# The Evolution of Smalltalk

From Smalltalk-72 through Squeak

DANIEL INGALLS, Independent Consultant, USA

Shepherd: Andrew Black, Portland State University, USA

This paper presents a personal view of the evolution of six generations of Smalltalk in which the author played a part, starting with Smalltalk-72 and progressing through Smalltalk-80 to Squeak and Etoys. It describes the forces that brought each generation into existence, the technical innovations that characterized it, and the growth in understanding of object-orientation and personal computing that emerged. It summarizes what that generation achieved and how it affected the future, both within the evolving group of developers and users, and in the outside world.

The early Smalltalks were not widely accessible because they ran only on proprietary Xerox hardware; because of this, few people have experience with these important historical artifacts. To make them accessible, the paper provides links to live simulations that can be run in present-day web browsers. These simulations offer the ability to run predefined scripts, but also allow the user to go off-script, browse the details of the implementation, and try anything that could be done in the original system. An appendix includes anecdotal and technical aspects of how examples of each generation of Smalltalk were recovered, and how order was teased out of chaos to the point that these old systems could be brought back to life.

CCS Concepts: • **Software and its engineering → General programming languages**; • **Social and professional topics** → *History of programming languages.*

Additional Key Words and Phrases: Smalltalk, Squeak, Alto, NoteTaker, OOZE, BitBlt, Morphic, EToys, Objects, Blocks, Bytecode, Interpreter, Virtual Machine, Bootstrap

**ACM Reference Format:**

Daniel Ingalls. 2020. The Evolution of Smalltalk: From Smalltalk-72 through Squeak. *Proc. ACM Program. Lang.* 4, HOPL, Article 85 (June 2020), 101 pages. https://doi.org/10.1145/3386335



## 1 PRELUDE

I am the luckiest person I know in computer science. Over a period of more than thirty years I have had the opportunity to design and implement six generations of Smalltalk systems, and the joy of working with wonderful like-minded people who shared my excitement and commitment to making every one of these systems remarkable in one way or another. In each case it was Alan Kay who carried the standard of realizing the potential of computers to make us better thinkers, and

Author's address: Daniel Ingalls, Independent Consultant, Manhattan Beach, CA, 90265, USA, DanHHIngalls@gmail.com.


2475-1421/2020/6-ART85
https://doi.org/10.1145/3386335

committed his time and energy to secure the resources to do what it took, year by year, to realize that dream. This paper can be read as a companion to Alan Kay's earlier HOPL paper [Kay 1993]; it presents my personal recollections of the work and the forces that shaped it.

Many people have been involved in the work on Smalltalk, and many interesting Smalltalks have been created over the years. This paper focuses on the work that began with Smalltalk-72 at Xerox Parc, and traces its evolution through six generations, culminating in Squeak and the EToys tile-programming environment that it hosted. The core team of half a dozen people or fewer remained remarkably stable over more than two decades, and this has helped me to tell the story in a coherent manner. All the Smalltalks except Squeak are named by the year in which they were designed, and in each case they were running and in use the following year. Squeak was designed and released in 1996.

Contents



My involvement with Smalltalk began in 1971. I was selling a system that I wrote as a spin-off from a project I had worked on with Don Knuth at Stanford University. This was the day when Fortran programs were still written on punched cards and read into room-sized computers to carry out the early automation of business and research. My system would read in those programs, insert monitoring statements at branch points, and then run the program as it was intended, while accumulating data about the execution. When the program terminated, my system would correlate the monitoring results with the original program and print out a source listing annotated by exact execution counts and estimates of time spent on each statement. Because it presented its results on the programmer's original listing, it was simple to understand, and therefore easy to use. It was possibly the first source-level execution profiling tool.

To develop this profiler on different platforms of the day, I would rent time at the various service bureaus around Palo Alto to make sure that it worked well on IBM, CDC, GE, Univac, and other mainframe computers. Time was cheaper at night, and the turnaround time was better, so that is when I carried out this testing. One night I ran into a fellow named George White who had just been hired to work on speech recognition at a new lab called Xerox Parc. He was working on a program to segment an utterance and identify phonemes based on the frequency spectrum at each instant. He wasn't sure where it was spending its time, so I offered to make him my test case for the evening. The results were enlightening to him and, though he didn't offer to buy my program, we parted as good friends.

A few days later I got a call from this George White, explaining that he had just taken delivery of a pile of computer equipment including a Xerox Sigma 3 minicomputer, and he thought I might be able to help him turn it into a speech lab at Xerox Parc. This sounded like fun, so I took the job. With my Fortran knowledge I was right at home on the Sigma 3. George had also hired a Stanford intern named Ted Kaehler, and we worked on the project together. I wrote a text editor and job

entry system so that in a couple of months we could do rapid prototyping of George's speech recognition schemes, and display the results on a Tektronix graphics terminal.

In our spare time, Ted introduced me to APL [Iverson 1962]; he had access to an APL system at IBM [Pakin 1968] from an earlier internship. In turn, I introduced Ted to Meta II, Val Schorre's amazing tiny compiler compiler [Schorre 1964]. Schorre's paper shows how to write a simplified Algol compiler in about a page, and finishes by exhibiting a Meta compiler that will compile itself in less than half a page. The day I read that paper, I wrote a version of Meta in Fortran for our Sigma 3.

At the time, my experience with computer science was limited. I was familiar with Fortran and APL, and had written assembly code for the PDP-8 and HP 2116. I had never really used a Lisp system [McCarthy 1978; McCarthy et al. 1965], except to write a REVERSE function as an exercise, and neither had I read about nor used Simula [Nygaard 1970; Nygaard and Dahl 1981]. I think, though, that both these deficiencies may have worked as strengths. Knowing less, I generally added something only if we actually needed it. Compared to Lisp, Smalltalk made classes special, whereas they could have been just another level of scope. Ironically I think this made things easier to understand. Just as serifs make some fonts easier to read by offering recognizable islands in text, so I think it helps to think about classes and instances, even if there is a way in which "everything is the same." Compared to Simula, Smalltalk may have been a "weaker" language in terms of types, concurrency, and other behavior, but from the beginning Smalltalk was "alive" in a way that Simula was not.[1]

In spite of my relative innocence, I will note that Alan knew just about every computer language and system; if I were doing something wrong or leaving something out, he would give me a nudge to rethink things.

There was an air of excitement everywhere around Xerox PARC. Major resources were available for an open agenda to invent "the office of the future," and remarkable people were being collected to discover what it was and to make it real. We would frequently bump into Butler Lampson and his crew from Berkeley Computer Corp (developers of the Scientific Data Systems SDS 940 computer [Lampson et al. 1966]), Bill English and his crew from SRI (prime movers behind the "greatest demo ever" [Doug Englebart Institute 2008; Engelbart and English 1968]) and Gary Starkweather, the genius of quality laser printing [Starkweather 1997]. Alan Kay had also been hired, with ideas about personal computing that resonated well with the Butler Lampson contingent.

While working for George White, Ted Kaehler and I used an office across the hall from Alan Kay's office, and we frequently found ourselves more interested in what we overheard from Alan than in our work on speech recognition. We fell into a hallway conversation that I think was started by talking about the beautiful compactness of APL and Meta; the spirit of *Small is Beautiful* [Schumacher 1973] was in the air.

We knew that Alan wanted to make a computer for children—"children of all ages," to use his phrase [Kay 1972]—based on sending messages. Alan had just come from the Stanford AI Lab, where various computing systems had been cobbled together, and he had seen that sending messages was the key to reducing complexity by not allowing one part of a system to depend on the internals of other parts. This kept the level of complexity proportional to the number of components rather than to the square of the number of components. And so our hallway discussion came around to

[1] In the 1970s, we spoke of Smalltalk as being "reactive," but I will also use the current term, "live," to denote the property of being immediately responsive to changes.

the question of how small a language one could write, based on messages, that would be capable of creating the entire worlds that we foresaw in the future of computing. Alan, who knew about McCarthy's Lisp interpreter in a half page [McCarthy 1960, §3f], declared "No more than a page of code." Ted and I said "show us," and Alan went home for the day.

For the next two weeks, Alan would come in around 4am, work on the design, and then chat with several of us about his progress. Around day 8 he had something that seemed workable. It was more symmetric with regard to send-receive, but for various reasons this was abandoned in favor of a more expression-oriented functional return style. The essence of Alan's original design appears in Appendix II of his paper at HOPL II [Kay 1993, Figure 11A.3]. It was not much more than a page, and we all agreed he had won his bet.

## 2 SMALLTALK-72

The possibility of such a simple and novel language being within reach was incredibly exciting. I immediately went home myself and wrote a simulation of Alan's interpreter in Basic (the only language to which I had easy access at home), as a proof of concept. I coded up a token scanner, list maker, and a couple of class tables. After a few days it evaluated `6 factorial`,[2] and we could see that it was going to work. In Alan's words, "all of a sudden there was something new to program." This was the birth of the language Smalltalk-72.

The rules of Smalltalk-72 are few, and well-summarized in Alan's HOPL paper [Kay 1993, p.534].

- Everything is an object.
- Objects have their own memory (in terms of objects).
- Every object is an instance of a class (which is an object).
- The class holds the shared behavior for its instances.
- Objects communicate by sending and receiving messages (streams of tokens).
- To evaluate a token stream representing an expression, control is passed to the object represented by the first token, and the remainder of the stream is passed to it as a message.

### 2.1 Implementation

I managed to wrangle access to a Data General Nova computer from the Polos group.[3] It had a hard disk, and I got one of the techs to hook up a modem so that I could call in to it from home. This was all I needed to write a serious version of the interpreter in Nova assembly language. The main components of that interpreter were

- a routine to read and tokenize text,
- a managed object memory,
- an architecture for classes and instances,
- a structure for code (we used vectors and sub-vectors), and
- the evaluator that Alan had designed.

[2] This test sent the message `factorial` to the object `6`, which would in turn send `factorial` to the object `5`, and so on, testing recursive calling and returning in the context of a message-sending system.
[3] Polos (for Parc Online Office System) was a project at Parc led by Bill English. Bill had come from Doug Englebart's group at SRI, and the project was exploring Englebart-like systems for office use.

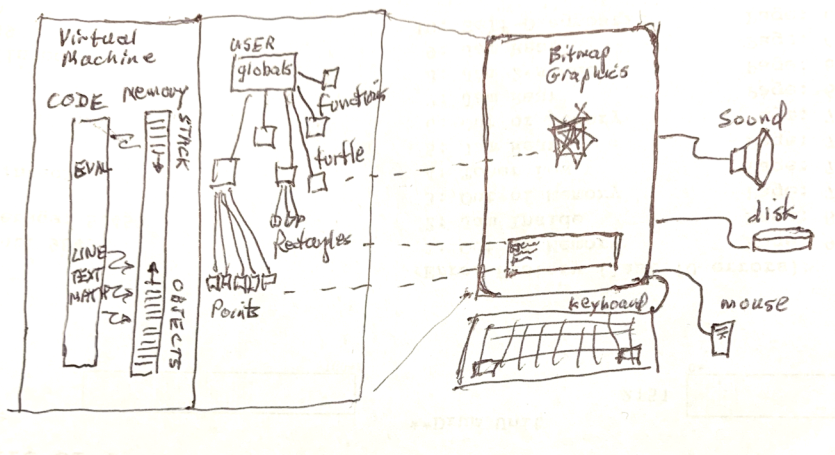


Fig. 1. The basic architecture of the Smalltalk-72 system. Shown are the Virtual Machine, containing the object memory and activation stack, the USER class pointing to global classes and other objects, the screen, with a dispframe for type-in and print-out, and the disk, mouse, keyboard, and a speaker.

The fledgling universe created by this interpreter consisted simply of objects that could send messages as sequences of tokens, and which could receive and recognize messages, somewhat in the spirit of Meta II. This created a language extensible in both its syntax and semantics.

The makeup of the Smalltalk-72 system is shown in Figure 1. It consisted of an object memory, or *image*, organized with direct 16-bit pointers to each object. Each object began with a class pointer followed by instance state. Strings (within string quotes) and vectors (within parentheses) had a length word following their class pointer. Classes were hash tables that described their instances, and a function was simply a class with no instances. Each name in the table was associated with one of three values:

- an offset for temporary variables stored in the activation
- an offset for variables stored in an instance
- a direct pointer for variables stored in the class

Each class included several class variables not shown to users:

- DO: the code for this function or class
- TITLE: the name of the function or class
- ARSIZE: the size needed for the activation record, with temps
- SIZE: the size of instances of the class; all instances of a particular class were of the same size, except for arrays and strings

One class, USER, held all the global objects as its class variables. Thus it served as an "index" of all the classes and other objects in the image. The state of each message send was specified by a stack frame that we referred to as an *activation record*, or simply an *activation*. The activation records formed a stack that grew down from the top of the object memory, while all other objects were allocated upward from the bottom. The interpreter itself followed the rule for interpretation with objects sending messages to each other to change their state, create new objects, or invoke primitive behavior. Primitive behavior was invoked by the token CODE followed by an integer; this

induced the abstract interpreter to perform concrete behavior such as arithmetic, logic, input from the keyboard, and output to the display and disk devices.

I worked on the object memory, the read routine, and the basic evaluator. The read routine read characters from the keyboard or elsewhere into a vector of atoms (names and non-alphanumeric characters), numbers, strings, and subvectors. The name "vector" was a holdover from APL. In source code the contents of a vector were delimited by parentheses, and stored as simple arrays with a length field. Diana Merry, who had been with Alan since before I joined Parc, worked with Alan on text display and some rectangle operations necessary for scrolling and clearing areas on the screen. Ted Kaehler came back from graduate school over Christmas 1972 then wrote line-drawing routines for the bitmap display.

Meanwhile, Altos (described in Appendix F) were being built, and would soon become available to us. The Alto had a 606×808 pixel black and white bitmap display and the possibility of writing microcode for important kernel routines. Butler Lampson and other folks in CSL had put together a development system for the Alto that ran on our Data General Nova minicomputers. It provided an operating system, text editor, assembler and loader, and SWAT, a debugger that we used mainly for its ability to dump all of memory to the disk in only a second, and to load it all back with similar speed. This development system enabled us to write and debug code while the Altos were still being built, since the Alto microcode included an emulator that would run the same Nova code that we had written and tested. Finally, in April of 1973, the first few Altos became available to us. Within a few days we had Smalltalk-72 up and running on the beautiful Alto bitmap display.

### 2.2 Hop into the Tardis[4]

Let's go back to 1973, put our disk-pack into an Alto, and start up Smalltalk-72. To run the live Smalltalk-72 simulation, visit https://smalltalkzoo.thechm.org/HOPL-St72.html?snippets.

### 2.3 Simple Examples

Figure 2 shows an Alto Smalltalk-72 screen. At the bottom is a scrolling text area where a miniature icon of the Alto beckons users to type a Smalltalk expression. Several such expressions can be seen along with their printed or graphical results. We discuss them here for the benefit of readers without access to the simulation.

```
3+4 !
7
```

We always typed this as our first evaluation after a new build. It tests a lot of the system; reading from keyboard, building and `eval`'ing vectors (more later), and printing to the display. This is a chance to explain to a first-timer that this sequence of symbols means: start with the number `3`, and pass it the message `+4`, which is something that `3` understands. After the *do-it character* (the bold "**!**"), it prints the result and waits for more input.

```
355.0/113 !
3.14159292
```

This second test runs floating-point arithmetic and the complex floating point print routine. At this point we could be more confident that things were working right.

```
go 100 !
```

[4]The "Tardis" is a time-travelling telephone box central to the television series *Dr. Who*.

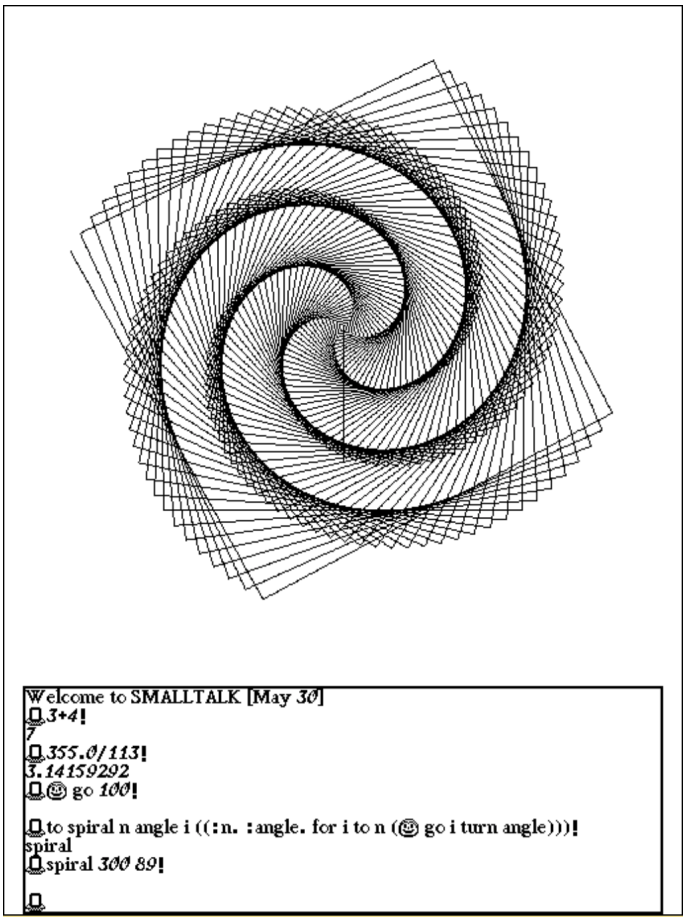


Fig. 2. A screen-shot from a (simulated) Smalltalk-72 system. The text area at the bottom shows several Smalltalk commands; the line drawing is the result of go 100 and spiral 300 89. To run this system live, click the image or visit https://smalltalkzoo.thechm.org/HOPL-St72.html?snippets

Yay, a line appears—the vertical line at the center of Figure 2. For new learners, this was often the first thing typed. We greatly respected the work of Seymour Papert, and made sure that Smalltalk-72 included a class named "turtle," of which ☺ (we called her "Smiley") was an instance. Already the learner is thinking about where ☺ is located, and what go 100 looks like as a line.

Next in Figure 2 is a function designed to give Smiley some real exercise.

```
to spiral n angle i (:n. :angle. for i to n do (go i turn angle))!
```

We followed the Logo usage of "to" as the command for defining a function, as we do with ordinary verb definitions in English ("to speed is to drive fast"). The name of the function being defined is followed by declarations for temporary variables—n, angle, and i—and then by a vector containing the actual code for the function. The defining function "to" returns the name of the function.

The code begins with the pattern :n.. The function : here fetches the next value from the incoming message stream, and binds it to the following name, n. The same thing happens with :angle, after

which we have n iterations of the for-loop body (🐢 go i turn angle) with the distance i increasing by one on each iteration.

```
🖳spiral 300 89❗
```

This expression looks up the name spiral, activates the function so named, and passes the remaining tokens (300 89) as a message stream to that function. With these values bound to n and angle, the for-loop produces the gratifying image in the upper portion of the screen shown in Figure 2.

### 2.4 Message Parsing

We can learn more about the message parsing mechanism of Smalltalk-72 by taking a look at the definition of for. We can do this by asking the Smalltalk system to show for:

```
🖳show for❗

to for token step stop var start exp (:☞var.
    (◁←⇒(:start.) ☞start←1).
    (◁to⇒(:stop.) ☞stop←start).
    (◁by⇒(:step.) ☞step←1).
    ◁do. :#exp. CODE 24)
```

This function declares six temporary variables; they are all accessible to the primitive CODE 24, as in dynamically bound Lisp or Logo.

Functions and other objects in Smalltalk-72 need to be able to recognize (i.e., parse) their input message streams. To accomplish this, several primitive functions are available.

The primitive function : fetches values from the message stream; it is seen here in three variants.

The variant that appears first is :☞ (pronounced "fetch-quote"); it fetches the very token that is next in the message (the atom i in our spiral example) and binds it to the following named variable var.

The most common variant is simply : (pronounced "fetch"); it will begin a new evaluation in the message stream, and bind the resulting value to the variable following :. We see this in the definition of for as :start, and :stop, just as with :n and :angle in definition of spiral.

The final variant shown is :# (pronounced "fetch-bind"); it fetches the next token in the message stream, (an embedded literal code vector in this example) without evaluating it. The :# variant differs from :☞ in that, if the next item in the message stream is an atom , then :# will look up its value, whereas :☞ will not.

It's worth noting that all these forms of fetch may be used with no target variable, in which case they simply return their value, presumably to be consumed in some other expression. Thus :stop could have been written as ☞stop ← :. With hindsight the latter pattern is better style. This "autobind" feature is particular to the fetch primitive, and not a general pattern; thus 10 start will not bind start to 10. We often discussed doing without the autobind feature of fetch, as it would have made the language a bit simpler to implement and to teach, but we had a bit of an APL-ish taste for the brevity of this idiom. For example, a factorial function might be defined as:

```
🖳to fact n (:n = 0 ? (⇑1) ⇑n * fact n–1)❗
```

Here, :n fetches the value from the incoming message stream, binds it to the variable n, and proceeds with the message = 0 as the beginning of the code body. Our young programmers were quick to

learn the autobind pattern and they, like us, were happy with its brevity. I am not an educator, and I care mainly that syntax be simple. My sense is that people learn more by copying patterns than by learning rules, and in this case the pattern `:stop` was easier to learn than ☞`stop` ← `:`. In addition to APL's brevity, we were also influenced by its infix operator syntax. It was natural to use infix for arithmetic, and we liked how infix operators worked together with the rest of the syntax.

### 2.5 Atoms and Their Values

An atom in Smalltalk-72 is any alphanumeric identifier. It may have an associated value or not. Assignment is accomplished by sending the message ← to an atom, as in ☞`alpha` ← `value`. This statement looks up `alpha` in the current context (it might be a temporary variable, an instance variable, or a class variable), and assigns the value to `alpha`. If not found, it will create a new global variable for alpha and assign the value to it.

And now a word about quote (☞). Note that if `alpha` does not have a value, then `alpha` ← `13` would cause an error, because it would send the message ← `6` to `alpha`, which has no value. More precisely, its value would be `nil`, and nil does not understand ← . On the other hand, if alpha were bound to a text frame on the display, then `alpha` ← `13` would add a newline character to that text. Quote picks up the next token in the message stream (the hand is pointing at it) without evaluating it. In this case it finds the atom `alpha`, which **can** understand ← `6`, and the assignment works.

Finally a word about left arrow and assignment. I always felt that there was something beautiful about assignment not being built into Smalltalk-72. Assignment is simply a result of how atoms interpret the message left-arrow. We decided to have the atom find its definition in a context and change it.[5] The kernel definition of Smalltalk-72 is remarkably open ended.

### 2.6 More on Message Parsing

Consider the middle three lines of `for` from Page 8; they say the following.

- If you see "←", then fetch a value for `start`, otherwise set it to `1`.
- If you see "`to`", then fetch a value for `stop`, otherwise set it to `start`.
- If you see "`by`", then fetch a value for `step`, otherwise set it to `1`.

From this it should be apparent that ◁ (read as "peek for"), followed by some symbol, tests for a matching symbol as the next token in incoming message stream. If the symbol following ◁ is found, then the message stream is advanced and `true` is returned; if not, then the message stream remains undisturbed, and `false` is returned.

Note also the expression ◁`do` on the last line of `for`. This does what it appears to do—it simply consumes the token `do` if it is found, affording the user of `for` the option of putting the "buzzword" `do` before the loop body.

The object `false` is a singleton instance of `falseclass`[6] known to the interpreter. A conditional test in Smalltalk-72 consists of an "implies-arrow" (⇒, read as "then") in the message stream followed by a parenthesized subvector containing the consequent code. If any value other than `false` (including `nil`!) precedes a "then", evaluation proceeds in the consequent subvector instead

[5] I talked with Alan about this while writing this paper, and he said "We could have done more with the first version of this; it's an interesting way to guard meaning with something between dynamic and static types."
[6] "CamelCase" was not widely used in the early '70s, and it made typing more difficult for young learners.

of continuing in the outer vector. When execution hits the end of the consequent subvector, it concludes evaluation of the *outer vector* as well. While the vector structure is not symmetric, the subvector, and the continuation of the outer vector, behave as two equal forks chosen by the implies arrow.

Finally we come to `CODE 24`, which invokes primitive code in the Smalltalk interpreter. This and other such "code escapes" in the interpreter exist partly to get things going from scratch, partly to maintain reasonable performance, and partly to effect primitive behavior that cannot be described in the source language. In this case, `CODE 24` looks up the value of `var`, adjusts it by `step`, compares it to `stop` and then, while in range, it stores the new value and evaluates the code vector, `exp`. Note that the activation of the code vector for the body of `for` must have a "caller" pointer back to the activation of `for`, but a "message" pointer to the code vector (which will be parsed afresh each time through), and a "global" pointer to the caller of `for`, where the binding of `var`, in our `spiral` example `i`, exists. In that way expressions in the body of the loop will see the value of `i`, but not the values of the other variables internal to the implementation of for.

Before leaving Smiley behind, let's look at one more definition that is evocative of the fun we had with Smallltalk-72 on the Alto.

It was possible to read the Alto's mouse position from `mx` and `my`, and `mouse 4` returned the state of the top mouse button (the three mouse buttons were accessed by bits 4, 2, and 1). Here is a rudimentary paint program of which many embellishments were built:

```
to paint (☺ home erase.
    repeat (☺ penup goto mx my pendn.
            repeat (4=mouse 4⇒(done) ☺ goto mx my))) !
```

```
to turtle var : pen ink width dir xor x y frame : f (
    CODE 21 'goto(draw a line of length :. ↑SELF)
            turn(turn right :. ↑SELF)
            goto(draw a line to :x :y. ↑SELF)'
    pendn(&pen ← 1. ↑SELF)
    penup(&pen ← 0. ↑SELF)
    ink(←. :ink. ↑SELF)
    width(←. :width. ↑SELF)
    xor(&xor ← (off(0) 1). ↑SELF)
    is(ISIT eval)
    home(&x ← frame frmwd/2.
            &y ← frame frmht/2.
            &xf ← &yf ← 0. &dir←270. ↑SELF)
    erase(frame fclear. ↑SELF)
    up(&dir ← 270. ↑SELF)
    isnew(&ink ← &black. &pen ← &width ← 1. &xor ← 0.
            (frame(&frame ← :) &frame ← f)
            at(:x. :y. &dir←270) SELF home) )
```

Fig. 3. A screenshot showing the `turtle` class rendered in the original Alto Smalltalk-72 fonts. The same code is shown more readably in Figure 4. Note that the quoted string following CODE 21 is a comment descriptive of the code escape.

```
to turtle var : pen ink width dir xor x y frame : f (
    CODE 21 'This code escape is equivalent to...
        ◁go⇒(draw a line of length :. ⇑SELF)
        ◁turn⇒(turn right :. ⇑SELF)
        ◁goto⇒(draw a line to :x :y. ⇑SELF)'
    ◁pendn⇒(☞pen ← 1. ⇑SELF)
    ◁penup⇒(☞pen ← 0. ⇑SELF)
    ◁ink⇒(◁←. :ink. ⇑SELF)
    ◁width⇒(◁←. :width. ⇑SELF)
    ◁xor⇒(☞xor ← (◁off⇒(0) 1). ⇑SELF)
    ◁is⇒(ISIT eval)
    ◁home⇒(☞x ← frame  frmwd/2.
        ☞y ← frame  frmht/2.
        ☞xf ← ☞yf ← 0. ☞dir←270. ⇑SELF)
    ◁erase⇒(frame fclear. ⇑SELF)
    ◁up⇒(☞dir ← 270. ⇑SELF)
    isnew⇒(☞ink ← ☞black. ☞pen ← ☞width ← 1. ☞xor ← 0.
        (◁frame⇒(☞frame ← :) ☞frame ← f)
        ◁at⇒(:x. :y. ☞dir←270) SELF home) )
```

Fig. 4. The `turtle` class of Figure 3, but typeset in the style of this paper.

### 2.7 Classes

So far we have seen the evaluation of simple expressions, messages sent to a `turtle` object Smiley, and some locally parsed syntax defining a for-loop. Because we have been sending messages to Smiley, let's take a look at the `turtle` class to see how it is defined, and thus how Smiley gets its behavior. The code is shown in Figures 3 and 4 (the same code is shown in both figures).

While `turtle` is a function, just like any other, it includes an `isnew` phrase that works as a factory to make instances. An evaluation of ☺ (or any other `turtle`) runs the code as shown, with the distinguished variable `SELF` bound to that instance. A call on `turtle` as a function runs its code with `SELF` set to `nil`. The `isnew` function detects a `nil` value of `SELF`, creates a new blank instance for `SELF`, and returns `true`, thus running the consequent code to initialize that instance. In this way, Smalltalk-72 functions play the rôle of functions, factories, and classes.

This definition declares a function named "turtle" with one temporary variable `var`, eight instance variables, and one class variable (preceding the code vector and delimited by colons). Following that is the code for evaluating instances of `turtle`. As in modern Smalltalks, temporary variables are local to each call (here one code block that includes many "method" sections), instance variables are stored and persist separately in each instance, and class variables are stored in the class and are shared by all instances.

Notice that most `turtle` methods end with ⇑`SELF`. This means that, not only is `SELF` the value returned, but that it is returned "actively," so that if any code remains in the message stream, this instance will again try to parse and execute it (as with ☺ `go 10 turn 90`). This cascading of messages is frequently convenient.

Take a look at the phrase ◁`is⇒(ISIT eval)`. We can't know what this is doing without knowing what `ISIT` does.

```
🎩ISIT❗ ( ◁ ? ⇒ ( ⇑ TITLE ) ⇑ TITLE = :☞)
```

Thus, this method provides answers regarding the type of a `turtle`:

```
🎩🐢 is ?❗
turtle

🎩🐢 is number❗
false
```

This same phrase is copied into each class so that all objects can answer their class (informally represented by the title of their class). In some sense it would have been simpler just to copy the same code in every class, but copying code is bothersome, and a bad habit as well. Because of examples like this, we already felt the need for a superclass object with behavior that could be inherited by all classes. This sharing of the global method `ISIT` was a weak second best.

In fact, each instance did have a pointer to its class object, but there was no access to it in the language. Even if there had been, the class objects did not point to a class themselves, and thus had no `DO` property with code that might have been useful. This explains why, earlier, we used the expression `show for` to print the definition of `for` instead of `for show` which would have been more object-oriented: there was no code to run `for` as a proper class-object. Smalltalk-72 was in that regard a proto-oop world, but it gave immediately the oop experience, which in turn inspired the real thing.

Let's define a class of our own—one that should be familiar to users of the language Lisp.

```
🎩to cons : car cdr (
    isnew⇒(☞car ← :. ☞cdr ← :)
    ◁ car ⇒(◁ ← ⇒ (☞car ← :) !car)
    ◁ cdr ⇒(◁ ← ⇒ (☞cdr ← :) !cdr)
    ◁ print ⇒(☞[ print.
            car print. ☞. print. cdr print.
            ☞] print))❗
🎩☞a ← cons 2 5❗
[2.5]
🎩☞b ← (a cdr ← cons 3 7)❗
[2.[3.7]]
🎩b cdr car❗
3
```

### 2.8 disp, Evaluation and the REPL

We have explored examples of Smalltalk-72 syntax and graphical functions that can be typed and executed in the text frame in the lower part of Figure 2.

We turn now to the Alto environment and look at how the world was organized to provide for reading and displaying function definitions, creating and accessing global variables, and some details of how the language interpreter actually works.

We begin with the object `disp`, an instance of `dispframe`. `dispframe` provides a rectangular area in which text can be read in from the user, evaluated, displayed, and scrolled. The starting frame is only a small part of the display, leaving a large space available above for turtle geometry examples

and other graphic explorations. If we wish to display more text, we can create a larger dispframe with

```
🖷☞disp ← dispframe 16 480 8 670 string 2000❗
```

These parameters define the $x$-position, width, $y$-position, and height of the frame; `string 2000` provides a buffer to hold the text being displayed. In Smalltalk-72, and throughout the Smalltalks of this paper, rectangles are characterized by the $x, y$ position of the upper-left corner on the screen, where $x$ increases to the right and $y$ increases downward on the screen. This is consistent with both the layout of text and with the scanning of the Alto display electronics, and therefore also with the layout of the display bits in Alto memory.

While we're talking about conventions, for children and most end-users, I preferred, and I think everyone in Alan's group pretty much agreed, that strings, vectors, and other collections should be indexed with 1 as the first element. That's our story, and we're sticking to it!

Next, we need to know about all the objects in this world, how our code interacts and adds to them, and how the code actually works. At the center of Smalltalk-72's "object architecture" is the class `USER`; `USER` has no temporary variables or instance variables, but many class variables. It thus serves as a dictionary of global variables in a manner that I will describe shortly. Whenever Smalltalk is started or restarted, the interpreter is wired to look up the code in `USER` and run it. As a default, this code is

```
(cr. read eval print)
```

This can be understood as follows:

- `cr`: start a new line (print a "carriage return"). The function `cr` is followed by a period to prevent any resulting value from parsing the remaining three tokens.
- `read`: prompt with an Alto character, read from the keyboard into `disp`'s text buffer (handling backspaces, etc.), and finally call an internal read routine that converts a character string to a sequence of tokens stored in a vector.
- `eval`: sent as a message to the code vector result of read.
- `print`: sent as a message to the result of evaluating the code. Most classes define a method for `print`; invoking it will normally print the result at the place where the user typed.

Because this code runs in `USER`, global variables are accessible to any code that is typed in. Thus, `USER` is effectively the global symbol table for this Smalltalk. Modern Smalltalk users can think of this as the origin of the `Smalltalk SystemDictionary`.

It is interesting that modern parlance condenses `(read eval print)` to REPL, but at the dawn of Object-Oriented programming, most existing systems used a `print(eval(read))` loop. The difference is not a minor one. A conventional print routine was full of logic to test for many possible types of objects. This made it complex for new learners to modify, and if they made a mistake the whole system could die because the print routine was broken. In Smalltalk, adding a new class causes no global perturbation other than defining that class. If you mess up in its print method, it affects only your new objects.

Let's return to `vector ◁ eval`. This is the heart of Smalltalk evaluation but, like everything else here, it is surprisingly simple. Each activation record points to the current code vector and contains a program counter (pc) The vector and the pc together provide the function of a message stream. Here is the pseudocode for vector `eval`:

As long as there is content remaining in the stream:

- fetch the next token (ignoring periods)
- evaluate it
  - atoms evaluate by looking themselves up in the current context
  - vectors make a new activation and evaluate themselves as a subexpression
  - everything else just returns itself
- apply the result to the remaining stream. This means creating an activation to run this object's code on the unconsumed part of the message stream.

Variable lookup is simple in Smalltalk-72. Whenever a message is sent, the interpreter builds and uses an activation, which is similar to a "context" in later Smalltalks. The *global* links in each activation form a chain that ends at the activation of USER's REPL. At every level of this *global* chain, a name is looked up in that class table. If it is one of

- a direct value link (a class variable),
- an activation offset (a temp variable), or
- an instance offset (an instance variable),

then the corresponding value is returned. If the name is not found, then the *global* link is followed to the next level, and so on up the chain. Ultimately this chain ends with the activation of USER, and this explains why USER is the home of all global variables.

An important special case (not an exception) of Smalltalk's variable lookup is ☞, the "quote" function. Quote picks up the next item in its message and returns it unevaluated. This means that an atom representing a variable has a chance to peek for a following ← (just another token). In the case of ☞`x ← expr`, `x` fetches the following value, looks itself up in the global chain, and stores the value wherever the atom `x` is found. If `x` is not found anywhere, it would make a new entry for itself in the USER dictionary, and store the value there.

Before leaving this section, we should talk about how this all comes together to make a running system. Smalltalk came in two parts: a pile of Nova assembly code that was assembled and loaded into memory by the Alto OS, and a bootstrap file of Smalltalk definitions known as ALLDEFS (see Appendix B). The assembly code would start running, and immediately read in ALLDEFS using a simplified form of `(read eval print)` that could not read strings or floats and that could print only the name of the most recent function defined. During the process, many functions and objects were defined (and stored in the USER dictionary), and along the way Smalltalk became more intelligent and able to parse strings and floats. At the end, the `disp` object was created, and its read-eval-print loop launched after a "Welcome to SMALLTALK" message appeared. If you read ALLDEFS, you will note places where a simple definition in the early part is then extended or supplanted later on. For instance the final version of `to` includes a feature that collects the names of any classes that have been defined or redefined, so that the `fileout` command would know to write those definitions out to save one's work in a text file.

After reading in the bootstrap file, we would hit the Alto's SWAT key. This invoked the Alto OS's code-swapping debugger to write a copy of the entire memory onto a file such as *small.sv*. That file could later be restarted by typing *resume small.sv* to the Alto OS. The simulation that is available on the CHM website loads one of these files as an exact replica of the Alto memory (65k 16-bit words) and simulates the Alto Nova emulator to run the Smalltalk image file that was saved over 40 years ago.

## 2.9 Under the Hood

Once Smalltalk was working, we found ways to use it for introspective system programming tasks. This informed the later evolution of Smalltalk systems.

*2.9.1 Object Layout and Linking of Activations.* In Smalltalk-72, there is a function `leech` that can attach to any object and return its contents either as Smalltalk pointers or as the actual bits in memory. This made it possible to access classes and activations even though they had no proper access methods.

As a simple example of such investigations, we could define a function `seebits` as follows:

```
to seebits x lx n i xbits (:n. :#x. 'Fasten seat belts'.
    ☞lx ← leech x. ☞xbits ← vector n+1.
    for i ← 0 to n do (xbits[i+1] ← base8 lx[i]○).
    ⇑xbits)!
seebits

seebits 4 (3+4)!
('0005352' '0000004' '0004215' '0003654' '0004216' '0177777')
```

The special "bits" symbol ○, seen in `lx[i]○`, returned the actual bits in memory. The numbers are octal, as was common usage in Alto software. From this we can learn that the address of class vector is `05350`, and that the reference count of this vector at the time was 2. The 4 is the instance size, which includes a nil at the end (to facilitate end-of-stream checks).

*2.9.2 Reference Counts.* Classes were given blocks of storage on 8-word boundaries so that the low three bits of an object's class pointer were always zero. In this way, we were able to fit an abbreviated reference count in the low three bits of any object's first word (its class pointer). Reference counts that overflowed that field were entered into a separate table until they returned to a lower value.

*2.9.3 Numbers and Strings.* We set aside a range of addresses below the object memory to represent small integers as immediate pointers (the actual range varied from one version to another). Integers outside that range were represented as real objects with a full 16-bit integer value. We can infer from this example that 04212 was the pointer value associated with the integer 0. The reader can probably imagine a small program to figure out the range of immediate integers in this version.

```
seebits 2 8192!
('0005262' '0020000')
```

Here we can tell that class number has an address of `05260`,

```
seebits 4 'ABC'!
('0005472' '0000003' '0040502' '0041777')
```

showing us that `05470` is class string, the second word gives the character count, and an odd last character is padded with `0377`. These are the kind of investigations that enable us to use the power of a general computing system to look inside itself and, in the case of this one, bring it back to life.

*2.9.4 Activations.* In Smalltalk-72, four pseudovariables point to activations, which can be "leeched". The pseudovariables are

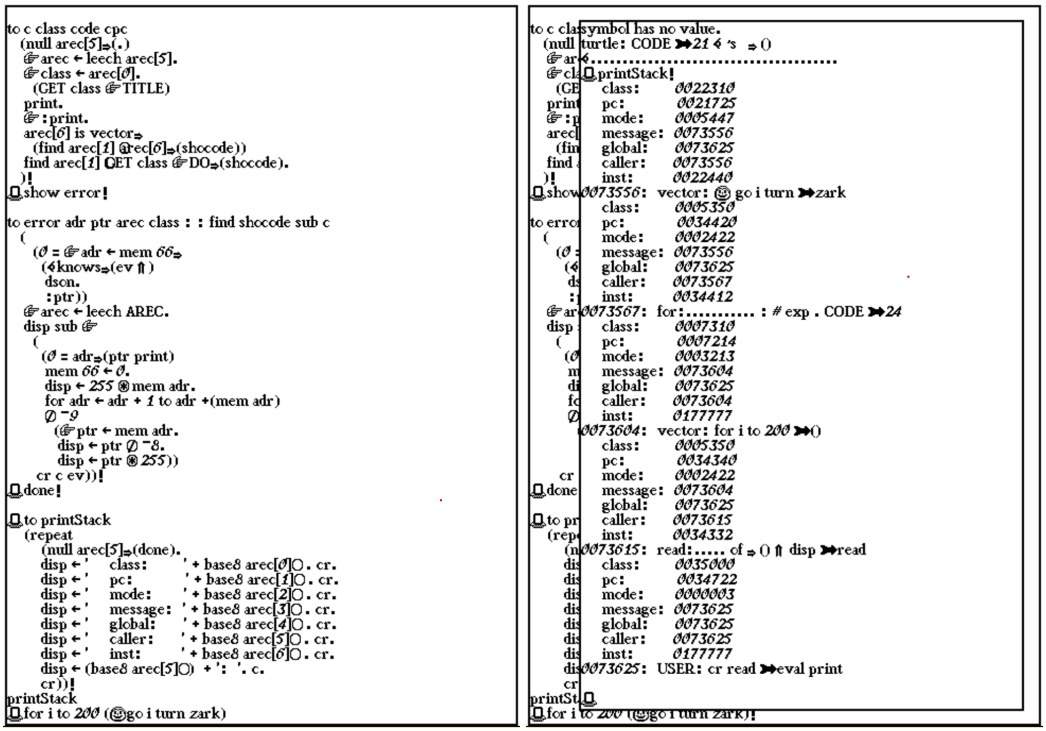


Fig. 5. Stack trace of an intentional error in Smalltalk-72 showing how the sender, message, and global pointers provide the structure for message passing when `eval`'ing a code vector.

- `AREC`—the current activation,
- `RETN`—the caller,
- `MESS`—the activation that serves as message stream for, eg, peek, fetch, etc., and
- `GLOB`—the next activation up in the global lookup chain.

Each activation also includes pointers to the following:

- the class (or function),
- its instance if any,
- the caller of this method (another activation),
- the message being received (another activation),
- a link in the global lookup chain (another activation),
- the code being run (a code vector), and
- a pc into the code.

The left side of Figure 5 shows the definition of the `error` routine that was called by the interpreter when it could not make sense of the next token in a message stream. It began by creating a leech on the current activation (`AREC`), copying an error message from inside the interpreter code,[7] and then calling `c` (for "caller"), a function that advances the activation leech up to its activation of its

[7]We made the special "bits" circle-symbol have a width of one, so that it would display on top of the next character to represent logical operations. Thus the two lines near the end of this function can be read as
disp gets ptr bit-shift -8.

caller. Also shown is a `printStack` function that I have written in the present day to provide a stack trace. It calls `c` repeatedly and shows how the message, global, and caller pointers work together in Smalltalk-72. The numbers are memory addresses in octal, as was common in the Nova assembly code tools.

On the right of Figure 5, I have executed an intentional error in our familiar spiral `for`-loop, and invoked `stackTrace` to show the various activation links work during interpretation. Each `caller` pointer threads back to the original read-eval-print loop in USER, and each `message` pointer points to the activation with the code vector being parsed. If we had been running the function `spiral`, the activations called from there would have had `global` links to the activation of `spiral`, but in this example, all the `global` links go straight to the activation of `USER` (`for` is written specifically to prevent called code from "seeing" its local variables).

### 2.10 Speed and Space—Not

With a 5MHz processor clock, the Alto was an exciting machine for its time [Wadlow 1981]. This however was the speed of the microcode; the memory cycle time was one fifth of that, and not all available to our fledgling programming system. The Alto devoted roughly 50 percent of the memory bandwidth to refresh the display from main memory. Application programs on the Alto were mostly written in Nova assembly code, which took on the average of two memory cycles per instruction, so the Alto Nova emulator was effectively a 0.5 MHz machine when not competing with display refresh, and more like 0.3MHz when running Smalltalk. 0.3MHz; not 0.3GHz.

The Alto seemed well-endowed with 128k of main memory, but half of that was allocated to the bitmap display, and roughly 32k more for the lean OS and our interpreter. So our Smalltalk system really had only 32k bytes for working storage—all strings, code vectors, and actual user structures. 32k bytes; not 32M bytes. When running these old Smalltalks, you will at times see the display shorten or go black altogether. This was to give the processor more cycles and make the program run faster. We used to joke that the Alto ran faster with its lights off.

Managed storage was a value inherited from our inspiration systems. APL and Lisp both provided garbage-collected workspaces. However, at this level of performance, we did not want to risk hiccups caused by mark-sweep garbage-collection. Moreover, with scarcely 10k words of free storage, mark-sweep reclamation would have been triggered almost constantly. Instead, we chose reference counting, which gave us smooth real-time behavior at the cost of occasional memory "leaks" due to circular structures not being reclaimed. The modest text editing code for "dispframes" was all written in Smalltalk-72. While we had to work hard to keep up with typing, storage management hiccups due to a scavenging garbage-collector were never a problem.[8]

Smalltalk-72 objects were compact, and have remained so in each subsequent generation. The Smalltalk *code* was also nicely compact. Notice that the code for the spiral example

```
for i to 400 (☺go i turn 89)
```

is only two vectors: one of five tokens, including a pointer to the second one, which is also of five tokens. With class pointers, vector lengths, and nil pointer at the end, this example took up a total

disp gets ptr bit-and 255,
where they extract a character string from 16-bit words in memory.

[8] Diana and Alan experimented with simple yet performant text editing schemes. A clever one that I recall was to populate the string buffer with long sequences of non-printing nulls which could be simply replaced during type-in rather than ever having to copy the string buffer when inserting or deleting characters.

of sixteen 16-bit words, or 32 bytes. The null pointer at the end enabled faster checks for end-of-list in the evaluation loop. We soon learned that this overhead was insignificant, and so eliminated the terminal nil in code vectors in Smalltalk-74.

Althought slow by modern standards, our experience was anything but boring. Smalltalk provided nearly immediate turnaround time for changes. We would frequently make several changes in a minute, never having to reload the system or any of our applications. Our figures of merit were malleability, and turnaround time for simple changes.

From the beginning, we made sure that when you left the world of Smalltalk, it saved all your state so that when you started up again (in 5 minutes, or in 5 days) things were exactly as you left them. This was especially valuable for students who might be away from the computer for several days. From the time your session ended, everything, including the bits on the screen, was saved onto a removable disk pack. When you returned and reloaded your disk into an Alto, everything was restored, allowing students to resume their plans with minimal cognitive disruption.

### 2.11 Impact

Smalltalk-72 supported several courses hosted at the Xerox Parc facility that taught this early form of object-oriented programming to middle school and high school students. The courses were designed and taught by Adele Goldberg in our group. Adele had a wonderful talent for recognizing what was difficult for learners and how to make it easier, and the experience from those courses was valuable to us as teachers and to students as well. Several of the students went on to careers that began with their Smalltalk experience (Bruce Horn who worked on the Macintosh finder, Steve Putz who taught computer science, and Marian Goldeen who went on to work at Apple). It also supported an animation system consisting of KAOS microcode (written by Steve Purcell) and the SHAZAM composition system (created by Ron Baeker and Tom Horseley [Baecker 1976]), and the TWANG music system (with microcode by Bob Shur) and a composition system (created by Ted Kaehler and Bob Shur [Kay 1993, p.539; Saunders 1977]). The primitive computations of animation and music were remarkable for the period, and had to be performed in microcode, Smalltalk-72 provided an easily understood language interface for these capabilities. Alan Kay was a constant coach and consultant on these early precursors to later media work in Squeak, and he was always the first to try out any new capability.

I will never forget when we first got synthesis to work. We had not yet hooked a real organ keyboard up to the Alto, but we mapped the QWERTY keyboard in to the synthesis code. It was not a crazy thing to do, since the Alto keyboard was unencoded and could register any number of simultaneous keys down. Chris Jeffers who worked in our group managing at the time, was an amazing musical talent, and we invited him over to try out the QWERTY organ as soon as we got a sound out. At first he played a few notes with various mistakes as we expected, finding out where the octaves were. But then after a few more notes and a couple of chords, he was playing a reasonable polyphonic rendition of a popular broadway tune…I think from The Music Man.

### 2.12 Reflections

For those of us working on the system itself, Smalltalk-72 gave us a taste of the subtle power of simplicity. In a system that ran probably 50 times slower than other languages at Parc, we would routinely try out and succeed with experiments in less time than serious programmers using more

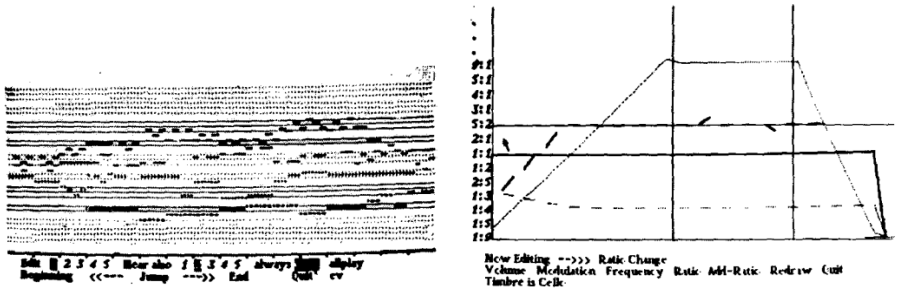

Fig. 6. Screen shots of the TWANG music system that ran in Smalltalk-72, using Alto microcode to perform 12-voice ASDR and 4-voice FM multitimbral synthesis. On the right is a real-time piano roll display, and on the left a rudimentary timbre editor for the FM synthesis parameters.

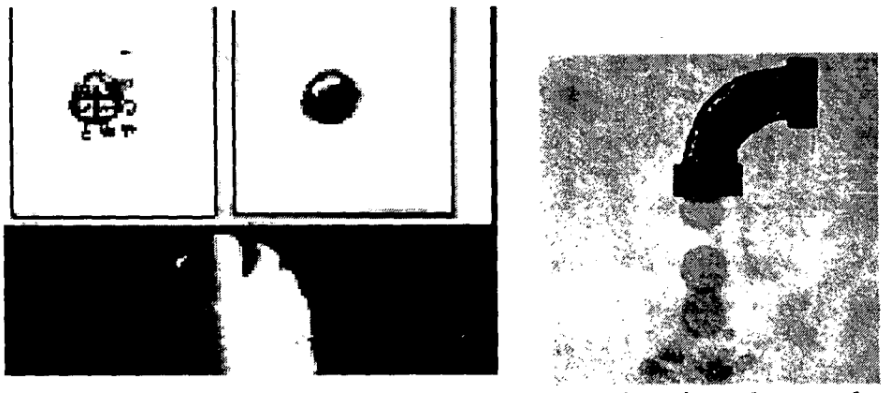

Fig. 7. Screen shots of the SHAZAM animation system. On the left, one of the images of a water droplet is being altered to show a splash—while the sequence is playing on the right. Such aliveness was a key value from the very beginning of Smalltalk.

complex systems. Much of this productivity resulted from the combination of a simple language, direct connection to bitmap graphics, and immediate response to program changes and user events.

Malleability was also important. Alan's language that fit on a page was a lot like clay—it came with little defined, but it was easy to shape into the forms we needed and wanted. From simple functions emerged object factories, and from message pattern recognizers grew the methods of OOP as we know it today.

The Alto also shaped our work. Alan and the other Alto designers had thought ahead, and knew the resources and processes that would be needed to create the future of personal computing. The bitmap display for graphics, the mouse for graphical manipulation, microcode for unanticipated primitive capabilities; all these enabled us to explore what we wanted this *Smalltalk* to be, as a language, and as a personal computing world.

## 3 WHAT IS A SMALLTALK?

When we talk about Smalltalk, we mean not just a language, but the entire gestalt conjured up by Alan's phrase "A personal computer for children of all ages" [Kay 1972]. Smalltalk is a vision that includes hardware and software and how they behave. I use the phrase "A Smalltalk" here to connote that larger gestalt. According to this view, "A Smalltalk" should have the following features.

**The system is immediately responsive**. This may seem obvious, but in the early days (and even today), many systems required one to leave the state of a project to edit and recompile code and then reload the state of the world before observing the effect of a change. In Smalltalk, everything is "live" so that changes to user code, or even to the system itself, take place immediately. This is important because it removes the cognitive interference of changing context from working on a project to jumping out to recompile it. Immediate response reinforces the experience of "being in a world," and this immersive aspect adds to focus and motivation. Both APL and Lisp systems of the time had this property; we never considered building a system that was not live in this way.

**Everything is an object**. It is important to understand how this uniformity extends into the whole experience of using Smalltalk. It means that anything in your world can be connected to anything else with relative ease, even when that connection has not been anticipated. This facilitates fanciful construction, whether it be pure art or a breakthrough in technology. The property of inviting unanticipated constructions is key to the power of Smalltalk and similar exploratory environments. To support this freedom obviously requires that the system be pointer safe, but it also requires that it be resilient in the face of absurd connections.

**SIX GENERATIONS OF SMALLTALK**

This paper focuses on work that began with Smalltalk-72 at Xerox PARC, and traces Smalltalk's evolution through six generations culminating in Squeak.

**Smalltalk-72** The first practical Smalltalk interpreter. Based on classes and messages and coupled to Alto graphics and sound.

**Smalltalk-74** Virtual memory, bitmap graphics, and a multi-window development environment prepared for the next generation.

**Smalltalk-76** Established the language, architecture, and virtual machine (VM) of all modern Smalltalks.

**Smalltalk-78** A reengineering of the Smalltalk-76 VM to run on an Intel 8086 in the Xerox NoteTaker computer.

**Smalltalk-80** A refinement of Smalltalk-76 and -78 aimed at commercial release. Standardization and documentation led to significant adoption in industry and academia.

**Squeak** VM entirely in Smalltalk. Flexible media support and extremely portable. Introduced Morphic graphics framework, project worlds, and EToy language extensions.

**Objects communicate by sending and receiving messages**. A system of objects communicating by messages is not invalidated when code or objects are added or removed. Smalltalk programs, including the later "compiled" ones, are nothing but sequences of message sends. About the only error that can be caused by changing or adding code is that a message is not understood, and this can be handled easily. Thus, robustness is a direct result of the message sending paradigm. The necessity to recompile and relink to avoid invalid pointers never arises in Smalltalk. Even in a large system with thousands of methods, the turnaround time to change and install a new method is typically just a second or two. Performance, especially performance of the development cycle, is thus another positive result of the message sending paradigm.

**Every object contains its state**. This organizing principle tends to pull together a lot of what would otherwise be rather random unconnected "data"—the screen extent, the height of a menu, the width of a line-drawing pen, etc. Not only does this tend to reduce the number of such apparently unconnected values, it also provides a natural place to look when trying to make sense of some aspect of the system.

**Every object is an instance of a class, which specifies the behavior of the object**. Organizing a world in terms of classes of objects is such a basic intuition as to be easily overlooked. Our natural language itself names things by their classes—a table, a person, a tree, etc. This tends to be even more so in a synthetic environment where it is natural to build things from a small number of parts—points, rectangles, windows, menus and the like. The early Smalltalk bootstrap definitions describe meaningful worlds made up from very few classes of objects.

**Everything is preserved from one session to the next**. We had experienced this comfort in the notion of a "workspace" that preserved not only function definitions but also variable bindings. We extended this to include even the bits on the graphics screen and partial results in any windows. It is not easy to articulate the benefit of this characteristic, but it relieves a stressful cognitive load to know that everything will be there when you come back; think about leaving notes to yourself on a desk and then having to worry about someone coming by and cleaning up before you have a chance to finish what you are working on. Any Smalltalk should preserve the state of work in progress across sessions. Smalltalk-72 accomplished this by capturing the entire state of the Alto memory. As later Smalltalks realized the ideal of "everything is an object," including the display bitmap, process scheduler and contexts, we were able to save everything by preserving the state of the object memory in a file, conventionally referred to as a "Smalltalk image file."

**Automatic storage management**. In the spirit of unanticipated creations, it must be possible to connect and disconnect objects without causing memory to fill up, and the programs must be free of "deallocation" code if they are to be simple and maintainable. We never considered a world without managed storage.

**Simplicity and generality**. Simplicity and generality are the apple pie and motherhood of science, and thus of computer science; they are seemingly obvious positive values. If a language is simple then it must also be general if it is to be of any use. This implies, for instance, that there should not be many distinctions other than that between one object and another, and it means that a user can build objects that participate in the world as first-class citizens. Even in the first Smalltalk it was possible to write your own class of character sequence objects, and these new objects would work in existing code that expected system-provided string objects.

## 4 SMALLTALK-74

Smalltalk-72 had sprung to life quickly, and gave us, as Alan said, "something to program." On the one hand, the experience was enormously positive: from Alan's sketch of proto-OOP, we had assembled a whole system adequate for teaching the elements of object-oriented programming in a simple world with engaging bitmap graphics; a world that could be shaped by the user and then saved and restored with complete fidelity. It had the feel of a personal computer. On the other hand, much of our gratification was tinged with a feeling that things were missing or not quite right. In the backs of our minds these various issues percolated.

### 4.1 Unfinished Business

There was no inheritance in Smalltalk-72. It was for this reason that we had copied into many classes the code ◁is⇒(ISIT eval) mentioned in Section 2.7. It didn't feel right to copy code for behavior common to all objects. Copying code leads to fragility due to the difficulty of maintaining consistency across all copies. The sharing of ISIT solved the consistency problem, but left the need to write this code stub in every class, which seemed burdensome and inelegant. This was behavior that every class could have inherited from a superclass 'object' if it were possible. Other examples arose, such as default printing behavior. Elsewhere in the fledgling class library we had to do a similar workaround for the extended form of subscripting (subscripting ranges in substr) desired for both strings and vectors, and number (integer) and float had no way to share common methods, such as max.

Classes in Smalltalk-72 were not real objects. They were essentially dictionaries assembled by the primitive code in to. That code parsed the arguments, and identified the set of temporary variables, instance variable and class variables declared for the class. With each name, the dictionary associated a pointer in the case of class variables, or a descriptor with offset in the case of temporary or instance variables. Because classes had no associated code, there was no way in the language to access these properties. The few places that required such access (such as the ability to print a class for editing or fileout) used the unprotected access afforded by leech; naturally that code was difficult to read and maintain.

Activation records were not real objects. Like classes, activation records had no class pointer and hence no code offering user-accessible means to be inspected. Here again, I relied on leech to enable the c function (short for caller) to climb the stack in error situations.

The various message-parsing primitives (peek, fetch, etc.) were useful and did the job, but it seemed that we should be able to write more of that code in Smalltalk itself, because they were simple manipulations of the streaming capability of activations.

Increasingly, we felt the need for more speed. We have discussed above the modest speed of the Alto, further burdened by maintaining the display. The flexibility of being able to simulate messages by on-the-fly parsing was a remarkable feat, but the cost of constantly parsing the code stream could not be ignored. As we increasingly evolved to the idiom of named methods, it seemed that we should somehow be able to use a dictionary of methods to speed things up.

We were also strapped for space. Once Smalltalk was loaded there was little more than 10k words of working storage. For teaching Smalltalk this was barely adequate,[9] but not for serious text editing, graphics editing, and system programming. While the lack of speed was an occasional frustration, the lack of space was a real barrier to progress.

We were also starting to experiment more with bitmap graphics, using the few primitives available in Smalltalk-72. On the one hand we wanted more control over the combination rules (or-ing, and-ing, complementing, etc.) for any operation, and on the other hand it seemed that it required a discouragingly complex mass of code for character display, line drawing, and scrolling. They all dealt separately with the problems of bit-wise alignment on a display that was organized as 16-bit words.

[9]The space available was iffy even for teaching, but Adele designed her "Box class" curriculum to be successful with minimal resources.

### 4.2 Progress

I've been asked how Smalltalk-74 came about. On the one hand we were living with these various linguistic frustrations, and on the other we were just fun-loving computer crazies working on the next thing, whatever that might be. For Dave Robson, who had just joined us, it was to do a new interpreter. For me it was to simplify and generalize all the bitmap manipulation code. For Ted Kaehler, who had just returned from grad school in August, it was the thought of grafting a virtual memory onto Dave's interpreter. Meanwhile, Alan was working with Diana Merry on a galley editor that handled both textual and graphical content, and with me on how to make a simple multi-window scheduler. I generally guided the detailed technical work, and Alan was in the wheelhouse. This was a time when he could tell that we were all "cooking," so he let the reins loose. Meanwhile he was busy writing an article for Scientific American [Kay 1977] that looked forward to the day of the "notebook-size computer."

Dave got to work on a new bootstrap that made it possible for classes and activations to be instances of real classes with useful descriptive code. `Class` was an instance of itself. Classes contained dictionary objects for class variables and for methods, and they also supported protocol for defining and redefining individual methods. While the expression syntax remained the same, the class-defining function `to` was supplanted by an assignment of the form

```
☞className ← class((temps)(instvars)(classvars)(globals))
             ((initCode)(list of selector–method pairs))
```

Moreover, since we had identified classes as a significant semantic structure, we made a separate entity for the simpler (non-factory) functions:

```
☞functionName ← function (temps) (code)
```

While `isnew` had performed satisfactorily in allowing functions to act as factories and classes in Smalltalk-72, it had always been confusing where to put it. Should it be placed early in the code to avoid running methods before any instance had been created, or should it be placed at the end where the test would not slow any of the most common methods? In Smalltalk-74, `initCode` took the place of the former `isnew` test, and it was dispatched primitively with no effect on the time to run instance methods.

The list of selector-method pairs canonized the dominant style of class description in which incoming messages were dispatched based on their leading token. It also gave us the structure we needed to redefine methods and classes on the fly, without invalidating existing instances. This was not yet reshaping instances, but only editing the class—methods, temps, instvar names, etc. But it was a *taste* of things to come.

While we made classes and activations proper objects, the expression syntax of Smalltalk-74 was unchanged from Smalltalk-72, so most of our old code ran in the new system. This allowed us to migrate most important projects forward with relatively little effort.

Dave's new bootstrap[10] for Smalltalk-74 included the first process scheduler for Smalltalk. It was modeled along the same lines as the priority scheduler of the Alto hardware, and it made it possible for us to interrupt, inspect, and resume running Smalltalk programs. While a fledgeling capability in Smalltalk-74, it became the core of the process scheduling mechanism in the Smalltalk-76 virtual machine and all its successors.

[10] A copy of the file "MDefs St-74.pdf" is available as auxiliary material for this paper.

### 4.3 Windows and Events

While Alan and Diana had endowed the Smalltalk-72 system with the capability to display multiple windows with independent text content, this capability was not used in the daily programming environment other than in a rudimentary sub-window feature used for trying code snippets and for displaying error notifications.

By this time, though, our text windows had evolved to offer mouse-driven selection and WYSIWYG text editing. We coupled the windows to the file system so that entire files could be read in, edited, and saved; and individual methods of a class stored in a file could be edited and redefined in just a few seconds. It was suddenly much easier to work on serious code with substantial logic. Our process was accelerating with its progress.

Smalltalk operated as its own world, without any real operating system support for mouse and keyboard events. This meant that much of the user interface code was complicated by polling loops that tested for mouse or keyboard activity. There was no consistent coding style for this kind of user interaction. With this in mind, Alan and I in one afternoon came up with a paradigm we called "loopless programming." The idea was to establish a protocol of three methods for client objects being scheduled:

- `firstTime`—Respond true if the client wants control, else false
- `eachTime`—Do this whenever the client has control; return true as long as the client wishes to keep control
- `lastTime`—This will be called once before the client loses control

This established a partition between the scheduling code and the "client" code for the objects being scheduled. It also set up a natural partition between initialization, live activity, and finalization for the client code. We referred to it as "loopless" because the polling loops were removed from the client code.

In a world in which event-driven programming did not really exist, this gave us an event-based paradigm for client code, and an almost trivial way to drive it from above:

```
For every client object in a list being scheduled...
obj firstTime ⇒
    (repeat (obj eachTime ⇒ (again) done).
    obj lastTime)
```

This immediately suggested variants relevant to window scheduling, such as `enter` and `leave`, with the scheduler promoting active windows to the head of the list and thus appearing on top of other windows as we expect of most systems today.

### 4.4 OOZE

One of the most important contributions of Smalltalk-74 is nearly invisible: the OOZE virtual memory. Ted and I developed this scheme to enable Smalltalk to address over a megabyte of data within the limited confines of our Alto memory (roughly 30k bytes). We did this by using the 16 bits of an object pointer to index a hash table of resident objects, along with elaborate schemes for how to retrieve non-resident objects from the disk and how to make space when the local cache was full. In a simple memory system, an indexed object table requires only one or two instructions to dereference an object pointer (i.e., resolve the object pointer to the memory address of the

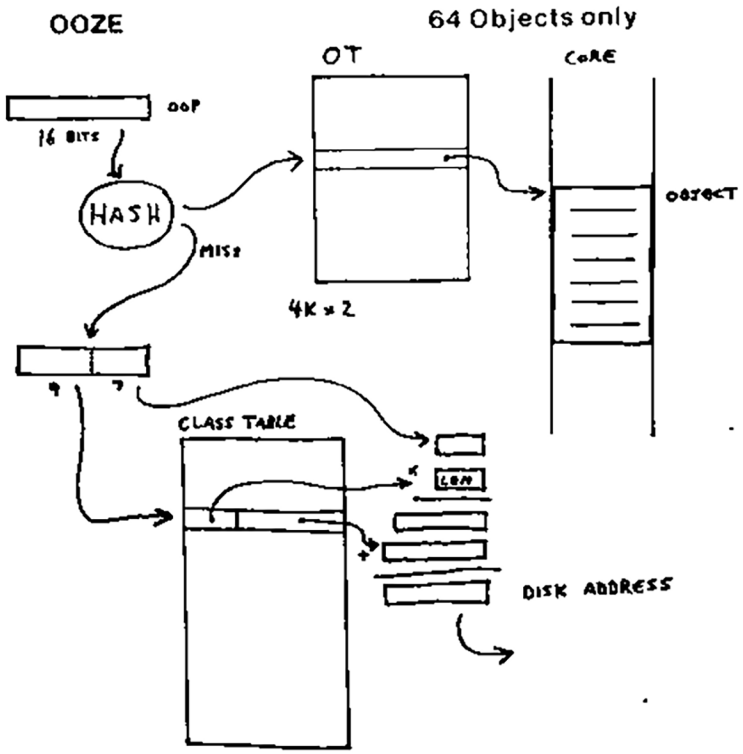


Fig. 8. Mapping object pointers in OOZE. Resident objects were found by hashing the OOZE Object Pointer (OOP) into the object table (OT). Non-resident objects were found via a pointer to one of 512 zones (the "Z" of "OOZE") on the disk allotted to 128 instances of their particular class. The low 7 bits of the OOP provided an index into that zone. Figure taken from *Byte* [Kaehler 1981].

referenced object). To dereference an OOZE object pointer would have tripled this overhead or worse, except…we had the option to put this vital piece of logic into microcode. With the help of a bit of microcode to access the hashed object table, Smalltalk was barely slowed at all, yet we suddenly had roughly 50 times more free space in which to explore object-oriented programming and bitmap graphics. While Smalltalk-72 had only 20k free bytes at startup, now OOZE could address over a megabyte.[11] That this scheme worked for us was partially due to the relatively small size of most Smalltalk projects. While the Alto's moving-head disk drive occasionally thrashed like a washing machine at major context changes, the system hummed along quietly most of the time we were at work.

The most ingenious feature of OOZE—a typical Ted invention—was "safing." A background process was always writing dirty objects to the disk and marking them clean, although many would immediately be made dirty again as the system continued its normal operation. Periodically, the "safing" operation would stop the system and write every last dirty object out to the disk, and then write a checkpoint image of the entire Smalltalk to the disk. The beauty of this scheme was that, regardless of the complicated structures in memory and on disk, Smalltalk could be restarted from

[11] Typical Smalltalk objects were about 20 bytes, and OOZE could address up to 60k objects. (We reserved 4k of the full 16-bit pointer space for immediate integers.)

this "safe" state until the next time an object needed to be written to the disk. Because all objects in memory were clean at this time, new space was available in memory simply by discarding clean objects, so it was a long time before the need to write to disk. During all this time, Smalltalk was "safe" in that it could crash and be rebooted, suffering only the effect of going back to the most recent checkpoint. Occasionally when someone said "Oh, no!" because of having caused a fatal error, you would hear a nearby experienced user say "Quick, boot and resume!" While this was a wonderful parachute, it had to be deployed promptly, because the safing process, on its next cycle, would dutifully enshrine the damaged state in a permanent checkpoint.

Throughout our time at Parc, we of the Learning Research Group (Alan's crew, LRG) enjoyed a friendly competitive relationship with Bob Taylor's Computer Science Lab (CSL). We were invited to their weekly "Dealer" brainstorming meetings, and frequently joined in the discussion. While they were working on serious system programming languages such as BCPL and Mesa, and on the Alto and Dorado operating systems, our Smalltalk work was viewed superficially as a toy. We gradually earned their respect by actually building things that were only talked about as "interesting ideas." In spite of the stately pace of Smalltalk, it was often much quicker to try something out in our world than in the tools of CSL. While Alan and Adele Goldberg were the only members of LRG with PhD's, we were fearless about writing microcode and other nitty-gritty projects like OOZE and BitBlt.

When we were designing OOZE, Alan suggested that Ted and I run the design for OOZE by Butler Lampson (a lead technical powerhouse in CSL), which we did one day when he was passing through our area. He heard us out, asking various questions, and at the end he said "You'll never get it to work." Coming from Butler, that one comment was so motivating for Ted and me, that I give Butler partial credit for making OOZE a success. More details on OOZE can be found in Ted's article in *Byte* magazine [Kaehler 1981].

### 4.5 BitBlt

In my periodic code reviews, I was bothered by the large and complex body of code devoted to various manipulations of the Alto's bitmap display. Drawing a line for a turtle graphic design, copying a character of text to the screen, or scrolling a block of text; in every case detailed manipulation was required at the left and right ends of the image to deal with the bit alignments within words, and similarly all along the horizontal extent of any block transfer, as necessary for scrolling.

Figure 9 shows a common situation where Smalltalk must copy a rectangle from one source point to a different destination location, and where source and destination are in different objects with different raster widths as well. Looking at the figure, one can see the need for a special mask for the bits at the left of the source, and another for storing it into the destination, and the same machination at the right side, with even more special cases when the area is narrow and the edge regions overlap. Of course the offsets within the two bitmaps may be different, and the raster widths may also be different in many cases except when moving or copying a rectangle from one location to another in the same bitmap.

Things got complicated when the source and destination overlapped, and of course the loops had to run in different directions when scrolling up or down. In dealing with these problems, the existing scrolling code in Smalltalk required an extra buffer the size of a scan line, and a computer scientist does not sleep well when extra buffers are required.

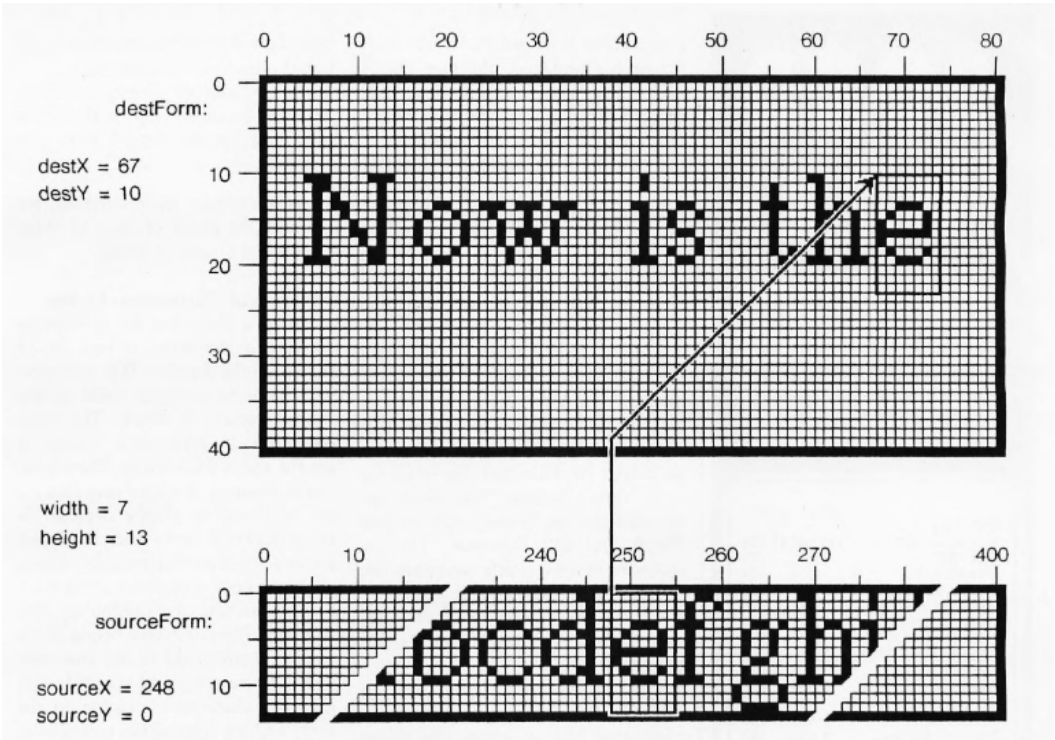


Fig. 9. Copying a character from a font (bottom) to the screen (top) requires complex manipulation of the stored bits to account for different alignment of source and destination in the 16-bit memory words, and different widths of the scan lines in the source and destination. Figure taken from *Byte* [Ingalls 1981, p. 171].

I had been wrestling with a solution to this problem, and especially the problem of how to avoid the need for temporary storage when moving large blocks, when my son brought home an intriguing rolling rubber stamp wheel, like the one on the left of Figure 10.

I realized immediately that one could think of the source and destination of the block transfer as two sheets passing over one another with the wheel in between, as shown on the right of Figure 10. The wheel picks up images from the top and lays them down on the bottom—not unlike the insides of a Xerox machine! The only temporary storage required is one compete circumference of the wheel (two full words that can be in registers) and, except at the beginning and end, full words can be loaded into, and stored out of, the two registers.

This point of view enabled me to greatly simplify and merge code for all these manipulations, including text display (offering a more compact font format), line drawing, scrolling, menu display, and movement of sprites (including dragged windows) on the display. The resulting operation, now known as BitBlt, was a major contribution to the evolution of Smalltalk as a computing tool, not just as a language. BitBlt was embellished over time to provide clipping and support for halftone fills as well as all possible combinations of source and destination pixels.

The *Byte* article on "The Smalltalk Graphics Kernel" [Ingalls 1981] details Smalltalk methods that, with BitBlt at their core, can lay out and display text, draw lines, and magnify and rotate images—each in little more than a dozen lines of code. The article concludes with a frivolous method for

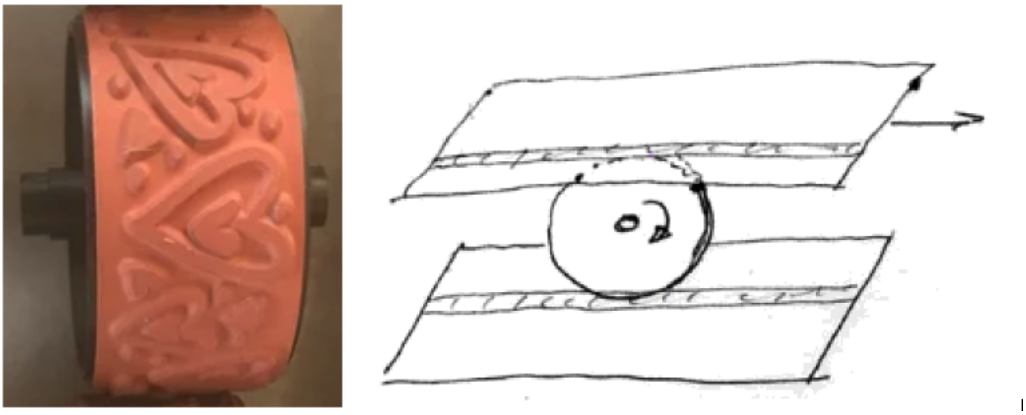

Fig. 10. On the left, a rubber stamp in the shape of a wheel, which can be rolled to make a long strip image on paper. On the right, a sketch of the rolling image transfer model of BitBlt. The "wheel" is a register two full words long. It picks up one full word unshifted, rotates by the pixel offset, and then stores a full word unshifted.

computing the next generation of Conway's "Game of Life" on any size bitmap in just 60 calls on BitBlt.

Fig. 11. 90-degree rotations accomplished by masking, shifting, and storing operations, all of which can be implemented with BitBlt. Figure taken from *Byte* [Ingalls 1981, p.188].

A notable instance of technology transfer took place when I gave a demo at one of the CSL "Dealer" meetings at Parc that featured overlapping windows with BitBlt in Smalltalk-74. In the process, I showed a pop-up menu, made with BitBlt, that restored the underlying image when done. Peter Deutsch,[12] who was in the audience at the time, called out, "Wait a minute…did you just do what I think you did?" I had just finished putting BitBlt into the Alto's microcode, and the menu response was instant. At the time, it seemed like magic. The very next day I was visited by Smoky Wallace from the Xerox's product division, and BitBlt was subsequently integrated into graphics projects there, and throughout Parc.

### 4.6 Microcode

The Alto was an ingeniously simple machine [see Appendix F] that used microcode to emulate the instruction set of the Data General Nova minicomputer. Most of the software that ran on the Alto was written in Nova assembler, or BCPL that compiled into Nova code. The Alto's microcycle was 200 ns and the memory cycle time was 1 $\mu$s, so well-written code could execute up to 5 simple logic

[12] Peter was a member of CSL at the time, although he later joined our group for a while before leaving Parc.

instructions for every cycle of the main memory. This could produce enormous gains: a block move loop might take three nova instructions totaling at least seven memory cycles per word, whereas the same move could be done in two memory cycles per word in microcode. To a programmer, this was a magic wand.

The tools for writing and debugging microcode were crude, but a number of us in Alan's group tackled the problem early on, achieving 4-voice real-time FM music synthesis for the TWANG music system, and real-time image compression and decompression for the SHAZAM animation system in Smalltalk-72.

It was Alto microcode that allowed us to consider using a hashed object table in OOZE, so that with 16-bit pointers could we could address up to 60k objects, as well as 4k immediate integers, and still manage the resident objects compactly in memory (as opposed to the breakage associated with a straight paging architecture).

The microcoded architecture of the Alto was also perfect for implementing BitBlt. With the ability to execute up to five logical instructions between each memory cycle, it greatly reduced the overhead of shifting and masking data at the left and right ends of the source and destination rectangles. In large scrolling operations, the "wheel" copying loop could read and write at the full speed of the memory, in spite of dealing with differing alignments and raster widths of the source and destination.

The writeable microstore of our Altos had a capacity of only 1k instructions; this meant that music and animation could not be run together, and neither could be run with other special microcode, such as that for OOZE. Microcode was a magic wand at the time, but our higher goals of combining live music, animation and programming had to wait for Squeak.

### 4.7 Major Smalltalk-74 Projects

A number of interesting projects were implemented in Smalltalk-74. Most of these were research projects internal to our group, but they all fed into the future work that was to carry us ever closer to the world outside Parc.

*4.7.1 Optical Character Recognition.* Smalltalk-74 was our slowest Smalltalk, yet BitBlt ran in Alto microcode, which meant that there were huge rewards for "thinking BitBlt" in image manipulation. For a while, I had dreamed of scanning Devanagari text as part of a project with my father, who was a Sanskrit scholar. With the ability to count bits (black pixels) in any rectangle it was possible to decompose images as well as to assemble them. Using BitBlt in this way, I was able to perform OCR on Devanagari text [Ingalls and Ingalls 1985], a task that would not otherwise have been possible in Smalltalk. A scan of one-pixel-wide rectangles produced a "profile" of black pixels; this made it possible to break a page into lines, the lines into "words," and the words into characters, as shown in Figure 12. BitBlt also served to align the characters, and to score their similarity, to achieve the final identification of each character glyph. A video of the project can be found at https://vimeo.com/4714623.

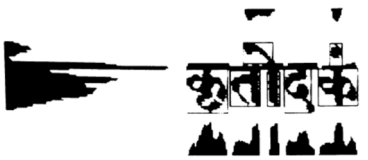

Fig. 12. Segmentation of characters in Sanskrit text. The "profiles" shown along the edges are used to separate the lines and characters.

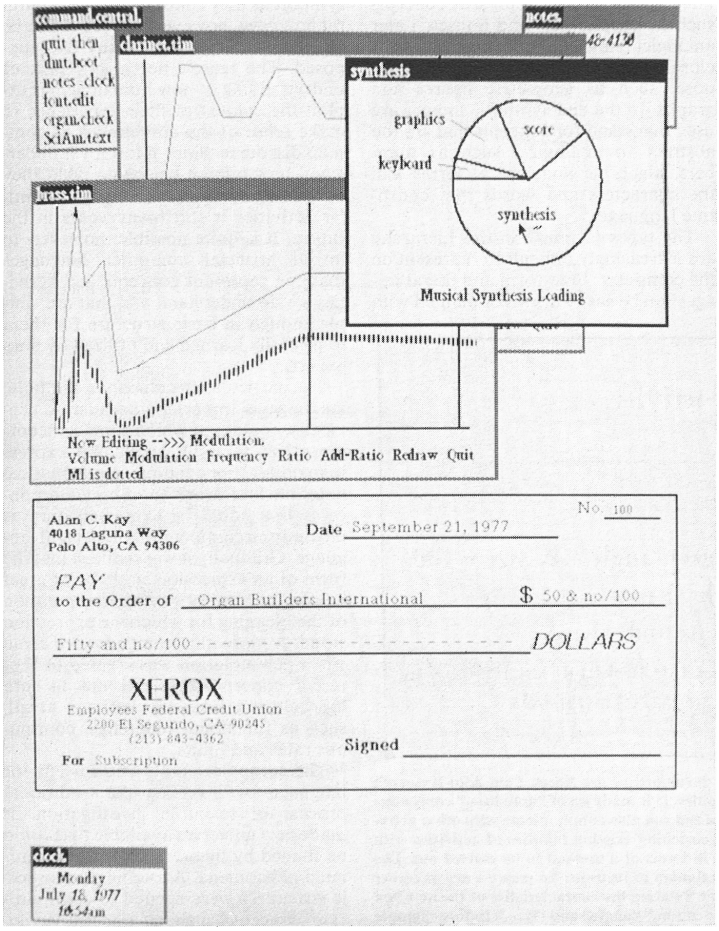


Fig. 13. The figure from page 234 of the September 1977 Scientific American, showing windows on an Alto screen.

*4.7.2 Scientific American Paper.* Alan Kay led a number of projects that fed into an article he wrote for Scientific American in September of 1977 [Kay 1977].

- Steve Weyer's Findit: Part of Steve's work on his PhD, Findit was an information retrieval system in which you filled out a card with any information you had; the system then returned all the cards that matched. This was an especially nice interface for a library, and in fact the Xerox PARC library used Findit for a number of years.
- Ted Kaehler's graphics editor: An early bitmap paint program that used BitBlt for its brushes and "stamps." It was cleverly designed to stripe large images across several smaller objects so that OOZE could handle them gracefully.
- Diana Merry's galley editor: This was our first foray into a real document editor. It built on the loopless scheduling model to provide a scrolling view of editable text, graphics, and other content, a rarity at the time.

*4.7.3 Development Environment.* Our event-driven scheduler and independent text windows were now at the heart of every serious Smalltalk project. Smalltalk-74's ability to redefine classes and methods on the fly made possible a new way to develop code. We put the code for each class in a separate file, and opened windows on each of the files being developed. Menu commands made it possible to read in the entire file contents, or to select a single method and read it as a new definition. In both situations, changes took place immediately with no need to reload the work in progress. This meant that we were seeing the entire code in context while editing—a refreshing change from the structured code editor of Smalltalk-72. If ever a fatal error caused a crash, the files were there for recovery. When this environment became functional, I remember saying to Ted, "if we just had this kind of coding environment at 10 times the speed, we would be unstoppable."

*4.7.4 Simulation of Smalltalk-76.* The other major project in Smalltalk-74 was, interestingly, the simulation and bootstrapping of Smalltalk-76. "Interestingly" because this project essentially killed Smalltalk-74. It both diverted the programming team away from Smalltalk-74, and became the host for most of the improvements we were contemplating. That Smalltalk-74 was so quickly abandoned explains why, out of 120 salvaged disk packs, we found only one with Smalltalk-74 on it. That disk contains pieces of the Smalltalk-76 simulation, and its screen is still full of content from the Scientific American article (see Appendix A).

*4.7.5 Reflections.* I sometimes see the period of Smalltalk-74 as one of displacement activity. We knew that we wanted to clean up the language, but instead we mainly worked on graphics and virtual memory. In the end, though, I feel this was for the best. Suppose we had implemented the message parsing functions in Smalltalk; it would have made things slower still. Suppose we had implemented inheritance; it would still have been in the context of slow message dispatch. As it was, we all did important work, and when the time came to clean up the language, we were ready.

## 5 SMALLTALK-76

In August of 1976, I spent a weekend at the beach with my family. In those three days all of the brewing forces and frustrations came together to suggest an entirely new language and architecture that promised true commercial speed while preserving an object-oriented style very similar to that of our familiar interpreted language. I began by sketching a syntax similar in style to Smalltalk-72/74, but that could be compiled. This quickly evolved into the now familiar keyword syntax of modern Smalltalks, along with a way to compile that syntax into bytecodes amenable to efficient execution in microcode.

I have been asked, "Why and when did the development of Smalltalk-76 start—was there any corporate or research team strategy?" When I got back to work after that weekend, I essentially declared, "I've got it." I felt I could answer any question on the way to an efficient Smalltalk. I was working closely with Ted, Diana, Dave, and Glenn Krasner, and we all basically dropped what we were dong that day. Alan knew when someone was on fire, and at that point he did what he does best: he made it our "corporate" strategy.[13]

[13]This was a period of feverish activity for me, and Alan allowed a "skunkworks" deviation from normal Xerox procedure so that I could burn midnight oil on Altos that were required to remain in the building. One night, we rolled an Alto onto the loading dock of Parc's building; we hoisted it into my station wagon, and I drove it home, where I kept it for several months. I would run my own diagnostics and replace memory boards as needed. To stay in sync with the group (before the Internet), I would carry a Diablo 33 disk in my backpack when I bicycled in to work. We figured it at 2.4 Mb/20 minutes = 2k bytes/second bandwidth each way.

### 5.1 Syntax

In order to compile Smalltalk, there needed to be a syntax. In the earlier Smaltalks, we had enjoyed using a word before before each argument, and this suggested a keyword-based form of expressions. Keywords provide the opportunity to name both the *what* (the keyword part) and *how* (the parameter name) of each parameter. By putting colons on each keyword, it seemed almost like natural language, as in "name: Philip"; it also happened to be reminiscent of Smalltalk-72's "fetch" function associated with most arguments.

This keyword syntax has remained essentially unchanged in Smalltalk ever since.[14] It seemed simple; the syntax rules are as follows:

- There are three kinds of messages, greatly reducing the need for parentheses:
  - a unary message with no arguments is a simple atom.
  - an infix message is an arithmetic token followed by an expression.
  - a keyword message is one or more colon keyword, each followed by an expression. A series of keyword phrases represents a message with multiple arguments, with each keyword descriptive of its accompanying parameter, as in `Point new x: 0 y: 5`. In such cases, the method name is the concatenation of the keywords: in this case, `x:y:`.
- Precedence is in the order above: unary > infix > keyword > assignment.
- A semicolon indicates a cascaded message to the prior receiver. I added this wrinkle because I liked the streaming style of Smalltalk-72.
- Assignment is of the form `<variable>` ← `<expression>`.
- Statements are expressions terminated by a period or close bracket.
- Square brackets delimit blocks of code (a series of statements).
- Expressions can be grouped by parentheses where needed for precedence.
- A method could be followed by a phrase such as `primitive: 24` to invoke primitive behavior.

I retained the old Smalltalk-72 pattern of representing conditionals by an arrow to the consequent block, as well as the convention of treating all objects (other than `false`) as `true`.

We adopted the use of camelCase in most places, and also the convention that local variables began with lowercase, while class variables and other global variables were capitalized. Both conventions aided readability and have endured for that reason.

Keyword syntax for messages gave us the ability to handle multiple arguments without the need for parentheses, and with readability close to that of natural language. It was a real breakthrough.

### 5.2 About Left Arrow

Smalltalk-76, as with Smalltalk-72 and Smalltalk-74, used the left arrow for assignment. For simplicity and speed, I made the decision that this would be compiled as a straight-out destructive store operation. However, I came up with a special "trailing left-arrow keyword" rule that harked back to the message-sending model of assignment in the earlier interpreted Smalltalks. This rule allowed the inclusion of a final left arrow with argument as a message part added to the end any of the Smalltalk-76 message forms: unary, infix, and keyword. This offered a pleasing read–write symmetry in patterns such as these:

[14] Small changes include the format of literal arrays, literal uniqueStrings, and extended infix (operators composed from a sequence of non-alphanumerics).

```
next  next←          "unary extended, as in stream next← item"
•     •←             "infix extended, as in array • i ← item
                     (• was the array subscripting operator)"
name: name:←         "keyword extended, as in name: 'Jones' ← 'CEO',
                      or row: a column: b ← item"
```

Beyond being merely a pleasant pairing, the trailing left-arrow bound the final argument more tightly. This meant that a number of expressions did not require parentheses, as with

```
greek wordFor: 'B' ← 'alpha beta gamma' word: 2
```

as opposed to

```
greek wordFor: 'B' put: ('alpha beta gamma' word: 2)
```

### 5.3 Classes and Contexts

Classes were first-class objects, as in Smalltalk-74, but now with a rich protocol that handled all the class-oriented aspects of the system kernel. These classes supported inheritance, so all objects were automatically endowed with a useful base protocol inherited from an Object class, and related entities like strings and vectors could share a common protocol for collections. Other natural inheritance groups were magnitudes and numbers, and streams and files. It had been frustrating to work without inheritance in the earlier Smalltalks, so inheritance was a central feature of Smalltalk-76 from the start. We had not encountered any real need for multiple inheritance, and it seemed to add a lot of complexity to a simple design, so we left it out. Simplicity was key.

Once I had a clear vision for the Smalltalk-76 architecture, I used the tools at my disposal, and proceeded to build an entire Smalltalk-76 image inside a running Smalltalk-74 system. As part of the new architecture, I changed from calling our stack frames "activations" to calling them "Contexts," since they captured the entire *context* of the state of the VM at each step of its operation. With Contexts being first-class objects, the entire operation of the virtual machine could be emulated by repeatedly "stepping" the current context:

```
repeat: (currentContext ← currentContext step)
```

This pattern is conveniently able to follow a growing and shrinking evaluation stack because the step method is written to return a different context in the case of a `send` (which creates a new context) or a `return` (which reverts to the calling context).

The trickiest part of getting this all going was writing the compiler, since it had to be written in the Smalltalk-76 language which did not yet have a compiler. However the simulation approach made it quite tractable: I wrote the whole compiler in Smalltalk-74, and could test that various code snippets were producing the right code. Then I just had to transliterate that compiler from Smalltalk-74 to Smalltalk-76 syntax, a relatively simple task, and thus one I could complete without introducing many bugs.

A paper at POPL in 1978 [Ingalls 1978] describes the Smalltalk-76 language, user interface, architecture and implementation of this system. Here, let's examine a few code snippets.

### 5.4 Example Code

```
Class new title: 'Rectangle';
    fields: 'origin corner'.

Access to fields
origin  [⇑ origin]          ``top left''
corner  [⇑ corner]          ``bottom right''
origin: origin corner: corner
    ["no code; just store into the instance"]

Testing
contains: pt        "return true if pt is inside me"
    [⇑ origin <= pt and: pt <= corner]
empty [⇑  corner <= origin]

Combination
inset: delta
    [⇑ origin + delta rect: corner − delta]
intersect: r
    [⇑ (origin max: r origin) rect: (corner min: r corner)]

Image
clear: color [primitive]       "display operation"
outline: width
    [self clear: black.
    (self inset: width) clear: white]
moveto: pt
    [corner ←  corner − origin + pt.
    origin ←  pt]
```

Rectangles have two fields: `origin` is the upper left corner, and `corner` is the lower right corner. Rectangle builds on the fact that `origin` and `corner` are Points.

```
Class new title: 'Point';
    fields: 'x y'.              "Cartesian coordinates"

Access to fields
x   [⇑ x]
x   [⇑ y]
x: x y: y

Testing
<= pt              "return true if I am below/left of pt"
    [⇑ x <= pt x and: y <= pt y]

Combination
rect: c           "make up a new rectangle"
    [⇑ Rectangle new origin: self corner: c]

Point arithmetic
+   pt  [⇑ Point new x: x + pt x y: y + pt y]
−   pt  [⇑ Point new x: x − pt x y: y − pt y]
```

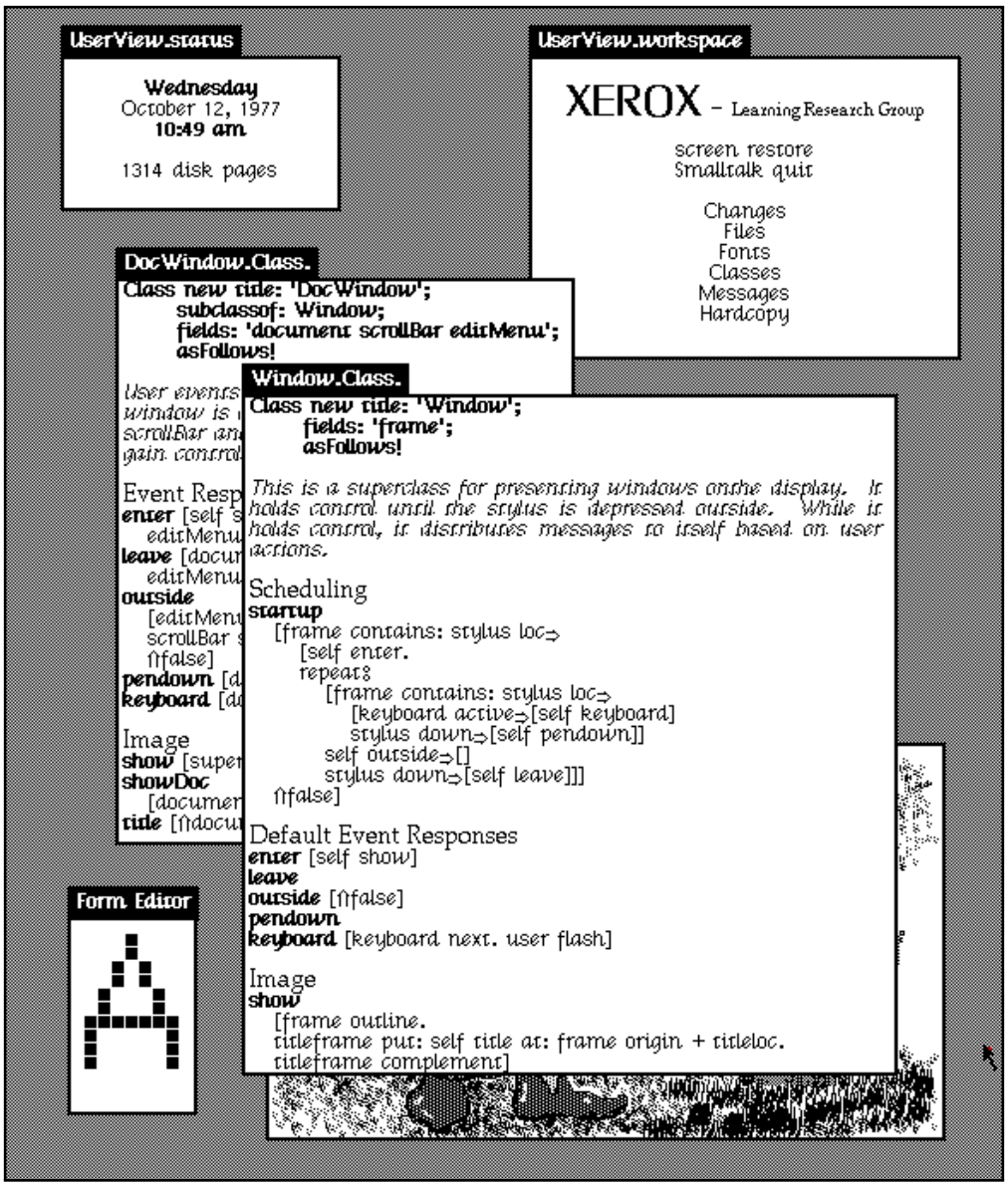


Fig. 14. A typical Smalltalk-76 screen showing two classes—`DocWindow` and `Window`—being edited. Figure replicates an illustration from Ingalls [1978, Fig. 3], but created on a Smalltalk-76 emulation.

The scheduling of windows shown in Figure 14 is implemented in Window's response to the message `startup` (visible in the figure). Central to that method is this expression:

```
frame contains: stylus loc
```

which checks whether the stylus location (a Point) is within the frame (a Rectangle) of the current Window. The code for Rectangle's `contains:` method builds on the meaning of <= for Points. The first test determines that `origin` is above and to the left of the Point `pt`, and the second test determines

that pt is above and to the left of corner. Following the precedence rule, arithmetic messages (+, <, etc.) are sent before keyword messages (the identifiers ending in colon). Therefore, after the two <= tests have been made, the conjunction of the two results (and:) defines whether the rectangle contains the Point pt. The arrow ⇑ returns this result.

### 5.5 Compilation

This vision of how to compile Smalltalk-76 included a byte-coded stack machine for evaluation of the code. Compilation of a message expression proceeded roughly as follows:

- evaluate any parameters, and leave them on the stack
- evaluate the receiver (a variable, or an expression in parenthesis), and leave it on the stack
- perform a send to the receiver with the selector name
- at the end of a statement, pop the stack
- at the end of a method or at a return arrow ⇑, pop the stack and return that value

The Rectangle method involved in the Window example above,

```
contains: pt        "return true if pt is inside me"
    [⇑ origin <= pt and: pt <= corner]
```

is compiled as

(1) corner—push (onto the stack) the rectangle's corner
(2) pt—push pt, the argument to contains:
(3) <=—send the message <= to the top of stack (origin) with argument pt
(4) pt—push pt, the argument to contains:
(5) origin—push the rectangle's corner
(6) <=—send the message <= to the top of stack (pt) with argument origin
(7) and:—send the message and: to the top of stack (a boolean result)
(8) ⇑—return the result returned from and:

The compiler has done some juggling so that the receiver of and: will be on the top of the stack when the message is sent. The compiled method for contains: appears at the center of Figure 15. The static structures through which the method is accessed appear in the upper half of the figure: class Rectangle points to a method dictionary which includes an entry for the selector contains:, and that entry points to both the source code and the compiled method.

### 5.6 Evaluation

I had heard of, but never seen, Peter Deutsch's bytecoded Lisp interpreter [Deutsch 1973]. The idea of a bytecoded interpreter appealed for several reasons. I liked compactness, and if we were going to compile this language I wanted it to be at least as compact as our earlier Smalltalks. Also, having written microcode for the kernel of OOZE and BitBlt, I had a visceral sense of the power of microcode. Stepping through byte-sized code syllables, and looking up offsets in an instance or method dictionary could be accomplished in microcode without wasting extra memory cycles to emulate the details in Nova instructions.

The operation of the virtual machine can be understood by examining the lower half of Figure 15. The message contains: has been looked up in the message dictionary of frame's class, Rectangle.

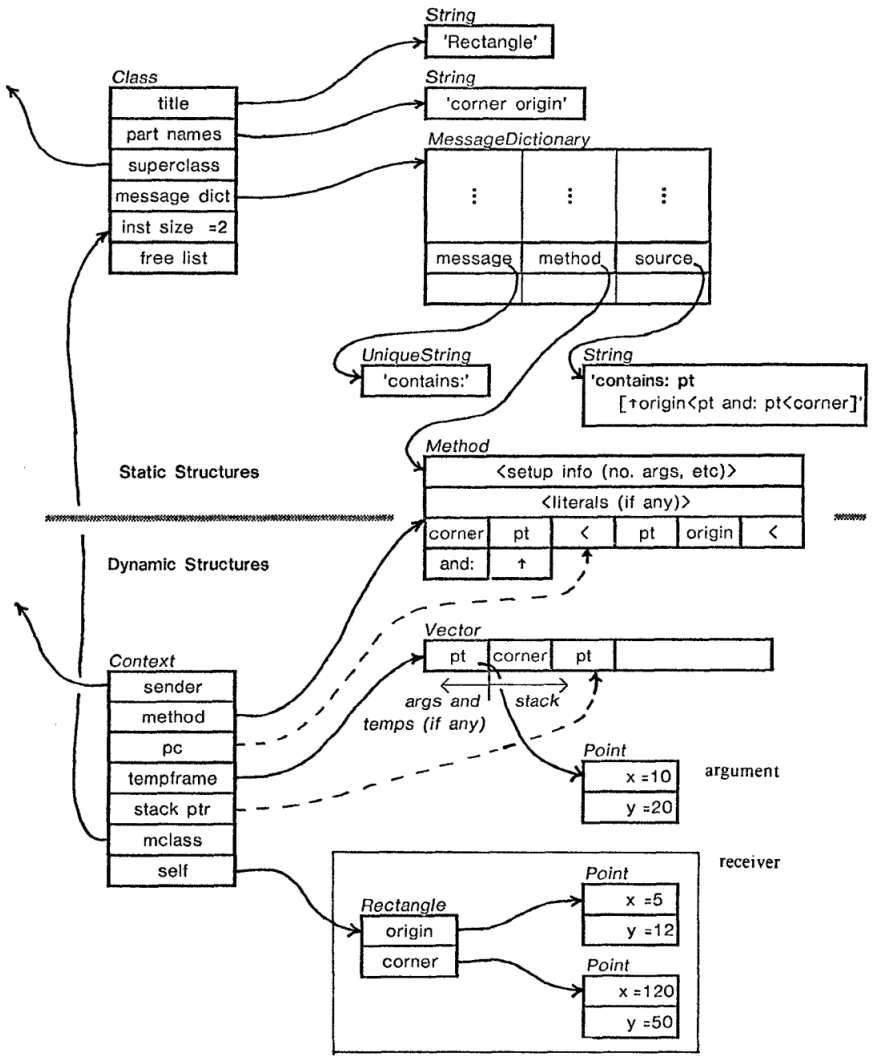


Fig. 15. The evaluation of `frame contains: stylus loc`. Figure from Ingalls [1978].

The *Method* object, shown in the middle, provides setup information from which a Context (think stack frame) has been set up as follows:

- sender—the "calling" context in which this expression was encountered
- method—the method found for `contains:`
- pc—the program counter that steps through the bytecodes compiled for this method
- tempframe—a vector allocated for arguments and temporary variables and the evaluation stack
- stack ptr—tracks the top of the evaluation stack

- mclass—the class in which this method was found. Note that mclass can be different from the receiver's class if the method is found in a superclass of the receiving instance[15]
- self—the receiver of the message; in this case the point value of 'stylus loc'

The instant depicted in the figure is when the first bytecode of the <= method is about to be executed. Returning to the operation of the virtual machine, we can see that '`currentContext step`' will pick up the next bytecode (the one for <=) and dispatch on it. In this case, <= is compiled as a special message, so it will run

```
self send: (SpecialMessages * lobits).
```

Here we encounter the world of trade-offs in speed and space that beckons to the virtual machine designer. The message <= is assigned a special bytecode because it is used frequently, and as such can be stored in a single VM table rather than requiring a literal in every method that sends that message. This confers a space benefit, and is also benefit in speed. The VM will have special code associated with this bytecode to first check if the receiver is an integer and, if so, to attempt a direct arithmetic operation without ever sending a message. In this case, it will fail because the receiver is a point, so it will retrieve the name <= from the SpecialMessages table, and send it as a normal message to that point, which will finally return a `true` or `false` to be pushed on the stack.

### 5.7 Contexts and Simulation

As mentioned earlier, Smalltalk-76 began its life as a simulation in Smalltalk-74 that repeatedly sent the message `step` to an instance of class Context.

```
repeat: (currentContext ← currentContext step)
```

When Smalltalk-76 came fully to life, this same pattern worked, and ran much faster! Figure 16 shows the code for Context-step in Smalltalk-76. This simple kernel capability enabled such system-level functions as tracing, profiling, and single-stepping to be written entirely as Smalltalk methods in class Context. The crucial step of making `Contexts` be first-class objects built a bridge between the language and its implementation.

### 5.8 Unevaluated Arguments and the Origin of Smalltalk Blocks

The syntax described so far covered almost all our examples of earlier Smalltalks in a manner that was readable and—for the same reason—compilable. The one important case not covered was that of unevaluated arguments, along with a mechanism for controlling their evaluation. For this I added the notion of a "non-evaluating keyword," signified by its ending with an "open colon," such as `do:` This was intended to be reminiscent of Smalltalk-72's "fetch-quote" function.

The argument following a non-evaluating keyword would not be evaluated before the message was sent, but would instead create a handle to the argument expression that could be evaluated one or more times by "remote control." The natural handle to use would be the current context, but we needed a different structure with its own `pc` so that it could be evaluated at a later time, or even repeatedly. Therefore I generated code to create a "remote context," a copy of the current context, with its own `pc`. To evaluate the argument expression, a client would send `eval` to the remote context, causing it to run the bytecodes of the argument expression exactly as it would have

[15]The pseudovariable *super* can be used to send messages intended to be looked up in a superclass of a given method. For that reason it is necessary to know the class in which that method was found.

```
Simulation of the Interpreter
step | byte lobits                          "dispatch on next code syllable"
   [byte ← self nextbyte.
   lobits ← byte|16.
   byte/16=1⇒[self push: receiver instfield: lobits];       "load from instance"
         =2⇒[self push: tempframe˙lobits];                  "load from temps (and args)"
         =3⇒[self push: (method literals: lobits)];         "load from literals"
         =4⇒[self push: (method literals: lobits) value];   "load indirectly from literals"
         =5⇒[self push: self instfield: lobits];            "load from this Context"
         =6⇒[self push: (☞"1 0 1 2 10 true false nil)˙lobits];  "frequent constants"
         =7⇒[⇑self send: (method literal: lobits)];
         =8⇒[⇑self send: (SpecialMessages˙lobits)];         "frequent messages"
         =9⇒[lobits<8⇒[pc← pc+lobits]                       "short jump forward"
               self pop⇒[] pc← pc+lobits-8];                "short branch if false and pop"
         =10⇒[lobits<8⇒[pc← lobits-3*256+self nextbyte+pc]"long jump forward and back"
               self pop⇒[pc← pc+1];                         "skip extension byte on true"
               pc← lobits-11*256+self nextbyte+pc];         "long bfp"
         =11⇒[lobits=0⇒[self pop];                          "pop stack"
               =1⇒[self store: self top into: self nextbyte]; "store"
               =2⇒[self store: self pop into: self nextbyte]; "store and pop"
               =3⇒[sender push: self top. ⇑sender]]         "return value to sender"
   ]
store: val into: field | lobits              "same encoding as above"
   [lobits ← byte|16.
   field/16=1⇒[receiver instfield: lobits ← val];     "store into instance"
         =2⇒[tempframe˙lobits ← val];                 "store into temps (and args)"
         =3⇒[user notify: 'invalid store'];           "can't store into literals"
         =4⇒[(method literals: lobits) value ← val];"store indirectly through literals"
         =5⇒[self instfield: lobits ← val]            "store into this Context"
   ]
send: message | class meth callee t i      "send a message"
   [class ← self top class.
   until⦇ (meth← class lookup: message) do⦈ "look up the method"
      [class← class superclass.                "follow the superclass chain if necess"
      class=nil⇒[user notify: 'Unrecognized message: '+message]]
   [meth primitive⇒                            "If flagged as primitive, then do it"
      [self doprimitive: meth ⇒[⇑self]]].      "If it fails, proceed with send"
   callee← Context new                         "create new Context, and fill its fields"
      sender: self method: meth pc: meth startpc
      tempframe: (t← Vector new: meth tframesize) stackptr: meth startstack
      mclass: class receiver: self pop.
   for⦇ i to: meth nargs do⦈                   "pass arguments"
      [t˙i← self pop]
   ⇑callee]                                    "return new Context, so it becomes current"
nextbyte       "step pc and return next code syllable"
   [⇑method˙(pc← pc+1)]

Stack-related Messages
push: val       "push value onto top of stack"
   [tempframe˙(stackptr← stackptr+1) ← val]
top    "return value on top of stack"
   [⇑tempframe˙stackptr]
pop | t          "pop value off stack and return it"
   [t← tempframe˙stackptr.
   stackptr← stackptr-1. ⇑t]
```

Fig. 16. An excerpt of the Smalltalk-76 virtual machine (written in Smalltalk-76) summarizing the operation of Context-step. Figure from Ingalls [1978].

```
while⦇ [input end⇒[false] (c←input peek) iskeyword] do⦈
    [t ← [t @ nil⇒[input next] t + input next].
       a append: [c isuneval⇒
                    [self remote⦇ (self compileterm: input)]
             self compileterm: input].
       self push]
```

Fig. 17. An argument is compiled as a remote context if its associated keyword ends with an open colon, causing `c isuneval` to be true.

in its original context, except that when execution reached the end of the block, it would return to the sender of `eval`, not to the sender of the home context.

The compiler knew to generate this special form of "remote return" for unevaluated arguments. Figure 17 shows the code that compiled a keyword selector `c` and its argument (the result of `self compileterm: input`), appending the resulting bytecodes to the instruction stream `a`. For a normal (evaluating) message selector, `<rcvr> <selector> <expr>` … compiled to

(1) push expr
(2) push rcvr
(3) send selector
(4) …

In Figure 17, the expression `c isuneval` checks whether or not the keyword bound to `c` ends in an open colon. If it does, the method `remote:` is used to wrap the bytecodes for `<rcvr> <selector> <expr>` … with the added mechanism to create a remote context for evaluation.

(1) push currentContext
(2) send remoteCopy
(3) jmp: (9)
(4) push expr
(5) push rcvr
(6) send selector
(7) remote retn
(8) jmp: (4)
(9) …

Here is the code that generated this sequence:

```
Compiler understands: 'remote: b | a t
    [a←Stream default.
    self push. b ← b eval. self pop.     "extra stack to push caller"
    a next← curcode; next← self encodeop: remoteCopy.
    t ← b length+3. a next← t/256+jmpcode+4; next← t\256.
    a append: b; next← endcode; append: (self encodejmp: jmpcode by: 0-t).
    ⇑a contents]'!
```

When this code ran, it created a remote context, and then jumped around the `expr` so that it did not get `eval`'ed at the time of the method call. The receiver of the unevaluating keyword would thus receive a remote context rather than the evaluated expression. The remote context could later be sent the message `eval` to evaluate the expression whenever desired. In fact, the expression could be `eval`led repeatedly, because the backward jump at the end of the setup sequence (item 8) would start a new evaluation.

Note, though, that if the code contained an explicit return statement, that would cause a return from the home method, effectively exiting from the remote control structure.

The other requirement of remote code was the ability to store into variables in the home context, as with the induction variable of a for-loop. This was handled by a primitive method `next←` in `RemoteContext` that looked for a variable reference at the starting pc, and stored into that variable.

The beauty of these unevaluated arguments, or "blocks," was that they provided a way for users to write their own control structures on a par with those in the base language. This little two-line method gives us a non-evaluating OR method:

```
Object understands: 'or: alternative | priorCursor value
    ["Non-evaluating OR evaluates alternative only when receiver is false"
    self ⇒ [⇑ true]
    ⇑ alternative eval]'
```

Here is an example that proved to be quite useful in the user interface:

```
Cursor understands: 'showWhile: activity | priorCursor value
    ["Display this cursor during the execution of activity, and then
            restore the prior one"
    priorCursor ← user currentcursor.
    self show.
    value ← activity eval.
    priorCursor show.
    ⇑ value]'
```

This `showWhile:` method appears in several places, such as:

```
readCursor showWhile: [file load]
upCursor showWhile: [self scrollUpWhile: [self clickInRegion]]
cornerCursor showWhile: [window moveTo: cursor location]
```

### 5.9 The Origin of Smalltalk SymbolTables[16]

One of the clever tricks in Smalltalk-76 that fell into my lap was what I called "Symbol Tables." A `SymbolTable` was a dictionary that had an indirect `ObjectReference` cell holding each value. One of these `ObjectReference` cells can be moved (e.g., to another dictionary), and have its value updated independent of the dictionary in which it resides.

The bytecodes for loading and storing *local* variables had direct access to the temporary variable frame, or the receiver (`self`) frame, that provided storage for those variables. However, *non-local* variables needed to refer to a dictionary entry, either in some class variable dictionary, or in the global `Smalltalk` dictionary. It seemed as though this reference would require both a pointer to the dictionary, and the name of the variable to use as the key, and a lot of access machinery to chase the reference at run time. However, putting an `ObjectReference` in the dictionary meant that the non-local variable reference could point directly to the `ObjectReference`, and the read and write bytecodes could access the variable's value in the `ObjectReference` directly, while still enabling the variable to be accessed by name using the appropriate dictionary. It took only 5 methods to implement a `SymbolTable`, using `ObjectReference` and `Dictionary`; the code is shown in Figure 18.

This `SymbolTable` could be used for class variables and globals alike, enabling the simple "indirect literal" bytcodes to read and write in those dictionaries. But wait, there is more! When such references could not be resolved as globals or class variables, they were placed in a special `Undeclared` dictionary. The `Undeclared` dictionary provided for forward references while compiling from a file. The beauty of the indirection through SymbolTables was that a forward reference or undeclared

[16]The use of the term "SymbolTable" here refers specifically to a Smalltalk dictionary with names as keys, and a level of indirection to the values.

```
Class new title: 'SymbolTable';
    subclassof: Dictionary;
    fields:''~
"SymbolTables have the same properties as Dictionaries, except that an indirect
reference is interposed between the value entries and the actual values. This allows
compiled code to point directly at a reference which remains valid although the value
changes. Notice that the define message checks in Undefined for unresolved references
which the compiler may have placed there previously."
SymbolTable understands: 'insert: name with: x
    [super insert: name with: (ObjectReference new value← x)]'!
SymbolTable understands: ': name
    [⇑(super'name) value]'!
SymbolTable understands: ': name ← x
    [⇑(super'name) value ← x]'!
SymbolTable understands: 'ref: name
    [⇑super'name]'!
SymbolTable understands: 'define: name as: x
    [self has: name⇒ [self'name ← x]
    Undeclared has: name⇒
        [super insert: name with: (Undeclared ref: name).
        self'name ← x.
        Undeclared delete: name]
    self insert: name with: x]'!
```

Fig. 18. Definition of `SymbolTable` from the Smalltalk-76 Annotated Bootstrap

variable could be declared as a global or class variable later on, regardless of pre-existing references, and nothing needed to be recompiled. The `ObjectReference` in question could simply be removed from `Undeclared`, and be placed in another dictionary, all with no effect on the code that had already been compiled, nor even on the values that might have been stored there.

Later, in Smalltalk-80, when Dictionaries were restructured to be Sets of Associations, this indirect access mechanism was moved to work through Associations, with the added convenience of providing access to the variable names required for decompilation.

### 5.10 Debugging Capability

A virtual machine design amounts to a specification of how each step of a computation shall be carried out. In Smalltalk-72, Alan's original design laid out the steps required to evaluate an expression. In Smalltalk-76, my design centered around a set of bytecodes that would carry out the data movement and communication functions that would achieve the same global goal of computation by sending messages. The Smalltalk-72 evaluation mechanism was simple enough that I could fairly reliably write assembly code from Alan's specification, even where it required some further clarification. By contrast, the Smalltalk-76 interpreter was a complex virtual machine. While it was as simple as I could make it, it nonetheless involved picking variable-length instructions from a byte stream, loading and storing data from numerous sources, looking up methods in an inheritance hierarchy of hash tables, etc. I don't recall ever considering writing a native code interpreter straight from such an architecture. Instead, I used Smalltalk-74, the only tool I had, to simulate the operation of Smalltalk-76. I have already alluded to this history, but it had an impact that I don't think any of us anticipated.

Almost from the day we got Smalltalk-76 running we realized that "something had happened." Not only was it more than 10 times faster[17] and more powerful (inheritance shrank the kernel while actually providing more functionality), but having Contexts as real objects suddenly put detailed control over the system into the domain of the user language. Thus, when an error occurred, we created a new Smalltalk process, with a handle to the context in trouble, and provided a simple browser-like interface to it. A simple two-line method could print out the stack, and with a bit more code, the variables at each level of that stack. Soon we had written an entire debugger in Smalltalk, and not long after, fleshed out the vestigial Smalltalk-74 logic to enable Smalltalk-76 to completely simulate itself. This in turn enabled tracing capabilities and source-level execution profiles with time spent and exact execution counts.

### 5.11 Message Not Understood

When the virtual machine failed to find a message, it would package up that message and send it to the intended receiver as an argument of the message `doesNotUnderstand:`. This began simply as a way for us to invoke the debugger with full information about the suspended context. However, it soon became clear that it could be used for other purposes, such as enabling a proxy object to be inserted between sender and receiver to notice exceptional events, or even to transmit the entire message stream to a remote receiver.

### 5.12 Inheritance

The Smalltalk-76 bootstrap[18] began with the definition of a class Object, from which all the other classes in the system inherited common behavior. After two generations of Smalltalk, this establishment of a superclass Object was finally a declaration in the language of the underlying principle that everything is an object. Figure 19 shows the protocol for class `Object`. I described Object and several other classes as "abstract" because their purpose was to establish an inheritable method protocol although there was no intention that instances would be made of exactly that class. Other abstract classes included `Array` and `Number`. In later systems, parts of `Object`'s protocol were pushed down into more specific subclasses such as `Collection` and `Boolean`.

There were no enforced properties of abstract classes, and a number of concrete classes served this same purpose of establishing a meaningful inheritance hierarchy. `Stream` was one such concrete class, whose subclasses benefited greatly from inheritance; the common protocol is shown in Figure 20. The messages for reading and writing, called `next` and `next←`, are written to send the messages `pastend` and `pastend←` if the stream is at its end. These return `false` in the read case, and grow the underlying array as required in the write case. While not shown here, a subclass `Filestream` inherits the same code, but of course responds differently, with `pastend` and `pastend←` advancing to the next page of a file on disk. The synergy goes even farther: the frequently used `next` and `next←` messages have primitive implementations allowing them to perform in native code, or even microcode, and this optimization works to the benefit of files, as well as any new kind of stream that may be added.

[17] The benchmarks in Appendix E.2 show about 28× improvement in simple operations and 2.5× in speed of full sends. Much of the code was written to minimize full sends and maximize the leverage of simple and especially primitive operations, so the change we experienced was dramatic.

[18] We use the noun "bootstrap" here, as we did in those days, to refer to a file of initial definitions that could be read into a raw interpreter, after which it became a usable programming system.

```
Class new title: 'Object';
    subclassof: nil;
    abstract~
"Object is the superclass of all classes.  It is an abstract class, meaning that it has no
instance state, and its main function is to provide a foundation message protocol for its
subclasses.  Three instances of this class are defined, namely: nil, true, and false."

"primitives"
Object understands: '@ x [] primitive: 4'!"test for identical pointers"
    "Font will be edited so this looks like ≡, meaning eq"
Object understands: 'hash [] primitive: 46'!"pointer as an Integer"
Object understands: 'asOop [] primitive: 46'!"Dont override this"
Object understands: 'refct [] primitive: 45'!"current reference count"
Object understands: 'class [] primitive: 27'!"class of this object"
Object understands: 'instfield: n [] primitive: 38'!"subscript any objct"
Object understands: 'instfield: n ← val [] primitive: 39'!

"boolean connectives"
Object understands: 'or: x [self⇒[⇑true] ⇑x]'!
Object understands: 'and: x [self⇒[⇑x] ⇑false]'!
Object understands: 'xor: x [x⇒[⇑self@false] ⇑self]'!
Object understands: 'eqv: x [x⇒[⇑self] ⇑self@false]'!

"following don't evaluate their arg unless necessary.
They are built for comfort, not for speed."
Object understands: 'or: x [self⇒[⇑true] ⇑x eval]'!
Object understands: 'and: x [self⇒[⇑x eval] ⇑false]'!

"default protocol"
Object understands: 'printon: strm
    [self@nil⇒ [strm append: "nil"]
    self@false⇒ [strm append: "false"]
    self@true⇒ [strm append: "true"]
    self class print: self on: strm]'!
Object understands: 'asString | strm
    [strm ← Stream default.
    self printon: strm. ⇑strm contents]'!
Object understands: 'print
    [user show: self asString]'!
Object understands: '= x [⇑self@x]'!
Object understands: '≠ x [⇑self=x@false]'!
Object understands: 'is: x [⇑self class@x]'!
Object understands: '_ x | v
    [v ← Vector new: 2.
    v·1 ← self. v·2 ← x.  ⇑v]'!
Object understands: 'startup  "loopless scheduling"
    [self firsttime⇒
        [while: self eachtime do: [].
        ⇑self lasttime]
    ⇑false]'!
Object understands: 'canunderstand: selector
    [⇑self class canunderstand: selector]'!
Object understands: 'copy      "create new copy of self"
    [⇑self class copy: self]'!
Object understands: 'recopy"recursively copy whole structure"
    [⇑self class recopy: self]'!
Object understands: 'error
    [user notify: "Message not understood."]'!
Object understands: "s code
    [self class understands: "doit [⇑[" + code + "]]".
    ⇑self doit]'!
```

Fig. 19. The inherited protocol for class `Object` in the original Smalltalk-76 system including comparisons, printing, boolean operations and even the loopless programming kernel. The "@" sign here is a stand-in for "≡," absent in the listing font of the day.

```
Class new title: 'Stream';
    fields: 'array position limit'~
Stream understands: 'of: array
    [position ← 0. limit ← array length]'!
Stream understands: 'of: array from: position to: limit
    [position ← position-1]'!
Stream understands: 'default
    [self of: (String new: 8)]'!
Stream understands: 'next"simple result"
    [self myend⇒ [↑self pastend]
    ↑array˙(position ← position+1)] primitive: 17'!
Stream understands: 'next ← x      "simple arg"
    [self myend⇒ [↑self pastend ← x]
    ↑array˙(position ← position+1) ← x] primitive: 18'!
Stream understands: 'append: x | i"Array arg"
    [for° i from: x do°
        [self next ← i].
    ↑x]'!
Stream understands: 'myend
    [↑position≥limit]'!
Stream understands: 'pastend
    [↑false]'!
Stream understands: 'pastend ← x
    [array ← array grow. limit ← array length.
    ↑self next ← x]'!
```

Fig. 20. The inherited protocol for abstract class `Stream` in Smalltalk-76.

In the first days of these inheritance groups we were sometimes careless in referring (in an abstract superclass) to methods that were not present, but which would be required in subclasses. Adele Goldberg, who was always aware of the learner's dilemma, instituted the practice of ensuring that such methods be added to the superclass as a stub `self subclassResponsibility` method. This made it clear, both to a reader of the class and to a programmer encountering an error, when a method was required in a subclass of an abstract superclass.

### 5.13 The Remarkable Synergy of Messages and Inheritance

Compilation is a lighter process in a message sending language. The reason is that messages are independent of the type of the receiver, and therefore the code is free of type information—the code itself is *abstract* in that sense. Let's take a look at the code for `max:` in class `Number` as an example.

```
<Number> max: other
    [self < other ⇒ [⇑ other]]
```

This means the following: if the receiver `self` is less than the argument `other`, return the argument, otherwise return `self`.[19] We can use Smalltalk to look at itself by evaluating `Number method: #max:` to reveal the bytecodes of the compiled method, as shown in Figure 21. The method is nicely compact, but it may not seem extraordinary at first. However, notice that the comparison is not an Integer less-than, nor is it a Float less-than; rather it is the *message* less-than, which is understood by both Floats and Integers…and MachineDoubles, and Dates, and Times, and by your own class Fraction if you choose to add it! This is the synergy between inheritance and message sending: many inherited methods in Smalltalk serve in multiple classes without needing to be copied, neither as source code nor even as the underlying bytecode. At runtime, the virtual machine needs only

[19] All methods return `self` implicitly if they run to completion.

```
max: other
    [self < other ⇒ [↑other]
    ↑ self]
"
(Number method: ↷max:) asBytes ==> '01 04 020 0161 0262 0231 020 0203 0161 0203 '
These bytes encode...
    2 header bytes (number of args, temps, and stack needed)
    push  other
    push  self
    send  '<'
    branch-if-false-and-pop  .+3
    push  other
    return
    push  self
    return
"
```

Fig. 21. Bytecodes compiled for Number-max: in Smalltalk-76

one entry for this method in its method cache; that means more space for other methods, and that means a little more speed: always a good thing.

### 5.14 The Multipane Window Interface

As we began serious development in Smalltalk-76, we wanted to be able to see the whole codebase "online." With over 50 classes and 500 methods, Smalltalk-74's class-per-file file editor did not allow one to jump around the system in a nimble manner. Before long, Larry Tesler came up with a multipane browser that made it easy to scroll to any class in the left pane, and then to any message selector in the right pane, and immediately see just that method in the bottom pane. This was already useful by itself, but with our overlapping windows, it was also easy to have several such browsers open on various parts of the system at the same time, and to jump between them with a click of the mouse. As the system grew, so did the browser, soon reaching the standard form that includes system categories and method categories for further organization.

The multipane paradigm also offered a perfect way to inspect any object, with a list of instance variable names on the left and a place to print or change their values on the right. Here again, several of these in separate windows made a wonderfully clear way to visualize the various parts of a system in development.

More than any other tool, the debugger benefited from this method of presentation. By putting the context stack in a scrollable pane, it became possible to see and explore the entire state of a suspended computation. Any context that was selected could reveal its details, such as temporary variables and instance variables, in separate panes. The ability to select a point in a suspended context, change a value, and then resume was the highlight of many Smalltalk demonstrations, and it changed the people who saw it.

### 5.15 Projects

Soon after we had Smalltalk-76 running, it occurred to me that if we saved the window scheduler as an object, we could jump between two or more complete views of the system just by restoring the screen from the different window lists. The system kept track of changes for the purpose of being able to save them on file and collaborate with others. By replicating the change tracking along with the scheduler for each of these "project" views, it suddenly became possible to work on

a number of different tasks without ever having to leave the Smalltalk environment. I remember being able to create this whole facility in an afternoon because the scheduler, the windows, and the change sets, were already simple and independent objects. In that one day, Smalltalk-76 acquired the feel of a designer's notebook.

### 5.16 Summing up Smalltalk-76

Smalltalk-76 was all we had hoped for, and more. The combination of the terseness of Smalltalk and the compactness of the OOZE object memory meant that we could enjoy an address space of 60,000 objects on a computer that barely offered 32k bytes of working storage. We also had a compilable library of general classes that represented all the data structures needed to support a personal computing kernel with a graphical user interface.

Among its remarkable properties for computing systems of the day:

- all code was live and editable, with a turnaround time of a few seconds
- inheritance and messages combined to produce compact compiled code
- the entire system could be emulated with augmentation, as for tracing or profiling
- the debugger was written entirely in the source language, and could run as a live window in the system being debugged
- the system could be interrupted at any time (with ctrl-C) to open a debugger and learn about all the objects at work
- it was possible to enumerate all the instances of any class
- when arithmetic on `SmallIntegers` overflowed, the result became a `LargeInteger`; inherited operations continued to work with the `LargeInteger` values
- robustness and power came from the `doesNotUnderstand:` facility
- Smalltalk-76 ran over 10 times faster than Smalltalk-74, fulfilling our need for speed

We enjoyed presenting our integrated development system, and these demonstrations began to attract interest from industry. While this outreach would become our primary focus, we began with efforts to use Smalltalk's simplicity and visual interface to reach Xerox upper management.

### 5.17 Significant Projects

One of the most ambitious projects in Smalltalk-76 was the "Kearns Curriculum," later known as the "Simulation Kit"—a job-shop simulation designed by Adele Goldberg to teach Xerox executives about software in a hands-on manner. It was an amazingly gutsy project, and Adele led it from beginning to end, designing and implementing the curriculum,[20] and promoting it to the top executives at Xerox. She organized everyone in our group to contribute to it, and ultimately to sit each beside one of the executives as they went through the curriculum. The theme of the workshop attended by the executives—historically accustomed to creating specialized hardware with software as a sideline—was to understand the growing importance of software as the driving product of the computer industry. David Kearns, then president and later CEO, was an exIBMer who, as it turned out, knew how to program!

Figure 22 shows a screenshot of the Simulation Kit. Each station could have a different role, and jobs followed a scripted trajectory from one station to the next, waiting on an input queue until

[20] Adele was the second beneficiary of a skunkworks Alto at home since her second child had just been born.

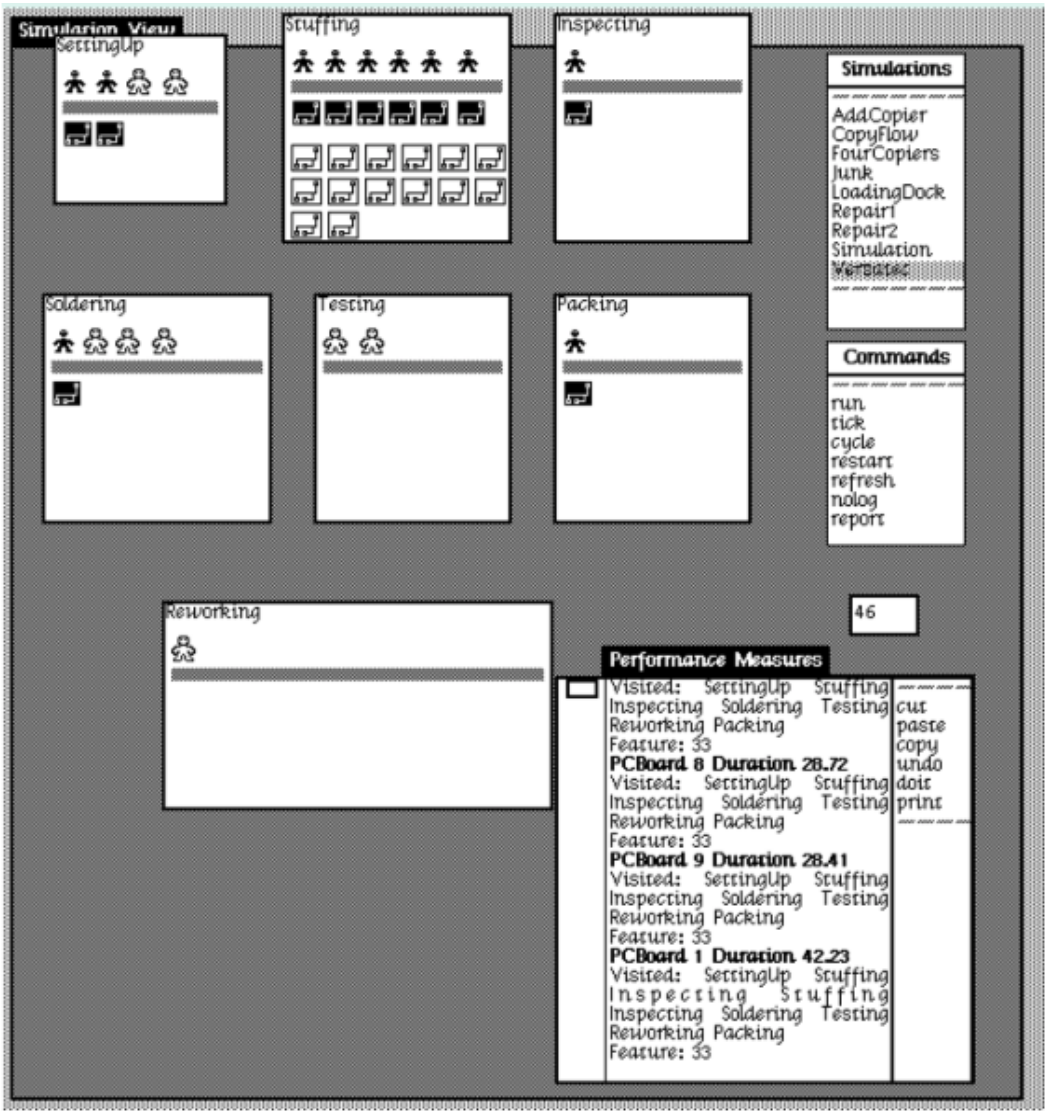


Fig. 22. A snapshot of the Simulation Kit. In this example PCBoard 1 was slowed down because it failed testing and had to repeat part of its agenda. It soon became clear that "Stuffing" needed more staff. To run the SimKit, click this image or run https://smalltalkzoo.thechm.org/HOPL-St78.html?simkit

the receiving station was ready to process another job. Each station had an input and output queue and specified allocation of workers. As jobs exited the simulation, they reported their history in the 'Performance measures' panel. It shows animated circuit boards being stuffed, inspected, soldered, tested, and packed for shipping. This particular example was chosen to be relevant to Versatec—a Xerox company at the time.

The Kearns curriculum came off without a hitch, in spite of a hugely risky change to the system just days before (described in Appendix B.4). Respect for our group (the Learning Research Group)

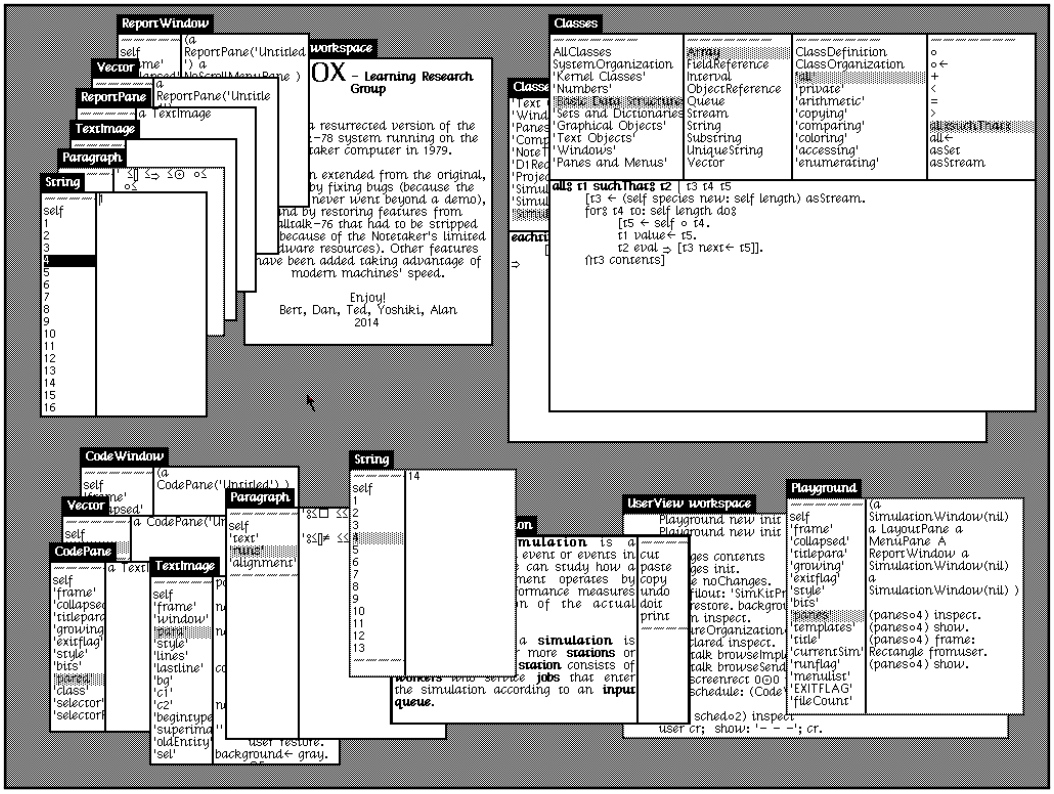


Fig. 23. A snapshot of my screen while debugging a problem with stylized text when converting the Simulation Kit code to run live in Smalltalk-78. I made the snapshot because I was struck that this was still a first-class debugging environment, yet it was written more than 40 years ago. All it lacked at the time was a bigger screen and faster processor, and we knew these were coming.

rose enormously as a result. Within the group too, we had a new-found sense that we could do anything that we put our minds to.

Fast-forward to 2019, when we found that one of the Alto disks appeared to have the Smalltalk-76 source files for the Simulation Kit on it. Since the Smalltalk-78 language is nearly identical to Smalltalk-76, it occurred to me that I might be able to get the Simulation Kit to run in our revived Smalltalk-78, emulated in a web browser. It was not a trivial task because the representation of rich text was changed in Smalltalk-78.

Another major project attempted in Smalltalk-76, which took full advantage of OOZE's large capacity, was PIE. This stood for "Personal Information Environment," [Goldstein and Bobrow 1981] and was a large information network architecture developed by Danny Bobrow and Ira Goldstein, researchers in CSL, and collaborators in our Smalltalk explorations.

As his thesis project at Stanford University, Alan Borning [Borning 1979, 1981] wrote his innovative Thinglab constraint-oriented programming environment in Smalltalk-76—Figure 24 shows a screenshot. This project has been resurrected in Smalltalk-78 and can be explored at https://constraints.cs.washington.edu/thinglab/. The quadrilateral visible in the ThingLab browser

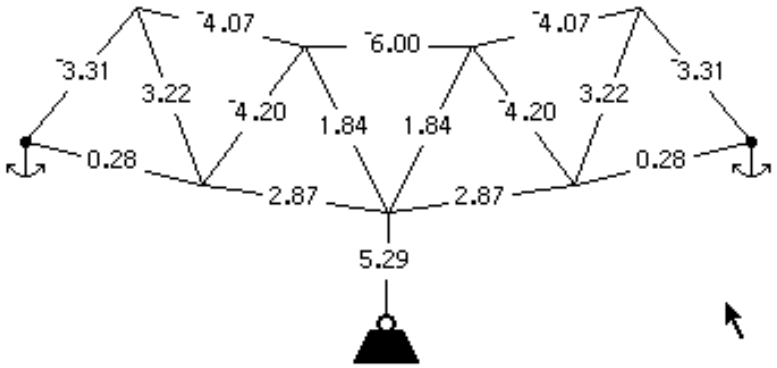


Fig. 24. A screenshot from ThingLab. The forces in the bridge framework due to the suspended weight are calculated as the weight is changed, or the anchors are moved. Note that some members are in compression, others in tension.

can be manipulated by choosing, e.g., "move" as the operation, and "point" to let you attach the mouse to a point. The example in Figure 24 is "FixedBridge."

### 5.18 Reflections

We could "smell" the coming microprocessor revolution, and the combination of a compact BitBlt graphics kernel and some work on the VM made it clear that this would soon be an interesting piece of technology for research and industry alike.

## 6 SMALLTALK-78

In 1978, 16-bit Microprocessors were coming into production, offering us the chance to build a portable Smalltalk computer. Memory costs were also dropping, so that it was reasonable to fit an entire Smalltalk image in main memory without OOZE. Alan managed to find some extra funding for a young Xerox engineer, Doug Fairbairn, to build a prototype portable machine. The machine was named "NoteTaker," and used three Intel 8086 processors on an innovative bus structure that Doug had designed, coupled to 256K bytes of resident memory.[21]

I was excited about anything in the direction of a portable computer because portable meant personal. The move from the Alto to a single-chip microprocessor was a step down in power, but it seemed that if we had enough RAM to get by without OOZE, we might get Smalltalk to run usably on such a machine. I started to think of how to port all our existing assembly code, and how to squeeze every last cycle out of Smalltalk-76 on the 8086.

[21] The NoteTaker actually had three 8086's; one to manage the disk, one to run Smalltalk, and a special "fast" 8Mhz 8086 to manage the Ethernet. Ever looking for more cycles, I wrote a special version of BitBlt that ran in the Ethernet processor, and took BitBlt commands off a queue from the main Smalltalk processor. This nearly doubled the speed of text-heavy operations.

### 6.1 Porting Smalltalk-76

The Smalltalk-76 virtual machine consisted at the time of roughly 16K bytes of assembly code. About half of this comprised the interpreter, memory manager including OOZE, OS, IO and process scheduler; the other half was devoted to line drawing, bitmap sketching operations, text display, and scrolling. I settled on the following strategy to make Smalltalk run on the new machine:

- Copy the bulk of the Smalltalk system verbatim from Smalltalk-76.
- Eliminate OOZE in favor of in-memory objects up to the size of available memory.
- Hand-translate the kernel virtual machine from Nova code and Alto microcode to 8086 code.
- Rewrite line drawing, text display, and all other graphics to use only BitBlt.
- Map Smalltalk's contexts onto the 8086 support for overlapping stack frames.

Copying the bulk of Smalltalk (after eliminating as much as possible) was simple. Ted retargeted the SystemTracer to output a new format for the NoteTaker. In the process, he eliminated OOZE by writing a direct-pointer object table and corresponding object data file. Because we were porting Smalltalk-76 verbatim, creating the kernel virtual machine required only a straightforward translation from Nova assembler and Alto microcode to 8086 assembly code. This reduced the porting task to rewriting text and graphics, and rewriting the management of contexts to use contiguous overlapping stack frames.

### 6.2 Unified Text Display with BitBlt

From our experiments in Smalltalk-74 and Smalltalk-76, we had learned that it should be possible to implement all the necessary graphics by calls on BitBlt, with the surrounding logic all in readable, malleable, Smalltalk code. Moreover, faced with the task of translating Smalltalk-76's assembly code from Alto Novacode to the Intel 8086 instruction set, the chance to reduce the task by half was appealing. Therefore, as we moved to the NoteTaker, we committed to implementing all of the graphics operations in Smalltalk itself, with calls on BitBlt. I had already written BitBlt in Alto microcode; surely 8086 assembly code would not be that bad?

Still in Smalltalk-76, I began to reimplement the layout of text entirely in Smalltalk, except for one kernel call on BitBlt. Managing the layout of characters in a string consists of three tasks: arranging the characters of a string in a rectangle (and possibly displaying them), detecting which character lies at a given position (for locating a mouse click), and determining the bounds of a given character in the string (for displaying a selection). I soon discovered that subtle coding differences could cause inconsistencies between these three operations. For this reason, I designed the kernel of text display around a *single method* `scanWord:`, which is shown in Figure 25.

The loop in `scanWord:` will scan and optionally print text (depending on the flag `printing`) until

- a given some horizontal position (`stopX`) is reached,
- a particular character (determined from `exceptions`) is found, or
- the end of a range in the text (`endRun`) is reached.

A great advantage of condensing all the logic into a single simple method was that it was easy to write an (optional) primitive for this method, so that the performance of all these functions was close to what would be achieved if the whole layout package had been written in primitive code. Once the `scanWord:` method and its surrounding interface were working, I segregated the old text code and left it all behind when we wrote the new image for the NoteTaker. BitBlt, the

```
scanWord: endRun
    | charIndex |
    < primitive >    "May be implemented internally for speed"
    [charIndex > endRun] whileTrue:
        [charIndex ← text at: textPos.                          "pick character"
        (exceptions at: charIndex) > 0                          "check exceptions"
            ifTrue: [↑ exceptions at: charIndex].
        sourceX ← xTable at: charIndex.                         "left x of character in font"
        width ← (xTable at: charIndex + 1) − sourceX.           "up to left of next char"
        printing ifTrue: [self copyBits].                       "print the character"
        destX ← destX + width.                  "advance by width of character"
        destX > stopX ifTrue: [↑ stopXCode].                    "passed right boundary"
        textPos ← textPos + 1].                                 "advance to next character"
    textPos ← textPos − 1.
    ↑ endRunCode
```

Fig. 25. A single method implements all layout, mouse-finding and display of text in Smalltalk-78 and subsequent Smalltalks. The method runs in a context where the source is a font and the destination is the display.

`scanWord:` text loop, and a similar loop for drawing lines (see *Byte* [Ingalls 1981]) comprised the entire graphics package for Smalltalk-78 and the Smalltalks that followed.

### 6.3 Linearized Context Stack

To achieve acceptable performance on the 8086, I reworked the `Context` architecture of Smalltalk-76 to use a single large `Process` object with overlapping stack frames, according to the stack discipline supported by the 8086 hardware. Using overlapping stack frames was not unheard of, but I had to work hard to map Smalltalk's independent `Context` objects onto that architecture. Special "outrigger" `RemoteContext` objects made it possible for this single large hardware-supported object to offer the same Smalltalk-level access as the more general architecture of Smalltalk-76 Contexts. This demonstration that the runtime system could use more efficient primitive structures—while still supporting the convenience and maintainability of clean object-oriented design—was never published. I think it was generally known to, or independently derived by, a number of people who worked on increasing the speed of Smalltalk on conventional processors [Deutsch and Schiffman 1984]. That said, the future implementations covered in this paper reverted to the cleaner, simpler, context-objects of Smalltalk-76.

While the conversion to a linearized stack with outriggers was a complication that we dropped in future Smalltalks, the move to BitBlt-based graphics stayed with us. It reduced the codebase that needed to be ported, not only to the NoteTaker but also to the Dorado when we finally moved away from the Alto as the platform for our Xerox Smalltalk work. Besides saving a lot of work in porting, the smaller graphics kernel also simplified all the nice user interface features such as overlapping windows, menus, animation, and high quality fonts.

This then became Smalltalk-78, also known as "NoteTaker Smalltalk," from the name of the (trans)portable machine being built to run it. The Smalltalk-78 architecture was nearly identical to that of Smalltalk-76; in fact almost the entire Smalltalk codebase was mechanically written out from Smalltalk-76 into a virtual image that could be read into the NoteTaker machine. The project was a success, demonstrating a code-compatible version of Smalltalk-76 running on a microprocessor.

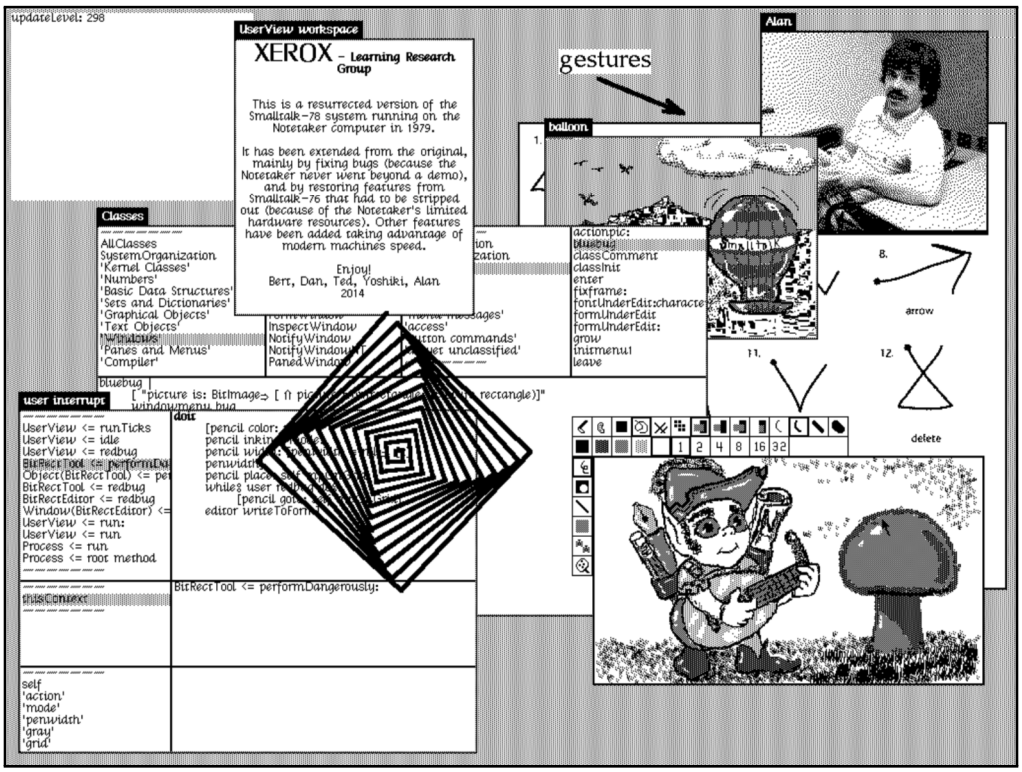


Fig. 26. A screen shot of the Smalltalk-78 system on a Dorado-sized screen (1024×808) that is larger than the 640×480 Xerox NoteTaker for which it was designed. With today's VM (see Appendix E), performance is excellent. To run this system live, click this image or visit https://smalltalkzoo.thechm.org/HOPL-St78.html?full

The virtual machine was half the size (roughly 6K bytes), and ran twice as fast on a single 8086 chip as the same language running on the Alto.

Being carefully winnowed to fit in the Notetaker with no virtual memory, Smalltalk-78 is a satisfyingly small kernel for a complete personal computing system. We are fortunate that, out of the 121 Alto disk packs that were saved, one happened to be the disk that wrote the object table for porting the Smalltalk-76 codebase to the NoteTaker. Working from another Smalltalk virtual machine in JavaScript, we were able to bring that system back to life as a running Smalltalk-78.

We actually did more. Rather than simply bring Smalltalk-78 back to life, Bert Freudenberg and I, together with Ted and Yoshiki Oshima, spent several weeks doing some of the further development that we never had the opportunity to do on the NoteTaker, to the point that this Smalltalk of 40 years ago became an interesting system for demonstrating the power of Smalltalk. The video at https://www.youtube.com/watch?v=AnrlSqtpOkw shows a presentation that would have turned many heads in 1978.

### 6.4 The Impact of Smalltalk-78

Smalltalk-78, like Smalltalk-74, was mainly important as a transitional system. To make it work, we had bundled up the best of Smalltalk-76 in a small and portable package, and made it runable on a single-chip microprocessor. In the process, the non-Smalltalk component of the whole system had shrunk from around 30K bytes of assembly code and microcode to just 6K bytes of 8086 code. Ironically, our little bundle was soon to run on what was arguably the fastest personal computer of the 1970s. Let's pause to review our story. Smalltalk-72 had launched us into a new way of thinking and building. The implementation was strictly "do what it takes," but the effect on us as researchers was profound. Out of that experience came the clean and powerful design of Smalltalk-76, but with two complexities remaining from earlier systems: the pre-BitBlt "do what it takes" graphics code, and the OOZE object memory required to work within the confines of the Alto. We have also seen how the challenge of porting to the NoteTaker led to a compact resident object memory and a simplified graphics system with the modern BitBlt graphics model.

### 6.5 Along Came the Dorado

In the late 1970s, while we were struggling with speed and space on the Intel 8086 microprocessor, another project at Xerox was putting the finishing touches on a machine with more speed and space than we had dreamed of. Dubbed the "Dorado," this machine boasted up to 16 megabytes of main memory and a 60 nanosecond processor cycle time. It was bulky for a personal computer (think small refrigerator), but we knew that it was only a matter of time before we would all have machines of this power on our desks, and then on our laps. We tweaked the NoteTaker memory exporter to revert to the simpler Context architecture of Smalltalk-76, but to retain the new BitBlt graphics kernel, and wrote a new object table file. Peter Deutsch ported the Smalltalk-76 virtual machine to Dorado microcode.

Smalltalk-76 on the Dorado was astounding to us: it ran Smalltalk at a million bytecodes per second. Coupled with our integrated development environment, this Smalltalk was dazzling to everyone who saw it. Not just children, but serious developers, including banks, AI companies, and publishers. Among the outsiders who saw Smalltalk was Steve jobs, who negotiated with the Xerox Venture Group for a possible Xerox-Apple relationship. I, along with Larry Tesler and Adele, had the pleasure of entertaining Steve with a demonstration of Smalltalk. What he saw was Smalltalk-76 running on the Dorado, which is very similar to the Smalltalk-78 emulation shown in Figure 26, because of the large screen and the execution speed.

### 6.6 The Jobs Demo

For my part of the demo, I began by evaluating the following:

```
Rectangle fromuser comp.
```

This prompts the user to select a rectangle, which it then complements (i.e., turns the black pixels to white, and the white pixels to black). I did this several times in an empty area, as shown in Figure 27, making the point that this is a system in which computing and play—even art—can be freely mixed. In the midst of this doodling, I arranged to overlap a couple of rectangles with a small offset, as shown in the upper center of Figure 27. We noted that complementing a region twice, with an offset, provides a simple way to outline a region on the screen. This is invention in a system that is designed for unanticipated creations.

I talked about how we display the selection in a block of text by complementing the selected text (changing it from black to white), and navigated in the browser to the text shown in Figure 28. I explained some of the details: how it complements the first partial line, then a block of internal lines, and finally the last partial line, in each case sending the message `compRect:` to do the job. All the while, I was selecting code to indicate the ranges, and I made the observation that some executives and their staff at Xerox were uncomfortable when large areas of the screen turn black, as they do when you make a large selection. Might it be better to outline the region instead? We pondered. How hard would it be to do that?

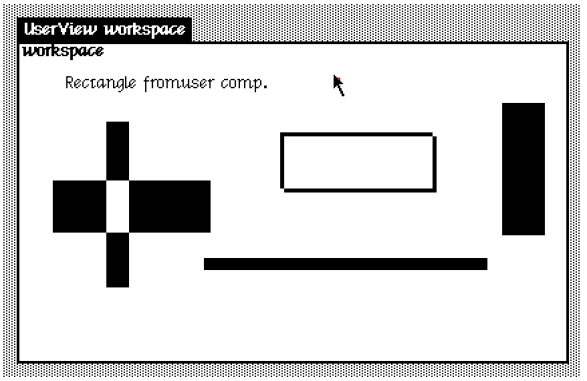


Fig. 27. Executing `Rectangle fromuser comp.`

I browsed to the neighboring method, `compRect:` shown in the lower left frame of Figure 28, and added the two lines shown in the lower right. As soon as I accepted the change, in that method and throughout the system, all selections in text appeared with an outline rather than as a black rectangle.

I had prepared this demonstration, as I used it frequently to show off the Smalltalk-76 debugging facilities. (In fact, I would sometimes hit the interrupt key in the process of making a selection, and

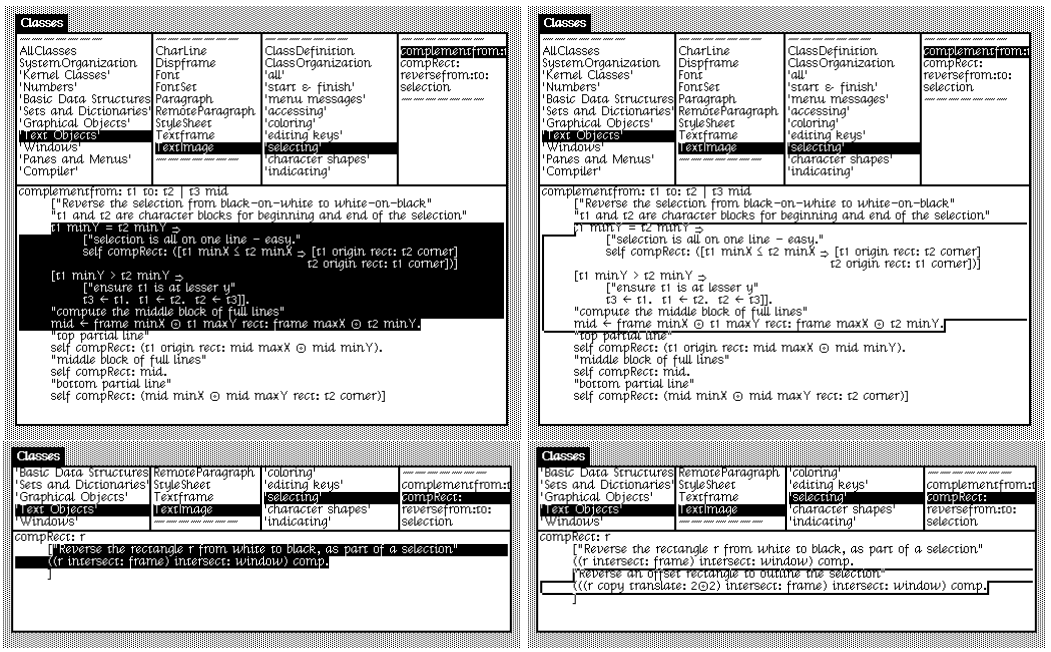


Fig. 28. Creating large black areas was found to be “scary” to non-technical computer users. As an experiment to solve this problem, a simple change in Smalltalk-76 causes all text selections to appear outlined.

make the change in the debugger.) However, what followed was unanticipated. When we viewed a large method in the browser and scrolled it up or down, the text moved jerkily, line-by-line. You can see this if you run the simulation. Steve wanted to see the text scroll smoothly. I said "I'll have to make a couple of changes, but I'll show you after lunch," since the suits were about to step out anyway. I spent my lunchtime rearranging a couple of methods, which remained in the system thenceforth. When the executives returned, I was able to change one line of `TextImage>scrollUp:` from

```
[self scrollby: delta / self lineheight]
```

to

```
[self scrollby: delta asFloat / self lineheight]
```

With this one change, the integer division became a floating point division, thus preserving the fractional part, and scrolling became silky smooth. Steve liked it.

Looking back from today, it is easy to miss the impact of these catchy demos. The mouse-and-menu driven user interface, running on a large paper-white screen with attractive bold and italic fonts, was already impressive to many. More amazing still was that the system itself could be explored without even touching a keyboard.[22] Finally, we showed that what we were navigating was not merely a passive view, but the actual live system, and that it was instantly responsive to the designer's or user's desires. This seemed like magic; everyone who saw it came away wanting it.

## 7 SMALLTALK-80

The Dorado experience showed us that Smalltalk had serious commercial possibilities, and the NoteTaker experience proved that microprocessors could support a system like Smalltalk. We had had the luxury of playing with computers of the future: now that future was becoming reality. Aware of a huge potential audience for our work, the group charter shifted for a time from education, graphics and music to building a product for public release. This public, sharable version of Smalltalk became known as Smalltalk-80.

The advance of technology was not just affecting big business: it was bringing real computer science into contact with the realm of clever hobbyists all over the world. Magazines such as *Creative Computing*, *Dr. Dobb's Journal*, and *Byte* nourished this burgeoning community of home computer enthusiasts. We shared the enthusiasm of that community and were eager to make Smalltalk accessible to those people.

In August of 1978, *Byte* magazine had published an issue devoted to the language Pascal, as part of a tradition that August issues generally featured issue-wide attention to a single computer language. *Byte* was arguably the leading publication for personal computing; other languages honored by this tradition included APL, Forth, and Lisp.

The Pascal issue of *Byte* affected us—especially me—in a visceral manner because of its cover. There, Robert Tinney's artwork depicted a fanciful scene showing "Pascal's triangle," as well as a number of other languages in a "sea" of programming languages. Smalltalk was depicted on an island, isolated from the rest of the world. In the words of Carl Helmers' "About the Cover" note:

[22] It was often after five minutes of opening windows, browsing code, executing and printing expressions, and finally introducing a change, that I would point out "this is the first time I have used the keyboard."

*Travelling upward (in the picture) through heavy seas we come to the pinnacle, a snow white island rising like an ivory tower out of the surrounding shark infested waters. Here we find the fantastic kingdom of small talk, where great and magical things happen. But alas, just as the impenetrable fog bank around the jungles of LISP hide it from our view, the craggy aloofness of the kingdom of small talk keeps it out of the mainstream of things.*

Nothing was more important to some of us than to get off that island and out into the world. A public release implied a number of changes. Among other things, we had to document the entire system and work toward getting Smalltalk accepted as a standard. Adele and Dave led the documentation and publication effort [Goldberg 1983; Goldberg and Robson 1983], and all of us contributed to what seemed to be the natural evolution of the language into a system suitable for a product.


Fig. 29. The cover of the August 1978 issue of *Byte* magazine. (Artwork by Robert Tinney.)

The work toward a "standard" was not done through professional societies, as had historically been the case for other languages. Rather, with the help of Bert Sutherland (to whom our group reported in 1979), Adele initiated a multi-corporation project to create VMs on multiple machines that would all run the same virtual image. That we managed to get cooperation from Apple, Tektronix, HP, and DEC, as well as several universities, guaranteed a documented working standard for the VM. This initiative also produced experience papers in the "Green Book" [Krasner 1983], and built a significant early community of implementors and users.

### 7.1 Drop the Special Characters and Go for ASCII

The move to ASCII went fairly smoothly, except for the loss of left arrow for assignment (for a while we were stuck with underbar), and the loss of an infix operator for indexing collections. I was not happy with those changes, but at some point we had to make a decision regardless of whether it was perfect or not. Sadly, with all the stress about left-arrow *vs* underscore (the ASCII character code suborned by Smalltalk-76 for a special ← glyph) for assignment, an important baby was thrown out with the bathwater: Smalltalk-80 dropped the terminal left-arrow keyword construct (see Section 5.2). With hindsight, the := combination now used for assignment could have replaced

the terminal left arrow keyword, and `!` or `@` could have replaced our • operator for indexing (`@` is even read as "at").

The example of copying an element from one array to another serves to illustrate these choices:

```
In Smalltalk-76
    a • i ←  b • i
In Smalltalk-80
    a at: i put: (b at: i)
Smalltalk-80-didnahappen
    a @ i := b @ i
or
    a ! i := b ! i
```

A prime example of the damage, `at:` and `at:put:` robbed subscripting of its former algebraic expressiveness. Moreover, without the assignment-level precedence of terminal left-arrow, `at:put:` requires parentheses arround the `put:` argument.

### 7.2 Change the Evaluation Order to Left-to-Right

Smalltalk-76 required message arguments to be evaluated before the receiver, so that the receiver would be on top of the stack at the time of the send operation. This post-evaluation of receivers made the send operation very simple—with both receiver and message at hand, looking up the method was straightforward. Unfortunately, this violated the otherwise-intuitive rule that Smalltalk executed from left to right; this bothered me.[23] But if the receiver were not on top, then how could it be found? During the years of Smalltalk-76, I had time to reflect on this problem and, as part of the changes to Smalltalk-80, I adopted the approach of encoding the number of arguments in the "send" bytecode. This meant that the evaluation order was left-to-right (as one might expect), and yet the receiver could be found even though it was not on the top of the stack.

### 7.3 Introduce Boolean Classes and More-Conventional Conditionals

From the beginning in Smalltalk-72, we had adopted the Lisp-like conflation of `nil` (in Smalltalk, the undefined object) with the Boolean notion of falsehood. In addition, the earlier pattern for conditionals was somewhat confusing, because of the asymmetry between the `true` and `false` branches. If we wanted Smalltalk to find acceptance in the world, it seemed that the old conditional pattern was a non-starter. I took on this problem, and was concerned at first with the likely need for special primitives or other ugliness. If `true` and `false` were to both be instances of Boolean, then they seemed to requires some circular logic inside them to test whether they were true or false.... Suddenly it occurred to me to make them instances of *distinct subclasses* of `Boolean`. By giving each of `true` and `false` their own class, the very mechanism of message dispatch does exactly what is needed, with no need for circular primitives. Here are the definitions of `ifTrue:ifFalse:` in the three classes; For convenience, Smalltalk also defines `ifFalse:ifTrue:`, as well as the single-armed conditionals `ifFalse:` and `ifTrue:`.

[23] This counterintuitive order of execution was almost never noticed, and was even less often a source of error. However I do recall a case where a 16-bit value was being read from a stream of bytes as `(strm next * 256) + strm next`. The high-order byte came first in the stream, but here the right argument to `+` was evaluated first, and so it was the second byte from the stream that was multiplied by `256`. Notwithstanding that this should do what one might expect in reading left to right, I hope the reader will agree that it is no way to write maintainable code.

```
Boolean
    ifTrue: trueBlock ifFalse: falseBlock
        ↑ self subclassResponsibility
True
    ifTrue: trueBlock ifFalse: falseBlock
        ↑ trueBlock value
False
    ifTrue: trueBlock ifFalse: falseBlock
        ↑ falseBlock value
```

### 7.4 Come up with a Better Form of Blocks

The blocks of Smalltalk-76 did the job required for simple control structures, but the open-colon syntax was subtle, and heaven forbid that we need another ASCII replacement for open colon! There were substantive problems as well. In Smalltalk-76, because the creation of a remote context for a block was associated with the open-colon keyword, there was no way to pass around a block as the argument of a normal send. This would be necessary, if, for example, one wanted to add the areas of every rectangle in a collection, and have the same code work for different kinds of collections. The solution was to introduce Smalltalk-80's bracket syntax for blocks. This created essentially the same remoteContext object as the open-colon, but as an explicit literal, rather than as a side-effect of the kind of keyword that preceded it. Such a block literal can be sent as an argument with a normal message. The block is not evaluated until it is sent the message `value`, (or `value:` if the block requires an argument, `value:value:` if it requires two arguments, and so on).

Figure 30 shows an example taken from the `RunArray` class. The "runs" are spans of the array for which the value is constant. This makes it useful for compactly storing, e.g., the emphasis attributes in rich text. The `fontsUsed` method calls `withStartStopAndValueDo:` to scan a text for all its font changes. Without the client needing to know how `RunArrays` work, the block protocol makes it easy to scan for changes in emphasis as well as the actual location of each change. In this example, only the fonts are of interest, but the same method could be used to find where a given font is used, or to locate an embedded object (also stored in the emphasis runs).

### 7.5 Support Class-Side Methods

For quite a while, Adele had been lobbying to introduce "metaclasses," that is, to make classes instances of other classes, so that each class would be able to have its own methods. She had good reasons for this, the main one being to minimize the likelihood that users would have to deal with uninitialized instances. At the time, `Point new` produced a point object with `nil` values for `x` and `y`. One could have every class interpret new as `↑self basicNew initialize`, with `initialize` installing default values. However, this would either make all sends of `new` run slower, or all performance-critical code would have to be rewritten to use, for example, `basicNew` instead of `new`. The situation of uninitialized instances could easily lead to errors in code, and Adele had borne the brunt of dealing with such errors in her experience introducing new users to Smalltalk.

I must confess to having been resistant on this issue, as metaclasses seemed to add a significant level of complexity. This was a juncture at which we might have moved from a class language to a prototype language, since working from a prototype gives you exactly this kind of pre-initialization, as well as offering some related benefits in revealing more of the type relationships inherent in the system. However, the prototype revolution was too much for us to confront, especially on the

```
RunArray >> withStartStopAndValueDo: aBlock
    "Evaluate aBlock with start, stop, and value for each 'run'"
    | start stop |
    start := 1.
    runs with: values do:
        [:len : val | "Will be called with length and value for each range"
        stop := start + len – 1.
        "Now call the original block with the start, stop and value"
        aBlock value: start value: stop value: val.
        start := stop + 1]
Text >> fontsUsed
    "return the list of fonts used in me"
    | fonts |
    fonts := IdentitySet new.
    runs withStartStopAndValueDo:  "Enumerate every run of emphasis"
        [ :start :stop :attribs |
        "attribs is the set of all attributes in this run"
        attribs do:
            [ :attrib | attrib isFontSpec ifTrue:
                [fonts add: attrib font]]].
    ↑fonts
```

Fig. 30. The upper method, `withStartStopAndValueDo:`, in class `RunArray`, handles the details of navigating the array. The client method, `fontsUsed` (in class `Text`) has all the hard work done for it, so that it is simple to scan for all fonts used in a rich text instance.

eve of going public with what was already a fairly mature and capable system. It should be noted that prototypes did not seem overly daunting to a junior faculty member at Stanford named Dave Ungar. In collaboration with Randall Smith, his devotion to the prototype approach gave us the language Self [Ungar and Smith 1987].

When I finally agreed to put in metaclasses, I was delighted to find that I could do so with no change to the virtual machine. The key was to realize how metaclasses fit into the existing class hierarchy. Figures 31 and 32 illustrate the change that was needed for a small sample of the system. I wrote a snippet of code that ran through the system changing all the pointers in the class hierarchy and, after a few crashes, it all worked and I saved the result.

The next day Adele arrived to find a picture of the metaclass hierarchy (like that in Figure 32) on her door, and a Smalltalk complete with metaclasses on the server.

Because all the metaclasses inherit from `Class`, they all behave like classes. But because they are distinct, every class can have additional methods, meaning that it can have its own behavior. This is very useful for initialization, as well as other, class-specific, utility functions. Here are a few examples of expressions that benefit from the presence of metaclasses.

`Point x: 10 y: 20`—create a point with cartesian coordinates
`Point r: 100 theta: 90`—create a point with polar coordinates
`Number degreesToRadians: 90`—perform a useful conversion
`Rectangle enclosingPoints: myPolygon vertices`—create a rectangle from several points

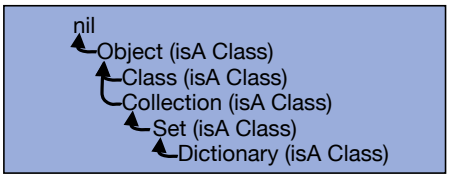


Fig. 31. A fragment of the class hierarchy before the addition of metaclasses. The black arrows show the subclass relationship.

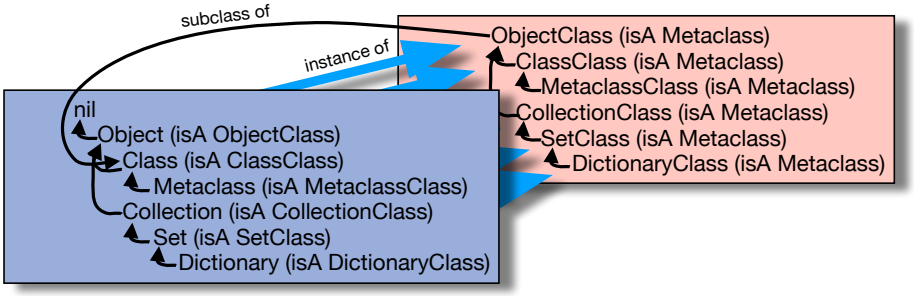


Fig. 32. After the addition of metaclasses, each class is now an instance of a new class—its metaclass. The metaclasses themselves form a new inheritance hierarchy that parallels that of their instances, except at the top. There, class `Object` is a subclass of `nil`, while class `ObjectClass` is a subclass of `Class`.

### 7.6 Improved Performance

As we looked toward more academic and industrial markets for Smalltalk, we sought to improve the performance of our systems. The original design of the interpreter, as specified by `Context step`, already provided ways to achieve decent performance of simple primitive operations such as arithmetic and array access. As we gained experience with this architecture, we learned other tricks that allowed us to improve performance of much of the other non-primitive code as well. The most significant case involved the use of a method cache indexed by message selector and receiver class. Once the interpreter had gone through the rather lengthy process of looking up a selector in one or more classes in an inheritance chain, the selector, receiver class, and method could be installed in a hash table. Once this was done, the interpreter could resolve the next such message by a single query in that table. Even a relatively small cache of 500 entries would typically bypass 90 percent of the message lookups. Further benefits can be achieved by including in the cache some of the other information encoded in the method, such as primitive index and number of arguments, that would otherwise require further decoding.

The Smalltalk virtual machine was a tinkerer's dream come true. In addition to the method lookup cache, considerable improvements could be made in the area of storage management, especially with regard to the allocation, initialization and release of Contexts. Then there were tricks to play with bytecode execution. For instance, one could maintain several different jump tables for dispatching on the various bytecodes, depending on whether the top of stack were known to be `true` or `false`, or, say, a small integer. Appendix E presents more discussion on performance.

### 7.7 Model-View-Controller

Nothing helps the success of a newly released programming environment more than a set of useful widgets and a paradigm for users to carry through the design of their own widgets according to the same pattern. The Model-View-Controller (MVC) paradigm [Lalonde and Pugh 1991, Ch. 2], developed by Trygve Reenskaug [1979], and later implemented in Smalltalk by Adele Goldberg, Jim Althoff, and others [Goldberg 1983], filled this vital need in the Smalltalk-80 release. The coordination of multiple views of a single model had been around in graphics for some time already, but the pattern that came forth in Smalltalk-80 was applicable throughout the universe of user-interface design, and answered the big question of "How do I get started building something useful in this new system?"

### 7.8 Enumeration and Mutation

Smalltalk classes retain a collection of their instances only when designed to do so (as with open files and windows on the screen). However from Smalltalk-76 on, the virtual machines have supported a method that will enumerate all instances of any class. This enumeration enables gathering of interesting statistics, and in Smalltalk-76 it was valuable in tracking down memory "leaks" when reference counting might fail to reclaim a circular structure.

When we left OOZE behind, Smalltalk-78 and Smalltalk-80 were designed around object tables that were indexed directly rather than hashed. This made enumeration trivially simple, and it also opened the possibility of mutation. The support for mutation that we put into Smalltalk-80 was simply to reverse the object table entries with the message `become:`. Thus one could add a new variable, say `borderColor`, to an existing `ProjectWindow` class. Once the new class was created, the old instances could be enumerated, copied into new instances with `nil` or some default value for `borderColor`, and then **`become:`** the new instance. The primitive method `become:` exchanged the object table entries for the old and new instances. By this means, every `projectWindow` instance in the system would become an instance of the new class with whatever initialization might have been specified. This facility added further to the malleability of Smalltalk itself, and to its ability to support exploratory development.

We would sometimes show off this capability by halting the system and then, in the on-screen debugger, add an `area` instance variable to class `Rectangle`. We would then open an inspector on a pane of the debugger to see that in fact all its rectangles now included that extra property, as did all the others in the system as we continued with the demonstration.

### 7.9 Robustness

We were always happy with the robustness of the earlier Smalltalks, but Smalltalk-80 definitely felt more solid than the earlier OOZE-based systems simply because its in-memory architecture was free of virtual memory delays. Only a few problems were ever likely to occur in Smalltalk-80:

- **message not understood**: This was always caught, and diagnosable by a debug window
- **failure of a primitive method**: Usually caught, and diagnosable by a debug window
- **filling memory due to failure of reclamation by reference counts**: Smalltalk is able to enumerate all instances of any class, and this enabled various tools that would help to diagnose such memory "leaks"

- **infinite loop filling memory**: This would typically fill memory very quickly, so the interpreter was equipped with safeguard heuristics that would almost always announce "I think you may be in an infinite loop" in a debug window.

The "feel" of these message systems was simply more forgiving than many other systems of the era. You were never confronted by meaningless bits when an error occurred—everything in memory is an object, and every object knows how to present itself in a meaningful way.

I will always remember a debug emergency call I received during one of our student classes. A girl was experimenting with numbers in Smalltalk-80, and suddenly her system froze when she tried to reframe a window. Although the UI was unresponsive, the "emergency evaluator" window was still operating so I was able to take a look at what had happened. Her window was trying to display itself, but was encountering an error because BitBlt, responsible for painting the border, had a width that was…wait a minute…a `Fraction`? We did not even *have* fractions in the student system, but she said, "Oh yes, I added a `Fraction` class."

Interestingly, BitBlt was written to be resilient, in that if it received a non-integer argument, it would call itself again after sending the message `asInteger` to that argument. However, her fractions did not have such a method, so the debugger had stopped at that point and was not even able to show the problem because of another such infraction (er, sorry). I asked whether she had a conversion to integer and she replied no, but that her fractions did have a `makeFloat` message. We talked about the problem, and I got her to suggest defining

```
Fraction understands: 'asInteger' as:  '↑ self makeFloat asInteger'
```

We were able to type this into the emergency evaluator, at which point the debugger miraculously displayed itself, the window she had tried to reframe reappeared, and everything in the system seemed to work again. Probably hundreds of methods in the system were now operating just fine with rectangles whose coordinates were instances of a student's newly defined `Fraction` class. This illustrates the astounding ability of message-sending systems to absorb unanticipated constructions.

### 7.10 Impact of Smalltalk-80

Smalltalk-80 is not far linguistically from Smalltalk-76, and its virtual machine architecture is nearly identical. This was good because it required all of our energy to carry out the character set changes, the documentation, and the preparation of a reference interpreter to guide developers in their own implementations at other companies. To seed the dissemination process, Xerox also granted royalty-free licenses to a kernel memory image full of development tools to four interested companies: Tektronix, DEC, Apple, and HP.

In celebration of the public release of Smalltalk-80, the August 1981 issue of Byte Magazine was devoted to Smalltalk, and featured on the cover a valiant Smalltalk crew finally leaving its ivory tower in a hot air balloon, shown in Figure 33. That issue, available online as https://archive.org/details/byte-magazine-1981-08, remedied the former aloofness by publishing twelve articles about Smalltalk-80 by those of us then working on Smalltalk at Xerox.

The release of Smalltalk-80 with accompanying books and documentation, and the articles in Byte Magazine, arguably started the Object-Oriented Programming revolution of 1980–1990. Directly or indirectly, they introduced the world to OOP. Outside of our group at PARC it inspired numerous individuals and organizations to build Smalltalk implementations or design other OOP

languages. Some of these Smalltalks were directly derived from licensed versions of Smalltalk-80 while others were created from scratch based upon the released documentation.

Dozens of organizations and hundreds of people spent the next 15 years establishing Smalltalk as a serious computer language. It is not possible in such a short section to adequately cover all that work. Dave Thomas's article "Celebrating 25 Years of Smalltalk" [Thomas 1995] is an excellent review of that history up to 1995.

### 7.11 About the Balloon

As soon as we had the agreement that Byte would cover Smalltalk in their August 1981 issue (another great achievement by Adele), I knew what I wanted on the cover.

My favorite adventure story from childhood was *The Mysterious Island* by Jules Verne. In this story, Cyrus Harding, a Union engineer held prisoner during the American Civil War, escapes from behind the enemy lines by commandeering an observation balloon, along with four fellow prisoners and a dog. Their plan goes awry when they are blown out to sea in a storm and nearly perish, landing finally on a deserted island somewhere in the Atlantic ocean. Through ingenuity and a bit of luck, Harding manages to recreate most of the accomplishments and comforts of civilization (with help from his friends, an amiable orangutan, and an unseen benefactor). Harding thus became my childhood standard for resourcefulness, and the conviction that anything can be accomplished if you are clever and persistent.


Fig. 33. The cover of the August 1981 issue of *Byte* magazine. (Artwork by Robert Tinney.)

The island connection made me think of a hot air balloon as the ideal vehicle for Smalltalk's release. When the opportunity came to design the cover for the Byte issue on Smalltalk, I made the suggestion of a balloon lifting off from the island shown on the Pascal cover, and everyone liked it. Robert Tinney executed the colorful graphic that we know so well, and the rest is history.

There is a coincidence in this particular literary allusion. It is revealed at the end of *The Mysterious Island* that the "unseen benefactor" is actually Captain Nemo, a childhood hero of Alan Kay.

### 7.12 After Xerox

Larry Tesler left Xerox for Apple in 1980, not long after the Jobs demo. In 1979, Alan took a sabbatical, and in 1981 became Chief Scientist at Atari. In 1984, I left Xerox and joined Apple, hoping to leverage Apple's existing nonrestrictive license to put Smalltalk on the Macintosh, and hence out into the world. I worked for several years on this, and released several versions of Smalltalk-80 that were available for free through the Apple Programmers and Developers Association (APDA). The next year (1988), Adele left, with Xerox's blessing, to form ParcPlace Systems, with the goal of making a commercial Smalltalk business.

The APDA Smalltalk-80 began as something of a toy, since we were running on relatively slow Macintosh and Lisa computers. However, an Apple Fellow, Rich Page, had a project to build a "Big Mac" which was a Motorola 68020 processor with a 1024×768-bit display. He liked what we were doing with Smalltalk, and he delivered a couple of early prototypes to us in 1985. I will never forget the impact of that machine on our group. As I've said before, Smalltalk in the early days was ahead of the computers that ran it. With the bigger screen and just that much extra processing power, Smalltalk-80 become like clay to a sculptor. It seemed that anything was possible, and the only thing slowing you down was the time it took to type in your ideas. Scott Wallace had joined my project, and was just learning Smalltalk when our first "Big Mac" arrived. Scott took it home, and I don't think he slept at all that week. Scott was a perfect match for Smalltalk. He would work for a while on some problem, and then when he was done, he would build all the tools he had wanted during that sprint. Smalltalk returned the favor to Scott by making it just as easy to change the system as to work on any other problem. In those first few weeks, Scott kept making our Smalltalk smoother and easier to use, while we were also improving its speed.

Xerox had made two public releases of Smalltalk-80, "Version 1" and "Version 2." Version 1 had a limited distribution—to Apple, DEC, HP and Tektronix—but allowed these companies unrestricted redistribution of any system they built. It was intended that way, to encourage the spread of Smalltalk. Version 2 had restrictions on its licensing, and fortunately Apple never signed on with Version 2. There was not a lot of difference that I can recall between the two versions, except that Version 2 had a spelling corrector that I had worked on earlier at Xerox, as well as a type inference system [Borning and Ingalls 1982c] and a multiple inheritance system [Borning and Ingalls 1982a,b] that Alan Borning and I had written. When I joined Apple and took over the Smalltalk work, Mark Lentczner worked with me for a while, and I had him build a spelling corrector. Meanwhile, I worked on redesigning the UI so that it could be more-easily used with Apple's one-button mouse, and with a tablet which I hoped would come soon (this was 1985).

I had a lot of fun developing the APDA Smalltalk and sharing it with the Apple developer community. As soon as it was stable, I used it to build a visual programming environment called Fabrik [Ingalls et al. 1988; Ludolph et al. 1988]. Fabrik mingled the metaphors of GUI-builder and dataflow, and a pen interface with pie menus for commands and wiring operations. It was possible to build a working file browser in five minutes, and I thought it would be a great tool for hand-held tablets. We published that work in 1987, and then I was drafted to take over a family business in the mountains of Virginia where there was no interest in computers, let alone in object-oriented programming.[24]

[24]I will always remember being at an Apple conference, walking between buildings, when John Sculley called over to give me a ride in his limo. He said: "I hear you're leaving us to go run *The Homestead*" (John had just attended a meeting of the Business Council there) and I said, "Yes, sir." He said, "Oh, don't do that—we'll find someone else to run *The Homestead* for you. You should stay here." With hindsight it probably would have been better all around if I had.

## 8 SQUEAK

It was 1995, when I returned from my hiatus in Virginia. I spent the next at Interval Research where I helped Glenn Edens's group to build a Smalltalk system by porting the APDA Smalltalk to their hardware. At the time I was also talking with Alan Kay and his group at Apple who were frustrated by the tools at their disposal and feeling the need for an expressive programming environment where they could control everything—like Smalltalk, only more so.

I rejoined Alan's group at the end of 1995 and thought that we could do another port of the APDA Smalltalk as a start, since it had no restrictions on IP or distribution rights. However the APDA VM was unusable Motorola 68000 assembly code, and the idea of writing and debugging another virtual machine in C was daunting. We were a team of four and we wanted something quickly. During my absence in Virginia, it seemed like nothing had changed much in computing except that everything ran much faster, had more memory and was getting cheaper. I remember driving to Apple in the rain one day, stopped at a light on Page Mill Road, and thinking, "OMG, we could run the reference interpreter and then translate it to C." I had actually worked a little on the reference interpreter in 1983, although Dave Robson had done most of the work. It was published in the "Blue Book" and freely available. With the free APDA image, and the free reference interpreter, all we had to do was to start typing and debug it into existence.

### 8.1 Poof

We didn't even have to do that. It turned out that the reference interpreter had already been typed in by students in a class taught by Mario Wolczko, so in a matter of days we had it running and loaded up with the APDA Smalltalk image. This ran slowly, entirely simulated on a commercial Smalltalk (VisualWorks), but it was actually fast enough to get work done. The reference interpreter was written in a style where the types of all variables are known; a subset of Smalltalk that is easy for any programmer to translate to C or assembly code because it does not use the polymorphic generality of message sends. We now call this subset "Slang." Knowing this, I knew that we could write a translator from Smalltalk to generate a version of the reference interpreter in C. That version would certainly be usable and would be easy to work on, optimize, and extend in many ways. This was the beginning of Squeak.

John Maloney set to work on a translator from Slang to compilable C code. Since Smalltalk already had a parse tree system (see Appendix D) capable of pretty-printing all the code in the system, it did not take long to ugly-print it to produce executable C code.

Scott Wallace simplified and reworked the file system. In a system where the source code is online all the time, it at first seemed as though we might have to give up on our pleasant live development style for a while until the file system was working. However, with Squeak running live we were able to rewrite the source code access to pull the entire source file into memory, and thus continue with full read/write access to the system sources while the file system was being worked on. This took only a day to complete, and it remains a handy trick when porting as with the Mitsubishi M32R/D experiment (see Appendix E) or otherwise running without full operating system support.

Ted Kaehler and I tackled the memory system. We had already noted that the move to 32-bit systems was taking the "small" out of Smalltalk, so we committed to doing a 32-bit system. For faster speed and more efficient use of space, we opted to do away with the object table approach and use direct pointers. In spite of the use of 32-bit pointers, we worked hard to keep the system

as compact as possible. A simple approach would be to allocate at least 3 words of object header (class, length, and storage management bits (including object hash)), but we arranged that up to 32 classes could be declared to be "compact" and thus over 80 percent of the objects required only a single word of overhead on each instance.

For garbage collection, we started with the two-generation garbage collection (GC) scheme of the APDA Smalltalk, inspired by Dave Ungar's SOAR [Samples et al. 1986] memory system. By noting stores into "old" objects, it became easy to trace all new objects, so incremental GC was very quick (under 25 ms even for machines of the 1980s). A pleasant surprise was that we were able to perform compaction as part of the incremental sweep, thus eliminating all the overhead and logic that had been required to maintain free lists for the reclaimed objects in our earlier design.

The one remaining challenge for the memory system was how to support the `become:` operation required for mutating instances of a class being reshaped. With direct pointers, the simple approach would require a scan of the entire object memory for each object pointer being changed. To begin with, we removed all uses of `become:` that were not really necessary in the base system, such as growing collections. This left only the need to mutate instances when a class has been reshaped. While several subtle approaches have been used on other Smalltalks, we opted for a simple variant, which was to scan the whole of memory, but to mutate an entire array of pointers in that one scan. This handled the dominant use of `become:`, and it was easily adapted to any other uses by aggregating the candidates into an array.

None of the four of us was aware of the details of the various high-performance VMs in existence at the time. Our goal was to pull off something clever in the very simplest way possible—so that we could actually do it, so that it would be easy for others to use and to port, and so that it would be easy for us to extend to quality graphics and sound and other special behavior that we might encounter in building interesting educational artifacts.

In only seven weeks, the entire Smalltalk was working, albeit slowly, and in the tenth week John's translator finally produced a virtual machine in C that ran. It ran 20 times as fast as our simulated lash-up, and Squeak became entirely self-supporting. More details can be found in the OOPSLA paper describing the Squeak Smalltalk design and experience [Ingalls et al. 1997].

The performance of Squeak was not stellar, but it was no toy either. We knew that it could be improved (as it has been), and its 32-bit design has stood the test of time in a number of serious applications. Even in the first release of Squeak, real-time animation, graphics and music ran without hiccups on machines of modest power. Most important for us were its simplicity, malleability and small footprint.

The malleability of Squeak was what really gratified us. The strategy of simulating a Smalltalk VM and then translating it to a low-level language left us with one of the most productive research environments we know of. We never would have tackled some of the technical challenges in Squeak if we had not been able to write and debug them all in Smalltalk, and then "push a button" to release them into the production interpreter. Among these otherwise-daunting challenges were:

- an enhanced BitBlt operation that included scaling, rotation and anti-aliasing,
- the ability to save the entire state and restore it on machines of different word size and endianness,
- large integer numerical support,
- real time music synthesis, and Fourier analysis of sound,
- the ability to instantly mutate all objects of a class in place,

- the ability to extract and save projects as "image segments" (see Section 8.8),
- a footprint for an entire IDE that is under 1 megabyte, and
- support for 1-, 2-, 4-, 8-, 16-, and 32-bit color depths.

Every one of these challenges answered one need or another from real users, and supported our goal of producing an exploratory software system for children of all ages.

### 8.2 The Beginnings of the Morphic User Interface

John Maloney joined Alan's group just before my rejoining in December of 1995. John had just come from Sun Microsystems where he had worked with Randy Smith and Dave Ungar on the Morphic interface for Self. Soon after John was done with the Slang translator and Squeak was starting to run well, he put together a demo of Morphic for Squeak that enchanted us with its simplicity. Morphic is about the simplest way you can compose objects and organize their behavior on a computer screen:

- `Morph` is a superclass of graphical objects; it has methods that let those objects respond to events and display themselves.
- All morphs have `bounds` (rectangles on the screen), and can have submorphs.
- The `world` (the screen or page) is itself a morph. This gives you the basic model for a scene graph displayed on the screen.
- A `hand` object, shown as the cursor (and also part of the world), can pick up other objects and drop them elsewhere, thus effecting not only motion but also changes in the structure of the scene graph. The hand is also the source of user event messages such as `mouseDown:` and `mouseMove:`.
- User events such as `mouseDown:` are passed as messages from the screen, down through the scene graph, to the frontmost morph that contains the location of the event.
- Morphs typically define methods that respond to user input events like `mouseDown:`, or to the passage of time through a `tick` message from the world.
- When a morph's position or appearance changes, a `changed` message causes the Morphic display system to update the screen efficiently, and without flashing.
- Both normal screen changes and animations are handled by a simple iterative kernel:

    forever do:
        detect and dispatch user events, such as `mouseDown:`, `mouseMove:`
        run `step` methods defined for any morphs,
        compute the affected screen area, and update it using double buffering.

John fleshed out the implementation in a very small number of classes, and we began to build a whole new graphical environment for Squeak. I worked with John to get this new world working with Squeak's evolving BitBlt primitive, especially BitBlt's ability to scale and rotate images. I also worked on a number of morphs such as polygons, curves, and the full "run-around" text package.

The combination of Morphic's stepping and screen update mechanism provided "liveness": the ability for multiple morphs to be active simultaneously, as opposed to the single active window supported by MVC at the time. The liveness of Morphic was made possible by the increase in processing and graphics speeds since the Alto.

### 8.3 MVC and Morphic

The Smalltalk codebase that first came alive as Squeak was a direct descendant of the Version 1 Smalltalk-80 release from Xerox and ParcPlace, as modified and distributed through the Apple Programmers and Developers Association. Its user interface comprised the normal Smalltalk-80 MVC components of windows, menus, browser, editor and debugger.

A significant part of the early Squeak community were using—and liked using—the MVC framework. This was because of its relevance to their "day jobs," where they used commercial Smalltalks that were running normal Smalltalk-80 systems with an MVC architecture. The main interest in Squeak was its free and open-source license, its great self-supporting tooling, and its radical portability. The Squeak community was not large, and we welcomed all Smalltalk-80 users to discussions and contributions of code.

To keep open a bridge between Squeak and other Smalltalks, for a number of years I maintained a compatibility package, so that most MVC programming tools would work in Squeak. This effort at synergy not only increased the energy in our project, but it also infused new energy into the Smalltalk community at large, at a time when commercial Smalltalk endeavors were falling out of favor.

### 8.4 The Comfort of Smalltalk and the Power of Primitives

The ease of writing and debugging primitive code in Squeak meant that we could return to our former loves for sound and graphics. Dusting off our FM and waveform synthesis techniques from the days of Alto microcode in Smalltalk-72, we were able to rewrite these in understandable and debuggable Smalltalk, and run even more voices on the increasingly powerful hardware of the day. We collaborated on timbre editors, MIDI players, and piano roll displays, with all of these being able to run at the same time that music was playing. For sound input we now had sampling, along with filtering and clipping; some of these sampled sounds survive to this day. For example, there is a sound in the EToys programming environment that is used signify the dropping of a program element into a script. This is actually a slowed-down replay of the first sound ever sampled in Squeak: John Maloney clinking his coffee cup with a spoon. I had implemented a Fourier transform function, which enabled John to write a nice real-time spectral "sonogram" facility.

### 8.5 BitBlt and WarpBlt

BitBlt too flourished with the ability to write and debug complex primitive code. I began by supporting all the different color modes on the modern Mac displays. This freed a lot of pent-up creative energy in the group, and made it possible to support graphical interface facilities appropriate to the evolving expectations of children and other general computer users. Ted and others worked on new paint tools to explore this dimension of our system.

For many years, Alan had a tradition of convening a "Learning Lab" each summer at a music camp in New Hampshire. These were a delightful combination of fresh experiences and demos to which a number of computer and education people outside our group were invited (and always a contingent from Mitch Resnick's group at MIT). It was two days to relax, discuss blue-sky ideas, and, when the muse struck, to hack. John's real-time voicegram was produced at one of these getaways, as was the ability of BitBlt to perform scaling and rotation.

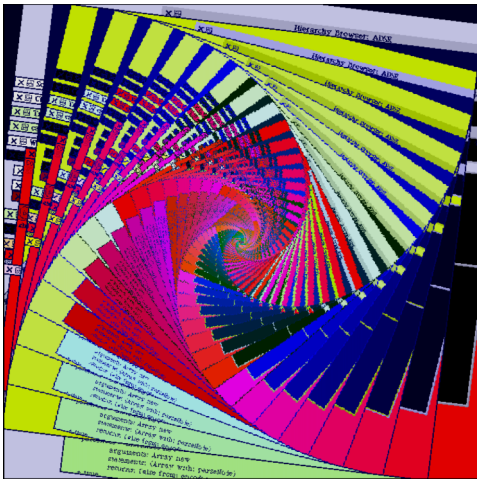

Fig. 34. The Squeak Mandala: WarpBlt copying its output into its input with scaling, rotation, and a change to its color map. Moving the mouse controls scale and rotation. To see the mandala live, click on the image or visit https://smalltalkzoo.thechm.org/HOPL-Squeak.html?mandala.

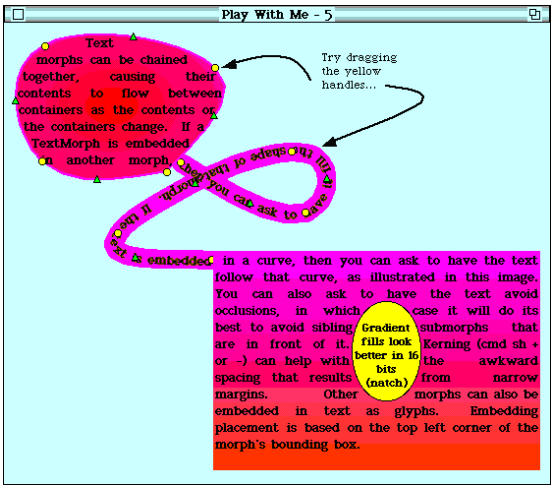


Fig. 35. Demonstration of text within a curved container, following a curved path, and flowing around a curved obstacle. To see the demonstration live, click on the image or visit https://smalltalkzoo.thechm.org/HOPL-Squeak.html?liquidText.

It had occurred to me that one could treat the input side of BitBlt as a relatively independent travelling point, delivering samples to the output rectangle side. I wrote a crude version of this in one evening at the music camp, and that experiment became the WarpBlt operation of Squeak. While it was not a breakthrough in the field of computer graphics, it was a remarkably simple piece of code which provided years of interesting graphical capabilities in Squeak and EToys. With the right parameters, it could rotate images by any angle, and expand them or shrink them, as shown in Figure 34. This function was provided as a property of any Morph, so that any of the objects built in Morphic could be scaled and rotated and, of course, animated in motion.

### 8.6 Run-around Text

For some reason, the implementation of text layout and display in Squeak fell to me. I remember driving somewhere with Alan when he remarked "Now that we've got paragraphs working, we need to be able to chain them together so we can get on to making real documents." I told him that shouldn't be too hard. Then, in typical Alan fashion, he added that it would be nice if I could do "something like what Word does going around other shapes." At first this sounded really difficult, and I just let it sit. Then I realized that, going back to the original request, text layout simply amounts to filling a chain of rectangles. I began by generalizing the existing text containers from rectangles to arbitrary polygons, and then added the possibility of having additional occluding shapes. Having done that, I needed only to find open rectangles in the resulting shape. I did this all with BitBlt in much the same way as I had done years before in my OCR project (see Section 4.7.1); first producing a shadow shape from the outline and the occluding figures, then scanning downward with a one-pixel slice to find the left and right edges of any open space, and finally developing a set of rectangles capable of holding text.

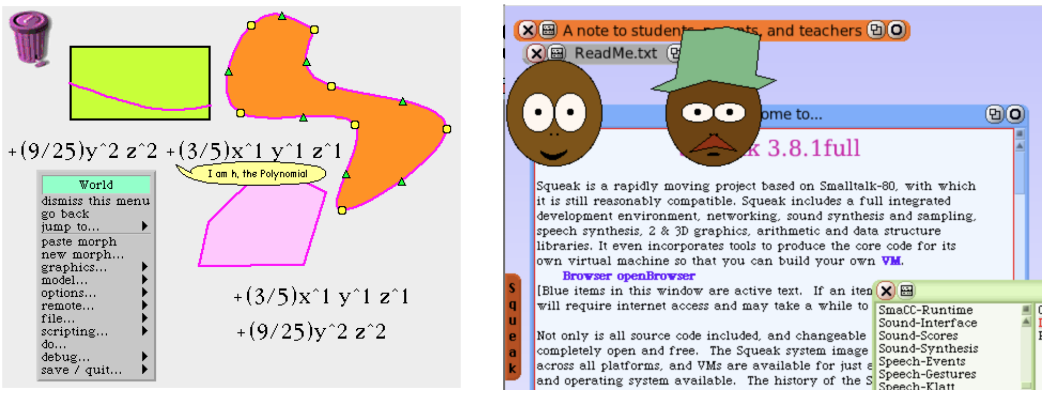


Fig. 36. MathMorphs on the left, and speech synthesis together with animated faces in Squeak on the right; two contributions from the University of Buenos Aires in Argentina. To try the speech synthesis live, click on the faces or visit https://smalltalkzoo.thechm.org/HOPL-Squeak.html?noel

I was so excited when I got this to work that I went on to add what Alan liked to call an "Oh, by the way…" capability. Since I already had polygons and curves working and, since WarpBlt could rotate anything, including text, it was not that hard to make a chain of rectangles along a curve and chain the text along those rectangles. Figure 35 demonstrates my final answer to Alan's request for chained text. The bounds of the round area, the linear curve, and the occluding ellipse can each be dragged, causing the justified text to flow back and forth though the garishly colored "pipe" in response.

### 8.7 Serendipity

Almost as soon as we had Morphic running in Squeak, we started getting messages from Leandro Caniglia and others at the University of Buenos Aires in Argentina. They had built a curriculum around numbers and math and later physics, with a very free user interface inspired by Morphic. They carried on relatively independently, but frequently contributed surprising and delightful tidbits to Squeak. One of these I will never forget: a tour de force by Luciano Notarfrancesco, consisting of a complete package with real-time speech synthesizer and animated faces, which is shown in Figure 36. It can be fully appreciated only by running this example, which he released in December of 1999. It demonstrates the capability for which we strove, which was that any enterprising user who knew Smalltalk could start from scratch and synthesize sound and images, all in the service of a personal artistic vision.

Another piece of magic in Squeak came from Ted Kaehler. Ted had always had empathy with first-time users. Years before, he had put an "explain" feature into Smalltalk-80 that would try to explain anything you typed, whether it was a Class, or some system object, or an expression in a method. In Squeak he returned to this project and produced the "MethodFinder," shown in Figure 37. This remarkable widget allowed you to type in one or two values and a result, separated by periods; it would find methods in the system that produced that result from those arguments. This was a very useful tool for anyone new to the language. For example, a new user might type `'ab'. 'cd'. 'abcd'` to find out how to concatenate strings. The response would come back: `'ab' , 'cd' --> 'abcd'`, informing the user that comma was the concatenation message.

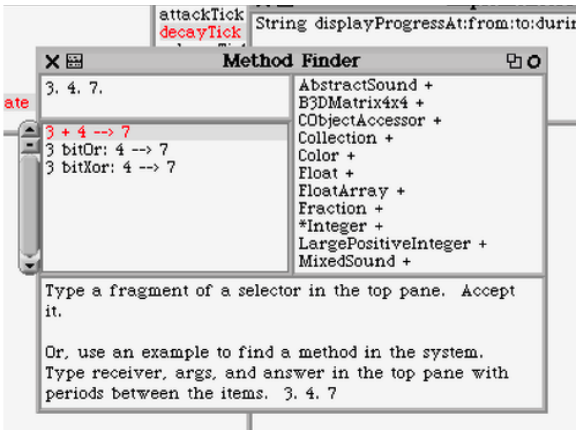


Fig. 37. The Method Finder shows the obvious response to `3. 4. 7`, as well as two other surprising but nonetheless enlightening results. In each case the Method Finder also lets you browse all the classes that implement the method in question.

### 8.8 Image Segments

Alan's never-ending thirst for creating and demonstrating examples of dynamic media shaped and stretched the Squeak system itself. The system that we demonstrated at OOPSLA in 1999 consisted of 72 linked *project worlds*, all live in the same image. Compact objects notwithstanding, that image file was 120MB, comprising 742,119 objects. The size of this image made it unwieldy to deal with as a file. Moreover, if a full garbage collection were invoked, the user could experience a noticeable delay. We figured a way similar to the VmemWriter to export project worlds and later re-import them, but the process was slow for large projects.

Since projects were already modeled as worlds unto themselves, they had, or could be made to have, very few objects that were accessible by pointers from outside the project's world. These objects were the "roots" of the world. I wanted to somehow use the garbage collector (GC) to mark all the objects accessible exclusively from the roots of a world. Running the GC would indeed mark all the accessible objects, but it would naturally continue and mark all the other objects pointed to from within the world—which would be almost everything. While thinking about this, I realized that if I *first* marked the roots of an inner world, and *then* ran the GC, it would mark all the objects in the image *except for* those in this inner world, thus providing a fast way of identifying a piece of the image that could be exported. The pre-marking worked like a parasol, casting a shadow over the inner world. By borrowing from other GC logic, it was then possible to copy the objects of that world into an image-like structure called an `ImageSegment` that could be exported and imported very quickly (often in less than a second). Not all projects were amenable to this form of export, but enough of them were that we were able to reduce such huge images to more manageable sizes.

Worlds so exported were replaced by a root stub that would cause a fault, bring in the image segment, and continue running, like a segment-swapping virtual memory. To make this work I defined an `ImageSegmentRootStub` that could replace (using in-place mutation) any class or instance or whole world that had been swapped out. While designed as a virtual memory facility, the same

mechanism made it possible to eject nearly all of the system, and then instrument the minimal working set needed to run any given application.

### 8.9 Licensing

With help from Jim Spohrer at Apple, Alan and Ted authored the Squeak Open Source License. The license allows commercial systems to be created in Squeak and sold royalty-free. Approved by Apple in 1996, this was one of the early open-source licenses, and one of the first granted by a profit-making company that did not have open source as its business model. Ted also built and maintained our website, Squeak.org, until it was later taken over by others in the Squeak community.

### 8.10 3D

No one did more with Squeak's ability to expand itself through primitives in Slang than Andreas Raab. Andreas earned early entrance to Squeak Central[25] by carrying out the port to Windows in little more than a week. He soon started to extend Squeak's graphics to provide anti-aliasing and gradient fills; this greatly improved our pixelated graphics front end. Andreas's work grew into the Balloon graphics system, around which he built a Flash graphics renderer capable of displaying sections from Lion King, and other beautiful graphic material. It seemed that there was nothing Andreas could not do! Balloon grew into Balloon-3D, and became the vehicle for Jeff Pierce's Squeak implementation of the Alice 3D composition system [Pierce 2002].

A meeting with Alan Kay, Andreas, David Smith, and David Reed spawned the idea of a live collaborative 3D system in Squeak. We would never have attempted such an ambitious project in previous Smalltalks, but the Slang primitive interface enabled Andeas to provide high-performance graphics with a clean interface, so that the other challenges of collaborative 3-D could be met with relative ease in Squeak's interactive world. The resulting Croquet [Smith et al. 2003] project went on to serve as an industrial multi-person communication and collaboration environment. Some highlights of Croquet include an escalator that would carry the user from one 3D room to another, a Verrazano bridge oscillation that morphed into a waving Canadian flag, and mirrors that would faithfully create the "infinite corridor" effect when properly aligned. A video is available at https://vimeo.com/257578323

### 8.11 The Squeak Update and Release Process

For the first few versions of Squeak, I oversaw the release process, after which this role passed to Scott Wallace. Various contributors would make their improvements known, and at some point I would merge all of those that I thought deserved inclusion, test the resulting system as best I could, and then post it as a new version with a descriptive message. The curious reader might enjoy following the series of these releases, all of which run in the SqueakJS emulator in any browser [Freudenberg et al. 2014], and many of which can be launched by their link on the SqueakJS launcher page https://smalltalkzoo.thechm.org/HOPL-Squeak.html?launch. Ted and Scott wrote

[25] Squeak was designed, implemented and released by four of us in Alan's group at Apple and then Disney. We managed the releases and related communications and soon came to be referred to as "Squeak Central." Others joined us along the way, but the inner circle always remained small and mostly acted as one.

the Squeak "ChangeSorter," which made it easy to manage a number of different ChangeSets.[26] The ability to move changes around in different files and detect conflicts between ChangeSets made it much easier to run the release process.

Even better, Ted and Scott developed an automatic update mechanism for changes that were candidates for the next release. With this feature, if someone had published a change since the last time you started Squeak, you would receive a notification that "Updates" (that is, new changes) were available. You could choose to ignore them (the safe choice), or you could choose to accept the updates. Most of us in the inner circle would opt to take the latest updates, and so were performing a constant beta test of the next release. If something went wrong, the problem was fixed, or the change was withdrawn if the problem was serious.

Each new update had a sequence number, and even when a new official release had not been made, it was easy to request changes through, e.g., update #1538. Making an official release was greatly simplified by this facility, since the person in charge already knew that, say, all updates through 1538 had been well tested, and little was required other than to update to that level, write a release message, and save an image as the new current release. Ted wrote the original update mechanism, using simple FTP. It did not require any new server code and ran very reliably as a result. As Squeak evolved and the community grew larger, the management of releases passed out of Squeak Central to the community at large. It is now generally managed using the Monticello change management system.

### 8.12 The Impact of Squeak

Squeak arrived at a time when Smalltalk needed a boost. Arguably, one of the biggest impacts of Squeak is that it kept Smalltalk alive and nurtured a world-wide community just when commercial Smalltalk endeavors were falling out of favor and failing. It was Squeak's simplicity and aliveness that re-energized the existing Smalltalk community, and led to significant adoption in education, as well as in a number of "stealth" projects.

Most primitives in Squeak Smalltalk could be written in Slang, and thus ported to other platforms without needing to be *re*written. This made Squeak extraordinarily portable. We had done all our Squeak development on Apple Macintosh computers, but within a week after we released the first Squeak, Ian Piumarta had it running on Unix, and in two more weeks, Andreas Raab had it running on Windows.

The ease of writing primitives in Squeak also enabled us to lavish attention on compatibility of images across the different ports. Thus, from an early date, a Squeak image could be halted in the middle of a debugger, saved on one computer, sent across the country, and resumed on a different computer with the opposite endianness. Everything would resume with bitwise fidelity.

Almost as soon as Squeak was released, the nascent Squeak community lit up with applications to the Internet. Georg Gollmann added a web server, and many Squeak images were downloaded just to get that free and elegant web server. The fact that most of the primitive code was written in Smalltalk, and thus easily ported to, maintainable on, any machine, made this a viral introduction. Soon after, Mark Guzdial at Georgia Tech and a student of his, Lex Spoon, put out a Wiki server ("Swiki") based on Ward Cunningham's work. Later Ted Kaehler added disk code to the Georgia

[26] A *ChangeSet* is a dictionary that records all the changes you make while working in a given world. It can be saved on a file, passed to others, and combined or taken apart with the aid of a ChangeSorter.

Tech Swiki, and this project became Comanche (now KomHttpServer). Yet another outgrowth of this early work was the AIDA Smalltalk web framework.

The Seaside web development framework is built around the ease of making continuations with Smalltalk's Context objects, and the ease of implementing web servers in Smalltalk. Squeak was also the progenitor of Pharo Smalltalk https://pharo.org, which began as a wholesale copy of the Squeak system. The Pharo project offers similar functionality with more emphasis on stability and commercial support.

Squeak and Pharo have been joint beneficiaries of the extraordinary Cog work of Eliot Miranda's team, who have pioneered techniques for just-in-time (JIT) compilation of Smalltalk [Miranda 2011; Miranda et al. 2018].

Finally, Squeak was the inspiration and the model for Lively, the live-coding JavaScript environment in which the simulations referenced here were constructed [Ingalls et al. 2016].

The following link allows you to run several versions of Squeak: https://smalltalkzoo.thechm.org/HOPL-Squeak.html?launch Choose the Squeak 2.8 image (from June 2000). It offers a number of fun demos that can be opened by clicking the icon in the upper right corner:

- Play With Me 3: a graphics demo world Shows clipping by the shape of a font, and anti-aliasing. The ellipse is a magnifier that you can move around the screen
- Play With Me 4: Drop the yellow ellipse in the text to see runaround text. Use right-click to get handles on the ellipse, and drag the brown one
- Play With Me 5: Even more text fun.
- In any text window, paste this text, select it, and choose `doit` (or cmd-D)
  ```
  Display restoreAfter: [WarpBlt test3]
  ```

The first couple of years of Squeak saw a joyful explosion of clever ideas and completion of many yearnings we had had in earlier systems, especially in the areas of sound and graphics. When Squeak came to life, it was a huge rush of energy for all of us. Suddenly we could work on all our favorite problems in a wonderful language and environment, and yet everything ran at decent speed. "Everything" included multicolor bitmap graphics with anti-aliasing, zooming linked worlds for multiple projects, performance and portability of the VM, waveform and FM music synthesis with many voices in real time, painting tools, programming tools, and more. Squeak epitomized our goal of a simple yet general programming system; one that was built to fulfill unanticipated computing ideas with ease.

### 8.13 EToys

When we set out to do Squeak, there were two main goals. The one I've written about so far was to do a small, fast, "cool" (i.e., with sound and graphics) Smalltalk—essentially to revisit all the things we thought we could do in Smalltalk before but hadn't. This was the goal for which I carried the flag. The other goal, hinted at earlier, was to build a software vehicle in which we could completely control our destiny. This was what Alan wanted, because he knew that if we were going to reach children—if we were going to get them excited, and keep them excited, about ideas and how to bring ideas to life in a computer—then it wasn't going to be with paned windows and text. Through all the excitement, Alan reminded us that Squeak was a vehicle, not the goal itself. Alan always contended, and we gradually came to see, that if something is good for first-time users, it is

generally good for experts as well. EToys was our reach out to the world of children and first-time users and, true to Alan's belief, the Squeak system just got better and better as we went forward.

EToys can be thought of as a subsystem of Squeak, because that is how it was implemented. It can also be thought of as new, graphical, Smalltalk language. A number of forces and ideas fed into the EToys system, and a number of improvements to Squeak emerged from it. The EToys project deserves a paper on its own, but I will summarize some of the innovations here.

*8.13.1 Inception.* We had a meeting at Ted's house to brainstorm the real goal. None of us can recall exactly how it went, but there was a napkin or scrap of paper that captured the notion of a split screen with a concrete scene on the left and various analytical controls on the right—or at least this is how I remember it. I have always considered this to be the real breakthrough that launched us into the world of EToys. When a student sees numbers for $x$- and $y$-positions change immediately while dragging an object, there is little need for explanation. The association between the object's location and the coordinates goes straight to the brain as unshakeable experience. And in case there is any question, scrolling the numbers in the analytical view confirms the association, as the concrete object moves in response.

This juxtaposition of concrete and analytical views was the beginning of EToys. As we worked on content and instruction, we developed a number of improvements to the user interface, and to the language as well. To my mind, the most important conceptual shift actually came with Morphic: the shift from "apps" like browsers, editors, and drawing tools, to independent (but interacting) objects in a world like circles, squares, sketches and images.

*8.13.2 Halos.* Squeak had already moved away from a global menu bar to local menus (for operations such as moving and editing), but even these seemed too "app-ish," that is, overly focused on global functions as opposed to local content. Local menus served us well for our early programming needs, but were too stilted for the kind of content we had in mind, both for children and ourselves. We wanted any object in the world to somehow "glow" with its capabilities, and we came up with the notion of a "halo" of handles. Some of these worked like the old menu commands (in fact, usually one handle would bring up a conventional menu), but others worked as direct manipulation handles for dragging, scaling, rotating and so on. Also important to the morphic model of objects in a scene graph was that clicking and dragging an object meant picking it up (that is, removing it from the scene graph), to be reinserted elsewhere in the scene graph when it was dropped. There were, of course, times when one wanted to move a morph without removing it from its owner; the halo of handles provided separate **grab** and **drag** handles to make this distinction.

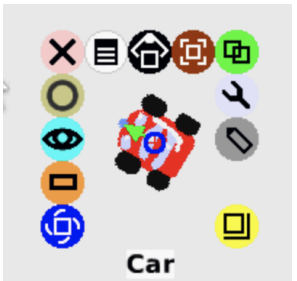


Fig. 38. Command-click on any object in a Squeak world brings up a "halo" of manipulation handles.

The halo of Figure 38 shows a Car morph with the following handles (clockwise from the bottom-left):

- Rotate the object.
- Make a tile representing the object.
- Open a "viewer"—an inspector with executable messages on a flap, as shown on the right of Figure 39.

- Collapse the object to the side of the screen.
- Delete the object—it can be retrieved later from the trash.
- Bring up a conventional menu with further commands.
- Pick up the object, removing it from its prior owner.
- Drag the object without removing it from its prior owner.
- Duplicate the object.
- Choose and open any of several inspectors on the object.
- Open a paint program on the object.
- Shrink or grow the object.

In addition, a little blue ring appeared within the halo; moving it had the effect of relocating the programmatic position of the object with respect to its image. The green arrow emanating from the ring could be moved to indicate the "heading" of the object (i.e., the direction in which the object would move if sent a "forward by" message) with respect to its image.

*8.13.3 Flaps.* As we and our users gained experience with EToys, we found the need for various "panels" that would not compete for valuable screen real estate. We have already alluded to the "control panels" that displayed such metrics as position, scale and rotation of an object, as well as push-button scripts, such as turn-by or rotate-by. Rather than put these in windows, we put them on "flaps"—off-screen areas that could slide on or off the screen as necessary. We soon enhanced this facility with labelled "tabs," so that a number of such flaps could be off screen, but instantly accessible by clicking a tab at the edge of the screen. This flap-tab facility was lovingly improved by Scott Wallace so that the tabs could be moved along the edge of the screen, and ultimately to any side of the screen, and settle themselves so as not to interfere with other tabs. They also came to remember how far out to extend when clicked. Finally, control panel tabs could display as a miniature image of the object under inspection as well as a textual label. Off-screen flaps came to serve a number of other uses, such as access to an editable menu of useful executable statements, or a palette of morphic shapes ready to be dragged into the world.

*8.13.4 EToy Affordances.* The Etoys project, being the direct progenitor of Scratch, has had more impact in education and end-user access to computing than any other application in Squeak. Etoys pioneered the brilliant juxtaposition of concrete and analytical views of dynamic processes for children. On the left we see the ball moving, on the right we see its numerical coordinates changing, whether in simulation or being dragged by the user's mouse. It also made a point of having the user create objects from sketches, so as to be personally invested in their later dynamic behavior. Finally, EToys introduced code as concrete manipulable tiles, so the same concreteness applies even to the construction of programs. The use of tiles mostly eliminated the need to type on a keyboard, and made syntax errors nearly impossible, both huge advantages for young learners.

*8.13.5 Worlds.* EToys also introduced several changes to the user experience and the language itself. For instance, the "project" mechanism that had been present since Smalltalk-76 (see Section 5.15) evolved into a Morphic "World" that could be saved and loaded independently (using the image segment facility of Squeak described in Section 8.8). For linear presentations, a BookMorph offered a simple page-flipping viewer of collections that could each be a separate world. But to build a personal universe, the choice was to add worlds to other worlds with small iconic links. Each world had a parent link so that all one's worlds could be navigated as a tree, and nice zoom-in/out effects enhanced the feeling of an infinite linked universe of live simulated worlds. Alan's amazing slide shows drove the creation and fine-tuning of many such facilities.

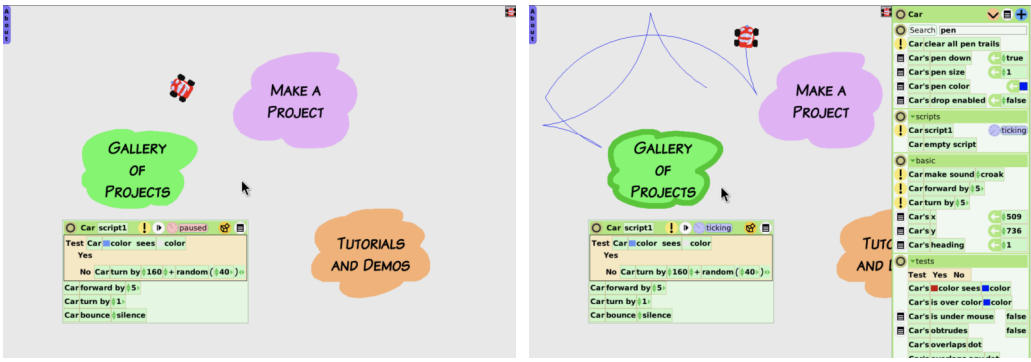


Fig. 39. On the left, an EToys scene with a script that tells the car to turn around if it encounters any color other than the background. On the right, a flap with the car's viewer has been deployed, and the car's "pen down" has been set to "true", causing it to make trails as it moves in the world. To run EToys live, click on the left or visit https://smalltalkzoo.thechm.org/HOPL-Squeak.html?etoys.

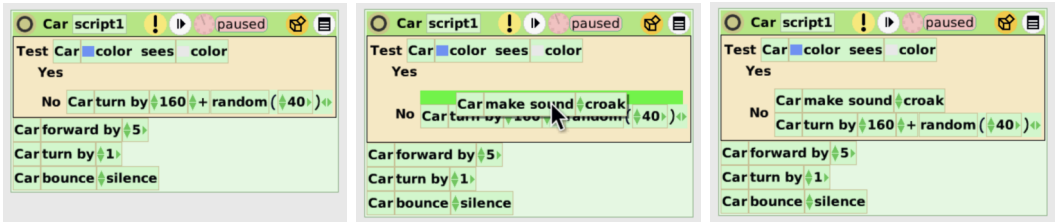


Fig. 40. On the left, we see a "scripter" that defines a script on a "Car" object. It can be executed once, or set to ticking at various rates. Further right, a "make sound" command has been dragged out of the control panel, and is about to be dropped into the green "drop zone." In the rightmost frame, the drop has taken place; now, every time the car runs into something, it will make a "croak" sound.

*8.13.6 Namespaces.* The morphic projects inherited the convenience of independent changeSets from earlier Smalltalks. But each WorldMorph also served as a namespace, so that various objects in a world could refer to each other by their world-local names, and even by class or other selection criteria.

*8.13.7 Control Panels.* With the availability of flaps came the ability to present much of an object's behavior in a visual and dynamically manipulable way. Control panels began as simple inspectors with manipulable presentations of, eg, position, heading, color and such. Soon we added a number of methods, not just as abstract methods, but with concrete arguments, ready to run, such as `car forward by 5` and `car turn by 5`. To these we added a little yellow do-it button, reminiscent of Smalltalk-72's bold exclamation mark. A sample control panel appears in Figure 39, as its tiny iconic flap-tab on the left of the figure, and as the fully extended flap on the right. We made the concrete arguments editable, and naturally we wanted a way to tie the methods together into meaningful scripts.

*8.13.8 Scripters.* We made it possible to spawn an editor from the control panel, and soon made it so that one could simply drop a method into an empty space in the world to create a "scripter" object with the script in it as shown in Figure 40. A scripter duplicates the do-it button functionality, but

adds to it the ability to repeatedly run the method or methods enclosed, or trigger them on various events. Most importantly, the scripter allows other scripts to be dropped into it, thus building-up a small sequential program. There is a place to name the script, thus putting it on equal footing with the system-supplied scripts, and making it visible in the control panel. The easy progression from dragging out one command, to having a full method, helps students to build their project as a group of scripts. The live morphic environment enables many such scripts to be active and interacting continuously. Figure 39 shows flaps, control panels, and scripters in use in a tutorial EToys world at https://smalltalkzoo.thechm.org/HOPL-Squeak.html?etoys

*8.13.9 Uniclasses.* The EToy/Morphic development style emphasized creating and scripting individual objects rather than developing classes. While Squeak's linguistic base was a class-based Smalltalk-80, we needed a simple way to create and program singleton objects, as in the Self and JavaScript languages. For this, Scott Wallace developed a scheme called "uniclasses" that made it simple to add a script on any object by making it a solitary instance of a new class. We also introduced an extension dictionary to enable the addition of arbitrary properties to any Morph without concern for Squeak's conventional model of fixed-sized instances.

### 8.14 Scratch

EToys were the direct progenitors of Scratch. After leaving the Squeak project, John Maloney, who had been active in developing EToys, worked at MIT with Mitch Resnick on Scratch. The first versions of Scratch actually ran in Squeak, and you can run one of these at https://smalltalkzoo.thechm.org/HOPL-Squeak.html?scratch. To reveal the underlying Squeak, shift-click in the loop of the "R" in the logo, and choose "turn fill screen off." This will reveal a border where all the Squeak menus, browsers, etc., can be accessed. Scratch benefited from key funding from the NSF; teacher experience and feedback from the Computer Clubhouse network gave the Scratch team valuable input over the course of almost three years of internal testing before the public release of Scratch in 2007. At the time of this writing, Scratch had over 50 million users (see https://scratch.mit.edu/statistics/).

### 8.15 My Favorite Etoy Project

My favorite Etoy project is one done by Alan Kay. Illustrated in Figure 41, it begins with a multi-timbral real-time MIDI player mostly built by John Maloney. It is not just reading a MIDI score; it is synthesizing every voice being played. As one moves the musicians left and right and near and far, they control (through Etoy tile scripts) the volume and pan parameters of the MIDI player. The sizes of the players automatically scale with vertical position, through another iconic tile script. One can also edit the parameters of each instrument's sound in a live panel—all while the piece is being played.

## 9 CONCLUSION

Even today, many of these Smalltalk systems seem remarkable. The ability to model objects, especially engaging objects with graphics and sound, in an environment in which they can be freely connected in almost any manner, invites innovation. Because it is all immediately reactive, it is exciting and fun.

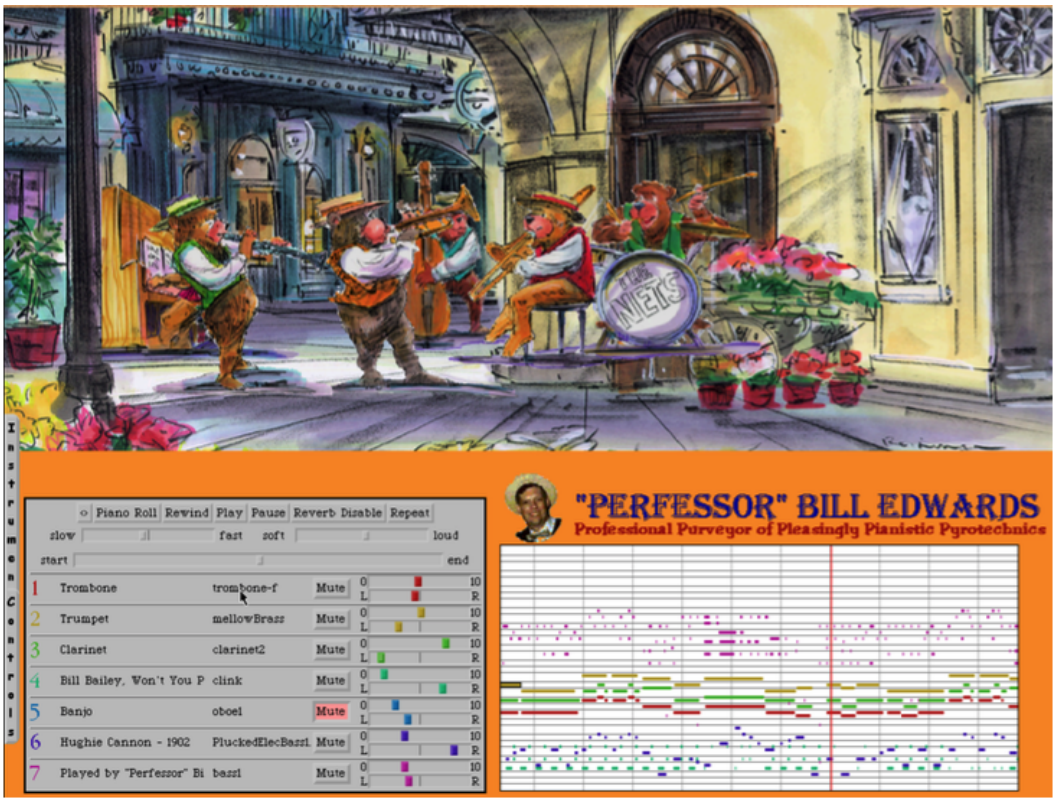


Fig. 41. A multi-timbral MIDI player animated with EToy scripts. Each player can be moved left and right with the mouse to pan the corresponding voice in the MIDI-player, and near and far to control the volume. To try this live, click on the image, or visit https://smalltalkzoo.thechm.org/HOPL-Squeak.html?bailey

I have sometimes said that we had a "free ride" in the Smalltalk project: the computers were getting just fast enough, and small enough, and cheap enough; the graphical displays were getting just good enough. All that is true. But the key was really people, not technology. Alan wanted something great: a personal computer to foster human creativity and understanding. Adele wanted to improve education, and to push technology out into the world. I wanted to make it easy to play with words and numbers and graphics and sound, and with the tool itself. As we grew and evolved, all the people who came to the project came with a passion to reify some aspect or other of Alan's compelling vision.

Ordinarily I would not talk about money in the context of Smalltalk, but none of this work could have happened without sponsorship. Alan and Adele both radiated a passion that was strong enough to co-opt even cost-conscious executives on a journey to a place they couldn't really understand.

Once you have the people, and the vision, and the money, then good things can happen. But there is one more crucial factor if you want to leave the world a better place, and that is engagement. People must have access to what you are building; they must use it, have trouble with it, complain about it, and, ideally, inspire you make it better.

Our group evolved, as did our engagement, over the years. It began with the our contribution to the "Office of the Future" vision at Xerox, while Alan and Adele were experimenting with teaching, and also giving talks.

With Smalltalk-80, Adele went squarely for commercial distribution. This led to a different kind of engagement, but it absolutely shaped the product and induced a new, and keen, focus: one that had major impact in the world. That engagement infected the industry, and I felt that we did change the world in several good ways.

When we reconvened around Squeak, we sought to engage a community of children who were hungry to make their own media, and yet we needed to retain the open field of raw materials and logical concepts to foster basic experimentation and learning. That was the genesis of EToys and later Scratch, and these are satisfying results.

And here we are, looking back on the journey, and evaluating the results. Nevertheless, this article it is not an ode: the thing is not dead. Although the community is small, many people still turn to Smalltalk, Pharo, or Squeak to build a little app, or try out some unanticipated idea with graphics or sound. Some people choose to start in a workspace, some in an inspector, some in the debugger. There's plenty more to do. What can we learn from this still glowing ember, that will lead us to a new tool of learning for children of all ages?

## ACKNOWLEDGEMENTS

I owe everything to the wonderful people with whom I worked on these projects. We had so much fun, and I never could have done the things I did without the excitement and joy we created together.

Special thanks to Alan Kay for giving me and all of us such great problems, for nudging me when I got off course, and for sustaining the vision that computers were made to help us think and learn and create.

Thanks to Andrew Black for his many great suggestions about the paper, and for bulldozing a path through LaTeX, BibTeX, and Git. I could never have done it without him. He was my friend in need as he "shepherded" me thorough the challenges of writing a HOPL paper.

Thanks to Bert Freudenberg for his great work on SqueakJS, for his work with Ted Kaehler, Yoshiki Oshima, and me on the Smalltalk-78 simulation, and for his help with setting up the smalltalkzoo website.

Thanks to Richard Gabriel, Adele Goldberg, Ted Kaehler, Alan Kay, John Maloney, Dave Robson, and Allen Wirfs-Brock for innumerable useful comments and suggestions on various earlier drafts.

Thanks to Chris Thorgrimsson and C. H. Huang for their warm encouragement of this writing.

The Learning Research Group at Xerox (Smalltalk-72 through Smalltalk-80):
Karla Garcia, Adele Goldberg, Dan Ingalls, Chris Jeffers, Ted Kaehler, Alan Kay, Glenn Krasner, Diana Merry, Steve Putz, Dave Robson, John Shoch, Steve Weyer, and Bert Sutherland, director of our lab at Parc.

The Apple Smalltalk group (APDA Smalltalk-80):
Yu-Ying Chow, Ken Doyle, Barry Haynes, Dan Ingalls, Frank Ludolph, Mark Lentczner, Scott Wallace.

The Squeak group at Viewpoints Research (Squeak and EToys):
Dan Ingalls, Ted Kaehler, Alan Kay, Aran Lunzer, John Maloney, Yoshiki Oshima, Ian Piumarta, Andreas Raab, Kim Rose, David A. Smith, Scott Wallace, Alex Warth.

Visitors to our group at Xerox over time:
Jim Althoff, Ron Baecker, Bill Bowman, Barbara Deutsch, Marian Golden, Laura Gould, Bob Flegal, Susan Hammett, Bruce Horn, Tom Horsley, Lisa Jack, Kathy Mansfield, Eric Martin, Steve Purcell, Steve Saunders, Dick Shoup, Bob Shur, David Canfield Smith, Bonnie Tenenbaum.

From other groups at Parc:
Daniel Bobrow, David Boggs, Bill Bowman, Larry Clark, Jim Cucinitti, L Peter Deutsch, Bill English, Doug Fairbairn, Ira Goldstein, Ralph Kimball, Butler Lampson, Bob Metcalfe, Ed McCreight, Mike Overton, Bob Sproull, Larry Stewart, Larry Tesler, Chuck Thacker, Truett Thatch, and Bob Taylor, director of the Computer Sciences Lab at Parc.

Help with the simulations:
Josh Dersch for use of his Alto Emulator,
Larry Stewart and Al Kossow for rescuing Alto disks,
Robert Krahn and Marko Roeder for work on Lively Kernel for the simulations.

Support from major organizations:
The Xerox Corporation for supporting all the early work on Smalltalk,
Apple Computer, Inc. for supporting work on APDA Smalltalk and the birth of Squeak,
The Walt Disney Company for supporting work on Squeak and EToys,
The Hewlett-Packard Company for supporting work on Squeak and EToys,
Applied Minds, LLC. for office space and other support,
The Computer History Museum for hosting the smalltalkzoo website.

Thanks and love to my wife Kat for continually encouraging me to finish this paper.

## A ARCHAEOLOGY AND RESURRECTION

We are fortunate to have just enough scraps of history to piece together a fairly full collection of running Smalltalk artifacts. When the Smalltalk group and other Alto users left Xerox Parc, machines and disk packs were discarded. A Parc researcher became aware of this at the eleventh hour, and took the initiative to spare a number of Alto disk packs from the dumpsters; another tech who had worked with us over the years noticed when a number of additional disks showed up at a Bay Area electronics salvage store. These were then copied onto then-standard media (CDs), and this treasure trove made its way to BitSavers. Many of these disks were not Smalltalk disks but, remarkably, it has turned out that we have representative software from every one of the early Parc Smalltalks.

How could we be so lucky? It is rare enough that 121 disk packs were saved from the dumpster. But, regarding Smalltalk's history, how could we ever have hoped that these included the following:

(1) one and only one disk with a running Smalltalk-74 system,
(2) one and only one disk with the construction of Smalltalk-76 in Smalltalk-74,
(3) one and only one disk with a running Smalltalk-76 system that includes complete Smalltalk sources, and
(4) one and only one disk that includes the two object files that comprise the last freshly written copy of Smalltalk-78.

### A.1 Smalltalk-72

Smalltalk-72 was widely used in the lab, especially in several longitudinal education projects, so many disks have running versions of the system, preserved as SWAT memory images.

The Alto OS provided an "outload debugger" named SWAT. It functioned by outloading the entire memory to disk and then enabling you to debug the program that had been in memory by reading and writing the disk. When you were ready to proceed, SWAT would load it all back and resume execution. Smalltalk quit and resume made use of this full-speed store and reload of the Alto memory image.

The interpreter for Smalltalk-72 was written entirely in Nova assembly code, and I began to look around for enough of the assembler files, or paper listings, to reassemble a working system. At the time I was working on Lively-Web [Ingalls et al. 2016], a Squeak-inspired live coding system in JavaScript. It occurred to me that, since the old Smalltalk executable files were simply snapshots of the entire state of memory, "all that would be needed" was to read the file into a big JavaScript array, and write a JavaScript program to emulate the Novacode operations, not unlike the microcode emulator on the Alto.

In a week I figured out how the memory snapshots were stored, and also how to get them into a JavaScript array with good access performance. In another week I had the instruction set working, and another week sufficed to format the code so that it was readable enough to aid in debugging. In about a month, an ancient Smalltalk-72 system again produced "bits on the screen."

You can run the live system at https://smalltalkzoo.thechm.org/HOPL-St72.html?snippets. When you start this page, you will see a panel showing either a memory snapshot with Nova instructions, or an Alto screen, depending on the setting of the Show Nova / Show Smalltalk button. If you choose "**Show Nova**", and press restart, you will be looking at the exact place in memory where the program counter was frozen. (The JSRII jump instruction leaves the PC just following the instruction where this Smalltalk last quit.)

You can the push "**step**" repeatedly to watch the Nova instructions executing and see the contents of the four registers change accordingly. If this gets tiresome, push "**run**" to let the Nova run without waiting after every instruction. If you still find it boring or meaningless, it is time for the "**show Smalltalk**" button. In this mode, emulation runs at full speed and the display shifts to show the contents of the section of Alto memory that normally appears on the Alto screen. It should show a REPL window with a character that looks like a tiny Alto beckoning you to type some Smalltalk expressions. Try "3+4\"—the backslash has the effect of typing the "doit" bold exclamation character.

For those who are interested in further exploring Smalltalk-72, we suggest that you use the button to "Open the Smalltalk-72 Instruction Manual" and continue to explore this environment. The examples in the text of this article will also give you a sense of how the language and system work.

The Smalltalk-72 simulation runs entirely live in the Lively development system [Ingalls et al. 2016]. Lively is a live-coding environment with a Class library and Morphic graphics model similar to that of Squeak, except that it is all written in JavaScript and runs live in any web browser. Readers curious to know more about Lively are referred to https://smalltalkzoo.thechm.org/welcome.html.

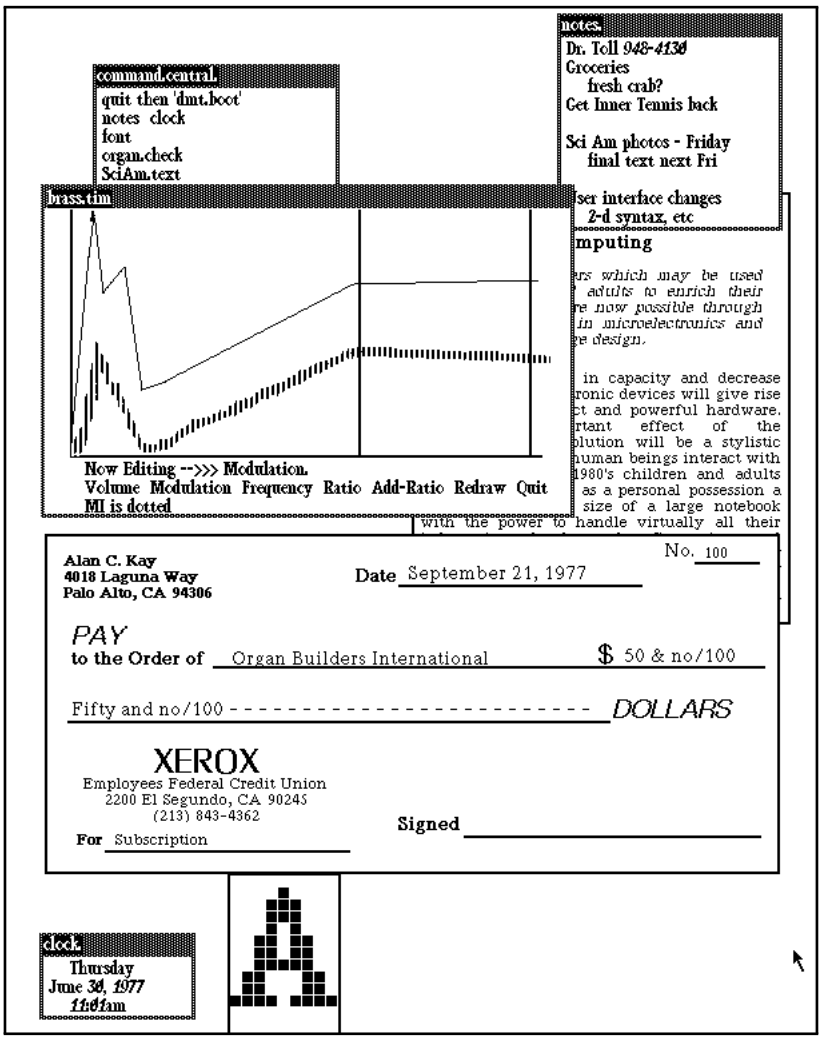


Fig. 42. A screen shot from the only Smalltalk-74 disk. Compare to Figure 13.

## A.2 Smalltalk-74

That Smalltalk-74 was so quickly abandoned explains why, out of 120 salvaged disk packs, we found only one with Smalltalk-74 on it. That disk is full of pieces of the Smalltalk-76 simulation, and its screen and scheduler is full of content from Alan's Scientific American article—see Figure 42.

I was able to get this Smalltalk-74 image to run on Josh's Dersch's Alto emulator (it required microcode support which I had not added to my Smalltalk emulator). Sure enough, there appeared on the screen several items reminiscent of the Scientific American article. I started to work with it, cleaning up the scheduler and figuring out how to do things again, but I could not get the edit menu required for evaluation of code snippets. It would always place the menu in the wrong place, and the selection rectangle elsewhere on the screen. By being persistent, though, I was able to infer what item was being selected and could, when lucky, get a few things to execute. As I was able to

clear the screen, I noticed in one of the windows a snippet of text that appeared to create the edit menu anew. I carefully managed to select this and invoke "doit" from a menu with its disembodied selection rectangle. *Mirabile dictu*, suddenly the edit menus worked correctly everywhere! Thrilled with this result, I executed `user quit` to save that image. I copied it to a backup, and returned to carry on my investigations.

Alas, when I resumed that Smalltalk, it had the same old problem with edit menus. Apparently, this was something that went amiss every time the system was restarted. Even in those early systems, the Smalltalk save logic saves and restores the exact state of computation in process at the time of executing the `quit` command. Knowing that, I tried (you would have too, right?) copying the bug fix that restored the edit menu directly after the `quit` command, as shown in Figure 43. This worked like a charm. Thereafter, when I wanted to quit, I just needed to select and execute both the `quit` command and the bug fix that followed it.


Fig. 43. The code that quits Smalltalk, and immediately after restart, fixes the edit menu again.

Now we have a Smalltalk-74 image that, while full of project fragments, is entirely usable. Until I track down the bug, it is necessary to use this circumlocution whenever exiting the system. Upon reflection, I remember an emergency bug alert from Diana when she was preparing Alan's article. I kept in my office a Radio Shack plastic helmet with a rotating red light and siren on top. On occasions such as this, I would don the helmet and rush to visit the scene where a bug was reported. With hindsight, I believe this was the very problem that occurred at that time, and probably why this disk was abandoned in its current state.

### A.3 Smalltalk-78

In the arc of Smalltalk implementations, two particular gems of simplicity and compactness stand out: Smalltalk-78 that ran on the NoteTaker portable 8086 machine and the Mini2.2 Squeak image that ran on a Mitsubishi microprocessor M32R/D with on-chip main memory.

As described in the text, Smalltalk-78 was nearly identical to Smalltalk-76, except for its more efficient context architecture and the move to a BitBlt-centered graphics kernel. This provided a 2× raw speed increase over the Alto, together with a much more immediate experience owing to the absence of swapping delays from a virtual memory.

Among the discarded Alto diskpacks, we found one that had been used to write an object memory image for porting to the NoteTaker. The Smalltalk-76 system on that disk was unusable, but the object memory file that it wrote for the NoteTaker was still there.

Given the success of the Smalltalk-72 revival in JavaScript and the SqueakJS emulation of Squeak (see below), Bert Freudenberg and I talked about performing a similar JavaScript resurrection of that NoteTaker image. When Alan heard this he became excited about a related project he had in mind, and so we set about producing a simulation of Smalltalk-78 to run in browsers.

We determined to bring it to life by reusing the object memory system and BitBlt code from SqueakJS (mainly Bert), and rewriting the interpreter to work with the Smalltalk-76 bytecode set and the unique contiguous-stack context architecture that was designed for the Intel 8086 microprocessor (mainly Me).

After a bit of sleuthing, we were able to decipher the file and read it into the Lively JavaScript development environment. We then began the process of debugging this little time capsule into existence. Bert read in the object table, dealing with integer and object formats and then went on to adapt the code to Smalltalk-78's earlier BitBlt conventions. Meanwhile I worked on the bytecode differences and the completely different Context structure. In only a couple of weeks we had "bits on the screen." We were able to browse and work with decompiled source code, and we began cleaning up various things that we had never had the chance to do on the NoteTaker itself. Ted managed to import a lot of missing comments from a Smalltalk-76 source code file, and Alan began his project to build a presentation system in Smalltalk-78.

Run the live system at https://smalltalkzoo.thechm.org/HOPL-St78.html?base

This version of Smalltalk-78 is identical to that which was ported and run on the Intel 8086-based NoteTaker, except for a few small code changes and additions, and the fact that it runs much faster.

You are encouraged to play around with this system. You can use the screen menu to open a browser and a workspace; these will give you a chance to explore the system. Note that if you make a new selection after editing in the browser, it will simply flash the pane until you choose cancel from the edit menu. "Doit" in the edit menu will evaluate the selected expression and print the result at that place in the text, as you might expect. You might want to try evaluating `100 factorial`. In case you're not confident of the result, evaluate `100 factorial / 99 factorial`.

Having revived Smalltalk-78 running in a browser, we have found its lean kernel and responsiveness to be a surprisingly usable (albeit black and white) media kernel. More details of the project to revive Smalltalk-78 can be found in our paper presented at IWST 2014 [Ingalls et al. 2014].

### A.4 Smalltalk-80

Although Smalltalk-80 is a current programming language, well-supported in the industry, I thought it would be good to resurrect its two earliest manifestations. For this I started with Vassili Bykov's elegant Smalltalk-80 Hobbes simulator. Hobbes uses the storage manager of the host language for its objects, thus considerably increasing its performance. I ported Hobbes to Squeak in order to run the ParcPlace Version 2 release, and then made the necessary changes to run the APDA Smalltalk-80 that I worked on at Apple. The APDA release is notable partly because it has a one-button user interface (menus can be invoked from scroll bars and window title bars) that enables use on tablets without mice, but its real claim to fame now is that it is the original image from which we started to build Squeak.

For fun, I have loaded both of these simulations in a Squeak system, where they run inside two active Morphic windows as shown in Figure 44. This simulation is all the more remarkable because the Squeak system is running in Bert Freudenberg's SqueakJS [2014]: two Smalltalk-80s simulated in Squeak, and emulated in JavaScript!

### A.5 Squeak

Although Squeak is still available for most computers, SqueakJS has become the easiest way to run Squeak for most users. It runs in just about any web browser, which helps in schools that do not allow the installation of non-standard software.

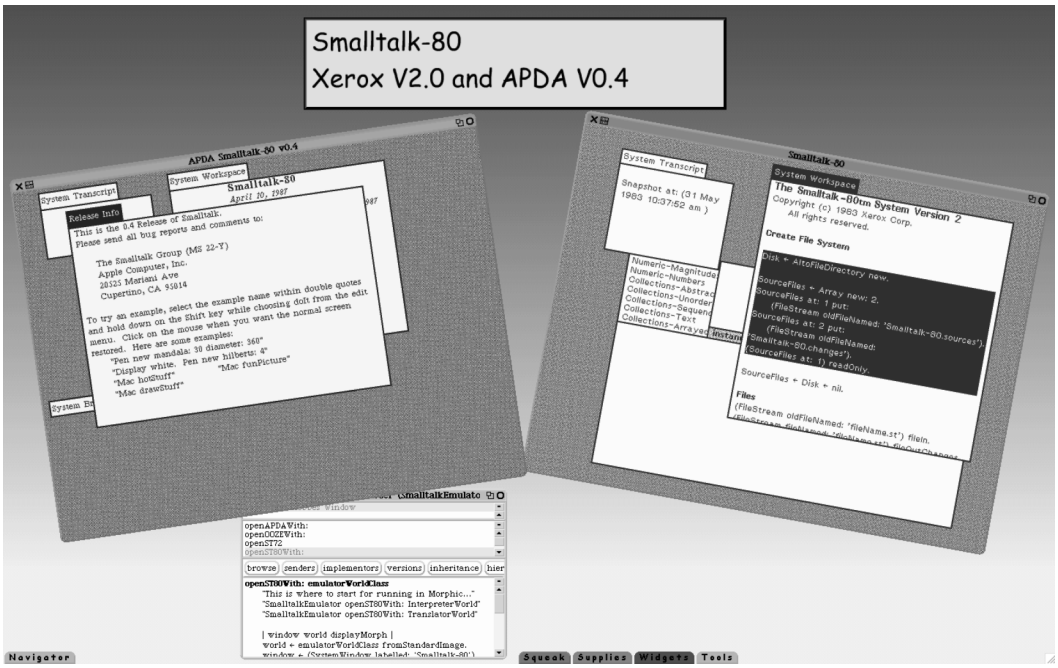


Fig. 44. The ParcPlace Version 2 release of Smalltalk-80, together with version 0.4 of the Apple APDA Smalltalk-80. Both are active in Squeak Morphic windows. To run this live, click the image or visit https://smalltalkzoo.thechm.org/HOPL-Squeak.html?earlyst80

The germ of the SqueakJS project began not long after I was hired at Sun Microsystems. I felt I should learn Java; casting about for a suitable project, I naturally chose to implement a Squeak VM. This I did; the result still appears to run at http://weather-dimensions.com/Dan/SqueakOnJava.jar. This VM is known in the Squeak community as "Potato" because of some difficulty clearing names with the trademark people at Sun. Much later, when I got the Smalltalk-72 interpreter running in JavaScript, Bert and I were both surprised at how fast it ran. Bert said, "Hmm, I wonder if it's time to consider trying to run Squeak in JavaScript." I responded with "Hey, JavaScript is pretty similar to Java; you could just start with my Potato code and have something running in no time."

"No time" turned into a bit more than a week, but the result was enough to get Bert excited. The main weakness in Potato had been the memory model, and Bert came up with a beautiful scheme to leverage the native JavaScript storage management while providing the kind of control that was needed in the Squeak VM. Anyone interested in hosting a managed-memory language system in JavaScript should read his paper on SqueakJS, presented at the Dynamic Languages Symposium [Freudenberg et al. 2014].

From there on Bert has continued to put more attention on performance and reliability, and SqueakJS now boasts the ability to run every Squeak image since the first release in 1996. To run the system live, visit this url: https://smalltalkzoo.thechm.org/HOPL-Squeak.html?launch

## B BOOTSTRAPPING OVER THE YEARS

### B.1 Smalltalk-72

Smalltalk-72 consisted of an interpreter and a text file (`ALLDEFS`[27]) with the initial code library. The interpreter was all written in Nova assembly code (which ran on the Alto), and comprised storage management, text and graphics routines, and the Smalltalk evaluator. When building a new system from scratch, the interpreter would parse incoming text from a bootstrap file named `ALLDEFS` into an array (a Smalltalk-72 "vector") of atoms (unique strings), numbers, and subarrays delimited by parentheses. A few definitions were hand-coded in the interpreter, such as `to`, which would assemble tokens into a function or class definition. With those minimal capabilities, the entire `ALLDEFS` file could be read in (with definitions for `string`, `vector`, `textframe`, `turtle`, `files`, etc.), and Smalltalk was ready to run. After a bit of testing, we would save the entire state as boot file so that it could be started up in its fully loaded state. Interested users are encouraged to start up the Smalltalk-72 simulation page which offers push-button access to the `ALLDEFS` bootstrap file. The entire file is only about 20 pages (38k), including comments and several pages of file and directory definitions for the Alto OS.

### B.2 Smalltalk-74

Smalltalk-74 was bootstrapped in a similar manner, from a file named `MDefs`.[28] That file devotes the first page or so to building a few classes such as `class`, `dictionary`, `for`, `function` and `arec` ("activation record," that is, stack frame), and populating their method dictionaries with basic methods. From that point on, `MDefs` is pretty similar to Smalltalk-72's `ALLDEFS`, except that it includes the basic protocol for access to BitBlt, and a rudimentary process scheduler for handling keyboard interrupts.

### B.3 Smalltalk-76

Smalltalk-76 began with a clear architectural plan. Our experience with Smalltalk-74 had taught us what was needed to support a nice personal computing environment with windows, menus, line and bitmap graphics, and basic text editing facilities. I wrote an entirely new bootstrap named `SysDefs`.[29] This file is comprehensive, with complete modern definitions of `Object` (inherited by all) `Class`, `Context`, `Number`, `Array`, `Stream`, `Point`, `Rectangle`, etc. It is the origin of the modern Smalltalk class library.

In Smalltalk-72 and Smalltalk-74, very little native code was required to read bootstraps; only enough to read code vectors, and to parse the arguments to the defining function `to`. Smalltalk-76, however, required a real compiler to read from the bootstrap definitions and create complete class structures with method dictionaries, byte-coded methods, and so on. The one tool at my disposal was Smalltalk-74, so I started by defining Smalltalk-74 proxies for each of the classes required to build the new image, called `GenClass`, `GenContext`, etc. Then I wrote a compiler in Smalltalk-74 that could compile the bootstrap method text into string objects that would become Smalltalk-76's bytecoded methods. Finally, Ted wrote a version of the VmemWriter that would create a fresh new OOZE file with all these proxy objects in the final format of a Smalltalk-76 image. The VmemWriter

[27] A copy of the file "ALLDEFS St-72" is available as auxiliary material for this paper.
[28] A copy of the file "MDefs St-74.pdf" is available as auxiliary material for this paper.
[29] A copy of the file "SysDefs St-76.pdf" is available as auxiliary material for this paper.

```
☞a← {nil}. a bitsinto 1 mem 67.
☞atomtable ← a[1].
a bitsinto 1 ⁻1.
Smalltalk define ☞USTable as atomtable.↩       "atom connexion"

textframe evals (   "unfinished business"
    STstyle's maxascent ← 11
    STstyle's maxdescent ← 3)↩

Smalltalk define ☞DefaultTextStyle as textframe evals (⇑STstyle).↩
Smalltalk define ☞mem as genCoreLocs 0 ⁻1.↩
Smalltalk define ☞kbMap as kmap.↩
Smalltalk define ☞CodeKeeper as stream of vector 4.↩
Smalltalk⊙☞CodeKeeper's (☞i ← ☞s swap i).↩ "new inst format"

"Make up a UserView with a new dispframe"
Smalltalk define ☞user as
    genUserView rectangle 0⊙0 576⊙768 20 16 1
    0 nil genDispframe (dispframe rectangle 10⊙10 300⊙200) nil.↩

"Make up an activation with the message ☞test"
☞Root← genContext. ☞superflag ← false↩
Root send ☞ev to Smalltalk⊙☞user from nil↩

    ☞ByteTrace ← false. dsoff.   "**only for simulation**"
    ☞Current ← Root. repeat do (Current step)

"Here are the oops needed by the microcode"
☞SpecialOops ← ☞( ↪(Object method ☞error) "error handler"
        ⁻1 0 1 2 10 ↪(nil) ↪(false) ↪(true)
        + − < > ≤ ≥ = ≠
        * / \ | min: max: land: lor:
        ⊙ ↪('⊙←'atomize) next ↪('next←'atomize) length ≡ ↪(nil) ↪
            (nil)
        class and: or: new new: to: oneToMeAsStream asStream).↩
```

Fig. 45. The file "launch.ft" set up the bootstrap image of Smalltalk-76 in Smalltalk-74, ready to be written as a new Smalltalk-76 image.

would also take any Smalltalk-74 object in the mix and write a clone of it directly into the new image. Finally I wrote Novacode and microcode for the VM, and we began the process of debugging the new system into existence.

Figure 45 shows the file "launch.ft"[30] that would set up a simulated image in Smalltalk-74, ready to be written as a new Smalltalk-76 image. Note the initial similarity to the earlier Smalltalks with a `dispframe` for text display, and a `Root` context that sends `ev` to `user`, an instance of `UserView`. The line

```
    ☞Current ← Root. repeat do (Current step)
```

[30] A copy of the file "launch.ft" is available as auxiliary material for this paper.

could be selected and executed in Smalltalk-74 to test the prenatal Smalltalk-76 before committing to run it with the real VM.

SpecialOops was a list of objects known by the virtual machine. These objects were chosen on a statistical basis to speed up common operations or to save space in the compiled code. The special constants enabled a single bytecode instruction to load that constant immediately, and with no need to provide space in the method for a literal object. This saved both time and space. Similarly, the special message selectors compiled to a single bytecode and required no extra literal reference in the method. These bytecodes were used for "high-frequency" constants and operations—those that were frequent both statically (number of occurrences in the codebase) and dynamically (number of executions per second). This special treatment improved both the compactness of the code and the speed of its execution.

Smalltalk-76 and its successors were themselves powerful development environments. Most of the evolution of these systems took place live in the system. With each notable set of improvements, after some testing, we would save a copy of the entire "image," and that would become the next release. From time to time we wanted to make an invasive change, such as changing the object format. This would be effected by running Ted's SystemTracer with transformation methods in place. We would save the system with a restart flag, transform the saved image, and then start it up again running new code based on the restart flag. From that time forward, we never built Smalltalk from a bootstrap, but instead maintained a fully running Smalltalk image along with this tool for writing out a mutated copy of the system.

### B.4 SystemTracer

The SytemTracer began as a tool (then called the VmemWriter) to effect the migration of an early Smalltalk-74 system into an OOZE virtual memory. Then, in the early days of Smalltalk-76 it became clear that we could make good use of the SystemTracer's ability to mutate one Smalltalk-76 to an altered form. Even with no mutation methods in place, the SystemTracer was often employed as an "overnight garbage collector" (at the time it could take over 6 hours to rewrite an entire Smalltalk system). This was valuable because normal work on the system could leave garbage that was not reclaimed by reference counting, such as killing an error in the debugger, or creating a circular structure without properly releasing it. Eventually Ted reworked the SystemTracer to write a new system directly from the predecessor system running live, rather than from an image file. This made it considerably faster.

The SytemTracer attended almost every birth of a new Smalltalk. It was used to write the non-OOZE object memory for the NoteTaker (Smalltalk-78), for the first Smalltalk-80 images, and especially for Squeak where we experimented with a number of compact memory formats. By that time, with live tracing, direct object pointers, and faster processors, the time required to create a mutated Smalltalk image was a matter of minutes or even seconds, depending on the size of the system being traced.

The most exciting use of the SystemTracer took place just days before we had to present the software Simulation Kit to Xerox executives. As we rehearsed things a week before the date, we found that we were running out of object space in OOZE. Not actually running out of space, but running out of "pclasses." OOZE was designed to partition Smalltalk's object space into 256 zones of 256 objects each. Each of these zones had instances all of the same class (and if variable, then all of a given octave size). We referred to these as "pseudoclasses," or "pclasses" for short. The full

Kearns curriculum stretched the limits of OOZE in this regard, and, if something weren't done, it began to look as if we might have Smalltalk crash in the middle of one of the most important demos in Parc's history.

Ted and I conferred about this, and it seemed that "all we would need to do" was to change to a mapping that allowed for 512 zones of 128 objects each, change the code that performed this mapping, and then, using the SystemTracer, read the whole old system in the old format, and write it in the new format. We started the SystemTracer off that night, and proceeded to edit the crucial mapping routines while it ran. In the morning we had a new image. When we loaded it up, it ran the first time. Once dangerously close to the 256-pclass limit, the new system used only about 350 of the 512 pclasses that were now available.

### B.5 Smalltalk-80

Smalltalk-80 essentially ***is*** Smalltalk-76, except for a number of cosmetic and a few not so cosmetic changes. There never was a file that suddenly made the changeover or that we would call a "bootstrap." Many of the changes were simply made live, after which we would save the new memory image.

More invasive changes, as before, were made using Ted's SystemTracer. By this time he had modified it so that it could run from within the image, and with the increasing system speed, it took only a couple of minutes to make even fundamental changes.

We have occasionally heard requests from others requesting something like a bootstrap for Smalltalk-80 (and Squeak as well). Something along the lines of "Well, can I just read in the sources file?" I have always felt that we should have had a good answer for that. When we built the Smalltalk-74 simulation of Smalltalk-76, a Smalltalk SystemDictionary was built with all the globals, and then the file "launch.ft" (shown in Figure 45) set up a "user" object that would run a REPL in a simple window. From that time forth, though, we never had a need to do this again. It would have been simple enough to do so, and it would have been useful for others to see this view of exactly what it takes, in addition to the system source file, to bring a modern Smalltalk to life. It would look very much like the Smalltalk-74's Launch file, together with a reader to compile the system sources into a new Smalltalk SystemDictionary.

### B.6 Squeak

The bootstrapping of Squeak is another topic altogether. In this case the challenge was how we could piece together a running system to get started. The main text details the process of running the Apple APDA image with a VM consisting of the reference interpreter running on a commercial Smalltalk. Once we had our translator running, it produced a performant version of the reference interpreter, and from that time on, Squeak was entirely self-supporting and able to evolve by changes made in the live system. It's true that some changes were not entirely "live," but for much of the time we still had a "SystemTracer" available that could write a new memory image out of a running system. This meant that if we wanted to change, say, how the bits were arranged in method headers, we would make the change in the SystemTracer, write a new image, make the change in the VM and then start it up with the altered image. The released Squeak memory system was the result of many such cycles of study and mutation. Much of the work that was done in Squeak to accommodate different word sizes, endianness, and color depths has this character of bootstrap programming, and the SystemTracer remains an invaluable tool.

## C PRIMITIVE ACCESS

No high-level language can produce practical results without making contact with basic computational or I/O entities. Each Smalltalk system had some way of doing this, generally for access to arithmetic, graphical, and user input functions. In Smalltalk-72, a reserved token, "`CODE`" was recognized by the interpreter, and it was typically followed by an integer indicating which of 50 or so native code routines should take over. Usually the native code performed some simple operation, such as fetching an argument, performing some arithmetic operation, and returning a result. In some cases, it might carry out the "isnew" clause as well, and in one particular case (string or vector recognizing a "[" token) it would return a value or proceed with either true or false back to the Smalltalk code. The file system and text display required so many primitive accesses that we used patterns like "4 `CODE` 51" or "6 `CODE` 51" just to keep the text and file dispatches together. As described in the main text, Smalltalk-74 code was nearly identical to that of Smalltalk-72, and the code escapes were also nearly identical. One exception is that in the case of text and files the multiple related dispatches were handled with a slightly different pattern, namely, eg, "`CODE 51 SUB 4`" or "`CODE 51 SUB 6`". Here, as in Smalltalk-72, the code escapes could be embedded either before or after normal code.

Smalltalk-76 introduced an entirely new interface to native code that was better suited to the higher performance expected of that system. Any method could include a phrase such as "primitive: 26" following its definition. This caused an immediate execution of the numbered primitive routine, which would often complete the method response and return without ever creating a context. In exceptional conditions the primitive could fail, causing the normal code body to run. Smalltalk-80 and Squeak follow almost the same convention as Smalltalk-76, except that they allow the insertion of primitive invocations, such as `<primitive: 26>`, *between* the method definition pattern and the code body. This has the somewhat more intuitive effect of indicating that the primitive will get control first, if it is present. This interface can be very effectively streamlined by including the primitive index (or even the address of the associated service routine) in the method lookup table.

Beginning with Smalltalk-80, we paid some attention to minimizing the number of essential primitives so that less effort was required to get Smalltalk running on a new machine or operating system. Non-essential primitives could simply fail, and the message would be handled in slower or more general Smalltalk code. Later on, these primitives could be implemented in native code to improve performance as desired.

## D THE SMALLTALK PARSE-TREE COMPILER

Smalltalk-76 introduced the modern Smalltalk keyword syntax. I wrote a compiler for it from scratch in Smalltalk-74, which was the system I had to work with at the time, and then transcoded that to Smalltalk-76 as things began running. It was simple and compact, but it offered no useful structure for extension to other features or variations.

Not long after we had Smalltalk-76 running, we began to want other features such as macro versions of various conditional and looping constructs. Larry Tesler took it upon himself to produce a new compiler built around a parse tree with nodes for all the Smalltalk syntax entities. While this may not have been quite as approachable as the original compiler, it was much better organized and made it possible to add these new features in an understandable manner. The Tesler compiler has remained at the heart of Smalltalk syntax experiments ever since.

Almost the first trick that Larry pulled off with the new compiler was to turn it around and make it into a decompiler by writing a parser for bytecoded methods that could regenerate a parse tree. Once you have a parse tree it is simple to decorate it with pretty-print methods to regenerate the Smalltalk code.

The decompiler made it possible to carry on development when, due to space or network limitations, it was not possible to access the full system source code base. We had a flexible strategy so that you could work with full sources for your particular project, and decompile when you needed to understand or debug code from elsewhere in the system. Because of Smalltalk's relatively verbose class and method names, decompiled code is very readable except for temporary variables being shown as, *e.g.*, `t1` and `t2`. Because *instance* variable names were retained, this was a minor shortcoming. Later, in Squeak, I added to the bytecode a compressed version of temporary names so that even these were preserved, while requiring almost no space. This gave us readable source code for the entire system.

Late in the era of Smalltalk-76, when we were using Dorados, it was clear that OOZE would not be adequate for the truly large applications that we could envision. This led Ted Kaehler to design a new virtual memory called LOOM, for Large Object-Oriented Memory. LOOM established "local" 16-bit address spaces (which we figured matched the "project" spaces in Smalltalk-76) embedded in a higher space with 32-bit inter-zone pointers. Ted wrote a simulation of the entire LOOM memory in Smalltalk-76, and was able to test it out that way. It was an interesting lash-up with a completely independent Smalltalk image simulating the LOOM code primitives. Ted wrote his code in a style that presumed all variables to be of fixed type, and he then used the decompiler logic to translate his Smalltalk code to BCPL, a C-like language supported on the Alto and Dorado. As each new feature was tested, it would be translated to BCPL and become part of the native virtual machine. The LOOM project was discontinued when competing 32-bit Smalltalk architectures started appearing, but the Smalltalk-to-C path for developing system facilities had been proven.

Around the same time, Alan Borning and I investigated a system for inferring the type of variables in Smalltalk methods [Borning and Ingalls 1982c]. Here we designed a set of methods for each of the parse tree nodes that constituted a system for abstract evaluation of Smalltalk. A given method could be evaluated for value, essentially interpreting the parse tree and producing the expected value object, or it could be evaluated for type (with types as input arguments as well), producing a type as its result. Having the analysis code as part of the living core of the system made it very easy to do experiments and gather statistics about many details, such as the tendency for types to "leak out" and reduce the entire inference system to knowing no more than that everything was an Object.

Much later, when building Squeak, we wished to generate a production virtual machine in C from our running and debugged Smalltalk reference interpreter. John Maloney adopted the same scheme to generate C code from the parse trees of the many methods of the reference interpreter. It was a very productive process, because we knew that the code in the parse tree was essentially bug free. When the code generator finally generated C code that worked, it *all* worked. It remained a live and malleable code body, so that when we wanted to try inlining certain methods, it could all be tested in Smalltalk and then translated to C and compiled with near certainty of success. The same was true of other experiments, such as whether to keep certain variables in registers.

Squeak eventually provided many such live views of code, all generated from the parse trees. These included the following:

Table 1. Available space in various generations of Smalltalk.

| System/year | Platform | Memory Architecture | Avail. space |
|---|---|---|---|
| Smalltalk-72 | Alto | 16-bit direct pointers, ref counts | 20k |
| Smalltalk-74 | Alto | OOZE: 16-bit hashed pointers; ref counts | 1.5Mb |
| Smalltalk-76 | Alto, Dorado | OOZE: 16-bit hashed pointers; ref counts | 1.5Mb |
| Smalltalk-78 | 8086 | 16-bit indirect pointers; ref counts | 200k |
| Smalltalk-80 | x86, 68K | 16-bit indirect pointers; ref counts or GC | 1.5Mb |
| Smalltalk-80 | multiple | Later 32-bit systems; direct and indirect; GC | >32Mb |
| Squeak(-96) | multiple | 32-bit direct pointers; generation GC | >32Mb |

- SourceSource—the actual stored text of the method
- prettyPrint—a conventionally formatted version of the stored text
- colorPrint—a conventionally colored and formatted version of the stored text
- altSyntax—a rendering in an alternate (colon-free) syntax experiment that Alan and I tried
- decompile—a prettyPrint version generated by decompiling the bytecoded method
- bytecodes—annotated bytecodes of the method
- tiles—draggable syntax tiles in the style of EToy scripts

## E SPEED AND SPACE OVER THE YEARS

Almost from the beginning, our work with Smalltalk was an attempt to answer the conceptual question: *Can we build a practical computer language around sending messages?* For that reason speed was not our first concern. Most important was to make it run; next, in the context of the Alto, was to provide enough space to do interesting work. Smalltalk-72 showed that a message-based system was possible, Smalltalk-74 provided enough space for interesting work, and Smalltalk-76 finally provided reasonable speed. From that point on, the main users of Smalltalk were either commercial or media-focused; both groups required large address spaces. Over the years, the Smalltalk-80 systems and Squeak grew to provide over 100 megabytes of live object space, without compromising the original goals of a simple language and relative compactness in the basic system.

### E.1 Sizes of Smalltalk Images

Table 1 shows the space available on various generations of Smalltalk systems, and Table 2 shows the number of classes and methods, and the sizes of the code, total image, and source.

The large "Squeak-large A" image reported in Table 2 was one of Alan Kay's demo images with all 78 projects loaded. It contained 729,000 objects at the time. The "Smalltalk-80-large" image is an industrial example running in VisualWorks Smalltalk, and containing over six million objects.

We were also interested in seeing how small we could make a Smalltalk image. "Squeak-min" in Table 2 is a Squeak 2.2 image that I stripped down by the iterative process of deleting classes and method that I didn't think were needed, and then mechanically removing all the classes and methods that were no longer accessible as a result.

Table 2. Sizes of various Smalltalk images.

| System | # classes[1] | # methods | code bytes | image bytes | source bytes |
|---|---|---|---|---|---|
| Smalltalk-72 | 20 | 200 | 15k | 30k | 38k |
| Smalltalk-74 | 20 | 200 | 15k | 30k | 41k |
| Smalltalk-76 | 50 | 500 | 20k | 100k | 93k |
| Smalltalk-78 | 82 | 1997 | 77k | 192k | ? |
| Smalltalk-80-V2 | 446 | 4495 | 159k | 681k | 1411k |
| Smalltalk-80-APDA | 472 | 4925 | 166k | 725k | 1476k |
| Smalltalk-80-large | 30080 | 314k | 10.67M | 57.1M | 316.7M |
| Squeak-min | 402 | 4592 | 240k | 602k | 856k |
| Squeak-large A[2] | 3186 | 26964 | 1519k | 121.9M | 8605k |
| Squeak-large B[3] | 5426 | 60592 | 6095k | 126.9M | 20.4M |

[1] For Smalltalk-80 and Squeak, this number includes metaclasses, so the number of user-facing classes is one half this figure
[2] Squeak 2.6 (1997) with many projects.
[3] Squeak 5.2 (2018) "all-in-one" image with 64-bit objects, source code, and VM; so not comparable with other images.

The Smalltalk-80 release image was around 1MB in size with 16-bit pointers. The Squeak object memory was so compact that I wanted to see if Squeak could also fit in 1MB, in spite of being a full 32-bit system. Indeed it could: the image was just over 600kB, including rich text editor, synthetic fonts, browser, debugger, and inspector. I added a feature that saved the names of temporary variables in compressed form, so the decompiled code was still extremely readable. The source code thus encoded in the image totaled 850kB: more than the size of the image itself! We joked that it was like source code compression with a complete Interactive Development Environment thrown in for free. Interested readers will want to try out this little image: https://smalltalkzoo.thechm.org/HOPL-Squeak.html?mini. Note that, although stored as a 640×480 monochrome display, this image can be immediately opened to 16- or 32-bit color, and 4k pixels, making it anything but a toy.


Fig. 46. Ceremonial luggage tag containing a Mitsubishi M32R/D chip similar to the chip that ran Squeak embedded in 1997.

The Squeak mini image was also the basis of a collaboration with Mitsubishi Research to put Squeak on their new M32R/D chip in 1997. The M32R/D chip was notable at the time for integrating dynamic RAM with a full CPU, and thus being capable of being a complete platform for Squeak and similar systems. John Maloney worked with an intern at Mitsubishi who implemented an adequate BIOS for the port. The whole Squeak group visited Mitsubishi Research in June of 1997 to see Squeak running on the chip shown in Figure 46. As part of the ceremony, Alan Kay was given the controls,

Table 3. An approximate comparison of the speed of various Smalltalk systems.

| System/year | Platform | Simple ops per second | Sends per second |
|---|---|---|---|
| Smalltalk-72 | Alto | 694 | 54 |
| Smalltalk-74 | Alto | 607 | 46 |
| Smalltalk-76 | Alto | 16.13k | 118 |
| Smalltalk-76 (est) | Dorado | 1.0M | 50k |
| Smalltalk-78 (est) | 5MHz 8086 | 30k | 250 |
| Squeak(1996) | 110MHz Power PC Mac 8100 | 4.1M | 175k |
| Squeak(2020) | 1.7GHz MacBook Pro Core i7 | 1.1G | 23.4M |
| Squeak(2020) | Same with 64-bit JIT | 3.7G | 290M |

and (with the aid of an external memory chip that anticipated their next-year's RAM capacity) was able to run a complete "Drive-a-car" EToys demo on that chip.

Craig Latta, a long-time Squeaker, has made images as small as 16k as part of his Spoon project. Such images are not full systems, but are minimal kernels sufficient to import the rest of a full system. Spoon is a project for creating minimal live object systems. It is motivated by a modularity effort called *Naiad* (for "Name And Identity Are Distinct") [Latta 2015], which seeks to make Smalltalk more understandable, collaborative, and deployable. Spoon has produced novel tools for code transfer and removal, and also for visualization of live systems.

### E.2 Performance Benchmarks

Early in the history of Squeak, we defined a simple measure of performance called `tinyBenchmarks`. The method `tinyBenchmarks`, in class `Integer`, times the execution of two other methods, `benchmark` and `benchFib`, and returns the results in terms of immediate operations (such as arithmetic and subscripting) per second, and in terms of the approximate number of activating message sends per second. Figure 47 presents the code for this benchmark. It would be a mistake to attribute a precise meaning to its results, but we have found them to be reasonable measure of *relative* performance of most Smalltalk virtual machines.

Although `tinyBenchmarks` was conceived in Squeak, it runs in Smalltalk-80, and I have done my best to back-port it to the earlier Smaltalks covered in this paper. The results are shown in Table 3. I had to take some liberties, such as using a string instead of a vector in `benchmark`, because Smalltalk-72 did not have enough space for a vector of 8192 elements. In the case of the Smalltalk-78 on the NoteTaker, with no hardware at my disposal, I have listed the overall performance as being roughly twice that of Alto Smalltalk-76. This is an estimate based on my recollections; the NoteTaker produced better numbers for send and return because of its optimized stack architecture. I have scaled the numbers for the Dorado to match my recollection that it ran around a million bytecodes per second. Interestingly, casual communication among VM implementers of the time (especially surrounding the public release of Smalltalk-80) characterized the performance of a virtual machines as, e.g., "1.6 Dorados," meaning 1.6 million bytecodes per second.

```
<Integer>tinyBenchmarks
    "Report the results of running the two tiny Squeak benchmarks."
    "0 tinyBenchmarks"
    | t1 t2 r n1 n2 |
    n1 ← 1.  "Double the argument to benchmark until >= 1000 milliseconds"
    [t1 ← Time millisecondsToRun: [n1 benchmark].
    t1 < 1000] whileTrue:[n1 ← n1 * 2]. "Note: #benchmark's runtime is
            about O(n)"

    n2 ← 28.  "Increment the argument to benchFib until >= 1000
            milliseconds"
    [t2 ← Time millisecondsToRun: [r ← n2 benchFib].
    t2 < 1000] whileTrue:[n2 ← n2 + 1].
    "Note: #benchFib's runtime is about O(k↑n),
        where k is the golden ratio = (1 + 5 sqrt) / 2 = 1.6180..."

    ↑ ((n1 * 500000 * 1000) // t1) printString, ' bytecodes/sec; ',
      ((r * 1000) // t2) printString, ' sends/sec'! !

<Integer>benchmark  "Handy bytecode-heavy benchmark"
    | size flags prime k count |
    size ← 8190.
    1 to: self do:
        [:iter |
        count ← 0.
        flags ← (Array new: size) atAllPut: true.
        1 to: size do:
            [:i | (flags at: i) ifTrue:
                [prime ← i+1.
                k ← i + prime.
                [k <= size] whileTrue:
                    [flags at: k put: false.
                    k ← k + prime].
                count ← count + 1]]].
    ↑ count

<Integer>benchFib  "Handy send-heavy benchmark"
    "(result // seconds to run) = approx calls per second"
    ↑ self < 2
        ifTrue: [1]
        ifFalse: [(self-1) benchFib + (self-2) benchFib + 1]
```

Fig. 47. The source code for tinyBenchmarks, used to compare the relative performance of Smalltalk virtual machines. It measures the rates of message send and of execution of immediate operations.

### E.3 High Performance

One of the benefits of the commercial release of Smalltalk was to put real financial incentive into work on Smalltalk implementations. From the earliest days of the Smalltalk-80 release, there was significant interest in the dynamic translation work of Peter Deutsch and Alan Schiffman [1984]. Their work produced a performant implementation on the Motorola 68020 microprocessor. This in

turn put Smalltalk in the hands of many more researchers because of the lower cost of hardware capable of running the system.

As ParcPlace Systems gathered momentum, they were able to pay for the development of a series of just-in-time (JIT) compilers that offered performance improved by factors of 3 to 5 over naïve implementations such as the one that I made available free from Apple's Programmers and Developers' Association (APDA).

With the development of Squeak, it became easier to experiment with JITs for Smalltalk. Several were produced by Ian Piumarta and Marcus Denker, but none was ever stably deployed. Later, the spin-out of Croquet to Quaq, and then TelePlace, motivated improved performance, and provided the capital to invest in it. This led to the excellent work on the Cog Smalltalk Virtual Machine by Eliot Miranda [Miranda 2011]. For more than a decade, the standard Squeak release has come with Cog, which has contributed greatly to Squeak's speed and stability. Squeak would not be the same without Eliot's work.

The Pharo branch of Squeak, overseen by Stéphane Ducasse https://pharo.org/ at INRIA, and Gilad Bracha's excellent work on NewSpeak https://newspeaklanguage.org/ at Cadence, have also afforded motivation and opportunity to tune the Squeak VM even further. Eliot's latest work with Clément Béra and Elisa Gonzalez Boix was reported at VMIL'18 [Miranda et al. 2018].

## F THE ALTO COMPUTER

An important part of the Parc zeitgeist was to be able to make and support about 100 of anything that was invented. Thus, most inventions weren't just "demos," but were engineered well enough to be replicable and usable tools. A big decision was to abandon calligraphic (line-drawing) displays in favor of an "omni-display" of a page's worth of pixels that could display anything (perhaps a little coarsely). We calculated that the minimum would be about 500k bits (808×606), which is a bit less than 64k bytes. Packaged memory for the Nova in the early 1970s was about 2.5¢/bit, so just the display memory would be about $12,500 in 1971 dollars (about $78,000 in 2020 dollars).

Many parts of the system had to be fast enough to paint an entire page of stylized text rapidly, synthesize many real-time voices of music, to run bit-map animations at "Disney frame rates" (about 10 frames/second), and especially to run efficient interpreters for the higher-level languages we planned to invent and use at Parc.

Chuck Thacker's solution was to use as much as possible of the first viable integrated circuit memory (the Intel 1103), at a little less than 1¢/bit, and design a "meta-computer" in which as many needs as possible could be emulated in microcoded software rather than hardware. The Alto made use of the Texas Instruments 74181 bit-slice processor chips that had just become available. Using a reloadable microstore of 1024 microinstructions, 5–6 microinstructions could be executed in one memory cycle.

The many different emulations that had to be carried out "concurrently" were facilitated by 16 interleaved program counters—tasks—with a small amount of "lookaside" hardware to decide which one to use to select the next microinstruction. These included emulators for a disk controller, a display controller, an input scanner for 1024 possible input lines, keyboard and mouse translators, a default "computer" emulator (most of a Data General Nova). There was even a software task to ping different parts of the Intel 1103 memory to keep the dynamic RAM chips from losing their contents! Soon added were emulators for graphics rendering, sound synthesis, 2.5D animations,

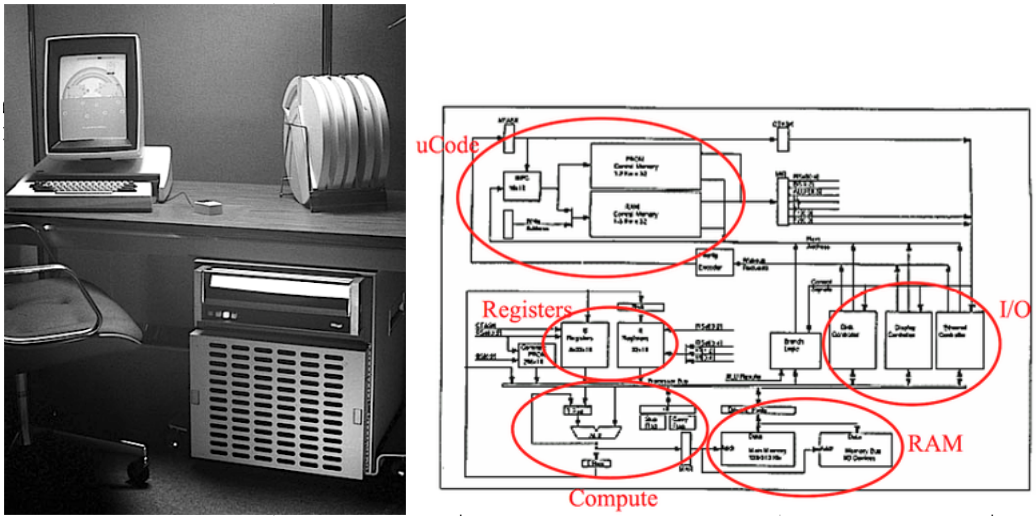


Fig. 48. The Alto. On the left we see a desk supporting the Alto display and keyboard, and a rack with five exchangeable disk packs. The Alto itself is the box under the desk that looks like a small refrigerator. On the right is a block diagram showing the major components of the Alto.

and virtual machines for high-level languages designed and built at PARC. More information about the Alto can be found in the Byte article by Thomas Wadlow [1981].

For many years, only 1024 microinstuctions were available, so not every emulator could be used at once. A lot of effort went into making efficient use of what was there. For example, the BitBlt screen-painting primitive used only 300 microinstructions, but was able to do all the painting and compositing processes needed for a complete graphics system. The guts of the first Smalltalk virtual machine emulator were about the same size.

Another limitation was the combination of only 64K of main memory for program and data storage. Secondary storage was a very slow Diablo Model 30 disk drive; this was really intended as a third-level file storage medium, not a swapping disk. Nonetheless, Smalltalk, which acted as its own operating and development system, did the best it could with cleverness in its OOZE virtual memory [Kaehler 1981].

## REFERENCES

Ronald M. Baecker. 1976. A Conversational Extensible System for the Animation of Shaded Images. In *Proceedings of the 3rd Annual Conference on Computer Graphics and Interactive Techniques* (Philadelphia, Pennsylvania) (*SIGGRAPH '76*). Association for Computing Machinery, New York, NY, USA, 32–39. https://doi.org/10.1145/563274.563281

Alan Borning. 1979. *Thinglab—a constraint-oriented simulation laboratory.* Ph.D. Dissertation. Stanford University, Dep. Computer Science (March).

Alan Borning. 1981. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *TOPLAS* 3, 4, 353–387.

Alan H. Borning and Daniel H. H. Ingalls. 1982a. *Multiple Inheritance in Smalltalk-80.* Technical Report Technical Report 82-06-02. University of Washington, Department of Computer Science (June).

Alan H. Borning and Daniel H. H. Ingalls. 1982b. Multiple Inheritance in Smalltalk-80. In *Proceedings of the Second AAAI Conference on Artificial Intelligence* (Pittsburgh, Pennsylvania) (*AAAI'82*). AAAI Press, 234–237.

Alan H. Borning and Daniel H. H. Ingalls. 1982c. A Type Declaration and Inference System for Smalltalk. In *Conference Record of the Ninth ACM Symposium on Principles of Programming Languages*. ACM Press, Albuquerque, NM, USA, 133–141.

L. Peter Deutsch. 1973. A LISP Machine with Very Compact Programs. In *Proc. 3rd International Joint Conference on Artificial Intelligence* (Stanford, USA) (*IJCAI'73*). Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 697–703. http://www.softwarepreservation.org/projects/LISP/interlisp-d/Deutsch-3IJCAI.pdf

L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *Conference Record of the Eleventh ACM Symposium on Principles of Programming Languages*. Salt Lake City, 297–302.

Doug Englebart Institute. 2008. The Demo @ 50. https://thedemoat50.org NON-ARCHIVAL.

Douglas C. Engelbart and William K. English. 1968. A Research Center for Augmenting Human Intellect. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I* (*AFIPS '68 (Fall, part I)*). Association for Computing Machinery, New York, NY, USA, 395–410.

Bert Freudenberg, Dan H.H. Ingalls, Tim Felgentreff, Tobias Pape, and Robert Hirschfeld. 2014. SqueakJS: A Modern and Practical Smalltalk That Runs in Any Browser. In *Proc. 10th Symposium on Dynamic Languages on Dynamic Languages* (*(DLS'14)*). ACM, New York, NY, USA (Oct.), 57–66. SIGPLAN Not. 50*(2)*.

A.J. Goldberg. 1983. *Smalltalk–80: The Interactive Programming Environment.* Addison-Wesley, Reading, MA (May). xi+516 pages.

Adele Goldberg and David Robson. 1983. *Smalltalk–80: The Language and its Implementation.* Addison-Wesley. xx+714 pages.

Ira P. Goldstein and Daniel G. Bobrow. 1981. *PIE: An Experimental Personal Information Environment.* Technical Report Technical Report CSL-81-4. Xerox Palo Alto Research Center.

Daniel Ingalls, Tim Felgentreff, Robert Hirschfeld, Robert Krahn, Jens Lincke, Marko Röder, Antero Taivalsaari, and Tommi Mikkonen. 2016. A World of Active Objects for Work and Play: The First Ten Years of Lively. In *Proc 2016 ACM Int. Symp. New Ideas, New Paradigms, and Reflections on Programming and Software* (Amsterdam, Netherlands) (*Onward! 2016*). ACM, 238–249. https://doi.org/10.1145/2986012.2986029 More information on Lively can be found at https://lively-web.org/welcome.html.

Dan Ingalls, Bert Freudenberg, Ted Kaehler, Yoshiki Ohshima, and Alan Kay. 2014. Reviving Smalltalk-78: The First Modern Smalltalk Lives Again. In *Proc 6th edition International Workshop on Smalltalk Technologies* (*IWST*), Alain Plantec and Jannik Laval (Eds.). 109–118. http://freudenbergs.de/bert/publications/Ingalls-2014-Smalltalk78.pdf

Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, A Practical Smalltalk Written in Itself. In *Proceedings Twelfth ACM Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM Press, Atlanta, 318–324.

Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. 1988. Fabrik: A Visual Programming Environment. *SIGPLAN Not.* 23, 11 (Nov.), 176–190. https://doi.org/10.1145/62084.62100

Daniel H. H. Ingalls. 1978. The Smalltalk-76 Programming System: Design and Implementation. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Tucson, Arizona) (*POPL '78*). Association for Computing Machinery, New York, NY, USA, 9–16. https://doi.org/10.1145/512760.512762

Daniel H. H. Ingalls. 1981. The Smalltalk Graphics Kernel. *Byte* 6, 8, 168–194. https://archive.org/details/byte-magazine-1981-08/page/n181/mode/2up

Daniel H. H. Ingalls and Daniel H. H. Ingalls, Jr. 1985. The "Mahābhārata": Stylistic Study, Computer Analysis, And Concordance. *Journal of South Asian Literature* 20, 1, 17–46. http://www.jstor.org/stable/40872708

Kenneth E. Iverson. 1962. *A Programming Language.* Wiley. (4th Printing, May 1967).

Ted Kaehler. 1981. Virtual Memory for an Object-Oriented Language. *Byte* 6, 8 (August), 378–387.

Alan C. Kay. 1972. A Personal Computer for Children of All Ages. In *Proceedings of the ACM Annual Conference - Volume 1* (Boston, Massachusetts, USA) (*ACM '72*). Association for Computing Machinery, New York, NY, USA, Article 1, 11 pages. https://doi.org/10.1145/800193.1971922

Alan C. Kay. 1977. Microelectronics and the Personal Computer. *Scientific American* 237, 3 (September), 230–245. https://www-jstor-org.proxy.lib.pdx.edu/stable/24920330

Alan C. Kay. 1993. The Early History of Smalltalk. In *The Second ACM SIGPLAN Conference on History of Programming Languages*. ACM Press, New York, NY, USA, Chapter XI, 511–598.

Glenn Krasner. 1983. *Smalltalk–80: Bits of History, Words of Advice.* Addison-Wesley Publishing Company, Reading, Massachusetts.

Wilf R. Lalonde and John R Pugh. 1991. *Inside Smalltalk.* Vol. II. Prentice-Hall International, Englewood Cliffs, New Jersey.

B. W. Lampson, W. W. Lichtenberger, and M. W. Pirtle. 1966. A user machine in a time-sharing system. *Proc. IEEE* 54, 12 (Dec), 1766–1774. https://doi.org/10.1109/PROC.1966.5260

Craig Latta. 2015. a detailed Naiad description. January 2015. https://web.archive.org/web/20161113033728/https://thiscontext.com/naiad/

Frank Ludolph, Yu-Ying Chow, Dan Ingalls, Scott Wallace, and Ken Doyle. 1988. The Fabrik Programming Environment. In *IEEE Workshop on Visual Languages*. Pittsburgh, PA, USA, 222–230. https://doi.org/10.1109/WVL.1988.18032

John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Comm. ACM* 3, 4 (April), 184–195. https://doi.org/10.1145/367177.367199

John McCarthy. 1978. *History of LISP*. Association for Computing Machinery, New York, NY, USA, 173–185. https://doi.org/10.1145/800025.1198360

John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I Levin. 1965. *LISP 1.5 Programmer's Manual* (2 ed.). MIT Press. http://www.softwarepreservation.org/projects/LISP/book/LISP%201.5%20Programmers%20Manual.pdf

Eliot Miranda. 2011. The Cog Smalltalk virtual machine — writing a JIT in a high-level dynamic language. In *Proc. 5th Workshop on Virtual Machines and Intermediate Languages* (*VMIL 2011*). 7. https://zenodo.org/record/3700608 Presented at the workshop, but not archived in the ACM Digital Library.

Eliot Miranda, Clément Béra, Elisa Gonzalez Boix, and Dan Ingalls. 2018. Two Decades of Smalltalk VM Development: Live VM Development through Simulation Tools. In *Proc. 10th ACM SIGPLAN Int. Workshop on Virtual Machines and Intermediate Languages* (Boston, MA, USA) (*VMIL 2018*). Association for Computing Machinery, New York, NY, USA (November), 57–66. https://doi.org/10.1145/3281287.3281295 HAL Id: hal-01883380.

Kristen Nygaard. 1970. *System description by SIMULA — An introduction*. Technical Report S-35. Norsk Regnesentral / Norwegian Computing Center (November).

Kristen Nygaard and Ole-Johan Dahl. 1981. The development of the SIMULA languages. In *History of programming languages I*, Richard L. Wexelblat (Ed.). ACM, New York, NY, USA, Chapter IX, 439–480. https://doi.org/10.1145/800025.1198392

Sandra Pakin. 1968. *APL\360 Reference Manual*. Science Research Associates (SRA).

Jeff Pierce. 2002. Alice in a Squeak Wonderland. In *Squeak: Open Personal Computing and Multimedia*, Mark Guzdial and Kim Rose (Eds.). Prentice Hall, Chapter 3, 69–96.

Trygve Reenskaug. 1979. *Models–Views–Controllers*. Technical Note. Learning Reseearch Group, Xerox Palo Alto Research Center (December). 2 pages. https://doi.org/10.5281/zenodo.3676092 Also at NON-ARCHIVAL http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf

A. Dain Samples, David Ungar, and Paul Hilfinger. 1986. SOAR: Smalltalk without bytecodes. *SIGPLAN Not.* 21, 11 (June), 107–118. https://doi.org/10.1145/960112.28708

Steve Saunders. 1977. Improved FM audio synthesis methods for real-time digital music generation. *Computer Music Journal* 1, 1 (February), 53–55.

D. V. Schorre. 1964. META II a Syntax-Oriented Compiler Writing Language. In *Proc. 1964 19th ACM National Conf.* (*ACM '64*). Association for Computing Machinery, New York, NY, USA, 41.301–41.3011. https://doi.org/10.1145/800257.808896

Ernst Friedrich Schumacher. 1973. *Small is Beautiful: A Study of Economics as if People Mattered*. Blond & Briggs.

D. A. Smith, A. Kay, A. Raab, and D. P. Reed. 2003. Croquet—a collaboration system architecture. In *First Conference on Creating, Connecting and Collaborating Through Computing (C5)*. (Jan), 2–9. https://doi.org/10.1109/C5.2003.1222325

Gary Starkweather. 1997. Birth of the Laser Printer (video; NON-ARCHIVAL). https://youtu.be/BZFaQiItckU (published 27 Oct. 2017; 80 minutes; retrieved 20 Feb. 2020). https://archive.computerhistory.org/resources/access/text/2017/10/102738575-05-01-acc.pdf (transcript, titled "Birth of the laser print [*sic*]," created Jan. 1999; CHM Reference number 102738575; retrieved 20 Feb. 2020).

Physicist Gary Starkweather discusses his work developing the laser printer at Xerox PARC in the early 1970s. Beginning with a discussion of his time at Bausch + Lomb, Starkweather describes how Xerox's earlier innovations, including the xerographic process, influenced the invention of the laser printer, initially at Xerox's Rochester headquarters, and later at PARC. Starkweather illustrates the printing methods used prior to the laser printer, the challenges he faced in developing it, and the methods he used to overcome them.

Collection of the Computer History Museum, Mountain View, California, USA.

Dave Thomas. 1995. Celebrating 25 Years of Smalltalk. *Mojowire*. http://www.mojowire.com/TravelsWithSmalltalk/DaveThomas-TravelsWithSmalltalk.htm (also at Internet Archive 12 June 2013 05:51:49).

David Ungar and Randall B. Smith. 1987. Self: The Power of Simplicity. In *Proceedings OOPSLA'87 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*. 227–242.

Thomas A. Wadlow. 1981. The Xerox Alto Computer. *Byte* 6, 9 (September), 58–68. https://archive.org/details/byte-magazine-1981-09/page/n58