten or twenty of them without feeling bad. The reason is that you can't even expect our hypothetical user of the future to write efficient. We'll be happy is the person is willing to program in any form and gets a tool that will work with him. So our performance goals for this machine are very high.

The movie shows a sequence of tools done by people of various ages and different walks of life. And ends up with a glimpse into a future of Smalltalk programming. You notice I'm not going to tell you about Smalltalk language tonight. I will just peak your interest, I hope, by saying that Smalltalk does not use as primitive concepts the idea of procedures and data structures. ---. If that interests you please come to a session tomorrow and we will tell you what a programming language is like to design these procedures and data structures.

Movie.

Comments during movie: Here's one of my favorite tools. This is done by a 15 year old. This is not - I would not consider this person to be an ordinary run-of-the-mill 15 year old. This is about the second program he ever did. Hand radio * and was frustrated by circuit diagrams so he sat down as a second project to build an illustrator that would allow him to draw the circuit diagrams. Notice there is a menu down at the bottom of the screen that is shapes that he can pick up and he is going to pick them up now. Pick up a resistor and putting it in - pick up a battery. You will notice occasionally where the pointer is will appear a little menu that has special features on it like open and close docs and erase and install(?) . Every once in a while it will flash - so that has little tools that he needs often. The reason it is one of my favorite systems is that this program is done by a 15 year old less than 10 years ago a PhD thesis was awarded for a system that wasn't as good as this. Shows that there is some hope for software after all. This is also a very short program in Smalltalk - about a page and a half or two pages.

This is an example of a typical tool. I claim that most people who want to learn to use personal computers are already interested in something else. And are seeking a way to maximize their enjoyment in this field they are interested and minimize the pain. Most of the tools that we have seen people do are ones that follow that philosophy. You can he can float the text in - he is typing it in and it is positionable right now so now he has finished the circuit one transistor receiver.

Here is another. This one is a philosophy major at Stanford. This is one of my favorite cases. We have him in the Summer. Adele has been working on a book, of course she has been putting a lot more work into it than we have and she wanted to generate some errors so we thought well this person has not written any kind of a program at all for seven years and he has just learned Smalltalk, we'll let him program up. What is this is is a Masters thesis done by a girl at Cal Tech for describing circuit diagrams in LSI. Little language. So what he has done is actually implemented an entire sub-language in Smalltalk and has asked that he now draw three of these nangates(?) at different size in a parallel array such as you need to lay out a circuit board. The heck of it was that he did all of this in one week, found a number of bugs in the masters thesis which was never actually implemented. And generated only a couple of errors and deemed the project a failure for that reason.

You might wonder how you create a musical instrument. Here is a picture of one. If you were going to make an instrument in front of your eyes, time goes horizontally. This is a graph of multiple parameters in time and the first we are drawing the amptitude. Nice soft and slow rise in volume. Now we are going to change the spectro characteristics - just drawing a little graph that says make the drawing a little more complicated then level it off. Now we are going to make a little bump in the middle section between the two bars which repeats over and over again and get a suble vibratto. Now a final parameter is sort of a general tonal family. For instance, here is what these families sound like as a wood wind. And as a kind of a bowed string. Again this is all done by programming. Actually done three years after the machine was built. This is just a specific example of the kinds of things we do in Smalltalk - here's another one that is almost like except right up to the last instance when the output gets translated into pictures rather than sound. This system was done by some professional animators who visited us for a summer. They did four systems in Smalltalk in eight weeks. This is the last one they did - they did this in about two weeks. So we drew a little picture there and we picked it up and put it on the mouse and showed a path - you can think of this as being just like playing a musical score with a clarinet tombre. Now we are single stepping the path that we just drew - of course the interesting point in any animation is the contact point. Again there is no hardware to do this graphics - very important. What we have done is replaced that bottom-most frame with a fresh painting window without picking up ** painting. You can think of it as being laid transparently over the frame we have there. Watch as the painting goes in and gets * in that bottom frame in real time as the animation proceeds. You can see why animators would do a system like this because this is one of the most frustrating aspects of animation - it's knowing whether the animation is going to work. Now since the specular reflections put in there will be enough hues to make it appear as though the ball is deforming when it hits. This system done by the animators is the most used adult system by children. This 12 year old girl has added some features to it and made use of it for her own purposes. You should be able to do at least this. Now we are taking a glimpse of a system done by a graduate student who is actually helping to design the next version of Smalltalk. What we are going to see here is grabbing onto a corner of a triangle and dragging that side around and the triangle is following because it has been constrained to stay together. This is a programming system in Smalltalk for dealing with constraints. Here we are looking at two views of this triangle - the one of the left is the picture form - the one on the right which is exactly equivalent is the class of constrained triangles. You see it has parts of various kinds and has a number of constraints which say how the system as a whole is supposed to stay together. Up on the top you see an information retrieval system called a browser. What has been retrieved here is a document consisting of text and graphics whose parts have been mutually constrained. In fact it's in the form of a table of values and a total and a bar chart. Of course one of the annoying things of any kinds of documents that have pictures and text in it is when somebody edits the text they are going to change the picture. In a constraint oriented system what you would like to do is to say -Well this numeric value - this slot for a numeric value here - is constrained to be proportional to the height of the bar graph and vice versa. What he is doing is changing that - as you can see - one of the figures in the total that is connected by constraints -- saying go ahead and do it - connected by a constraint to the last bar and the constraints system settles the two constraints - one the height of the bar and the other the total. In fact they are mutually constrained as we will see in a minute. The importance of the system - now he is changing the total and we will see what will happen there . In some cases changing the total, if that were an equation, it would make sense to change all the numbers that are feeding into the total - since this is just a sum the constraint is only one way and whatever you change the number to it has to spring back to the total numbers above it. Now he is reaching down and grabbing on to the top of the bar chart and notice as he drags it the number that he is connected to is changing and so is the total as he goes along. The other thing that is interesting about programming constraints is that most of the programming can be done by simply combining separate elements that have constraints on themselves and the system has to figure out what the entailment of all the mutual constraints means in the system. I think the next example shows that. Let me explain the browser a bit. To read the browser look on the left and scrolling a whole bunch of category names - he is picking one and that immediately retrieves something to the next window on the right- he is picking one there and browsing through a fairly complicated information structure - there are thousands of entries. Finally pick out a blank slate here to show an example of constraint programming by instruction. The first * that is picked up is an element that is called bit point line. They're sticky. When you get close to two parts that are the same they stick together. Notice that no matter how he pulls out the line there the point in the center always stays in the mid point because it is constrained to do so. So now he's made himself a quadro-lateral, now what he'll do is through these about a hundred different graphics kinds of objects that you can have he has selected out another one which is just a regular line. Notice how it jumps when the end points get close to something that can connect. If you think about the intra constraints of the system built like this are quite different from the constraints of the parts taken separately. All bit point line has to worry about is to make sure that the bit point dot goes in it's bit point. Think of what happens when you connect a complicated system together each of which has it's own set of constraints as to what * should be. In general, those constraints will entail that the system must figure out a new set of behaviors for the system as a whole. This is something we are very interested in because it's something that naive users are not very good at. In fact sophisticated users are not very good at it, if you've ever tried working with a typical operating system. Now, he has made a quadro lateral and connected all the bit points - all of the math teachers in the crowd should know what he is going to do next. There is a theorum that says that you always get a parallelogram in the center. Dragging one corner of it. By making these collection of parts a whole the system has had to figure out not in what I would call an artificial intelligent way, but in a straight forward way - notice that the theorum is true even when you pull the quadro-lateral inside out. Using the constraint system to show the kids about geometry. Here's another example. This is a graphics calculator. The calculator part is down below. This is a version from fahrenheit to centigrade and vice versa. And again ** are much more fun. The system will work with either numbers or thermometers and the little diagram down there has whatever number that is there multiplied by 1.8 and then added to 32 and as he drags one of the thermometers the other must follow. If you are wondering what the

Taylor

little anchors are there he does not want to change the constants with the equation when he constraints to satisfy. Grab the other side - * they work in groups. So this is going to be one of the futures. Video tape shown here. Here is an example of a window - you will see various windows - the window contains text - there is a very simple description in Smalltalk of what windows are. In fact all programming here - there is text being typed in and in real time the lines are being justified on the left. Now some text is being read - a little menu appearing right where the cursor was - remember the kids program had that also and the PUP command was invoked. So you think of in any of these systems all Smalltalk is programmed in itself so all of the low level systems programming stuff - so this is almost a command-less editor as far as text. Now over on the left you will see a scroll bar. Notice what is happening. The square box shows where you are relative to the text and by dragging on it you can jump your way through the document. Any piece of text can be interpreted as a Smalltalk program. Instead of saying paste or cut we say do it. And Smalltalk calculates the result, which is 7. Now we move the scroll finger up to the top and the text goes back up.to the top of the screen. Adele is pointing - these two windows down at the bottom here are port holes to other project windows and pointing to one we get a different set of screen windows. In Smalltalk there is only one description per text, no matter what the contents, whether the text is a document or program or something else the use can expect that the same editor will work with it. But windows can have other things in them. Like here is a picture and when we went into it we got a different kind of menu which is a drawing menu. Up on the top there are all sorts of different tools - we will show you a few of them. This is a system that was originally done by a professional artist. Here we pick some gray paint - if you like the idea that you can sketch as well as doing rip snorting computer graphics type stuff you will like the idea of the sketch and play with the design.

Here is a tool for drawing around a thin line blacking and the message here is that you should be free to play around and do the kind of doodling that you can do on paper as well the very crystaline things normally associated with computer graphics. Again, don't settle for anything less than this, because it's what you need to not have to tell kids and adults - you can't do this or you can't do that -' and I say why not. you can do it. The stuff was all possible six years ago. Now we showing various kinds of tools. One of the messages of the computer is not even in the gray stuff. You can edit it and change it and you can get rid of it. One of the significant differences between it and the media we are used to. Now we show you some of the ways of combining graphics with what is already there. Overlaying - underlaying. This one looks like an overlay but actually is - here we pick a slightely different paint and you can see it has evolved **. Here is one that just uses what is there as a mask. Here is a demonstration that anything that ** anywhere can be used - not as a picture but as a brush. Draw in the flower, and pick the main brush command. Brush appears over to the side. Now she can paint with it. That of course can be stored away in a repertoire of brushes. Here we are picking numbers and these numbers refer to the grid. So now she is painting constrained with a certain grid. This is not a feature of Smalltalk. In fact as you wil find out tomorrow even numbers are not a feature of Smalltalk - numbers are an extension to what Smalltalk is as a programming language.

Smalltalk is actually a programming language whose basis is that of communication. Things like numbers, simulations and drawings and all of those things are done as extensions to the kernal language. We figured that would be the safest way of not anticipating what people were going to do was to not try and guess * features in. One of our main docturnal points, if you will, is that the power of a programming system of any kind, of any kind of a computer programming system is determined almost entirely by how well it does in areas for which it does not have features. Think about APL is wonderful for trying to do * but try to do ** with one of your own functions. We designed Smalltalk so that the kernal really doesn't have any features except the the ability to create communicating inscriptions. What Adele has retrieved here is actually a document which we are going to use in a funny way - she is copying this heart into the first paragraph. Documents in Smalltalk mean a lot more than regular documents in text. You can think of a document in Smalltalk as really being an organization of windows in the * information space in Smalltalk. Trimming the top of the heart and drawing a smaller one. Cut and paste metaphor is one that we use in all of our editors and in fact most of the editing commands for text and for graphics are the same. Much alike as we can make them. Cut out a piece of text into a shelf that you can later retrieve you can do the same thing for any picture. If you can edit a brush for a picture you can do the same thing for a font character. Retrieve the font character as something that fills up real quickly in the linear way or you can treat it as a graphic entity - the reason is that because the part of the self doesn't know what a font character is. You can see the menus appearing. Now she copies in this last picture here and tells the system to do something funny with the document which is to rap it and swing it's way through the document showing everything at the top of the screen. Now she says run - getting animation. Now what she is doing is copying that heart just like she copied the text before and now moving the one that she had put on the shelf back in - now paste in the one that she just copied. Changed the order of the heart and she will tell the system to run again. Goes over to the * - picks up the filled in heart. Whjen you are dealing with entities in Smalltalk almost everything you do, even when you are doing stuff in Music as we will see tomorrow is always really in the context of retrieval finding things, browsing for them getting entities that you want and then editing them into the configuration that you are after. Now we going through another window into a more involved document . Demonstration to show you what it is like to make a page of fairly reasonable text and graphics with a little twist of the hand - here is the document and a number of windows. Zoom in to the information. Picture of a mop which is obviously going to be used in this. As you might have guessed, all the other tools that are available in the Smalltalk system can be brought together in any one of these project windows and browsed through one of these * ports. So the first thing we will do is to change the font from the script font to a much bigger font and make it bold so it will look like a heading. Next thing we do is grab all the text in the next paragraph and choose a Timesroman font of a certain size and boldness in order to make the text more legible. In fact as Adele mentioned in her seminar this afternoon we have had occasion to learn about five year old eyes, ten year old eyes, thirty year old eyes, and fifty year old eyes. Fifty year old eyes for one thing do not look at the display the same way as some of the other age groups do . This font was a font done for fifty year old eyes. Underline a few words in there. Come down to the bottom and do the same thing to it. I think the first thing we do is just change it into a bigger font and now go over and make the O's even bigger, cause that was said in the text was for and make them look bold. None of this stuff I am showing you is a feature of the Smalltalk system. All facilities added as extensions by users. A lot of these that have to do with document layout and so one are ones we have added ourselves. But we have used techniques developed by professional artists who have written their own programs in Smalltalk that are adaptable. Adele is picking up a butterfly and wants to place it in the text. System animates the butterfly down so she can see where it is going to go and she wants to place it right there. This whole page of text that we are putting together is all about butterflies and moths and how they differ from each other so we obviously have to get ourselves a butterfly since we have a moth in there. In fact there is a butterfly hiding under this moth. These windows on the screen overlap. Whenever you point into one it comes up to the top of the screen and occasionally you can include so we can choose a menu under which will bring up anything that happens to be hiding and lo and behold there is a butterfly hiding under there which Adele is going to paste in. Notice we have not typed during this whole demonstration except for the very beginning as the text was being entered. The reason is that typing is not the best way to give commands in general *a graphics display. It's much better to have context intended commands. Some things you can't see easily because we are not showing the hands or the mouse. There goes the butterfly. But, the middle button on the mouse in this particular system is context sensitive to a particular window in the area you are in. So that when you are in a painting window and you go to the middle button it will give you painting commands. When you are in a text window it will give you various kinds of text commands. Musical window - various kinds of music commands. The user can rely that they will get the menu and available options by simple pushing on the middle button even when the mouse is outside the window. Here's the twist in that after all this is a simulation system and we don't want it just because of the paper. So why not antimate the butterfly - cause that's what they do most of the time is fly. So any document in Smalltalk system can contain animated - you can think of documents in physics and so on where * your explanations that are running simulations be evoked at any given point. We are going to cut off the video tape right here. Another sequence after with similar context. That's a brief tour of the kinds of things we are trying to do. I'd like to conclude with a few remarks. We ourselves are not in the educational business. Speaking at least for myself and I think for Adele and many people in our group we sort of * when it comes to education. What education is is a social process that a whole bunch of people get involved in if it's going to work at all. Education is the kind of process that any kind of technologist from the book to the fanciest computer games only magnifies what is already there in the social process. As an amplifier the computer is the greatest information amplifier that has ever been produced. In a poor social process for education different uses worst stuff in copius quantities I have ever seen. So my plea here is - there is a tendency to get involved with gadgets because you can pick them up as you set at this new class at your desk. The problem is that the problems are not in our desks but in our heads and we need to work these out first. I am going to tell you a few things I think we have learned from our experience so far. I certainly don't think we have solved the personal computing problem at all. we constantly, I should say periodically - go off - the whole group of us to a resort for a retreat and sort of poujnd into our own heads that just because we enjoy some relative success that we have to keep on measuring what we are doing compared to what we wanted to do when we started this thing six years ago. That is one of the problems of a long project is that you occasionally forget what it was you were trying to do. What we have right now is by no means the Dynabook - either the hardware or the software. I hope it's in the right direction. Here are are some generalizations that I think are a little over-simplified but maybe they will do some good. First we believe that everybody can learn how to program and without a whole lot of effort. In fact we doing think its remarkable that people can - particularly children. So we shouldn't pat ourselves on the back cause kids can program and stuff I believe that it's * humanity to be able to program or to be able to construct any kinds of things. Human beings are basically constructors of one kind of a thing or another and programs are just another kind of construction. Second children about the ages of 3-4 have written subroutines in various langauges we feel that the implications of what programming is about, that is,the power of the machine and the generalities available don't really grasp children's imigination until around the age of 10-11. Adele pointed out this afternoon that we really haven't looked at enough people to make any kind of a statement like that except at a banquet speech.Most of the successful projects that we have done have been with kids from 11-12-13 **** I have some feelings as to why. A thing that came to us as a big shock a couple of years ago after we had been enjoying some success was to suddenly discover something that is really obvious when you think about it and that is the fantastic difference between programming and designing. We sort of knew that - but not from our gut. That is an incredibly important distinction. While programming is easy - just like brick laying is easy it is not everybody who can build a house from those bricks. Not everyone can design a house and not everybody can build one. One of the characteristics of many of the users that we deal with is that they are interested in other things. We feel that design is - while programming may be a 15-40 hour skill as far as learning the mechanics of it - design may be a 1000-2000 hour skill. I believe this is one of the essential difficulties in making personal computing a reality because right now with the level of the fraction we can talk about systems most of the useful tools that the user is aiming for want to use a more complex to make a design. Shucks. Really tough - one of the reasons we have added the constraint program becuase it is a way of bridging some of the gaps of how do you get users to design archs or the equivalent of them in programming when they haven't heard of arches before. Programming is by no means the most important the most useful interchange of having personal compujting - there are other ways of using it. We feel though that always you should have the opportunity to program and change on any personal computing system that you are using. You should always have that opportunity . Even if it's * most * somebody else's text editor and information retrieval - by God if you want to change a picture in it there should be a way for that personal computer to allow you to do it. So I wrote down five properties that I am willing to argue about or in the act of personal computing. They are in order of what I call distinctiveness.

Unfortunately the first one is the most distinctive thing about personal computing is the enormous 1
attention spans that everybody has noticed. The great uses of it in the next 5-10 years - may be
only because it has a great attention span - the equivalent of 76 trombones - you don't need it to do
music - but boy is it fun to see it marching down the street. Second, this is something that I don't
think anyone in computers feels ashamed about is what you just saw on the video screen is the 2
editing of everything - text, pictures and models is both the most indulging past time 80-90% and
one process for which personal computers will find value far beyond * media. Just down right true.
Third - modelling and simulation are to me what computers are all about - not worked out as well 3
as the first two I have talked about. I think there is where the ultimate content in using the
machine lies. Modelling the simulation - building to capture like to have done for thousands of
years in speech, hundreds of thousands of years in text for just a few centuries in mathematics and
a few decades on the computer evermore dynamic ever richer models that we can manipulate this is
how we grasp our own universe. We can't touch our universe - our brain is the thing that is doing
the thinking. What I am doing here is not touching anythning with my brain - I have to convert
whatever happened here to all sorts of electro impulses and things that aren't wood into something I
can perceive as something else. Always dealing with the media - learning about media and learning
what reactions happen to a single system I think is the most important content related area in 4
commputers. Fourth - the elimination of distance by computers. What I mean by that is in a fast
computer like this looking at it as a space - it is a space that is only topological - not metric. In
other words the every part of space is the same distance from every other part - not constrainted to
the linear relationship of the text but that we can organize things in far more complex ways than we
used to in books. This is poorly worked out - and again very important. Finally - I think the most
subtle use and most subtle value in interactive computing is the** human implication. I think 5
probably everyone has noticed that what we can take in aside from touch and smell and hearing -
there is an enormous set of bandwidths comapred what we communicate out with. This may be part
of human beings fading to media. See all this wonderful new information flooding in all of our
senses but the best we can do is sing. We can hear about cues but we only sing one line * - we can
see enormous distance but our ability to paint and so on is severly limited by our own physical
attributes - this is why I beleive human beings invented symbol systems to grasp far more than they
can communicate through the kinds of noises they can make . One of the things we have noticed a
little bit that is very very interesting is that very often in the classroom some of the experiments
Adele ran is that when a kid wants to explain something to someone else and very often in our own

lab a person likes to explain something to someone else they go to a machine. Why do they go to a machine - because there is a model there that with a few simple commands or waves of their hand they can cause a whole display to change and invoking bandwidth that is not much more closer than what a person can see. I think of that as an amplifier of the ability of humans to communicate with each other. A way of matching up the inadequate output bandwidths that we were born with. One final thought on progress and technology. One of the principles I have used as a guide when thinking about personal computing is a musical intruments. I used to be a musician a long time ago and still interested in it. Musical intruments have aesthetics with them that computer people would do well to follow. Nothing more * than a flute or violin - take them anywhere and play anywhere. You notice about a flute and violin is that there is no language to input - or output imagine what it would be like to play something serious on the flute where there was a 2 or 3 second lag. Equivalent of the musician going to the concert hall in the afternoon and playing a concert and going in the evening to hear what it might have sounded like. That's absurd. However there is another analogy of musical instruments which I think is very constructive and that is by and large most musical instruments most musical instruments were invented as prosthetics - not invented to make music per se but to make up for perceived deficiencies in the human voice. Making a lot of noise at roman stadium...making a lot of noise in a cathedral - in fact these early musical instruments were far inferior in many important parameters - like they couldn't even say words the range of * they could use were bad. Back in those days the musical instruments were worse than the things they were replacing except in once trivial thing and that was they were louder. Over a long period of time - hundred or two hundred years there were lot of interactions between composers, players and musical instrument manufacturers. Musical instruments found a value system of their own where now today they are not prosthetics but amplifiers. That is what I want to see the destiny of personal computers - amplifier - not prosthetics.

Alto II/Orbit/Dover (Menlo) Press file printer

Spruce version 9.200 -- spooler version 9.200

File: metaphors.pap

Creation date: April 20, 1979 1:29 PM

Name: adele

7 total sheets = 6 pages, 1 copy.

One way to design a programming language is to expand into the pocket universe of the computer simple metaphors from the world of human experience having to do with structure and time.

For instance, we know a bit about how atoms can be structured into complex molecules and this might serve as a model for a very construction oriented language like LISP: heavy on connexion and light on control and protection. On the other hand, we know that living material, the most complex structures in our experience, require much more than a molecular bonding philosophy in order to function. Though it would be presumptious to imitate that which we do not yet understand, might our designs still be fruitfully guided by some of the metaphors of living material?

It is a conceit of Western Culture to believe that the dissection of a whole into parts can reveal many of the secrets of the whole. When combined with an appreciation for the way new properties appear as parts are combined -- such as an arch formed from lowly bricks -- our Western conceit has been remarkably successful. We need not congratulate ourselves unduly on our insight, since it appears that in a universe not obviously controlled by a ... deity, the parts of wholes require considerable autonomy and limited interactivity in order for the aggregate structures to work at all.

A biological cell is a structure with more on the inside than on the outside. In fact, these organisms expend a considerable percentage of their energy and activities in just maintaining the distinction between inside and outside. Where almost all of a complex molecule's structure is exposed to the ravages of any environment in which it is put, a cell only exposes a small part of its structure: its cell membrane, a fabric specialized to keep all parts of the environment except those beneficial to the cell away from the more delicate structures inside.

The cell membrane also keeps the cell *in*. Inside is a miniature sea, a primordial soup which the cell keeps simmering to make more cells. There are recognizable *parts* within a cell. Some, like granules and a host of organic and inorganic molecules appear to have a simple relation to the entirety. Others, like the mitochondrial chemical factories and genetic material are so intertwined into the life cycle of the organism that it is difficult to fruitfully discuss their nature in isolation.

When a cell divides to produce a twin, the least important of its constituants, the majority of its cytoplasm and membrane, is simply increased and shared. Its most important parts however, the nucleus and mitochondria, are copied as exactly as nature will permit. Thus part of every kind of cell are closely similar structures which link their destinies coupled with parts that are those of individual cells alone.

The potential for differentiation within a fixed heritage is enormous. Every body-cell in human beings has the same ancestry, yet has let itself be differentiated by chemical messengers to form a marvelous variety of specialized tissues and organs to make up creatures which for the most part are blissfully unaware of their inner majesty.

If we think of a computer as an environment in which time, space, and structure can be fashioned, and a programming language as a vehicle for describing and building complex dynamic structures, then metaphors drawn from the most complex systems we know can be very helpful in guiding the principles of the language we will use.

Let us consider a universe consisting of *wholes* made from *parts*. Each part is a whole in its own right. One of a whole's parts is a *boundary* which determines the interaction of the *interior* with the *environment*. The interior consists of parts. Some of the parts are shared with every *sibling* of the whole. Some parts are rather inert, others are constantly in *process*, maintaining the interactive relationships between other parts that keeps a particular whole whole. Wholes can communicate by sending other wholes to each other. Communications are accepted or rejected by a whole's boundary. One whole's assumptions about another whole have to do entirely with expectations about the kinds of effects and return messages invoked by an initial communication.

A language is a medium for communicating about a world. Though it will be structured by the relationships of that world, much of its form will have to do with the linguistic range of its users.

For example, humans find it useful to make up figurative objects called *concepts* which, though only their *instances* can be pointed to, seem to help greatly the task of describing interesting formations. The *arch* is such a concept. If we step up to a natural or man-made one, we may vainly search high and low for the object: archness. All we can find is *one* of them, and an agreement in our culture to call such things: arches.

Another widely used human linguistic device is the statement consisting of a description of one or more performers in a scene of interest followed by a description of their action or inter-relation.

There are many points of view from which to judge human languages and it would do a disservice to treat them too lightly. But here we are not concerned with the ways of humans or biology, but only what we may draw through superficial analogy.

For example, let us consider the use of positional notation and its influence on vocabulary. In the natural development of vocabulary, we might guess that the tendancy of humans and animals would be to make up new word-noises for each concept. In animals such as monkeys this seems to be exclusively the case. In man, there is a trade-off between either making up a new word or trying to find a combination of old words which will describe the new idea. The social advantages of being able to describe new ideas through those one's compatriots already understand are obvious. Nonetheless, the uses of position and metaphor grew hard indeed. We find, for instance, in Latin and in the Roman numeral system a fine disregard for the advantages of position. The heavy inflections of Latin make word-order a stylistic, rather than a semantic, issue. That it was difficult to add Roman numerals, and almost impossible to multiply them, was no doubt regarded as a useful feature by the calculatory unions of the time, there being no better methods then known to give them their leave.

Mandarin Chinese, on the other hand, is a language innocent of word-inflections, in which the *aspect* of meaning a word is to contribute to a statement is selected by its position in the utterence as a whole. For our purposes, English, particulary the colloquial variety, is a language much more like Chinese than Latin. Today we can say: "It's a new kind of clean!", be understood, and escape with unrapped knuckles in the bargain. Though English still has word-inflections for number and tense, we may feel secure about following the Chinese model and dispensing with them all. Of course, languages which give up word-inflections are likely to have many more statement-level marker words such as tense indicators, prepositions, articles, and the like. These tend to be small in number and uniformly used across the language.

We also need to consider style of description. For most human purposes, concepts and their instances are discussed informally through their boundary appearence. When someone asks another what a house is, the description given back has more to do with the *goals* of a house rather than how one is built. Parts of a whole are most often described by giving their relationships to each other rather than how the relationships are maintained. One reason for this style of description in natural language is that the universe was here long before and much of our discovery process was first just to describe *what* we saw about us. It is only in the last few centuries that we have made any progress with the *hows.*

In computers, we start with a proto-universe in which space and time are yet to be described, photons and field-laws are creatures of our imagination, and everything which exists must be built.

This is why early computer languages have been imperative rather than declarative in style. Though constructive descriptions are much harder for humans to deal with, they have been and will continue to be the *lingua franca ex machina* until better ways are found to translate cool declaratives into the bustling imperatives of the machine.

A language system which required a prior understanding of the philosophical principles of physics, biology, linguistics, psychology, and theatre would attract few users. The trick in designing an easy to learn programming system which can grow with its users, is to find simple, easy to use forms and a few abstract ideas which contain both the simplest expressions of a novice and the most subtle descriptions of an expert.

Our approach to teaching a programming system is to get learners to build their own systems. Much of their early programming deals with using and modifying already given forms. Grammar is learned gradually and informally.

Since teaching consists primarily of finding ways to let a student learn, an essay is a poor vessal indeed for teaching: one persuades rather than teaches.

< Examples >

What Can be done by Dinking Around

- CHASE and Spacewar
- Retrieval/Calculation
- Queing Simulation
- BarGraph and Browser
- Layout/Document/Editing/Fonts, Pictures, ...

Language Principles

Finding

By supplying the actual whole:	3, 'this is text', }
By supplying a name:	a, x, Turtle1, ...
By supplying a statement:	3+4, Bob's Father's Age, ...

Grammar

Phrase ::= { Literal | Name } {sel

sel

sel exp

sel sel

sel exp sel

sel exp sel exp

sel exp sel exp sel

$a\ +\ b\ *\ c$

$a\ \underline{+\ b}\ \underline{*\ c}$

$\underline{a\ +\ b}\ *\ c$

- Concepts
    - Meta-Concepts
        - Viewing-Editing
        - Naming-Pronaming-Paths-Finding
        - Messaging: $\underline{\text{Whole | Message,}}$ $\underline{\text{Whole | Message | Reply-Catch}}$
        - Merging
        - Offspring-Twinning
        - Defaults-Initialization
        - Relations
        - Control
        - Exceptions
        - Events
    - Concepts
        - Quantity-Location-Movement
        - Collection

Summary of Useful Metaphors

Notions from Mathematics:

- Relations
- Spaces-Transformations-Points

Notions from Physics:

- No action at a distance -- messages are sent/received
- Local effects have local causes
- Time is information flowing through a boundary
- Observer/Observed is a symmetric relationship

Notions from Biology:

- No action at a distance -- messages are sent/received
- Wholes made from Parts which in turn are Wholes.
- One part is a boundary which protects the inside from the outside.
- Some parts are rather structural, others are process-like
- Some parts are relations between other parts

Notions from Linguistics

- No action at a distance -- messages are sent/received
- Positional Form
- no inflections
- Statement wide markers
- Subject-Predicate
- words as symbols for concepts with aspects

Notions from Computer Science:

- Information retrieval
- Graphics
- Domains, travelers, Decorations

Notions from the Psychology of Complex Skill Learning

- Learning by doing
- Learning with others
- The 1000-hour skill (paradigm learning)
- Short-Term Mem: easier to learn a few abstract ideas than to learn a lot of concrete ideas.
- Design as a central metaphor

- Stage Learning: meaning as: action, image, and symbol
- Creativity: a retreat to earlier stages
- Learning without remediation
- Spiral Curricula

Notions From the Performing Arts:

- No action at a distance -- messages are sent/received
- Theatre: stage, scene shop, plays, performers, ...
- Performer: costume, role, script, schticks, ...

Invited Paper: Session 3: Programming By Non-expert Users

Infotech State-of-the-Art Conference on User-Friendly Systems, London (March 28-30, 1978)

Ideas for Novice Programming in a Personal Computing System

Alan Kay

Learning Research Group

XEROX PALO ALTO RESEARCH CENTER

Abstract

The scope of the *Learning Research Group* at PARC includes most of the topics of interest at this conference. I will treat them in a slightly different order—and with a depth which reserves the most detail for my main subject of user programming.

Personal Computing. The initial section is a quick precis of our motivation, work, and relation to users. The first series of figures will illustrate some of our experimental hardware and personal tools programmed by novices in our system.

Interactive Novice Programming. My main focus. The text is an explanation of our strategies for dealing with users and a general description of the semantics of the personal computing systems we have designed. The figures will illustrate a variety of programming techniques we use, all related to the uniform metaphors of editing and search discussed in the text. A general simulation kit and experience with adult novice users is covered next. Following this is a brief look at some nonprocedural programming techniques we currently employ.

Conclusions covers some of our feelings about the place and values of personal computing in 1979.

Personal Computing

The Xerox Learning Research Group is concerned with human-to-personal-computer communication, particularly for novice programmmers. Our approach has been to envision the personal computing system of the 1980's, the *Dynabook*, as notebook-sized with enough computing power and capacity to carry out its owner's needs in the world of information (Figure 1). Though the hardware for the *Dynabook* is difficult enough, the major problems which must be solved are software related. The center of the personal computing problem concerns communication, description, and learning -- all "soft" problems, and our chief pursuits [*Dynabook*].

Over the years we have built many versions of a personal computing system called *Smalltalk*, occasionally collaborating with other groups at Xerox to develop experimental hardware on which *Smalltalk* can be tested. In 1973 we started to use the desk-sized personal computer shown in Figure 2. Now our focus is the equally powerful, but portable, system shown in Figure 3. The personal computing systems developed by Xerox are interconnected through an information network used to provide communication between users, servers (such as shared information utilities and printing), and other networks [*Smalltalk*].

Smalltalk does not use the usual building blocks of subroutines and data-structures as do BASIC, FORTRAN, and PL/1. Instead, everything is built from active communicating objects, organized by inheritance of behavior. *Smalltalk*'s semantics has been somewhat idealized and simplified for this paper. (The actual semantics of *Smalltalk-76* is similar but less unified that given here). All examples in this paper were programmed in *Smalltalk* and directly photographed from our display screens.

We are particularly interested in three aspects of this new metamedium: first, how can media be formed into tools such as the systems shown in Figures 4-7; second, we are interested in the nature of descriptions and communication; and third, we would like to learn how

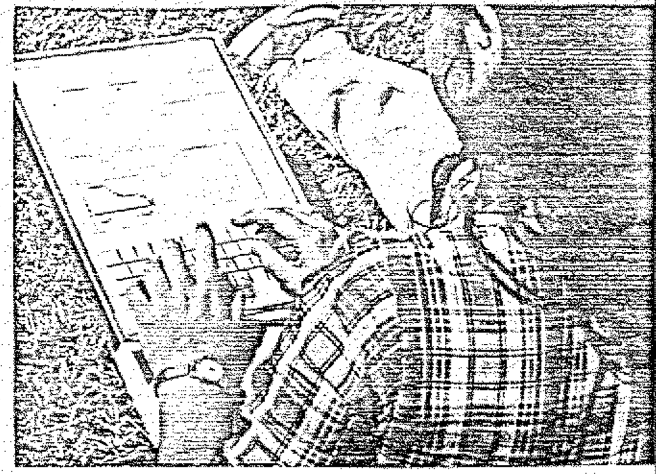


Figure 1. An appearance model of the notebook-sized Dynabook of the 1980s.

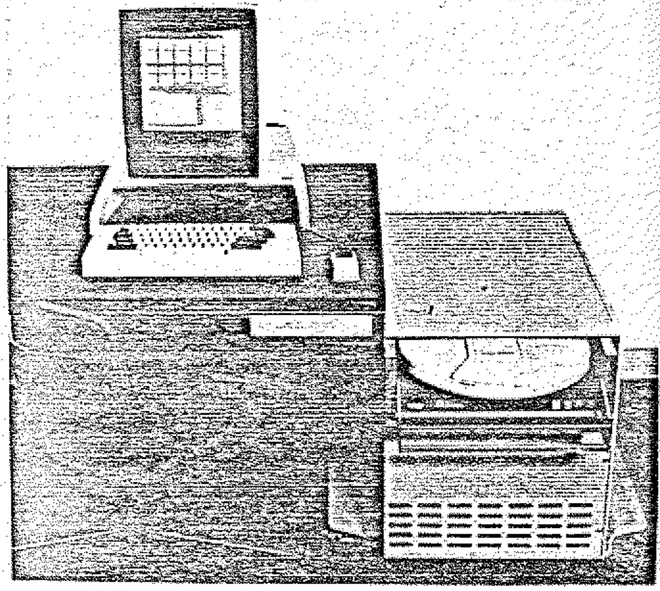


Figure 2. The Xerox PARC "Alto" Personal Computer.

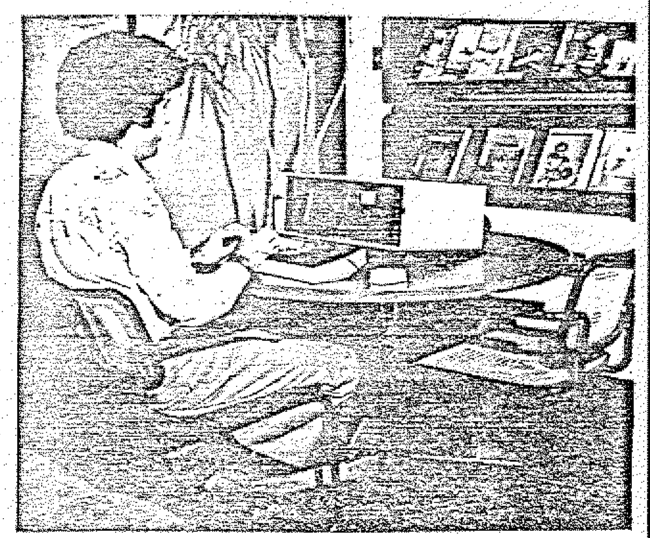


Figure 3. The Xerox PARC "NoteTaker" Personal Computer.

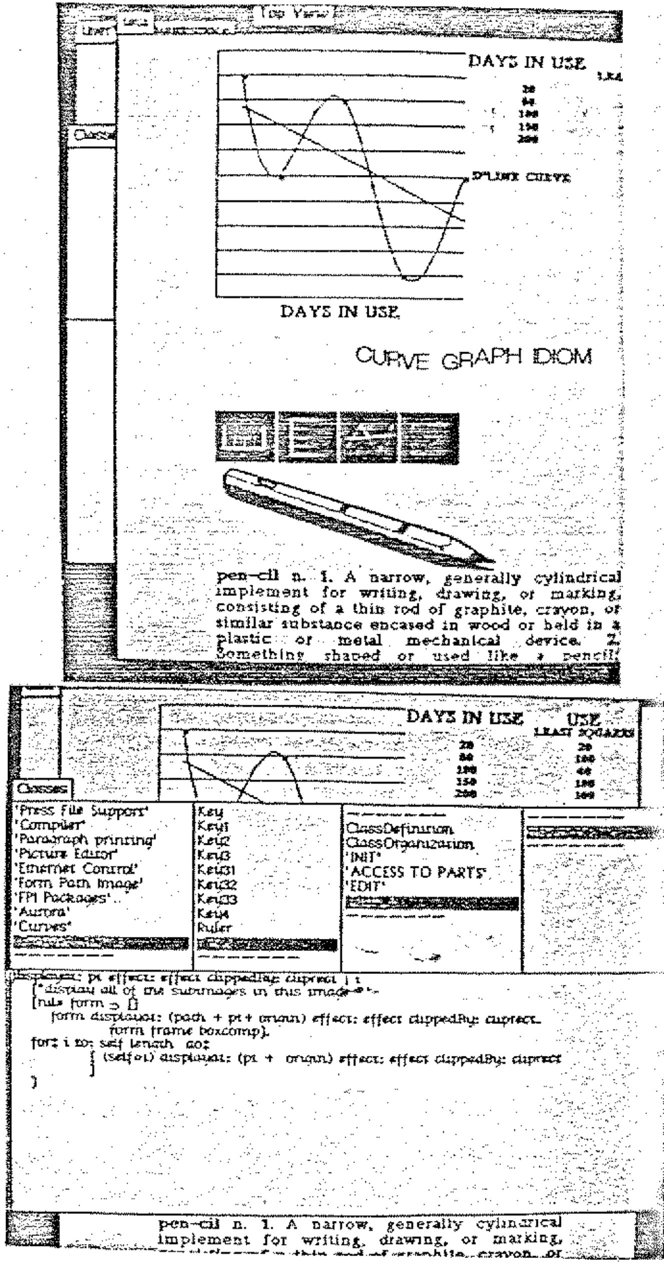
people may be taught design skills which enable them to shape their computer medium.

Interactive Novice Programming

The traditional differences between interactive control of computer tools and the programming of these tools have rarely been resolved in interactive systems. Yet, ever since JOSS in the early sixties, the only distinctions needed between *direct* and *indirect* control/programming have been the amount and the kinds of *delay* between specification and effect [*Joss*]. Thus, we feel that a programming language design is properly part of the user interface.

The problems faced by a novice programmer parallel those of a person who has books all over the house and wishes to have a bookcase. He may see an ad for one, go to the store, find that the bookcase is just right, and buy it. Fortuitous and satisfying. Often, though, the bookcase is not quite right. The buyer may decide to get it anyway, refinish it, and adjust the shelves to fit. If this is not possible, the next level of strategy is to find a kit which has much of the hard work already done yet still allows the builder some options. If no kit is available, many people would quit and go back to orange crates. Our resolute subject is brave enough to go to a lumber store to purchase materials and a manual about sawing, fastening and other properties of wood. In an extreme case (e.g., when a fancy veneer is desired), new materials must be constructed and the amateur must subcontract the task to an expert or must become more of an expert himself.

All of these situations will arise in a personal computing system; it must contain facilities for meeting each difficulty which range from directly usable tools, to "kits" and instructions, to raw materials and theoretical knowledge. In order of greatest return for least amount of work, users must learn: to characterize their current *need*, to *browse* for and *recognise* a facility that possibly meets that need, to *use* it, to *modify* it, to *fix* it, to *combine* it with other facilities in a *kit*, and to *make* a facility from scratch. The strongest technique we use is that of enforced

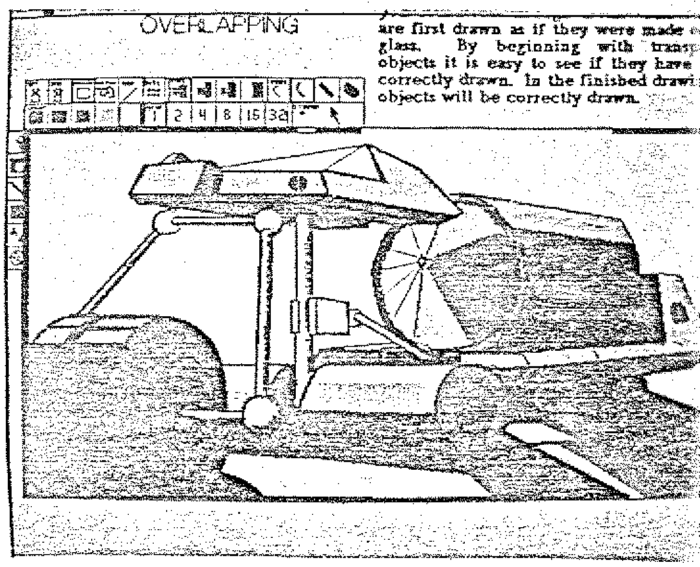


pen–cil n. 1. A narrow, generally cylindrical implement for writing, drawing, or marking, consisting of a thin rod of graphite, crayon, or similar substance encased in wood or held in a plastic or metal mechanical device. 2. Something shaped or used like a pencil.

```
displayat: pt effect: effect clippedBy: cliprect | i
  ["display all of the subimages in this image"
  [nil= form ⇒ []
     form displayat: (path + pt + origin) effect: effect clippedBy: cliprect.
           form frame boxcomp].
  for: i to: self length do:
          [ (self◦i) displayat: (pt +  origin) effect: effect clippedBy: cliprect
          ]
  ]
```

pen–cil n. 1. A narrow, generally cylindrical implement for writing, drawing, or marking,

Figure 4A-B. A typical Smalltalk screen showing overlapping viewport-windows. Included are views of formatted text, illustrations, curves, documents, and Smalltalk resources.



are first drawn as if they were made o glass. By beginning with transp objects it is easy to see if they have correctly drawn. In the finished drawi objects will be correctly drawn.

Figure 5. An illustration tool initially designed and programmed by an adult artist.

analogies. That is, we try to make every kind of material, process, and programming technique look abstractly similar. We carry through this fiction at every level of structure from the atomic to the architectural. The hope (for which we have seen some evidence) is that the additional burden of abstraction this "artificial chunking" tactic puts on the novice is more than repaid in the novice's ability to cope with many more necessary facilities than an unaided short-term consciousness can handle. In designing a language we wish to limit the *forms* of descriptions, but not their range; we want to crystallize a *style* of programming, not just arrive at a syntax for expressions. Though we search for ways to make our personal computing systems more abstractly simple, we don't teach our languages axiomatically, but through examples and incremental changes [*Teaching*]. The learner gradually notices the extreme uniformity of the system and begins to rely on guesses derived from concrete experience.

Stripped of their metaphysics, every *Smalltalk* language design has been concerned with how to deal with information organized in the form of parts-and-wholes, e.g., dictionaries which relate part-names to their active meanings. The first step in the development of this model was to give each dictionary-object a separate *inside* and *outside*. Users of the object can deal with a never changing *virtual* organization from the outside. Programmers of the object can, from the inside, change methods for carrying out its virtual meanings at any time, as long as the external behavior stays the same. The distinction between inside and outside means that users must *communicate* with an object to get it to do anything. Direct absolute commands no longer are possible; they are replaced by queries tendered in the form of messages. Objects from the outside thus act quite like digital computers in a communications network and are neither data-structures nor procedures.

Since there are many more objects in the universe than are contained in any one dictionary (save the universal dictionary), every dictionary can be thought of as a *filter* that selects some objects from a universe and rejects most others. The "insideness" and "outsideness" of a dictionary-

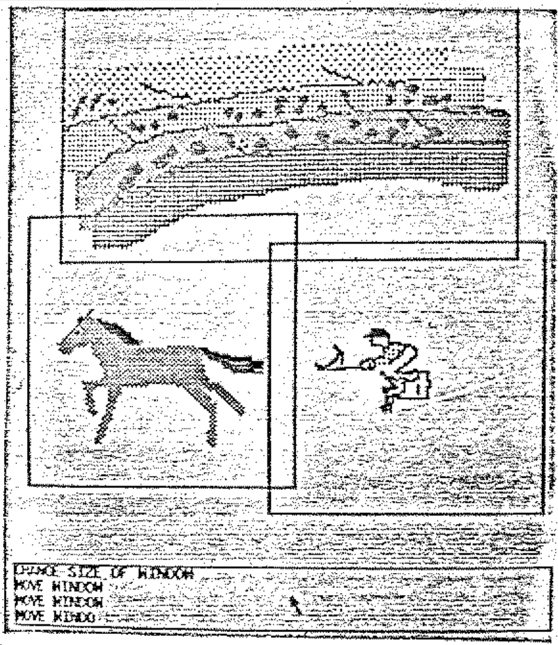


Figure-6A. An animation being composed by a 12-year old girl. She is combining a horse with a rider to be animated over a background painting.

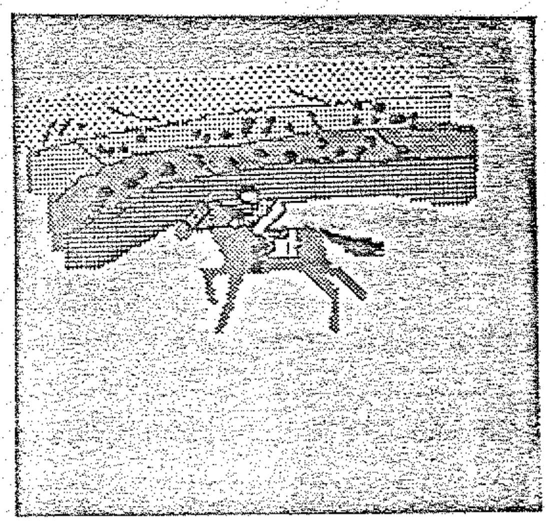
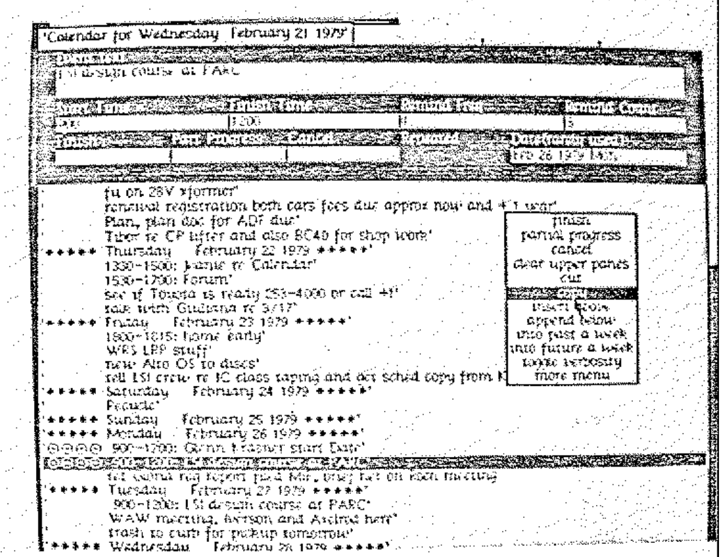


Figure 6B. The finished animation.



'Calendar for Wednesday February 21 1979'

LSI design course at PARC

1200

Feb 26 1979 14..

```
      fu on 28V xformer'
      renewal registration both cars fees due approx now and +1 year'
      Plan, plan doc for ADF due'
      Tiber re CP lifter and also BC40 for shop work'
***** Thursday    February 22 1979 *****'
      1330-1500: Jamie re Calendar'
      1530-1700: Forum'
      see if Toyota is ready 253-4000 or call +1'
      talk with Giuliana re 3/17'
***** Friday    February 23 1979 *****'
      1500-1615: home early'
      WRS LPP stuff'
      new Alto OS to discs'
      tell LSI crew re IC class taping and get sched copy from N
***** Saturday    February 24 1979 *****'
      Picnic'
***** Sunday    February 25 1979 *****'
***** Monday    February 26 1979 *****'
@@@@ 900-1700: Gunn. Frazier start Date'
[illegible]
      [illegible]
***** Tuesday    February 27 1979 *****'
      900-1200: LSI design course at PARC'
      WAW meeting, Arrson and Andre here'
      trash to curb for pickup tomorrow'
***** Wednesday    February 28 1979 *****'
```

Figure 7. A personal calender program devised by a PARC laboratory facilities manager.

object is itself constructed by filtering: the outside is portrayed by a filter that suppresses internal organization for a simpler external fiction.

Filtering provides an excellent mechanism for developing interfaces of all kinds, both to the human user and to other objects. Because every dictionary-object is ultimately dealt with as a picture on the display screen, it simplifies matters to consider every object as inherently an *image* with a variety of default and user-supplied ways to display itself for perusal and editing. In Figures 8A-B, we see a complex object in process being viewed by three objects; two are display-viewports. The different kinds of viewport-windows are developed as analogies to basic descriptions which contain methods for locating themselves relative to others. Each view of an object is constructed by filtering. The filters are not out of the ken of the object because edits passed though one view that change the state of the object must be continuously reflected back through all the other views of the object. Tools for user interaction with an object such as menus can be constructed as a filter on a dictionary which visually suppresses the meaning side of entries (and most of the entries) of an object. Pointing at the name side of an entry with the stylus connotes sending a message to that entry. The entry can prompt the user if additional parameters are needed.

Filtering is also used to provide meta-levels of description structuring, ranging from low-level parts, wholes, and messages; to inheritance of properties, to meta-building blocks; to kits and applications.

If the entries in the internal representation of an object can refer to each other, then a perfectly general systems network is possible. We can think of an entry as containing references to the name of the entry and its meaning. The meaning of an entry is thus an object which can be shared (it can be contained by other entries in other dictionary-objects). This parts-wholes semantics is also used by the external representation of an object but, as mentioned, its details may be completely different from the object's internal strategies.

An important property of filtering is whether the

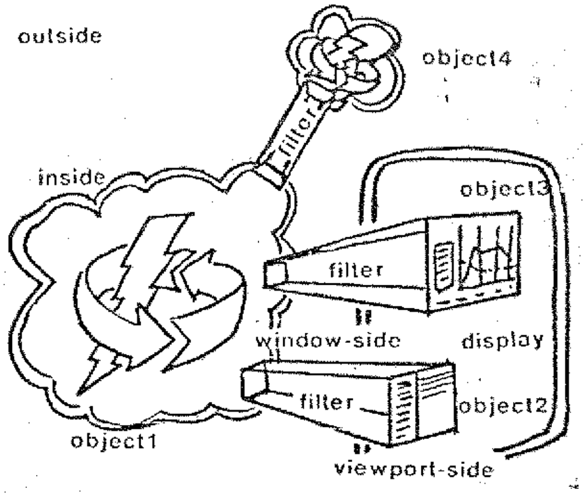
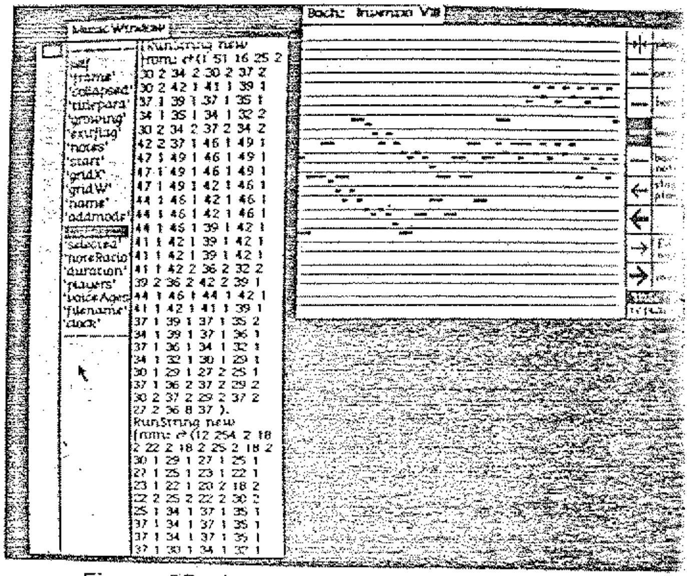


Figure 8A. An object being viewed by three other objects.



```
RunString new
from: #(1 51 16 25 2
30 2 34 2 30 2 37 2
30 2 42 1 41 1 39 1
37 1 39 1 37 1 35 1
34 1 35 1 34 1 32 2
30 2 34 2 37 2 34 2
42 2 37 1 46 1 49 1
47 1 49 1 46 1 49 1
47 1 49 1 46 1 49 1
47 1 49 1 42 1 46 1
44 1 46 1 42 1 46 1
44 1 46 1 42 1 46 1
44 1 45 1 39 1 42 1
41 1 42 1 39 1 42 1
41 1 42 1 39 1 42 1
41 1 42 2 36 2 32 2
39 2 36 2 42 2 39 1
44 1 46 1 44 1 42 1
41 1 42 1 41 1 39 1
37 1 39 1 37 1 35 2
34 1 39 1 37 1 36 1
37 1 36 1 34 1 32 1
34 1 32 1 30 1 29 1
30 1 29 1 27 2 25 1
37 1 36 2 37 2 29 2
30 2 37 2 29 2 37 2
27 2 36 8 37 ).
RunString new
from: #(12 254 2 18
2 22 2 18 2 25 2 18 2
30 1 29 1 27 1 25 1
27 1 25 1 23 1 22 1
23 1 22 1 20 2 18 2
22 2 25 2 22 2 30 2
25 1 34 1 37 1 35 1
37 1 34 1 37 1 35 1
37 1 34 1 37 1 35 1
37 1 30 1 34 1 32 1
```

Figure 8B shows multiple views of a musical score. The right-hand viewport contains a score with its menu-controlled editing interface. The left-hand viewport shows the same score but in its default view as a Smalltalk object. Its menu is in the left pane and selects a particular entry to be seen in the right pane.

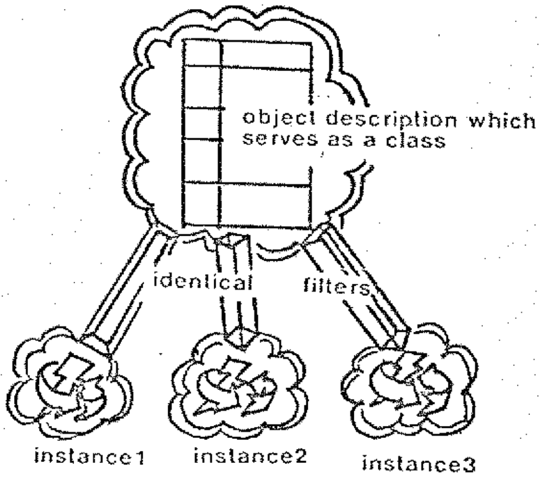


Figure 9. Deriving instances from a protolypical class description is an identity-changing filtration.

filtration is severe enough to completely mask the identity of the filtee. For example, deriving a new instance of a class, Figure 9, or a new subclass inheritance, Figure 10, both create distinctly new objects. An identity-preserving filtration was shown in Figures 8A-B: multiple views of a musical score presented through different windows on the display. Identity-changing filtration facilitates a second level of systems structuring: ways to describe the kinds of objects we wish to gather together to carry out our wishes. A successful framework we employ is to use filtering to create new objects that are dynamic analogies of objects we already know about. All of the notions of instances of classes and (multiple) super- and subclassing are examples of such filtering. The analogy filtering mechanism permits *differential programming*, that is, programming by saying: "... I want something just like you, except ... ". System designers can devise far-reaching object descriptions from which analogies may be extracted. Having these descriptions already in the system is a two-edged sword. On the one hand, users do not have to invent classes for numbers, interaction windows, schedulers, and the like, every time they want to use the system. On the other hand, the possible great number and variety of furnished descriptions requires that a much greater attention be given to providing ways for users to browse through them. Otherwise, a large preloaded system full of promise but difficult to wander through will be considerably more trouble than a stripped clean but easily understandable version.

Systems designers themselves have considerable difficulty in devising an epistemological framework in which a small number of basic concepts can cover most user needs in models that the designer cannot possibly anticipate. The tendency is to admit more and more concepts to the system, regardless of their possible overlap to already existing ones. The result can be greater turmoil and complexity (situations we were trying to avoid at all costs) requiring better "help" and browsing assistance. Figures 11-15 show a variety of browsers derived from the same basic description [*Browsers*].

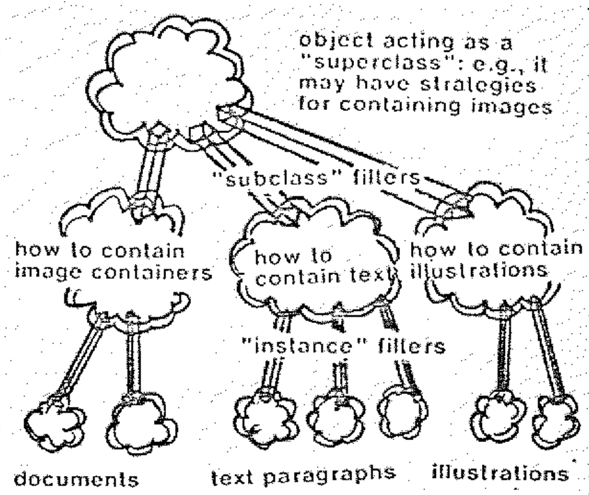
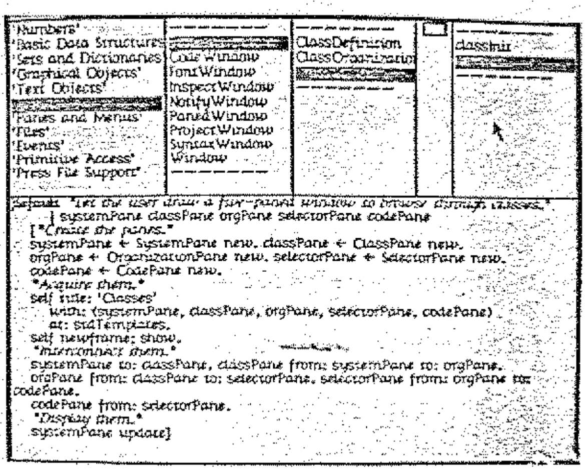

object acting as a "superclass": e.g., it may have strategies for containing images
"subclass" filters
how to contain image containers
how to contain text
how to contain illustrations
"instance" filters
documents
text paragraphs
illustrations

Figure 10. Object descriptions which serve as classes can themselves be composed by filtration.



```
default "Let the user draw a five-paned window to browse through classes."
    | systemPane classPane orgPane selectorPane codePane
  ["Create the panes."
  systemPane ← SystemPane new. classPane ← ClassPane new.
  orgPane ← OrganizationPane new. selectorPane ← SelectorPane new.
  codePane ← CodePane new.
  "Acquire them."
  self title: 'Classes'
     with: (systemPane, classPane, orgPane, selectorPane, codePane)
     at: stdTemplates.
  self newframe; show.
  "Interconnect them."
  systemPane to: classPane. classPane from: systemPane to: orgPane.
  orgPane from: classPane to: selectorPane. selectorPane from: orgPane to:
codePane.
  codePane from: selectorPane.
  "Display them."
  systemPane update]
```

Figure 11A. A Smalltalk resource "browsing" window used to peruse the Smalltalk system itself.

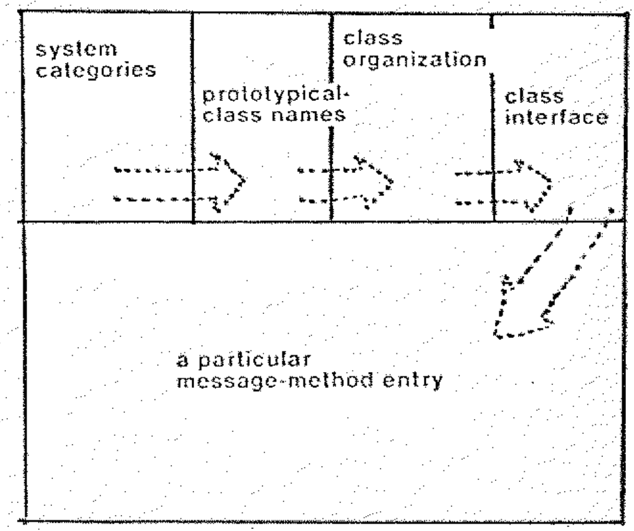
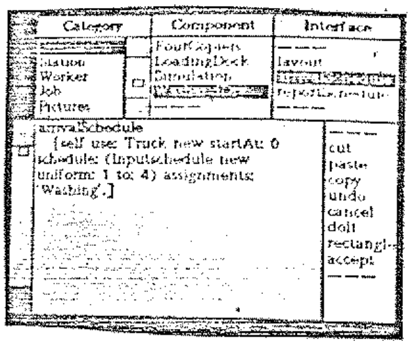


Figure 11B. A diagram of the dependencies between "panes" of the browser. When a list item is pointed at with the stylus, a retrieval of the next lower category is placed in a dependent pane.

Since most novice user's short-term memories are saturated by the newness of it all, it is paramount that the design of the system, and the introduction of novices to it, should carefully deal with just what a novice user's short-term memory should be saturated! We have chosen *editing* and *search* as the candidates for saturation. Editing is both the most indulged in pastime in interactive systems, and is a prerequisite for using or programming any tool. Search is the other activity that users are always doing. They search for desired effects, ways to cause them, reasons why they didn't happen, and how to fix them. Our system designs have attempted to find ways to permit everything else the user may wish to do, including the programming of arbitrarily complex behaviors, to be learned as metaphors of the two simple ideas of editing and searching for dictionary-objects.

To summarize, several levels of structuring have been discussed. The first includes the ideas of *communicating-objects, parts-wholes, insides-outsides,* and *searching-filters-editing.* The next structural level used the first level to provide objects that act as prototypes from which new objects may *inherit* analogous characteristics.

More levels are possible and needed. A third level of structuring is to impose sharply orthogonal properties on a small number of basic prototypical objects to provide a rich *domain* for building simulations. As an example, consider the following three orthogonal prototypes. First, a prototypical object which represents *spatial* (or locative) characteristics can be the basis for a wide variety of further descriptions, such as numbers and other algebraic systems, composition and setting for illustrations, documents, circuit diagrams, musical scores, animation scenarios, and programs themselves. A second prototype embodies the general notion of *traveling* in a spatial domain. Examples are a constrained numeric *variable*, a *cursor* in a document, a *paint brush* in an illustration, a *player* in a musical score, a *role* in an animation, a *process-point* in a program. Travelers may employ elaborate strategies in deciding when to move from one location in a space to another, and what to do when they arrive. A prototype which



```
arrivalSchedule
   [self use: Truck new startAt: 0
schedule: (Inputschedule new
uniform: 1 to: 4) assignments:
'Washing'.]
```

Figure 12. A Smalltalk browser which has been filtered to suppress most of the actual system organization. This browser was constructed for a simulation "kit" used by adult business managers.

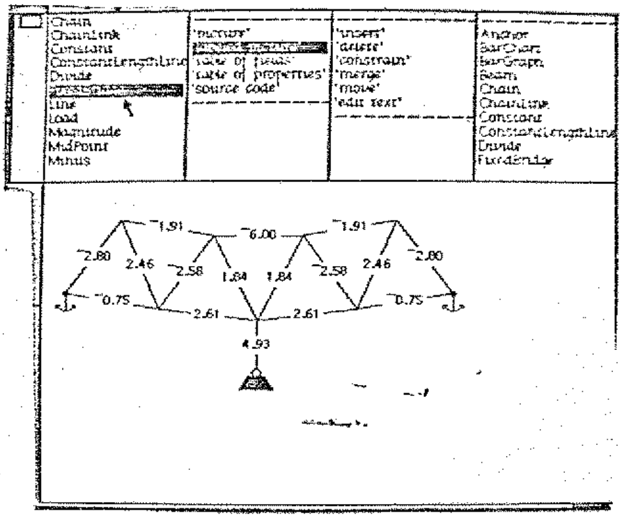


Figure 13. A Smalltalk relational-object browser. Programming in this browser is done by editing pictures of desired results. More examples are given later in the text.

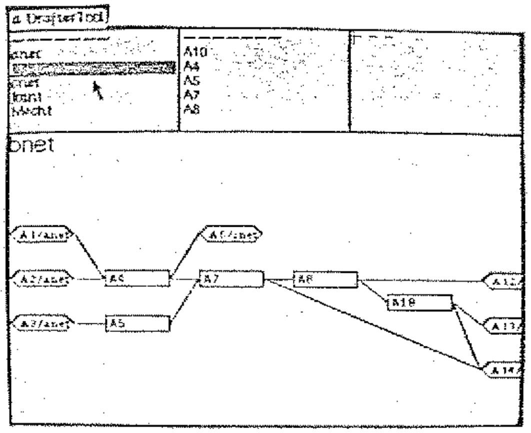


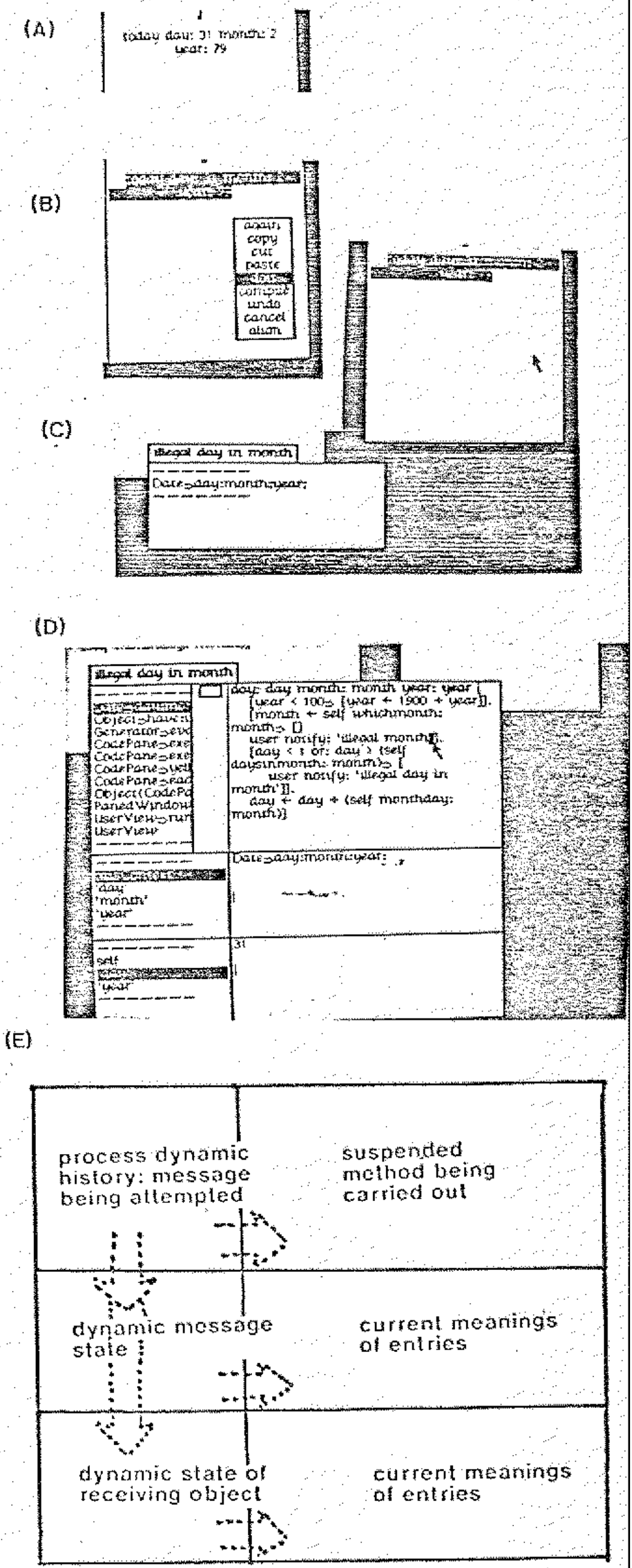
Figure 14. A Smalltalk planning system browser. Plans are constructed by forming diagrams which can then control simulations of the future.

supplies a third dimension to this model is that of a *decoration*. This is an entity which is not strictly necessary for the trip of a traveler through a space, except in a zero-dimensional-form (z-d-f), but when present, greatly enhances the journey. Examples are: a *physical-dimension*, such as length or weight attached to a changing quantity (z-d-f: a scaler); different *type-fonts* in a document (z-d-f: a default font); *tone/color* and *texture* in an illustration; the *timbre* of an instrument used to play a score (z-d-f: a pure sine wave); the *costume* of a role in an animation (z-d-f: a dot); the particular *pragmatic-primitives* employed by a program-interpreter, e.g., one way of carrying out arithmetic orders may be much faster than another (z-d-f: a "one-page" interpreter).

Kits are the fourth level of structuring we use. A kit is a further filtering on a domain which greatly focuses its degrees of freedom to a particular goal. Examples that we have built in *Smalltalk* for novice users include kits for: arithmetic, algebra, and geometry; document editing; drawing and painting; music; animation; and programming. All of the above can be understood as specializations (filtrations) of the three orthogonal prototypes mentioned previously. An example is the subsequently discussed simulation kit we devised shown in Figure 16 [*Kits*]. It has been used by manager-level adults to build specific application-simulations which explored and answered their questions about complicated situations in their own businesses.

A fifth level of structuring is that of an *application*, a system whose use is tailored to a particular problem. Applications require the least deep understanding by a user (because most degrees of freedom have been removed), and are the most difficult systems designs to anticipate correctly. Some of our more successful applications in *Smalltalk* have been a music system for composition and orchestration, the *Smalltalk* browsers, and an information retrieval system used for several years by our center's library.

Ideally, all applications should be built with a careful regard to each of the four lower levels of structuring we



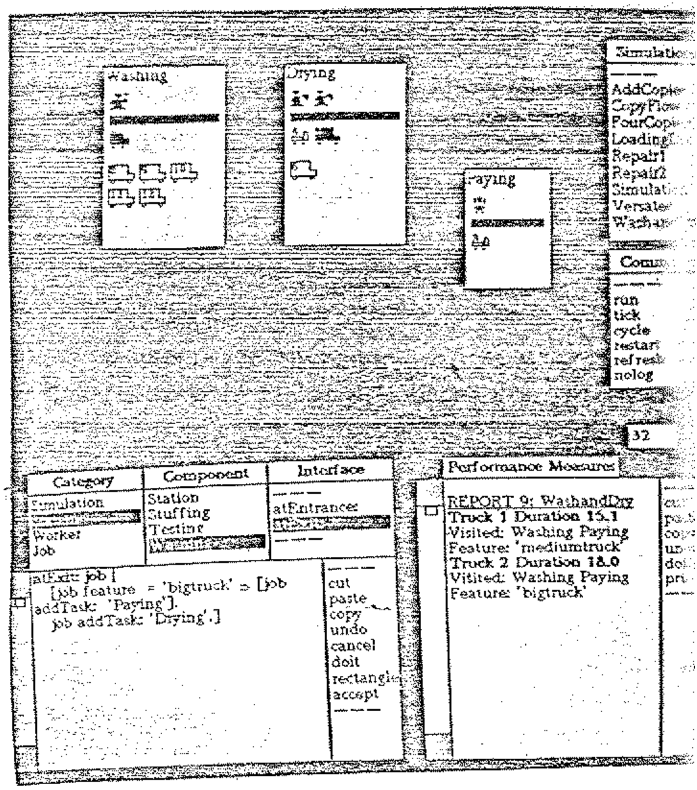
```
day: day month: month year: year [
    [year < 100⇒ [year ← 1900 + year]].
    [month ← self whichmonth:
month⇒ []
    user notify: 'illegal month'].
    [day < 1 or: day > (self
daysinmonth: month)⇒ [
        user notify: 'illegal day in
month']].
    day ← day + (self monthday:
month)]
```

Figures 15A-E. The Smalltalk diagnostic browser. (A) Typing a Smalltalk expression to update today's date information. (B) Smalltalk menu gives command "DOIT". (C) When a condition which requires the user's assistance occurs, the state of the particular process which contains the condition is set aside and a "notify window" appears on the screen. The user may continue with other activities. When the user wishes, the notify window may be interrogated. (D) If it is "drawn-out" larger, it changes into a paned window which is specialized to peer into Smalltalk's dynamic history. (E) The diagram shows the dependencies between panes.

have discussed. A user unhappy with features in an application could very likely fix them at the next (kit) level without having to descend to equally simple but more atomic levels. For example, a kit such as the graphic simulator can be specialized by a user into a specific application for estimating solutions to a business problem. Yet, the simulation application can again be addressed by the user at the kit level when changes are needed. Simulations of this kind have *jobs* flowing through the system that visit one or more *stations* to get service from *workers* at the stations. These entities can be displayed in graphic animation as the simulation is carried out. From my previous remarks we can see that simulations of this type are a special kind of animation. Jobs, workers, and stations are *travelers* in a two-dimentional *space*. Their graphic form, their icons, are *decorations*. Each traveler has its own collection of directives and strategies which global events may affect.

An extensive job-shop system was extracted from *Smalltalk* by filtering objects already present. Major facilities used by this kit were:

- pseudo-time schedulers
- prototypes for stations, jobs, workers, symbolic images
- a statistical distribution package for generating scheduling data
- an animation package
- new display and printing fonts
- a user interface which included
  - a filtered browser for manipulating simulations without having to deal with the rest of Smalltalk
  - a simple editor
  - menus
  - a new error window handler
  - a reporting facility which permitted summaries to be generated and objects on the screen to be probed for information.



```
atExit: job |
   [job feature = 'bigtruck' ⇒ [job
addTask: 'Paying'].
   job addTask: 'Drying'.]
```

REPORT 9: WashandDry
Truck 1 Duration 15.1
Visited: Washing Paying
Feature: 'mediumtruck'
Truck 2 Duration 18.0
Vitited: Washing Paying
Feature: 'bigtruck'

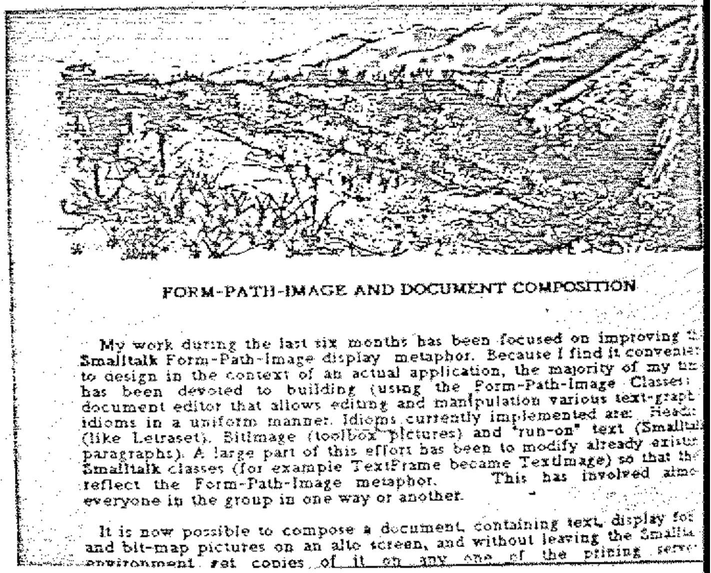
Figure 16. An application system programmed in the Smalltalk simulation kit. On the top right appear dynamically filtered menus for controlling the progress of a simulation. Only currently valid commands are passed through the filter. Below the menus is a viewport through which simulation statistics are reported. Above left is the simulation "playground" showing rectangular "stations" and traveling "jobs" and "workers". A car wash is in progress. Jobs flow in from the left, visit stations, and are serviced by workers. Below the playground is a filtered browser which only shows objects relevant to the simulation kit.

A Next-Higher-Level For Novice Programming

Each of the parts of an object is itself an object. Some of these play the roles of traditional parts-in-a-whole while others maintain relationships between them. For example, a scene with two chairs next to each other could be represented as three objects: one for each chair and one that dynamically maintains the relationship *next-to*. Relational objects are important since it is a lucky programmer with a rare problem indeed that does not need to coordinate the parts of a whole. Both novices and experts have difficulty with interacting parts. Experts can generally find a *kluge* to (more or less) make the right thing happen, while novices just crash and burn.

An example of a straightforward design that turns treacherous is that of a simple document model (Figure 17). In the *Smalltalk* programming style, this consists in making up two objects: the first exhibits the general properties of a paragraph-sized chunk of information; the second has the general properties of a document itself, a container of paragraphs. We can build documents by producing as many instances of these descriptions as needed. So far this is what we would call a "linear" model. That is, the document does not have to know about the details of the paragraphs, and the paragraphs do not have to know about each other or the document that owns them. Text or pictures put into paragraphs can grow or shrink without disturbing their neighbors. But now the user wishes to display parts of the document on the screen and immediately notices that what was once a separable phenomenon has now become intertwined: when paragraphs expand on the screen, they clobber the images of the paragraphs below them. This is the beginning of the end for the novice. A place to put in a conditional will be found for this case, though it will doubtless be the wrong place, and there will be no global strategy for taking care of these kinds of part-part interactions. The descriptions will balloon and reliability will depart.

If the system had relational objects in its part-whole semantics, it would only be necessary for the general paragraph description to state:



FORM-PATH-IMAGE AND DOCUMENT COMPOSITION

My work during the last six months has been focused on improving Smalltalk Form-Path-Image display metaphor. Because I find it convenie to design in the context of an actual application, the majority of my has been devoted to building (using the Form-Path-Image Classes) document editor that allows editing and manipulation various text-graph idioms in a uniform manner. Idioms currently implemented are: Head (like Letraset), BitImage (toolbox pictures) and "run-on" text (Smallta paragraphs). A large part of this effort has been to modify already exist Smalltalk classes (for example TextFrame became TextImage) so that the reflect the Form-Path-Image metaphor. This has involved alm everyone in the group in one way or another.

It is now possible to compose a document, containing text, display fo and bit-map pictures on an alto screen, and without leaving the Smallta environment get copies of it on any one of the printing serv

Figure 17. A Smalltalk document composed of text paragraphs, illustration paragraphs, and others not shown.

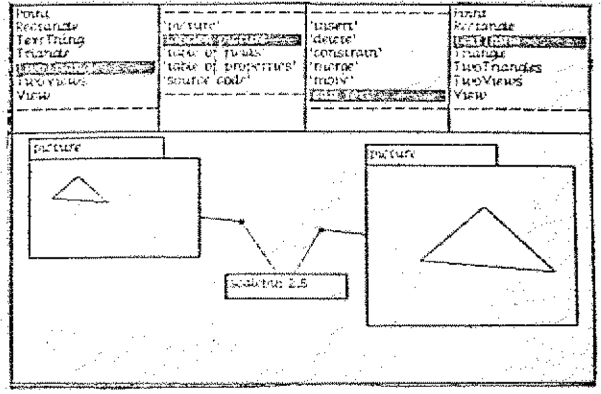


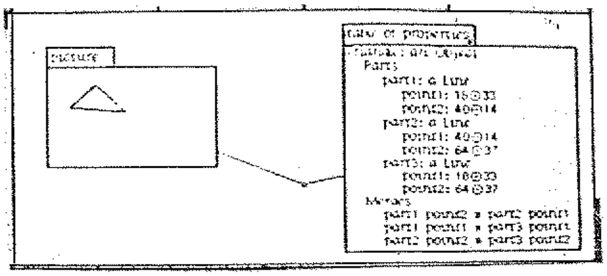
Figure 18A. Multiple views of a relationally described object: a triangle. Here the two views differ only by scale.

"I want the following continuous relation to hold:
my y = above's y + above's height".

Just how the relational object thus created manages to maintain the continuous relation under all possible circumstances is not the province of the user. The use of relational objects introduces an important idea for novice programming: new information in a relational form can be added to an already existing system as though it was linear. The system has to satisfy it or complain; users do not have to puzzle out race conditions and interactions for themselves. The following examples are taken from a *Smalltalk* extended to explore the relational programming of the future [*Relations*].

An example of a simple relation is a linkage formed by merging the endpoints of separate edges together. Polygons are closed linkages. If one part of a linkage is moved, the other parts must follow because they are constrained to be connected. Thus, a triangle, for example, must always remain a triangle no matter how it is pushed and pulled. As a simple example, Figures 18A-B show multiple views of a hand-constructed triangle including automatically generated relational code. An interesting geometric theorem is illustrated in Figures 19A-B. A quadrilateral is formed from edges which have been constrained to have midpoints. If the midpoints are connected, they form a parallelogram no matter how the quadrilateral is deformed, or even turned inside out!

In a similar manner a continuous calculator can be hand-constructed from graphical representations of variables, constants, and operators. The example in Figure 20A shows a conversion from Centigrade temperature to Fahrenheit. The anchors signify constants, that is, values that must remain unchanged as the desired relationship is calculated. If a new number is edited in on the left-hand (Centigrade) side, the result on the Fahrenheit side changes according to the diagram's relationship. The boiling point of water is transformed from 100°C to 212°F. The constraint-oriented nature of the diagram means that it also works in reverse. Editing in a new number on the right-hand (Fahrenheit) side will cause the



```
picture
table of properties
Triangle : an Object
  Parts
    part1: a Line
      point1: 16@33
      point2: 40@14
    part2: a Line
      point1: 40@14
      point2: 64@37
    part3: a Line
      point1: 16@33
      point2: 64@37
  Merges
    part1 point2 = part2 point1
    part1 point1 = part3 point1
    part2 point2 = part3 point2
```

Figure 18B. More multiple views of the triangle. On the left, its picture (and the form in which it was created by hand-editing). On the right, its relationally described code, automatically generated by the composition process.

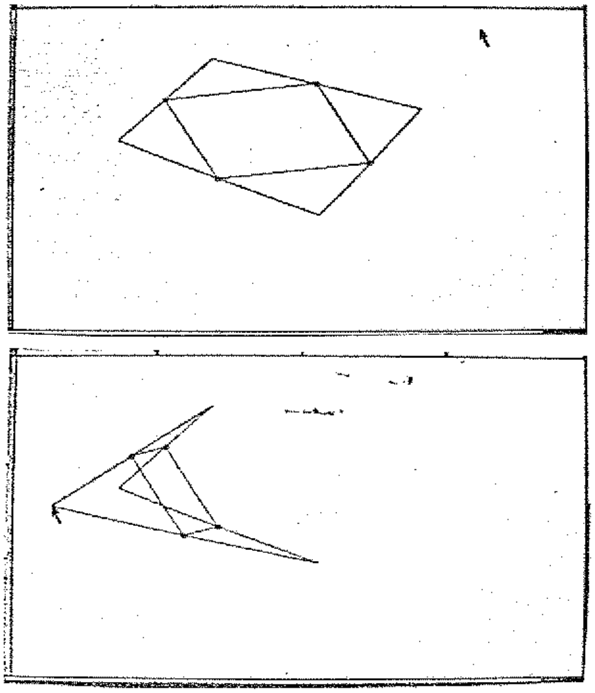


Figure 19 A-B. A hand-constructed midpoint quadrilateral being pulled inside out. The relational system in which this description was formed has no intrinsic information about geometry.

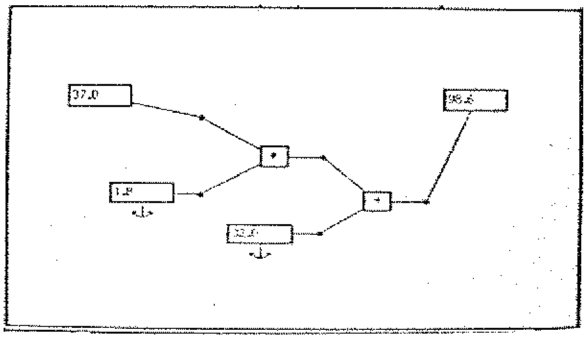


Figure 20A. A hand-constructed graphic calculator. Numbers entered in one side are transformed to balanced numbers of the opposite side.

Centigrade side to change. The temperature of the human body 98.6°F is transformed back into 37°C. Diagrams which have the same behavior may be freely substituted. Thermometer gauge diagrams, themselves constructed by constraint programming, can replace numbers as shown in Figure 20B. The "mercury" in a gauge can be grabbed by the mouse pointer and dragged to change its value. The corresponding gauge on the other side of the calculator changes in response according to the constraints between them.

Another example of constraint programming is "automatic forms", Figures 21A-B. A summary of divisions of a fictitious corporation can be presented as a combination of a tabulation and a bar graph. The height of each bar is constrained to be proportional to the value in the table. If either is changed, the other will change in response. The *Total* field in the table is constrained to always be the continuous sum of the values above it. If the value of *Long* division is edited from 40 to 1940, both its bar height and the total change automatically.

From the standpoint of the novice computer user, constraint programming will likely provide a way to describe and control situations that have not been accessible to this group of users. There are several reasons for this. First, constraints themselves are usually simple expressions of desired relationships. They are *goals* rather than strategies, or, as is most usual in programming languages, actual tactical descriptions. Second, constraints can be added without regard to what descriptions are already present. It is up to the system to either figure out how to merge the new constraints with the old or to complain. Finally, many constrained systems can be constructed without contact with constraints at all. Given a collection of diagrammatic building blocks such as operators and place-holders for values, constructions such as the metric converter can be built by simply "pasting" together a composite diagram with a layout editor.

To sum up, the programming style we think is most successful for novices could be termed "results-mode" or "training-analogies". That is, the user edits a picture which

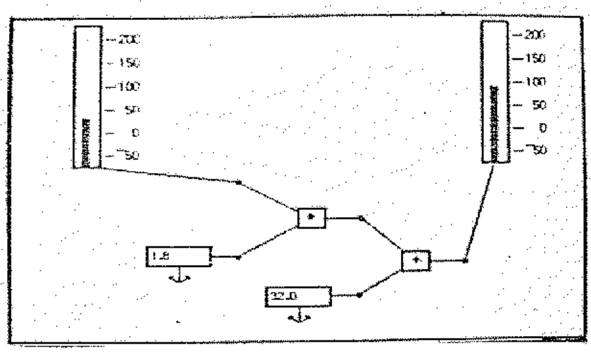
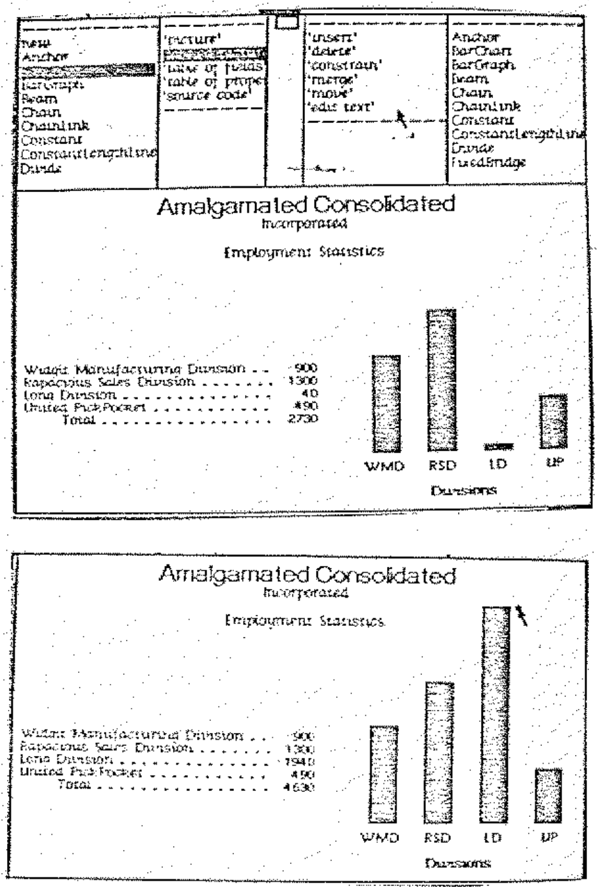


Figure 20B. Thermometer gauges are sematically equivalent to textual numbers. Thus they may be used wherever numbers previously appeared.



Amalgamated Consolidated
Incorporated

Employment Statistics

Widgit Manufacturing Division . . . 900
Rapacious Sales Division 1300
Long Division 40
United PickPocket 490
Total 2730

Amalgamated Consolidated
Incorporated

Employment Statistics

Widgit Manufacturing Division . . . 900
Rapacious Sales Division 1300
Long Division 1940
United PickPocket 490
Total 4630

Figures 21A-B. "Automatic Forms" contructed using relations. Numeric values, bar heights, and totals are continuously related.

has as many properties of the desired effect as can be brought together. Often, as in relational programming, this is sufficient information for the actual program-structure to be searched out by the system, as in Figures 18-22. If the system cannot find a way to carry out the user's wishes, the next level of strategy is invoked. Perhaps the user knows a way to do a critical subpart. If so, hints or actual sequential code may be supplied by the user. The system can use the original relations to check the user's more ad hoc contributions.

Conclusions

We think we have learned a few things from our experience so far.

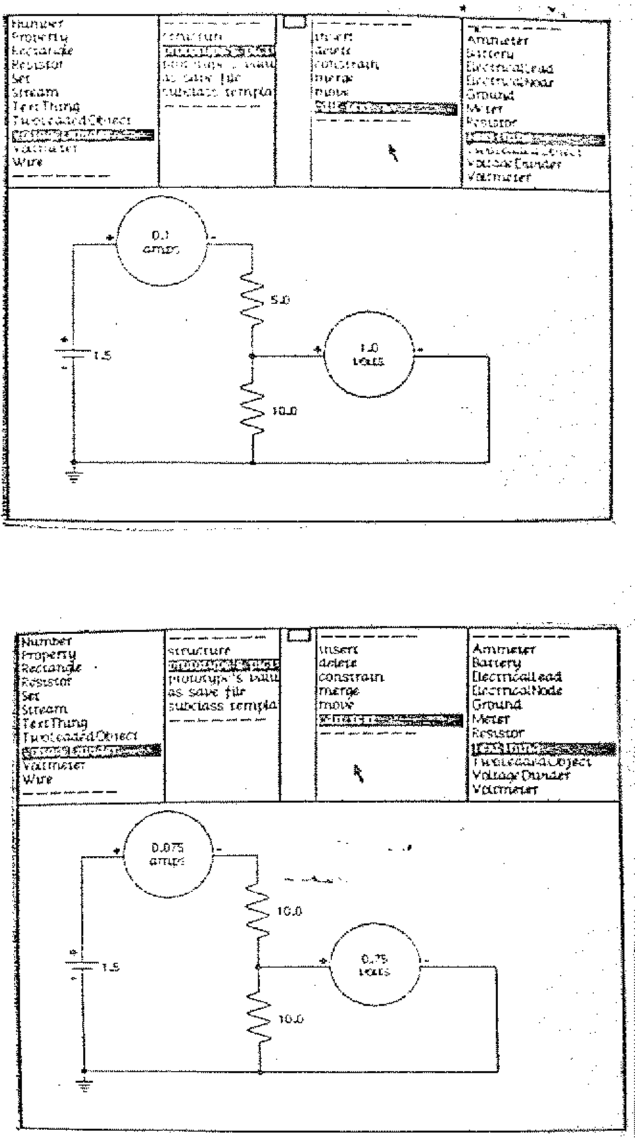
First, we believe that everyone can learn how to program -- and with little effort if the curriculum and programming environment is suitable. Children are particularly good at it.

Second, though young children have appeared to use the concept of subroutining, and older children can accomplish more, it is not until age 11 or so that many of the important implications of programming -- such as general tool-building for future use -- seem to be fruitfully understood. Adult difficulties with learning to program seem to center about finding enough of the right kind of time per day to make a serious start, and to conquer "exposure" and "failure" fears.

Third, *Smalltalk* made it quite easy to notice the enormous distinction between learning how to become a proficient programmer and learning how to design useful programs. We would classify the first as a 10-100 hour skill and the latter as a 1000-2000 hour skill.

Fourth, we have encountered several important short-term memory problems.

- Most beginners can only handle a few pages of program regardless of the language.
- But, beginners can read, understand, and change longer descriptions written by others.
- For learning to program, more ground is covered if novices learn a few patiently taught abstract concepts which may be used everywhere in the future, rather



Figures 22A-B. A hand-constructed circuit simulation. The relations placed on the components dynamically "drive" the circuit: batteries must maintain a difference of EMF between terminals; and wires and resistors must obey Ohm's law.

> than to learn many, though simply taught, concrete ideas which have little range and seem to choke their short-term memory.

To us, this clearly implies that a beginner's programming language should be of a level as high-, abstract-, and as readable- as the state of the art permits.

What values can we find for personal computing in 1979?

Editing of everything -- text, pictures, models -- is both the most indulged in pastime and the one process for which personal computers provide undeniable value far beyond conventional media.

Search, the elimination of distance by connection, is what multi-indexed information systems provide. The user is always browsing for resources. The tyranny of single-dimensional paper organization of information is replaced by multidimensional relations.

Modeling and *Simulation* are what computers are all about. Only limitations in current day user programming prevent this from being the area of most value.

Amplification of Human Communication is the most subtle value of personal computing. The input bandwidths of humans -- visual, audio, and tactile -- are enormous when contrasted with their output bandwidths of gesture and speech. But, complex simulation models producing graphics and sound can approach a viewer's ability to take in information, yet can be easily controlled by gestures.

From these values it is obvious that we should help users learn to:

- edit it until it's as right as they want,
- capture their world and learn how to design new ones,
- find out what's been going on, and
- show others their thoughts by bringing them to life.

We haven't seen much adult or child fear of technology even in those proverbially technologically cautious adults, teachers and businessfolk. I would like to point out that most people who are supposed to be afraid of science and technology are quite willing to drive a car in freeway traffic. To me this shows great faith both in technology and in other people! And it suggests an approach to the technologically fearful adult. I believe that once our field is willing to find destinations and routes for teachers, businessfolk, and other people, which are at least as relevant to their lives as driving, they will happily hop aboard and elbow us out of their way.

One final thought on values and technology. Many musical instruments were first introduced as prosthetics for the human voice: mostly to make more noise and carry further on a battlefield, in a church, in the sports stadium, and theatre. They were worse than what they replaced with regard to dynamic range, change of timbre, being able to sing words, and general musical technique. Gradually, though, they gained value. As composers began to explore their intrinsic artistic worth, they were physically improved and new ways to think about them entered the culture. Musical instruments ultimately became a technology which has brought great value to human life -- but, only when they ceased to be used as prosthetics and instead became amplifiers for human ideas. This is the destiny of the personal computer.

References

[*Dynabook*] A. Kay, *Microelectronics and the personal computer*, Scientific American, Sept., 1977.

A. Kay & A. Goldberg, *Personal Dynamic Media*, Computer, March, 1977

[*Smalltalk*] A. Goldberg & A. Kay, *Smalltalk-72 instruction manual*, Xerox PARC-SSL-76-6, June, 1976

J. Shoch, *An overview of the programming language smalltalk-72*, Convention Informatique, Paris, 1977

D. Ingalls, *The smalltalk-76 programming system: design and implementation*, Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, Jan., 1978

[*Joss*] C. Shaw, *JOSS: a designer's view of an experimental on-line computing system*, AFIPS Proc., Fall, 1964

[*Teaching*] A. Goldberg & A. Kay, *Teaching smalltalk*, (2 papers), Xerox PARC-SSL-77-2, June, 1977

A. Goldberg., *Educational uses of a dynabook*, U. Exeter Symp. on Comp. Ass. Learning, Exeter, Eng., April, 1979

[*Browsers*] L. Tesler, Personal communication, 1977. T. Reenskaug, Personal communication, 1978.

[*Kits*] A. Goldberg & D. Robson, *A metaphor for user interface design*, Proc. U. Hawaii 12th Symp. on Syst. Sci., Honolulu, Jan., 1979

[*Relations*] A. Borning, *ThingLab--an object-oriented system for building simulations using constraints*, 5th IJCAI, Cambridge, Mass., Aug., 1977

A. Borning, *A constraint-oriented simulation laboratory*, PhD Thesis, Comp. Sci. Dept., Stanford U., 1979

3/6 Smalltalk Lesson 2

$\overbrace{\text{frontDoor}}^{\text{object}}$ $\overbrace{\text{withLock:}}^{\text{Keyword}}$
(lock init: 101)

[literal parts end in colons]

Std Form for Statements:
object message

Msg Types

1. unary — front door open
   [single token, single msg]
2. Keywordmsgs — brush paint: pic color: red
3. special — obj ← value
4. binary — $\underset{\text{obj}}{3}$ $\underset{\text{msg}}{+4}$

Method composed of:

1. msgs
2. 2 other objs
3. msg to myself
4. chgs to my state

# of colons = # of parameters
msg keyword becomes unambiguous w/ ()
Values can be assigned to names — Dictionary

To create instances of a Class:

new
init } obj new init
init: } obj new init:

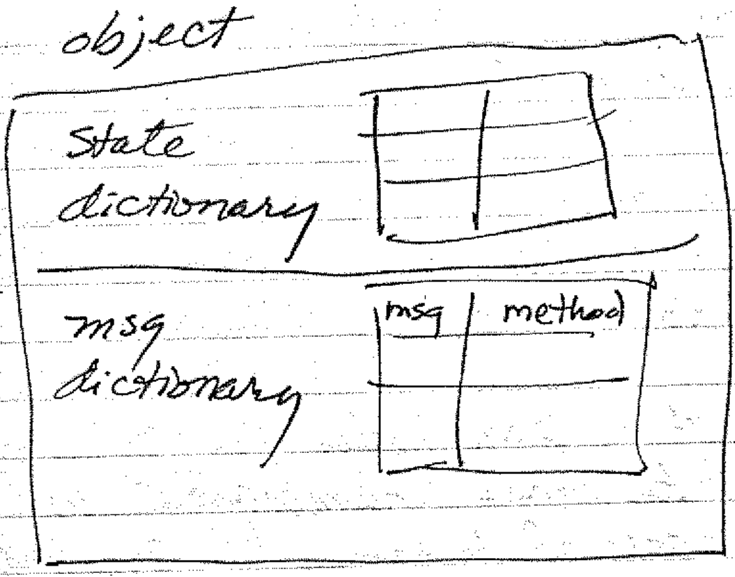
Objs can be strung together w/ ;'s
do not include 1st obj in subsequent occurrences.
(Lock init: 101) setlock
obj can be a parenthetical thing)

Bunch of unary things:
Front Door open close open close
(semicolons need not be included)

When you're programming, you're inventing syntax.

Obj compose of
has behavior + that's shown in msg dictionary
has memory, so there's a state dictionary
Dictionary
has ways of looking things up

Class Like-behaviored things use same msg dictionary



Class Door
Instance   If no value assigned, nil

To create instance of obj: frontDoor ← Door new init
Back Door ← Door init
my Door ← front Door

Name things so that they are retrievable.

Task write def. of obj. in Inventory

Inventory

Problem: track parts ~~items~~ ordered
quantity on hand
when ordered
type/category
according to functionality

Parts: 1. Transistors
Generic P/N
2. Resistors
wattage
value
tolerance
3. Capacitors
capacitance
voltage
type
4. Diodes
Generic P/N

State

names of fields for state dictionary

Problem: 1. Order Date — date
2. Qty. on hand — 0 or more
3. Qty. ordered — ~~received date~~
~~4. Obj context (?)~~

Messages (Kinds of things you want to ask)

1. have you been ordered?
2. Qty on hand?
3. Order level: do we need more of you [order threshold #]
4. When was last shipment rec'd?
5. Qty ordered?
6. What are you? (Print yourself)

Inventory System (remembers all parts)

1. has part p been ordered?
2. Qty. p ordered?
3. List Qty. p to have been rec'd by today?
4. Automatic ordering
5. List all parts ordered since ⟨date⟩
6. " " " needing orders [those over threshold]

...

7. List all parts on hand [current inventory]
what do I have?

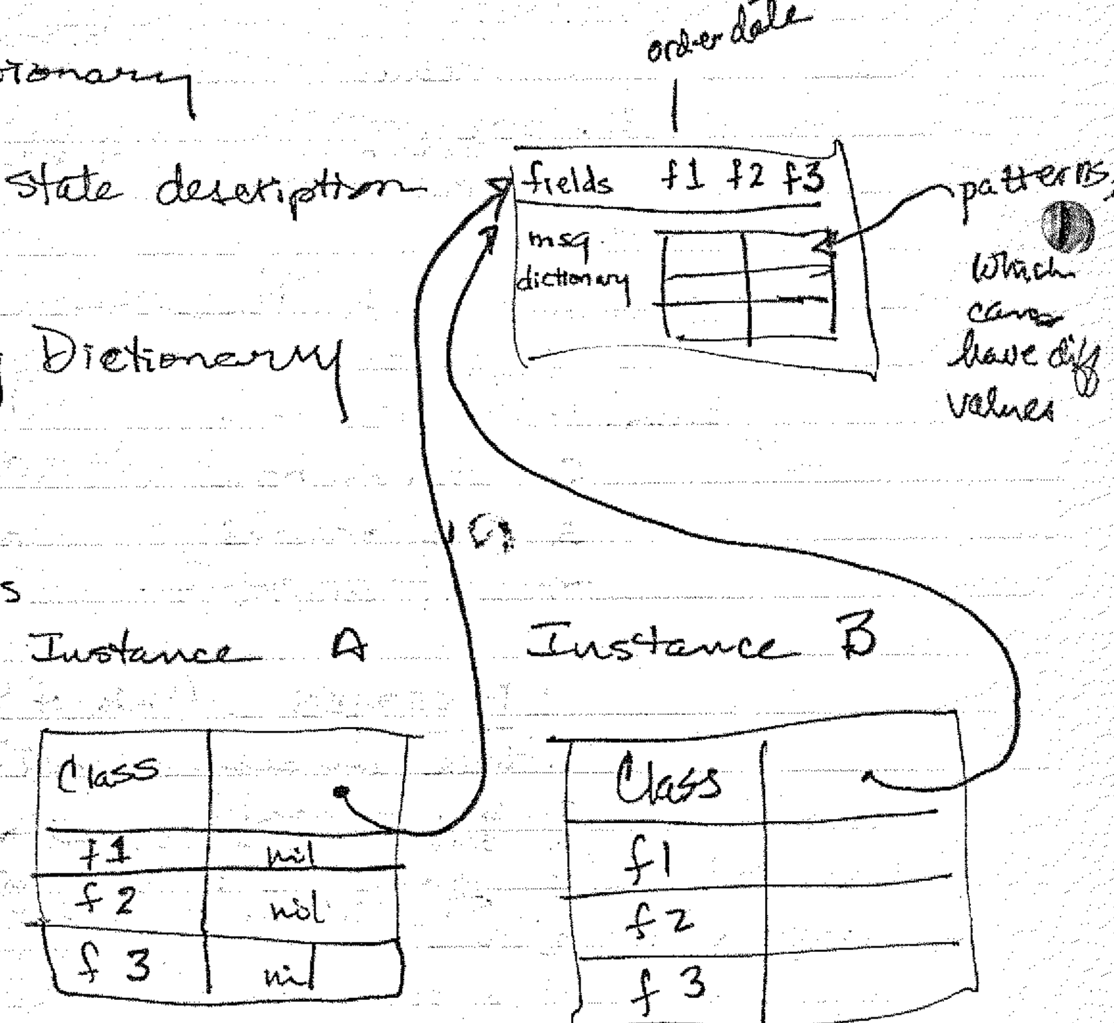
Syntax (Chapt. 4)

2nd pane displays names of all classes
Hash set
Dictionary

Class
Dictionary

Msg Dictionary

Class



Class	
f1	nil
f2	nil
f3	nil

Class	
f1	
f2	
f3	

Special constants: nil, true, false

Instances of same class: state dictionary will be same

Class

Dictionary

field	object	values
	look up :	name
msg	insert :	name
	delete :	name
	insert :	~~name~~
	with :	value

← msgs

← You can insert something w/o value

Instance A lookup: 'joe'

Go to msg dictionary

Every Obj. { state description + msg dictionary

Create an instance of class:

dict

obj	'joe' 'jim'
values	' ' , '

dict lookup: 'joe'

message pattern

lookup: name | X

[X gets* ← self find: name then ⇒ ⇑ values 0 x] ⇑ false]

returns the answers

answers

[ ] groups statements

| divides msg pattern from list of temp. variables

* is assigned

activation record

X	nil

Everything starts out nil

variable - something that takes on diff. values value

lookup: name | x   1

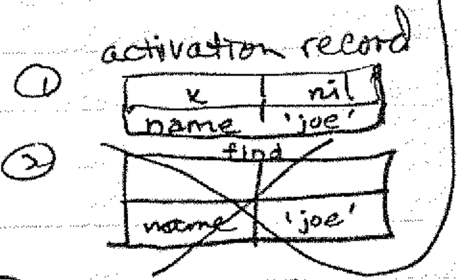
[x ← ~~self find: name~~ ⇒

$[\underbrace{\uparrow \text{values} \circ x}_{\text{then-stat}}] \; \underbrace{\uparrow \text{false}}_{\text{else-stat}}]$

name is an argument; it's temporary

① dict lookup: 'joe'

② self find : name



Is 1 false? No, it's the same as true. Therefore, do what's in 1st set of [ ]

x ← 1

Conditional ~~statements~~ : if-then-else
if-stat ⇒ [then-stat]
else-stats

Iteration Statement

open colon

When you see keyword followed by parameter, value is passed on.

⚬ pass-by reference

: pass-by

How to do Iteration

find : name

3 parameters

① for loop [smalltalK] for : i to: object's length do:

[ ]

for can be considered an obj in smalltalk

obj - expression
expression is sent msg 'eval'
: - evaluate

Forms of for:

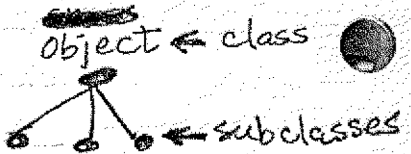
```
for: i to:

for: i ← 3 to: 6
     do: [ ]

for: i ← 1 to: by: 2 do: [ ]
```

② while statements while something is true, do the following

③ until loops until something becomes true, do the following

begin end
[ ]

Why can't you find 'find'?
Subclasses



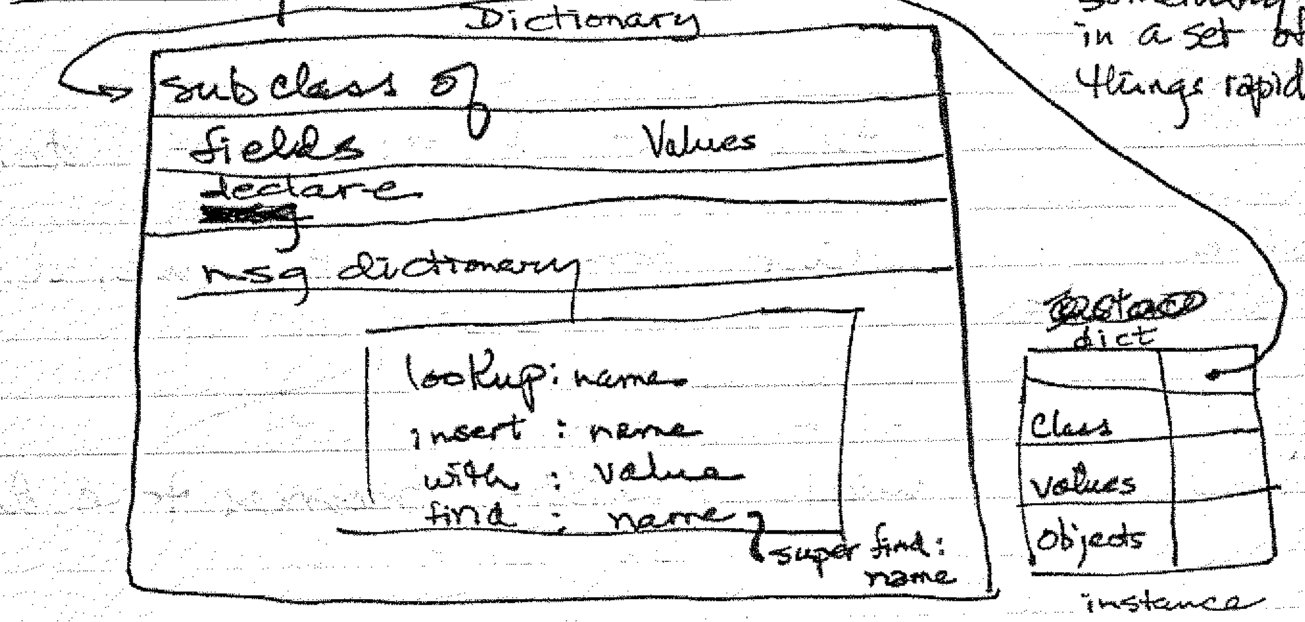
Class (Kind of obj.)

sub class (differs in certain ways)

ways subclass differ from a class
1. added capabilities
2. lesser capabilities
3. diff. method for handling msg
4. added properties state capability

2 definitions
Object Class

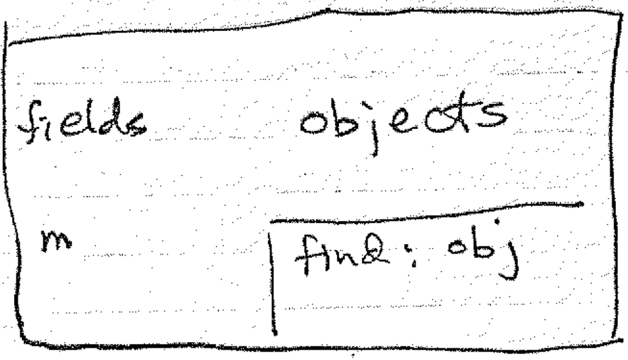
Dictionary is a subclass of Hashset (way of finding something in a set of things rapidly)



Hashset ~~is an object a~~

Hashset is a subclass of obj

Hashset



Instance

dict lookup: 'joe'
self find: 'joe'

If something can't be found in dictionary:
obj doesn't understand msg

Super - special object
super is me, but start w/super class (look higher)

Class Object

In its msg dict: we put things to which it should respond

You can make copies of objects

When you create something, he has no instances + no state (he's abstract), but he's a subclass of object + therefore has minimal properties
Obj is an instance of class

Class Class

- is a subclass of Obj
- allows mechanism for creating instances

Example:

```
Class new title: Door    created as an instance of class
          subclass of:
          fields:
          declare:
```

Instances can't create instances
You can create instances of classes + subclasses.

Task: Implement the class Inventory

- make up msgs
- Define class
- Begin dictionary + methods

3/7 **Smalltalk Lesson #3**

Browsing
Filing / Printing
Define Class/Part
Put text in a rectangle

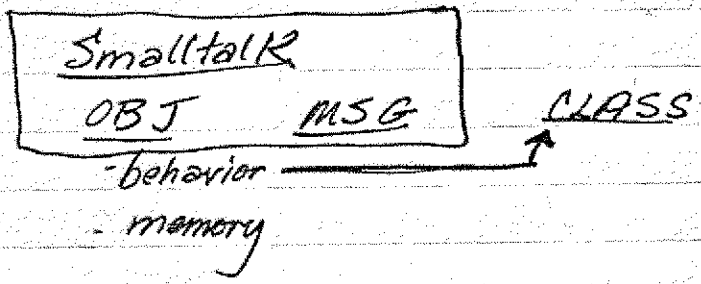
} To be covered today

Class - way of grouping set of objs
description of a Kind of obj

Instance - member of class
particular Kind of obj

Obj : external view - msg protocol
way of defining behavior

internal view - memory
state dictionary
+ methods for responding to msg



Syntax

1. Syntax for msg sending
2. Templates for obj description
3. Syntax for control

1. obj message .

a. unary msg : joe jump
b. binary msg : 3 + 4
c. Keywork : contain :'s, parameters

Point: way of storing 2 numbers in relation to ea. other

2. msg pattern | temp
[statements]

3. : passes a reference to a # rather than a value
for : i to 10: 10 do: [ ]

- ● while-loop
- ● until-loop
- ● assignment statement

<
≤ dg ← object
>
≥ a ← 10

; enables cascading (stringing msgs together)

● if-~ ⇒ [~] false

" " used for comments
' ' used for strings
↱ followed by a token means that what follows it is a name
{ } to make list of things
{↱ a, ↱ b, 10}

Homework Discussion

Blue Bug: prints whole thing
Yellow Bug: prints relative to editor - contents

Classes Window

All Classes - alpha listing of classes

Classes
Examples
Fields: orderDate
on Hand
Ordered
order Level } compile

Class Organization

The inventory system is made up of parts. This class handles order dates and inventory count
(As got unclassified)

Initialization

UserView Workspace

don't compile in UserView workspace

```
diode ← Part new
  select DoIt

diode inspect
  Select DoIt
  [inspect opens state descrip dict (It has no
    state descrip; it's nil)]

diode order: 5 on: date.
diode onHand.
diode orderWaiting.
diode needed.
diode howManyOrdered.
diode print.

select diode print (not .) and DoIt
```

Return to Browser

Teach it how to tell us what's on Hand

Msg name and arguments: onHand

Short comment: Tell how many parts are onHand

[redacted] [↑]

compile

OnHand should have been a Query,
so cut it out of incorrect place and
paste it in Query

Initializetion

```
init: onHand | "initialize a part w/qty currently onHand"
    [numberOrdered ← 0.
     orderLevel ← 0.]
```

compile

User View workspace

```
    diode ← Part init: 200.
```

bug diode inspect.

Classes Browser Window

```
Class new title: Diode
    subclass of: PART
    fields:        Part No.
    declare:

    compile
```

UserView WorkSpace

```
    diode ← Diode init: 200.
```

compile

do it (to inspect)

Classes Browser Window

bug Examples

bug diode

bug Class organization

Enter: This class is a Kind of part Ea instance has a generic P/N.

Initialization

compile

```
Initialization
    init: partNo wite: onHand |
      [super init: onHand]
      [orderLevel ← 5.
      numberOrdered ← 0.]
    compile
```

User View Workspace

```
    diode ← Diode new init: ('IN914' unique) with: 200
```

do it

```
diode order: 5 on: (Date new day: 27 month:
    2 year: 79).
```

doit

Classes Browser Window

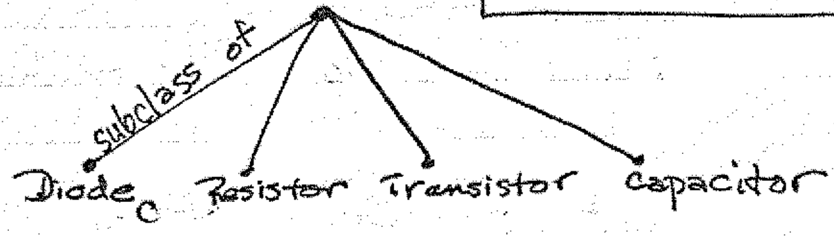
```
Order: numberOrdered on: orderDate
```

3/8 Smalltalk Lesson #4

Class Part
State orderDate
onHand
numberOrdered
orderLevel

init: onHand	*
order: numberOrdered on: orderDate	
onHand	
moreNeeded	

```
⇑ onHand < orderLevel
*orderLevel ← ∅
numberOrdered ← ∅
```



state Part No.
init: partNo with: onHand

To create an instance of a class: new
To create new part:
Part init: 200

To create new diode:
p ← Diode new

new
init ≡ new init
init: p ≡ new init: p
default ≡ new default

Error msg

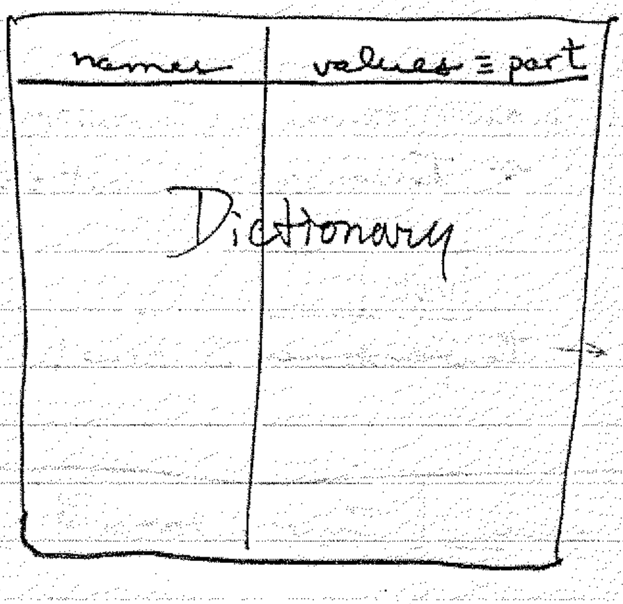
Obj being sent msg / Class
init: with:

Class new title: Diode
subclass of: Part
fields: 'part No'
declare: ''

Diode is a subclass of Part, Obj
Diode is an instance of Class

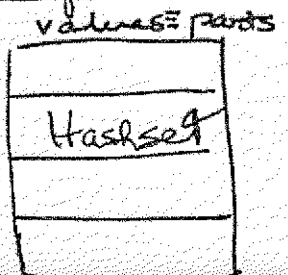
Object every object
↑ ↓
Class all classes are its instances

~~Classes new~~
Create a
Dictionary



or

List of values, where values are parts



class Inventory A

state	portsList	

is a Dictionary

subclass of: object

class Inventory B
subclass of Dictionary

Is P in inventory? (P = P/N)
# of P's onhand?
# of P's ordered?
When more P's ordered?
Yield transistors needing to be ordered

To create Inventory B:
Tech lab is an instance of Inventory B / Dictionary objects / Hashset Values subclass of:
tech lab ← InventoryB init: 50.

init	

techlab ← Inventory B new!

Class	Inventory B
Objects	
Values	

To put things in Dictionary

```
                  name
techlab  insert: ('1N924' unique)
         with: (Diode new
           value
         init ('1N924' unique)
         with: 300).
```

```
techlab lookup: name
techlab contains: name
```

contains: — must be chg'd in msg protocol for Inventory B's Dictionary

```
techlab onHand# for: name
techlab lookup : name
                sets
   [⇑ (lookup:name)
              onHand]
```

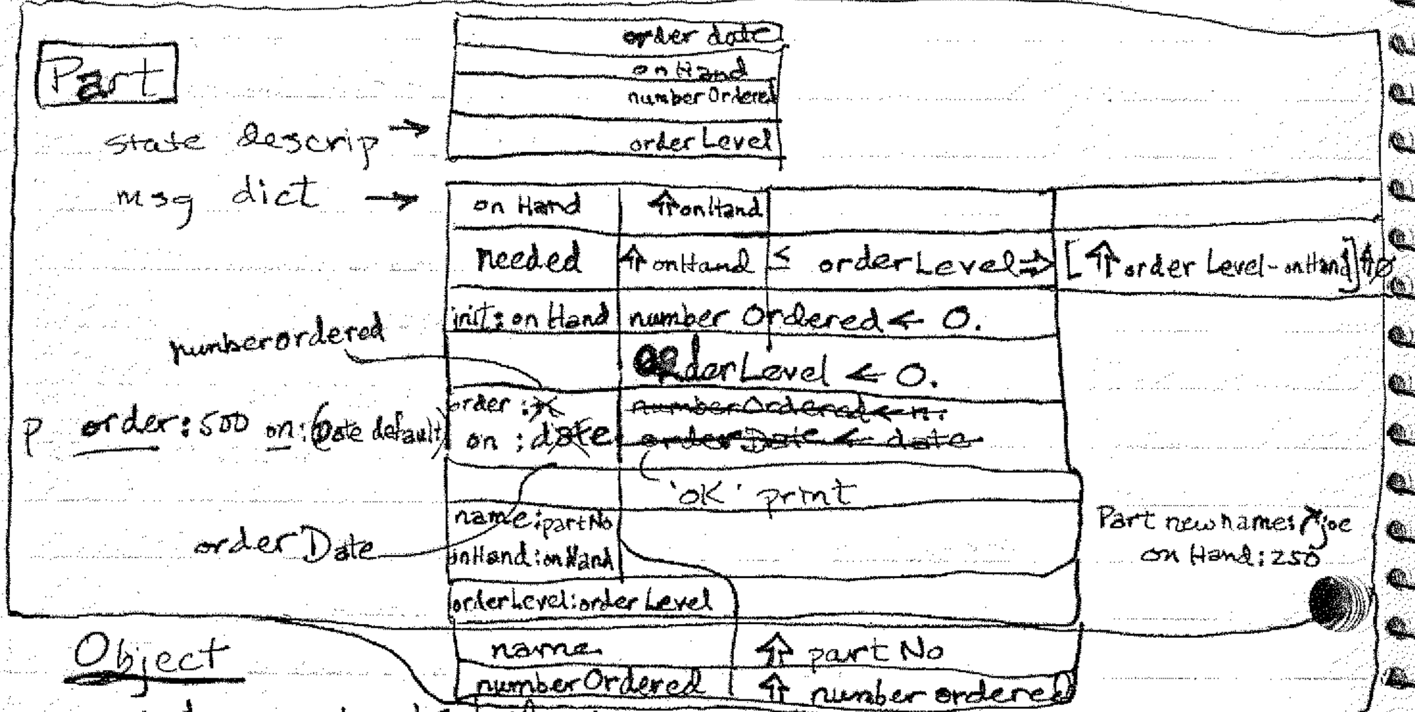
```
onHandfor : name | t

   [t ← self  lookup: name ⇒
      [⇑ t onHand] ⇑ false]
```

Categories of Class Definitions

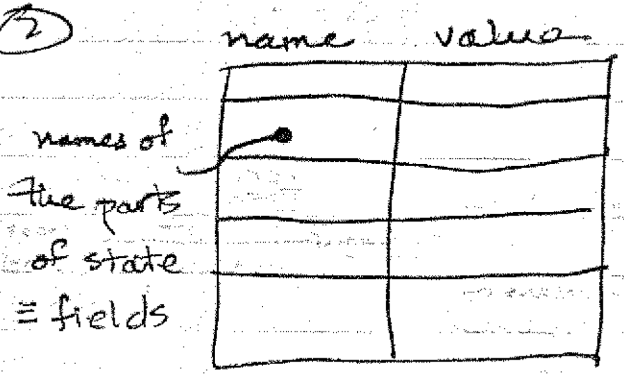
- Text Objects
- FPI (form path image) — Form, Path, Image — Document — BitImage - to draw pix (most things I'll do will be a subclass of Document
- FPI Packages
- DispFrame, Stream (subclass of)
- Paragraph (formatting)
- Reader (reads Keyboard & allows one to use dispFrame)
- Textframe
- TextStyle (to stipulate font)
- Text Image (subclass of Text frame) - handles editing, e.g., cut, Paste

BitRect Editor to draw pictures
(preferable to use Bit Image)

3/9 Smalltalk Lesson #5



Object
- demonstrate behavior
in terms of msg patterns / responses ①
- have memory
in terms of state description ②



Diode subclass of: Part

msg	name: partNo onHand: onHand	partNo ← ('D' + partNo as String) unique. supername: partNo onHand: onHand

'D' + 'Joe'
'DJoe' unique
↱ DJoe

Strings understand + to mean concatenate.

$5 \equiv \boxed{\overset{\text{obj}}{\text{diode}} \overset{5}{\ } \overset{\text{msg}}{\text{onHand}}}$ print

$\text{aDiode} \equiv \boxed{\underbrace{\overset{\text{obj}}{\text{diode}}}_{\text{obj}} \overset{5}{\ } \overset{\text{msg}}{\text{onHand}}};$ print

msg

diode gets the msg print
; cascades msgs to 1st obj.

Diode, Part, Object

write out comments in English first.

PartNo ← ('D' + partNo as String) unique
'1n56'

techlab insert: ↱D1n56
with: (Diode new init: '1n56' with: 200)

techlab add: (Diode new init: ↱Joe with: 200)

This msg must be in Dictionary

add: self insert: (n name)
with: n.

techlab ← Inventory init: 5

add 'add:' to ~~msg~~ dictionary of Inventory

techlab allPartsOnHand. "list all parts & their # onHand"

msg

comment

for every obj in dictionary, print out name and number ordered

[for: i to: objects length do:
    [CR (objects o_i) print. ~~tab~~
    ~~(values o_i)~~ onHand ~~print.~~]
    (values o_i) number ordered
    (values o_i) needed ⇒ [ ]
    ∅ print]

yields

Joe	TAB	200
Jim	TAB	150

print all: name of class