2

The Evolution of the Smalltalk Virtual Machine

Daniel H. H. Ingalls
Software Concepts Group
Xerox Palo Alto Research Center
Palo Alto, California

Introduction

In this paper we record some history from which the current design of the Smalltalk-80 Virtual Machine springs. Our work over the past decade follows a two- to four-year cycle that can be seen to parallel the scientific method and is shown in Fig. 2.1. The paper appears in two

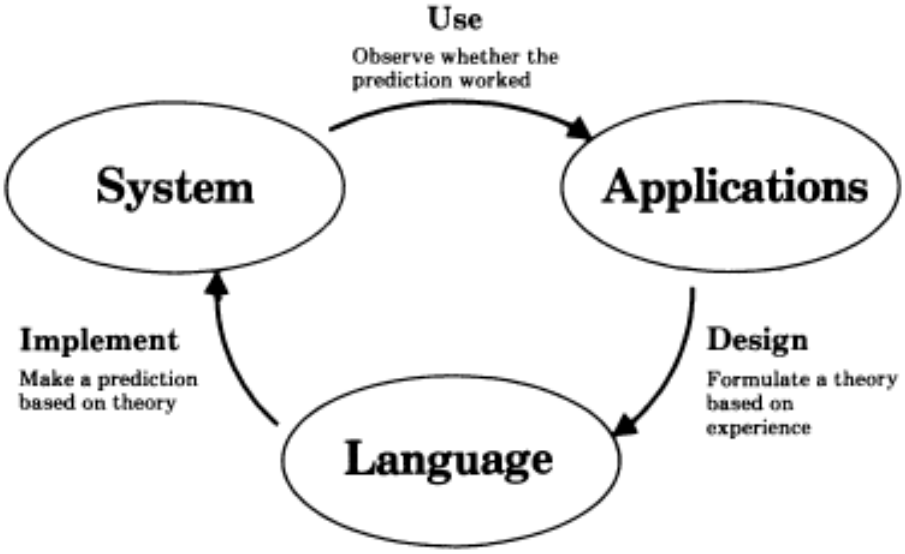

Figure 2.1

sections that are relatively independent of one another. The first section traces the evolution of the current design from the perspective of form following function. It follows the major implementation challenges and our solutions to them. The second section relates some of the *methodology* which evolved in pursuing this cycle of reincarnation. Readers who are less interested in the details of Smalltalk can skip to the second section and interpret our experience relative to other programming languages and systems.

Form Follows Function

From the first Smalltalk interpreter to the definition of the Smalltalk-80 Virtual Machine, the Smalltalk language has been characterized by three principal attributes:

- Data stored as objects which are automatically deallocated,
- Processing effected by sending messages to objects,
- Behavior of objects described in classes.

In spite of other opinions to the contrary, we consider these to be the hallmarks of the "object-oriented" style of computing. In this section we shall trace the evolution of the underlying machinery which has supported language systems in this style over the last ten years. Some of the changes have augmented the power of the language, and some have increased its efficiency. Each change can be seen as an attempt to bring the underlying machinery more into harmony with the day-to-day demands of object-oriented programming.

Smalltalk-72

The very first Smalltalk evaluator was a thousand-line BASIC program which first evaluated 3+4 in October 1972. It was followed in two months by a Nova assembly code implementation which became known as the Smalltalk-72 system[1].

☐ *Storage Management* Objects were allocated from a linked list of free storage using a first-fit strategy. Objects which were no longer accessible were detected by reference-counting. They were then returned to the free storage list, with adjacent entries being automatically coalesced. Since pointers were direct memory addresses, compaction would have been complicated, and was not attempted. Contexts, the suspended stack frames, were managed specially as a stack growing down from high memory while normal allocation grew up from low memory. This separation reduced the tendency to leave "sandbars"

when returning values from deep calls, a problem in the absence of compaction.

☐ *Token Representation of Code* All code was stored in a single tree made up of instances of Array (it was called Vector then), a variable-length array of pointers. The code in this tree represented a pattern description, similar to Meta. Fig. 2.2 presents the Smalltalk-72 definition of a class of dotted-pair objects, followed by a few examples of its use. Responses printed by the system are underlined.

```
to pair                                  to is the defining word, as in LOGO.
   | head tail                           | declares instance variable names.
   (isnew ⇒                              isnew is true if an instance is just created.
      (" head ← :. " tail ← :)           " means literally the next token, here the
   ◁ head ⇒                                 names head and tail.
      (◁ ← ⇒                               ← is a message like any other.
         (" head ← :)                    : fetches the next value from the incoming.
      ↑ head)                                message stream.
   ◁ tail ⇒                              ◁ matches the next literal token
      (◁ ← ⇒                                like the Smalltalk-80 message
         (" tail ← :)                       peekFor:
      ↑ tail)                            false ⇒ (body) does nothing, but
   ◁ print ⇒                             true ⇒ (body) evaluates the body, and
      (" [ print. head print.                then leaves the outer scope.
       " . print.                            In this way several such constructs
      tail print. " ] print))                work as a CASE statement.
pair

" a ← pair 2 5                           Here a pair is created, called a.
[2.5]
a tail ← pair 3 7                        a gets its tail changed.
[2.[3.7]]
b tail head                              a's tail (=[3.7]) gets the message head.
3
```

Figure 2.2

The code was viewed by the interpreter as simply a stream of tokens. The first one encountered was looked up in the dynamic context, to determine the receiver of the subsequent message. The name lookup began with the class dictionary of the current activation. Failing there, it moved to the sender of that activation and so on up the sender chain. When a binding was finally found for the token, its value became the receiver of a new message, and the interpreter activated the code for that object's class.

In the new context, the interpreter would begin executing the receiver's code, matching it with the token stream which followed the original occurrence of the receiver. Various matching operators would select a route through the code which corresponded to the message pattern encountered. The matching vocabulary included matching a literal token, skipping a token, picking up a token literally, and picking up the value of a token. The latter operation invoked the same dynamic lookup described above for the receiver.

☐ *Classes* Most class properties were stored in a single dictionary. Instance variable names were denoted by a special code which included their offset in the instance. Class variables appeared in the same dictionary as normal name/value pairs. Another entry gave the size of instances, and another gave the "code" for the class. When a class was mentioned in code, it would automatically produce a new instance as a value. The unfortunate result of this special behavior was to make classes unable to be treated as ordinary objects.

☐ *Applications* Smalltalk-72 was ported to the Alto[2] as soon as the first machines were built, and it provided a stable environment for experimentation over the next few years. The Alto provided a large bitmap display and a pointing device, and thus made an ideal vehicle for working with graphical user interfaces.

Development of the Smalltalk-72 system began with Textframes and Turtles. Textframes provided text display with independent composition and clipping rectangles; Turtles gave us line drawing capability, modeled after Papert's experiments with turtle geometry[3]. In both cases, Smalltalk's ability to describe multiple instances added considerable leverage to these primitive capabilities. Soon many interesting and useful applications were written, including a mouse-driven program editor, a structured graphics editor, an animation system and a music system. Finally, Smalltalk-72 served as the basis for an experimental curriculum in object-oriented computing for high-school children[4].

☐ *Shortcomings* The Smalltalk-72 system was used heavily by a dozen people for four years. The many practical applications gave us a lot of experience with the power of classes and the message-sending metaphor. In the course of this work, we also became increasingly aware of many limitations in the Smalltalk-72 system.

Dynamic lookup of tokens was both inefficient and unmodular. The dynamic lookup tempted some programmers to write functions which "knew" about their enclosing context. This code would then cause subtle errors when apparently innocent changes were made in the outer level.

The message stream model was complicated and inefficient. One could not tell what a given piece of code meant in isolation. This became a problem as we attempted to build larger systems in which modularity was more of an issue. Also, a considerable amount of time was wasted on execution-time parsing (determining whether the result of a receiver expression should gobble the next token in the execution stream).

As mentioned above, classes were not first-class objects. Also, as our experience increased, we came to realize the need for inheritance. This was felt first in the commonality of behavior of Arrays, Strings, and sub-Arrays. For the time being, we referred to common functions from these similar classes so as to factor the behavior, but it was clear that some sort of inheritance mechanism was needed.

'ltalk-74

In 1974 we produced a major redesign of the Smalltalk interpreter with the aim of cleaning up its semantics and improving its performance. While the redesign was a mixed success, Smalltalk-74 was the site of several advances which fed into the later systems.

☐ *Message Stream Formalism* We succeeded in formalizing the operation of the interpreter, a step in the direction of simplicity and generality. For instance, we were able to provide a programmer-accessible object which represented the incoming message stream. Thus, not only could all the message stream operations be examined in Smalltalk, but the user could also define his own extensions to the message stream semantics. While this was a local success, it did not solve either of the real problems: token interpretation overhead, and non-modularity of receiver-dependent message parsing.

☐ *Message Dictionaries* Classes were given a message dictionary to allow primary message matching to be done by hashing. This did not do much for execution speed, since the previous interpreter had tight code for its linear scan, but it did help compilation a great deal since a single message could be recompiled without having to recompile all the code for the class. Unfortunately classes were still not able to be treated as normal objects.

☐ *BitBlt* Smalltalk-74 was the first Smalltalk to use BitBlt as its main operation for bitmap graphics. The specification for BitBlt arose out of earlier experience with Turtle graphics, text display, and other screen operations such as scrolling and menu overlays. Our specification of BitBlt has been used by others under the name RasterOp[5]. While the general operation was available to the Smalltalk programmer, much of the system graphics were still done in machine-coded primitives, owing to inadequate performance of the token interpreter.

☐ *OOZE* Smalltalk-74 was the system in which the OOZE ("Object-Oriented Zoned Environment")[6] virtual memory was first implemented. OOZE provided uniform access to 65K objects, or roughly a million words of data. Smalltalk-74 served as the development environment for OOZE, so that when Smalltalk-76 was designed, OOZE was debugged and ready for use.

☐ *Applications* In addition to the previous applications which we had developed, Smalltalk-74 served as host to an information retrieval system and complete window-oriented display interface. Owing to the virtual memory support, it was possible to integrate many functions in a convenient and uniform user interface.

Smalltalk-76

In 1976 we carried out a major redesign of the Smalltalk language and implementation[7]. It addressed most of the problems encountered in the previous four years of experience with Smalltalk:

- Classes and contexts became real objects;
- A class hierarchy provided inheritance;
- A simple yet flexible syntax for messages was introduced;
- The syntax eliminated message stream side-effects and could be compiled;
- A compact and efficient byte-encoded instruction set was introduced;
- A microcode emulator for this instruction set ran 4 to 100 times faster than previous Smalltalks; and
- OOZE provided storage for 65K objects—roughly the capacity of the Alto hardware.

The design for this system was completed in November of 1976 and seven months later the system was working. This included a full rewrite of all the system class definitions.

Experience with Smalltalk-76

The Smalltalk-76 design stood the test of time well. It was used for four years by 20 people daily and 100 people occasionally. A large portion of the design survives unchanged in the Smalltalk-80 system. However, the Smalltalk-76 design did have some snags which we encountered during our four-year experience.

☐ *Block Contexts* Smalltalk-76 had to provide a mechanism for passing unevaluated code which was compatible with a compiled representation. A syntax was devised which used open-colon keywords for

passing unevaluated expressions (the semantics were the same as the square bracket construct in the Smalltalk-80 language). This approach was supported by block contexts which allowed executing code remotely. Since the Smalltalk-76 design had no experience to draw from, it was weak in several areas.

One problem which was discovered in the process of supporting error recovery was that block contexts could not be restarted because they did not include their initial PC as part of their state. This was not normally needed for looping, since all such code fragments ended with a branch back to the beginning. Happily, we were able to fix this by defining a new subclass.

Two other problems were discovered with remote contexts when users began to store them as local procedures. For one thing, there was no check in the interpreter to recover gracefully if such a piece of code executed a return to sender after the originating context had already returned. Also, the system could crash if remote contexts were made to call one another recursively, since they depended on their home context for stack space, rather than having their own stack space.

There were two other weaknesses with remote code. There was an assymmetry due to use of open-colon keywords. For example one would write

```
newCursor showWhile: [someExpression]
```

to cause a different cursor to appear during execution of someExpression. But if the code contained a variable, *action*, which was already bound to remote code, one wanted that variable to be passed directly, as with a closed-colon keyword. The only way to handle this without needing a pair of messages with and without evaluation was to write

```
newCursor showWhile: [action eval].
```

This would do the right thing, but caused an extra remote evaluation for every level at which this strategy was exercised. Besides being costly, it was just plain ugly.

Another weakness of remote contexts was that, while they acted much like nullary functions, there was no way to extend the family to functions which took arguments.

Finally, there was a question about variable scoping within remote code blocks. Smalltalk-76 had no scoping, whereas most other languages with blocks did.

All of these problems with RemoteContexts were addressed one way or another in the Smalltalk-80 design.

☐ *Compilation Order* The Smalltalk-76 interpreter assumed that the receiver of a message would be on the top of the execution stack, with arguments below it. The number of arguments was not specified in the "send" instruction, but was determined from the method header after message lookup. From the designer's perspective this seemed natural; the only other reasonable choice would be for the receiver to lie underneath the arguments, as in the Smalltalk-80 system. In this case it seemed necessary to determine the number of arguments from the selector in order to find the receiver in the stack, and this looked both complex and costly to do at run time. There were two problems with having the receiver on the top of the stack. First the compiler had to save the code for the receiver while it put out the code for the arguments. This was no problem for three of the compilers which we built, but one particularly simple compiler design foundered on this detail. The second problem with post-evaluation of receivers was that the order of evaluation differed from the order of appearance in the code. Although one should not write Smalltalk code which depends on such ordering, it did happen occasionally, and programmers were confused by the Smalltalk-76 evaluation scheme.

☐ *Instruction Set Limitations* The Smalltalk-76 instruction set was originally limited to accessing 16 instance variables, 32 temps, and 48 literals. These limits were both a strain on applications and on the instruction set. A year later we added extended instructions which relieved these limits. This was important for applications, and it also took pressure off the future of the instruction set. With extended codes available, we had the flexibility to change the instruction set to better reflect measured usage patterns. For example we found that we could get rid of the (non-extended) instructions which accessed literals 33-48, because their usage was so low. Such measurements led us eventually to the present Smalltalk-80 instruction set.

Experience with OOZE

☐ *Address Encoding* In OOZE, object pointers encoded class information in the high 9 bits of each pointer. This had the benefit of saving one word per object which would have been needed to point to the class of the object. In fact, it actually saved two words on many objects because classes contained the length of their instances. Variable length objects had separate class-parts for common lengths (0 through 8). However, the address encoding had several weaknesses. It squandered 128 pointers on each class, even though some never had more than a couple of instances. It also set a limit on the number of classes in the system (512). This did not turn out to be a problem, although an earlier limit of 128 did have to be changed. Finally, owing to the encoding, it was not possible to use object pointers as semantic indirection. For this reason, Smalltalk-76 could not support become: (mutation of objects through pointer indirection) as in later Smalltalks.

☐ *Capacity* While the OOZE limitation of 65K objects is small by today's standards, it served well on the Alto. The Alto has a 2.5 megabyte disk, and with a mean object size of 16 words, OOZE was well matched to this device.

☐ *Interpreter Overhead* OOZE had a couple of weaknesses in the area of performance, which only became significant after our appetites had increased from several years' experience. One was that the object table required at least one hash probe for every object access, even just to touch a reference count. Another was a design flaw in the management of free storage which required going to the disk to create a new temporary object if its pointer had been previously placed on a free list. We designed a solution to both of these problems. Temporary objects would be treated specially with pointers which were direct indexes into their object table. Freelists would only be consulted when an object "matured" and needed a permanent pointer assigned. Because temporary objects account for many accesses, much of the overhead of probing the permanent object table would be eliminated. Since Smalltalk-76's days seemed numbered, we did not take the time to implement this solution.

Efficiency and Portability: Smalltalk-78

In 1977 we began a project to build a portable computer capable of running the Smalltalk system. Known internally as NoteTaker, it began as a hand-held device for taking notes, but ended up as a suitcase-sized Smalltalk machine. Several factors converged to define this project. We wanted to be able to bring Smalltalk to conferences and meetings to break through the abstractions of verbal presentations. With the Intel 8086 and other 16-bit microprocessors (the Z8000 and MC68000 were coming, but not available yet), we felt that enough computing power would be available to support Smalltalk, even without microcode. Finally, portability seemed to be an essential ingredient for exploring the full potential of personal computing.

The design challenge was significant. We were moving to an environment with less processing power, and the whole system had to fit in 1/4 Mbyte, since there was no swapping medium. Also we faced transporting 32K bytes of machine code which made up the Smalltalk-76 system, and it seemed a shame not to learn something in the process. The result of these forces was the design of Smalltalk-78.

☐ *Cloned Implementation* The Smalltalk-78 implementation was significant in that it was not built from scratch. We were happy enough with the basic model that we transported the entire Smalltalk level of the system from Smalltalk-76. In order to do this, we used the system tracer (see p. 24) which could write a clone of the entire system onto an image file. This file could then be loaded and executed by the Smalltalk-78 interpreter. The tracer had provisions in it for transmut-

Figure 2.3

ing object formats as necessary, and even for changing the instruction set used in the methods.

☐ *Indexed OT* The Smalltalk-78 design centered around an indexed object table, which is the same design as in the Smalltalk-80 system. This greatly simplified object access and yet retained the indirection which made for easy storage management in Smalltalk-76. Reference counts were stored as one byte of the 4-byte table entry. Given an object pointer in a register, a reference count could be incremented or decremented with a single add-byte instruction with an overflow check.

☐ *Small Integers* Since there would not be room in core for more than 10K objects or so, it was possible to encode small integers (-16384 to 16383) in part of the pointer space. Since object table indices would all be even (on the 8086, they were multiples of 4), we encoded small integers as two's complement integers in the high-order 15 bits, with the low-order bit turned on. With this design, allocation of integer results was trivial, integer literals could be stored efficiently, and integer values did not need to be reference counted.

☐ *In-line Contexts* In order to save time allocating new contexts, and to take advantage of the stack-oriented instructions available in most microprocessors, the representation of contexts was redesigned. Instead of having a separate object for each context, a large object was allocated for each process, in which contexts could be represented as conventional

stack frames. This special representation complicated the handling of blocks and the debugger, requiring an interface which referred to the process and an offset within the process.

In addition to reduced allocation time, the time to transfer arguments was eliminated by allowing the contexts to overlap; the top of one context's stack (receiver and arguments) was the base of the next context's frame.

☐ *Reduced Kernel—The Leverage of BitBlt* We have always sought to reduce the size of the Smalltalk kernel. This is not only an aesthetic desideratum; kernel code is inaccessible to the normal user, and we have always tried to minimize the parts of our system which can not be examined and altered by the curious user. In this particular case, we were also moving to a new machine. While writing a certain amount of machine code seemed inevitable, we did not relish the idea of transcribing all 32K bytes of code which comprised the Smalltalk-76 kernel. Fortunately, much of that bulk consisted of various routines to compose and display text, to draw lines and implement Turtle geometry, and to provide various interfaces to bitmap graphics such as moving rectangles, and copying bits to buffers as for restoring the background under menus.

The definition of BitBlt grew out of our experience with text, lines and other bitmap graphics. Now the constraints of the NoteTaker implementation provided the motivation to implement all these capabilities in Smalltalk, leaving only the one primitive BitBlt operation in the kernel. This was a great success in reducing the size of the kernel. The full NoteTaker kernel consisted of around 6K bytes of 8086 code. This figure did not include Ethernet support, real-time clock, nor any significant support for process scheduling.

☐ *Performance* The performance of the NoteTaker was interesting to compare with the Alto. The Smalltalk instruction rate improved by a factor of two, and yet the display of text was much slower (being in Smalltalk, rather than machine code). By adding a small primitive for the inner loop of text display and line drawing, this decrease was largely compensated. User response for such actions as compiling was significantly improved, owing to the faster execution and to the freedom from the swapping delays of OOZE.

☐ *Mutability* Smalltalk-78 used no encoding of object pointers other than for small integers. Class pointers and length fields (for variable-length objects) were stored just as any other fields. It was therefore possible in this design to allow mutation of objects, and this was made available as the primitive method for become:.

☐ *Relevance* We learned a great deal from the NoteTaker challenge, even though only 10 prototypes were built. We made the system much more portable, and had demonstrated that the new generation of microprocessors could indeed support Smalltalk. The decision not to continue the project added motivation to release Smalltalk widely.

TinyTalk

At the same time as the NoteTaker implementation, we performed an experiment[8] to see if a very simple implementation could run on a conventional microprocessor such as a Z80 or 6502. This implementation used marking garbage collection instead of reference-counting, and was able to use simple push and pop operations on the stack as a result. A method cache largely eliminated the overhead in message lookup and, since primitive codes were included in the cache, access to primitives was fast. The system did actually fit in 64K bytes with a little bit of room to spare. Another experiment which was done in conjunction with this implementation was to demonstrate that a special case of BitBlt for characters could run much faster than the general version.

Smalltalk-80

With Smalltalk-78 behind us, few changes were made to the Virtual Machine to produce the Smalltalk-80 Virtual Machine. The main change was an increase in power from allowing blocks with arguments. Beyond this, mostly we cleaned up many details, some of which supported more extensive cleanups in the Smalltalk level of the system.

☐ *Contexts Again* We felt that the optimized contexts of Smalltalk-78 did not justify the loss in clarity which they entailed. So in the Smalltalk-80 language we reverted to Contexts as objects, leaving such optimizations up to implementors clever enough to hide their tricks entirely from Smalltalk. In order to simplify the management of Contexts in the Virtual Machine, we decided to use two sizes of contexts instead of making them truly variable-length. This meant that, if separate free lists were managed for these two lengths, storage for contexts could be allocated and freed with relatively little fragmentation and coalescence overhead.

☐ *Blocks with Arguments* While the syntax changed little in the Smalltalk-80 language (open colon and other non-ASCII selectors were banished), our extended discussions of syntax led to the current description for blocks with arguments. In fact, this required no change to the Virtual Machine, but it had the feel of such a change in the language.

☐ *BlockContexts* We re-engineered BlockContexts in the Smalltalk-80 language. Smalltalk-78 had already handled their recursive application by providing independent stack space for each invocation. Beyond this, mechanisms were defined for checking and diagnosing such anomalous conditions as returning to a context which has already returned.

☐ *Compilation Order* Smalltalk-78 had perpetuated the post-evaluation of receiver expressions so as to avoid delving into the stack to find the receiver. In the Smalltalk-80 language, however, we encoded the number of arguments in the send instruction. This enabled strictly left-to-right evaluation, and no one has since complained about surprising order of evaluation. We suspect that this change will yield further fruit in the future when someone tries to build a very simple compiler.

☐ *Instruction Set* In addition to revamping the send instructions, we made several other improvements to the instruction set. We completed the branch instructions by adding branch-if-true. We put in 2- and 3-byte extensions to retain reasonable compactness without restricting functionality. We also added a few compact codes for returning true and false, and for pop-and-store into temps and fields of the receiver.

☐ *Methods* The encoding of method headers followed the earlier Smalltalk-78 design. In order to simplify the allocation of contexts, a bit was included to indicate whether a large frame was necessary to run the method or not.

Future Directions

While the present Smalltalk design has evolved over a decade now, that does not mean it is finished. As when one climbs a large mountain, the higher reaches are gradually revealed and it seems there is as much to do now as when we started.

☐ *Virtual Memory* An obvious shortcoming of the Smalltalk-80 specification is that it does not include a virtual memory. There are several reasons for this. Our experience with OOZE suggested that object-oriented approaches might be significantly better than simple paging, and we did not want to commit ourselves to one or the other. From our experience with porting the system from one interpreter to another, it was clear to us that implementors could experiment with the virtual memory issue fairly easily, while still working from the Smalltalk-80 image specification. The current object formats allow a simple resident implementation, and yet lend themselves to extension in most of the obvious directions for virtual memory.

☐ *Reducing Concepts* It is always useful to reduce the number of concepts in a language when possible. Smalltalk distinguishes many levels of refinement: subclassing, instantiation, blocks and contexts, to name a few. It is likely that some of these distinctions can be dissolved, and that a cleaner virtual machine design would result.

☐ *Typing and Protocols* While the Smalltalk-80 language is not a typed language in the normal sense, there is nonetheless an implicit notion of variable type in the protocol (full set) of messages which must be

understood by a given variable. We have described an experimental system based on this notion of type[9], and a serious treatment of this approach would likely involve changes to the Virtual Machine.

☐ *Multiple Inheritance* While the Smalltalk-80 system does not provide for multiple inheritance, we have described an experimental system which supports multiple superclasses using the standard Virtual Machine[10]. This is another area in which serious use of the new paradigm might suggest useful changes to the Virtual Machine.

☐ *Tiny Implementations* While, on one end of the spectrum, we seek to build vastly larger systems, we should not ignore the role of small systems. To this end, there is a great deal of room for experimentation with small systems that provide the essential behavior of the Smalltalk-80 system. Threaded interpreters offer simplicity and speed, and it shouldn't be difficult to capture the essence of message sending in an efficient manner.

Maintaining an Evolving Integrated System

We have had considerable experience maintaining an evolving integrated system. In this section we cover several of the challenges and our solutions which support the Smalltalk approach to software engineering.

Applying the Smalltalk Philosophy

One way of stating the Smalltalk philosophy is to "choose a small number of general principles and apply them uniformly." This approach has somewhat of a recursive thrust, for it implies that once you've built something, you ought to be using it whenever possible. For instance, the conventional approach to altering such kernel code as the text editor of a programming system is to use off-line editing tools and then reload the system with the new code and try it out. By contrast, Smalltalk's incremental compilation and accessibility of kernel code encourages you to make the change while the system is running, a bit like performing an appendectomy on yourself.

The recursive approach offers significant advantages, but it also poses its own special problems. One of the benefits is that system maintainers are always using the system, so they are highly motivated to produce quality. Another benefit is high productivity, deriving from the elimination of conventional loading and system generation cycles. Consistent with the Smalltalk philosophy as articulated above, things are

also simpler; the tools and the task are one, so there are fewer versions to worry about. The complementary side of this characteristic is that if the only version is compromised, you are "down the creek without a paddle."

Figure 2.4

The Snapshot Concept

The Alto had a particularly nice characteristic as a personal machine: being based on a removable disk pack, once you had installed your personal pack, any machine you used behaved as your personal machine. When we built the Smalltalk environment, based on an extensible programming language, we arranged the system so that when you terminated a working session, or *quit*, the entire state of your system was saved as a *snapshot* on the disk. This meant that as Smalltalk came to be a stand-alone environment, containing all the capabilities of most operating systems as well as the personal extensions of its owner, any Alto instantly took on that specialized power as soon as you inserted your disk and *resumed* your working session. The snapshot also served as a useful checkpoint in case of fatal errors.

In the later virtual memory systems, OOZE automatically saved a snapshot from time to time, which could subsequently be resumed following catastrophes such as loss of power or fatal programming errors. The robustness of OOZE in this respect was remarkable, but owing to the finite latency period of the checkpointing process, it was necessary to act quickly when fatal errors were recognized, lest they be enshrined forever in the mausoleum of a snapshot. In such circumstances, the alert user would quickly reach around to the rear of the keyboard and press the "boot" button of the Alto before the next automatic snapshot.

Then he could resume his work from a previous state saved a few minutes before. This process was known as "booting and resuming." The term came to be jokingly applied to other situations in life, such as unsuccessful research efforts and other less serious endeavors.

Minimum Kernel for Maximum Flexibility

Most systems are built around a kernel of code which cannot easily be changed. In our Smalltalk systems, the kernel consists of machine code and microcode necessary to implement a virtual Smalltalk machine. You clearly want the kernel to be as small as possible, so that you encounter barriers to change as infrequently as possible. For example, in Smalltalk-72 it was a great improvement when the primitive read routine was supplanted by one written in Smalltalk, since it could then be easily changed to handle such extensions as floating-point constants.

Speed comes into play here, because if the kernel is not fast enough, it will not support certain functions being implemented at a higher level. This was the case for text display in Smalltalk-76. Similarly, generality is important, for the more general the kernel is, the more kernel-like functions can be built at a higher level. For example, the one BitBlt primitive in Smalltalk-80 supports line drawing, text, menus and freehand graphics.

The Fear of Standing Alone

While Smalltalk-72 and -74 were used as long-lived evolving images, the systems as released were always generated from scratch, by reading a set of system definitions into a bootstrap kernel. With the Smalltalk-76 system, we took a bold step and opted to ignore support for system generation. The system was built in two parts: a Virtual Machine was written in microcode and machine code, and a virtual image was cross-compiled from a simulation done in Smalltalk-74. Although this paralleled our previous strategy, we knew that we would soon abandon support for Smalltalk-74, and thus the Smalltalk-76 system would be truly stand-alone. In other words, if a bit were dropped from the system image, or if a reference-count error occurred, there would be no way to recover the state of the system except to backtrack through whatever earlier versions of the system had been saved. As the system became more reliable, we went for days and then weeks without starting over, and finally we realized that Smalltalk-76 was on its own. If this sounds risky to you, think of how we felt!

Standing Alone Without Fear: The System Tracer

While the foregoing approach may seem foolhardy, we actually had a plan: Ted Kaehler said that he would write a Smalltalk program, the *system tracer*, which would run inside of Smalltalk and copy the whole system out to a file while it was running. Considerable attention would have to be paid to the parts of the system which were changing while the process ran. Two months after the launch of Smalltalk-76, Ted's first system tracer ran and produced a clone without errors. While we

Figure 2.5

all breathed a sigh of relief at this point, the full implications only dawned on us gradually. This truly marked the beginning of an era: there are many bits in the Smalltalk-80 release of today which are copies of those bits first cloned in 1977.

The system tracer solved our most immediate problem of ensuring the integrity of the system. It caught and diagnosed inaccurate reference counts, of which there were several during the first few months of Smalltalk-76. Also, although it took four hours to run, it performed the function of a garbage collector, reclaiming storage tied up in circular structures, and reclaiming pointers lost to OOZE's zoning by class. The essential contribution of the system tracer, however, was to validate our test-pilot philosophy of living in the system we worked on. From this point on, we never started from scratch again, but were able to use the system we knew so well in order to design its follow-ons.

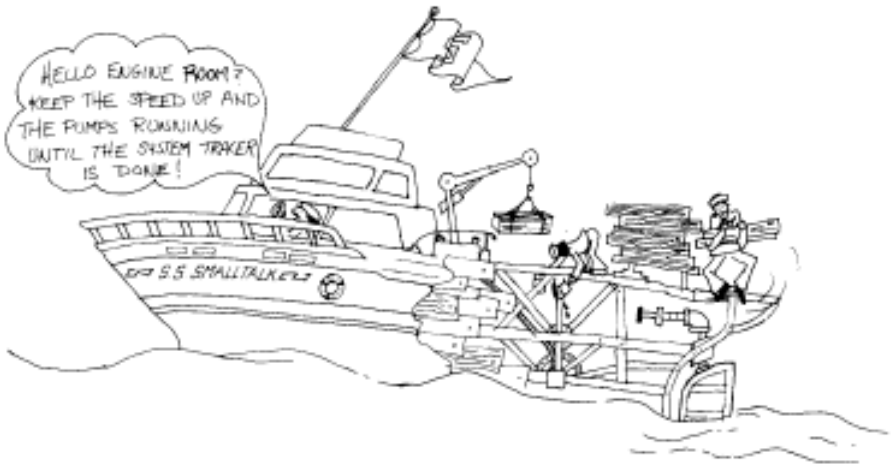

Figure 2.6

Spawning and Mutation

As time passed we found that the system tracer had even more potential than we had imagined. For one thing, it offered an answer to the problem of using a fully integrated system for production applications. This problem manifests itself in several ways: a fully integrated system contains many components which are not needed in production, such as compiler, debugger, and various editing and communications facilities. Also, at a finer grain, much of the symbolic information which is retained for ease of access may be wasteful, or even objectionable (for security reasons) in a production release of the system.

The system tracer could be instructed to *spawn* an application with all unnecessary information removed. This could be done *post facto*, thus freeing application programmers from the integrated/production dichotomy until the final release of a product. In actual fact, since the goal of our research is integration, we never pursued the full potential of the system tracer to drop out such "product" applications. The closest we came was to eliminate unnecessary parts of the system when we were short of space for certain large projects.

The possibility of using the system tracer to produce mutations became evident soon after its creation, and we took full advantage of this. For instance, prior to the Smalltalk-80 release, we wanted to convert from our own private form of floating-point numbers to the standard IEEE format. In this case, we simply included an appropriate transformation in the system tracer and wrote out a cloned image which used the new format. Then we replaced the floating-point routines in the Virtual Machine and started up the new image. Similar transformations have been used to change the instruction set of the Virtual Machine, to change the format of compiled methods, and to change the encoding of small integers. It was in this manner that Smalltalk-78 and -80 were built out of Smalltalk-76.

It is hard to say how far one should take this approach. Sometimes a change is so fundamental that it requires starting again from the ground up, as we did from Smalltalk-74 to -76. Even in such cases though, it seems easiest to simulate the new environment in the old, and then use the simulation to produce the actual new system.

Figure 2.7

The Virtual Image

When we decided to release the Smalltalk-80 system, the question arose as to what form it should take. From the discussion above, it should be clear why we chose the virtual image (a fancier term for snapshot) format. This was the one way in which we could be sure that the release would set a standard. Any implementation, if it worked at all, would look and behave identically, at least in its initial version. At the same time, we tried to decouple the image format as much as possible from such implementation-related details as reference counting versus garbage collection, machine word size, and pointer size. At the time of this writing, implementations exist which vary in all of these parameters. It should be possible to decouple similarly our choice of bitmap display representation, but this project was not of immediate interest to us.

Conclusion

The evolution of the Smalltalk system is a story of good and bad designs alike. We have learned much from our experiences. Probably the greatest factor that keeps us moving forward is that we use the system all the time, and we keep trying to do new things with it. It is this "liv-

ing-with" which drives us to root out failures, to clean up inconsistencies, and which inspires our occasional innovation.

References

1. Goldberg, Adele, and Kay, Alan, Eds., "Smalltalk-72 Instruction Manual", Xerox PARC Technical Report SSL-76-6, 1976.

2. Thacker, C. P., et al., "Alto: A Personal Computer", in *Computer Structures: Readings and Examples,* 2nd Edition, Eds. Sieworek, Bell, and Newell, McGraw-Hill, New York, 1981; (also Xerox PARC CSL-79-11), Aug. 1979.

3. Papert, Seymour, *Mindstorms,* Basic Books, New York, 1980.

4. Goldberg, Adele, and Kay, Alan, "Teaching Smalltalk", Xerox PARC Technical Report SSL-77-2, June 1977.

5. Newman, William, and Sproull, Robert, Principles of Interactive Computer Graphics, 2nd Edition, McGraw-Hill, New York, 1979.

6. Kaehler, Ted, "Virtual Memory for an Object-Oriented Language", *Byte,* vol. 6, no. 8, Aug. 1981.

7. Ingalls, Daniel H. H., "The Smalltalk-76 Programming System: Design and Implementation", Conference Record, Fifth Annual ACM Symposium on Principles of Programming Languages, 1978.

8. McCall, Kim, "TinyTalk, a Subset of Smalltalk-76 for 64KB Microcomputers", *Sigsmall Newsletter,* Sept. 1980.

9. Borning, Alan H., and Ingalls, Daniel H. H., "A Type Declaration and Inference System for Smalltalk", Ninth Symposium on Principles of Programming Languages, pp. 133–141, Albuquerque, NM, 1982.

10. _____, "Multiple Inheritance in Smalltalk-80", pp. 234–237, Proceedings at the National Conference on Artificial Intelligence, Pittsburgh, PA, 1982.