Deltatalk: An Empirically and Aesthetically Motivated Simplification of the Smalltalk-80 Language

Alan Borning
Computer Science Department, FR-35
University of Washington
Seattle, Washington 98195
U.S.A.

Tim O'Shea[1]
System Concepts Laboratory
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304
U.S.A.

Abstract. The Smalltalk-80 system offers a language with a small and elegant conceptual core, and a highly interactive programming environment. We believe, however, that it could be made more learnable and usable by a relatively small set of changes. In this paper, we present the results of a series of empirical studies on learnability, and also some informal studies of large implementation projects. Based on these studies, we suggest a number of changes to the Smalltalk-80 language and system.

1 Introduction

The Smalltalk-80[2] system [8] is the fifth in a succession of object-oriented programming environments which have been produced as part of the long term research programme [13] of the System Concepts Laboratory (originally the Learning Research Group) at Xerox PARC. The first, Smalltalk-72 [7] was designed for "children of all ages," but successive implementations of Smalltalk have become larger and more complex, and have required increasing sophistication on the part of the user [9]. At the same time, successive Smalltalks offer better engineered environments [6] and increased efficiency; the Smalltalk-80 system can now be implemented on an increasingly diverse range of hardware.

The Deltatalk proposal is a way of organising a set of thoughts about improving the learnability and usability of the Smalltalk-80 system. Our starting point in writing this paper is that the object-oriented programming style is particularly attractive for a very wide range of applications, and that the Smalltalk-80 language is the best of the various object-oriented languages that are currently available. In our view it has easily the most usable environment, and the language is based on a small, elegant and mostly consistent conceptual core. However, Smalltalk is obviously not perfect, and we believe that it could be made more learnable and more usable without sacrificing any expressive power or any useful features of the environment. In this paper, we primarily address Smalltalk as a language, and more peripherally Smalltalk as an environment and as an integrated system. Many of the linguistic suggestions we make are taken from the earlier implementations, especially Smalltalk-76 [12].

In general we are willing to trade more text and verbosity in programs against a smaller number of basic constructs. The overall spirit of this paper is to identify small (i.e. "delta") changes from the current Smalltalk-80 system, where there is some genuine evidence that the change might be beneficial. We have pruned our set of recommendations to ensure that they are mutually compatible, and have forced ourselves to not make proposals here (for example with respect to inheritance or blocks and methods) that would change the fundamental character of Smalltalk.

[1] Visiting from Institute of Educational Technology, Open University, Milton Keynes, MK7 6AA, United Kingdom.
[2] Smalltalk-80 is a trademark of the Xerox Corporation.

2 Background

The line of modest development from the Smalltalk-80 system suggested in this paper was initially motivated by a series of empirical studies on learnability [15]. The subjects included over thirty experts with substantial experience in the design and implementation of one of the versions of Smalltalk or one of the object-oriented extensions of Lisp. The experts were asked to make predictions about learnability, and these predictions were compared with the results of a study of over fifty teachers, students, and autonomous learners. The data collection methods included questionnaires (administered by post and by electronic mail), structured interviews, and the completion (by autonomous learners) of daily inventories. The inventories included questions on failures, misconceptions, and known areas of difficulty. The chronological inventory data was used to focus interviews with struggling students and to guide the analysis of student programs and project reports. The different data sources taken together revealed a consistent picture. No single source of learnability problems dominates the data, but a small number of difficulties (with a variety of causes) impacted the learning trajectories of nearly every student. The study also has a number of very positive results with regard to the Smalltalk-80 system. Students and teachers are enthusiastic about it, they can discuss all but one of the core concepts cogently and accurately, and they make effective use of the programming interface, particularly the browser.

It is important not to confuse learnability with usability. The learnability studies described above do not shed any light on how accomplished programmers use Smalltalk or what they can sensibly use it for. In order to shed some light on usability, we considered five large systems implemented primarily at Xerox PARC. These systems are ThingLab [2,3] (a simulation kit), Babar (an electronic mail interface), Rehearsal World [10,11] (an educational programming system for teachers), ARK [16, 17] (the Alternate Reality Kit—a visual programming language based on a physical metaphor), and, of course, the Smalltalk-80 environment itself. Considering these five systems made it possible for us to look at issues regarding programming-in-the-large and advanced programming techniques (e.g. programs that write programs). It became possible to collect statistical data regarding frequency of use of given language constructs in full scale applications. It also forced us to be realistic in considering issues such as sharing, version control, and change management.

Our discussion is also motivated by various aesthetic values that we would apply to the design of any language or environment. In order of importance, we value simplicity, uniformity, orthogonality, readability, economy, and tidiness. For example, in Smalltalk there is uniformity—every object is an instance of some class; simplicity—everything is an object; orthogonality—classes are first-class citizens; readability—with the use of unary message selectors; economy—through polymorphism and inheritance; and finally, tidiness—provided by the class hierarchy. In any attempt to apply these values to ameliorate learnability or usability problems one encounters design tradeoffs. To resolve these tradeoffs we apply various approaches, including one attributed to Alan Kay, viz. "in programming language design similar constructs should either be made identical or they should be made clearly dissimilar." Another principle that we try to follow is the "zero-one-infinity" notion [14], viz., that something in a language should be forbidden entirely, that exactly one occurrence of the thing is permitted, or that it should be possible to use arbitrary numbers of the thing. For example, in Smalltalk there are no methods that don't return a value; every object is an instance of exactly one class; and identifiers can contain an indefinitely large number of characters. An example of a language feature that does not fit this notion is the restriction in FORTRAN that identifiers have at most 6 characters.

We believe that learnability is an important issue in language design generally, and quite crucial if a wide community of users is to be established for languages that are built on innovative models of computation (such as the object model). We believe that it is foolish not to build on past experience and systems, and that one should take great pains to avoid the new language syndrome (i.e. only invent a major new language when one is really forced to).

3 Recommendations Arising from Empirical Studies

3.1 Metaclasses

We identified various learnability problems, but by far the most acute were associated with metaclasses. All the students and teachers reported that they were unable to understand metaclasses, and that they could not bring themselves to stop worrying about where the seemingly endless chain of metaclasses stopped. The solution to this problem is either to remove metaclasses or to find some way of illuminating their creation and operation.

3.2 System Size

Students all reported problems with the size of the system. Although they found the browser helpful for dealing with the class hierarchy, they often got lost and generally felt uneasy about the large number of classes (over 200) and huge number of methods (5,000) that they knew existed in the license version of the system. Limited studies of students working with LOOPS do not indicate that "dynamic browsing" would be helpful. The obvious solution is to introduce some form of layering through the use of modules. (Many students used a "spaghetti" metaphor for the existing system, so we are proposing a shift to "lasagne.")

3.3 Inheritance

Students also had difficulty with nonstandard forms of inheritance (super) and with abstract superclasses.[3] Within our self-imposed confines of delta-sized changes, there does not appear to be a good linguistic solution; so we favour environmental solutions that illuminate inheritance in the browser. However, students did master the general notions of object, message, class, subclass, and method inheritance without difficulty.

3.4 Other Difficulties

Some "minor" matters of syntax and convention turned out to cause numerous problems. The precedence rules and the significance of capitalisation taxed students, as did the variation in the tone and content of error messages. Students also had difficulty in keeping track of the variety of scoping rules in the language. The solution is to simplify the parsing, rigidify conventions where possible, and reduce the number of scoping rules.

Two other possible problem areas (for which we have no strong empirical support) concern keyword parameters and assignment. Keyword parameters might bring up images of more generality than actually exists in the language—that the keywords of a message can be given in any order, and that keywords can be elided and a default value supplied. (These properties hold, for example, for keyword parameters in Ada.) From studies of the difficulties experienced by learners of other languages, we know that assignment is often a source of confusion. In Smalltalk, the operation of assignment is defined as in other imperative languages and hence hidden from the student; one would instead prefer that it be handled via message-passing, as are other Smalltalk operations.

The empirical studies are reported in detail elsewhere. In this document we have confined ourselves to results that could in principle be acted on by making small changes to Smalltalk. Other courses of action include the elaboration of new teaching materials and making major changes to the language (such as removing the object-class distinction).

[3] An abstract superclass is like any other class as far as the Smalltalk language and environment are concerned. However, by convention, its purpose is only to define protocol and useful methods that can be inherited by concrete subclasses, and not to be instantiated. An example from the current Smalltalk-80 language is class Collection.

4 Recommendations Arising from Large Implementations

In addition to the insights arising from empirical studies, we have also accumulated (in a less formal manner) a number of recommendations regarding the language based on the experiences of implementors of large systems in Smalltalk.

4.1 Programs as Data

A number of applications, such as ThingLab, ARK, and Programming by Rehearsal, compose and install Smalltalk methods, and also store fragments of Smalltalk code that can be manipulated in a variety of ways. This is not particularly convenient in the Smalltalk-80 system. (Lisp provides an example of a language in which these sorts of activities are much easier.)

We examine ThingLab to pinpoint some of the problems. In early versions of ThingLab [2], code was stored as text strings. New methods were composed by concatenating these strings, adding parentheses, selectors, temporaries, and so forth as needed. The string that represented the new method was then given to the compiler. It was straightforward to compose methods in this way, but very difficult to analyze and transform code fragments. Because of these problems, ThingLab now stores code as parse trees. The parse tree classes built into Smalltalk make it easy to inspect and manipulate code. Nevertheless, it remains much less convenient to manipulate code than in Lisp. One problem is that the expressions to build pieces of parse tree are verbose (although straightforward to write). Another problem is that parse tree objects, as currently implemented, are tied to a particular method. For example, moving a part of a tree into a method for a different class requires that new literal nodes be created and noted in the literal dictionary for the new method.

4.2 Modules and Scoping Rules

Regarding modules and scoping rules, an obvious problem with the current system is that there is a single global name space for classes and global variables. When combining several large application packages, one can run up against conflicting class names. A related problem is that there is no way to create private messages for a class or a group of cooperating classes—there is a convention of providing a "private" category in the browser, but access to these private messages is not restricted by the language.

It has regularly been noted that there are too many kinds of variable names in Smalltalk: global, pool, class variable, instance variable, method argument, temporary, and block argument. In addition to variable names, there are selector names. Accompanying these names are a variety of scope rules. Global names are, of course, in a single global name space. Pool variables are shared by all classes that name the given pool, and can also be inherited down the subclass hierarchy. Class and instance variables are inherited down the subclass hierarchy. Method argument and temporary names are local to the given method. Block arguments are known only within the given block, but space for them is stored in the method context, resulting in blocks that are not reentrant.

For the novice, this plethora of kinds of names and scopes presents a learning problem; experts can learn to work with the kinds of names and scopes. One difficulty that does arise for experts is that these rules increase the difficulty of having programs that manipulate programs. For example, since there is no general hierarchical, lexical scoping method, one cannot easily do inline expansion of method invocations.

4.3 Blocks

Blocks are fundamental in supporting extensible control structures in Smalltalk, and also have more sophisticated uses (e.g. in process creation). However, they have some anomalous properties. Block arguments are not stored local to the block activation, but rather with the method activation, leading to various bugs and restrictions. As noted above, blocks are not reentrant. As an example of another

bug, consider the following code:

```
1 to: 10 do: [:i | [Transcript show: i] fork].
```

One would expect the digits 1 to 10 to be printed on the transcript (perhaps in some permuted order due to process priorities). Instead, if the forked processes have lower priority than the creating process, 10 will be printed 10 times, since i is not local to the iterated block. Another annoying limitation of blocks is that they cannot have temporary variables of their own. Following Alan Kay's principle mentioned above, on aesthetic grounds, blocks and method bodies are similar enough that they ought to be the same.

4.4 Multiple Inheritance

Multiple inheritance is a thorny issue. Borning and Ingalls [4] designed and implemented a straightforward multiple inheritance mechanism for the Smalltalk-80 system. This was subsequently included in the release system, but without environmental support. It has received almost no use. Whether this is because it is inadequate, not well supported by the environment, or both, is not known. Looking at other object-oriented systems, multiple inheritance is used extensively in e.g. Flavors in ZetaLisp [5]. Flavors is a system for experts only, and our informants tell us that the more unusual modes of method combination receive little use even by them, since it is difficult to predict how a combined method will behave. More generally, there is no consensus in the object-oriented programming community regarding what sort of multiple inheritance, if any, should be supported. It does seem that single inheritance, as used pervasively in Smalltalk, is a powerful mechanism for constructing object taxonomies and for doing "differential programming." Multiple inheritance, when used, seems better at a different role, namely combining many small packages of behaviour to make a new kind of object. Multiple inheritance works better when the behaviours can be made relatively independent, as in window systems, and not as well when they interact strongly, as in defining different kinds of collections.

4.5 Super

As noted in the previous section, super is confusing to novices. In the Deltatalk spirit, we considered simply eliminating it. Rather than writing super zork, one would need to define a method with a different name in the superclass, and invoke that method. To investigate the feasibility of that route, we gathered statistics on the use of super in a large system—namely the Smalltalk system itself. The current research version of the Smalltalk-80 system has a total of 8826 methods. Of this total, 428 methods use super. Of these 428, 81 methods send some selector to super other than the selector for that method. We concluded from inspecting these methods that eliminating super would result in a blowup in the number of method names (in a system with too many different method names already). There would also be some code that was less clear. Regarding sending super to some other selector, many methods did so for dubious reasons, but there were some legitimate uses. We therefore conclude that some mechanism like super should remain in the language.

4.6 Literals

Our last two recommendations are much finer-grained. The Smalltalk-80 language supports integer, float, string, symbol, and array literals. Some additional sorts are needed. For readability and editability, one would like bitmap literals. These should be seen and edited as bitmaps, not as strings that can be used to create the bitmaps. Parse tree literals would ease a few of of the problems listed above regarding programs that write programs. Classes are too "heavy" in the current system for some applications. One would like support for anonymous, in-line class definitions for simple classes, so that for example one could package up a pair of objects and return them as an appropriate unit. A notation for literal classes would handle this. Minor increases in efficiency could be gained if point and rectangle literals were supported. Regarding literals other than bitmaps, one could build each of

them into the language, but one would regularly come up with some new sort of literal that should be supported as well. Therefore, we instead recommend that there be a language construct that instructs the compiler to evaluate an expression at compile time and include the result as a literal. If a more convenient set of messages for constructing classes and parse trees is devised, this should handle the remaining cases, and also support such things as constant expressions in code.

4.7 Naming Conventions

Finally, to some programmers who have used a variety of languages, the Smalltalk conventions for building long names are peculiar. For example, in Smalltalk one would write a long instance variable name in a manner such as veryLongName. In most other languages, one could write very_long_name, which we regard as more readable. We therefore recommend that underscores be allowed as part of a token, and (as noted in the previous section), that case not be significant in names.

5 Recommendations Regarding the Environment

This paper is concerned primarily with changes to the Smalltalk-80 language. However, our studies have some implications for the programming environment as well, and so we believe that it is worth noting some areas for improvement and simplification.

Both the empirical studies and surveys of experts indicate that the Smalltalk browser is a major win. A few improvements would be useful, however. There should be support for an indefinite number of levels of class categories, rather than just one as at present. This support should be integrated with the proposed module mechanism. (The current naming conventions in effect result in two levels of classification—for example, in the current system there is in effect a category *Graphics* with subcategories such as *Editing* and *Text*, although the browser doesn't know about the two levels.) There should be a mode that, if desired, allows inherited methods to be seen in the list of methods for a class. There should be support for reorganising chunks of the class hierarchy, as well as for adding and editing classes. Some have argued that there should be explicit "protocol objects" representing the protocols that classes must obey, rather than just having a less formal category in the third browser pane.

Code viewing and display is tied a bit too tightly to the text strings typed in by the programmer. For example, if one views code in a browser with different width than the one in which it was created, lines will sometimes wrap around, with a loss of the clues given by indentation. Similarly, when typing comments, carriage returns and tabs must be inserted manually if the comments are to be indented properly; when the comment is edited, the formatting must be adjusted. To remedy these problems, we recommend that methods be stored as parse trees, and that they be presented using a pretty-printer. It should be possible to attach a comment node to any other node in the parse tree; comments should be formatted appropriately. However, formatting should not be completely automatic; for example, if the user indicates that two short statements are to be printed on the same line, this should be remembered and obeyed. Similarly, while case would not be significant in looking up names, the system should remember the particular way a name was written by the programmer in a given method, and show the name in that way when the code is displayed, rather than (say) showing everything in lower case.

The set of tools and conventions for handling concurrency is inadequate. A comprehensive remedy is well beyond the scope of Deltatalk. However, experience with e.g. the Sun and Cedar environments shows that associating a process with each window considerably enhances programmer productivity. In these environments, one can start up a long file-in process in one window, and then go on to read one's mail in another window while the file-in continues. Support for this modest level of concurrency is recommended. Nearly all of the necessary primitives are already available within the system (an obvious exception is reentrant blocks); what is needed is a consistent set of tools and conventions that make use of them.

The Smalltalk-80 environment provides the Model-View-Controller mechanism (MVC) for constructing user interfaces. MVC provides a toolkit of classes and conventions for user interface design, in which there is a clear separation between the underlying object (the model) and views on the model. The basic goals of MVC are clearly appropriate. In practice, however, MCV has proven difficult for both both novices and experts to use; it should be replaced. Finally, there are parts of the system that have grown and changed over the years, perhaps having been translated automatically from Smalltalk-76, and which are due for an overhaul.

6 Principal Language Changes Involved in Deltatalk

As described in the introduction, our starting point is a feeling that the Smalltalk-80 language is mostly right, and that a number of relatively small changes would improve the language considerably. In this section the proposed changes are outlined. The language would otherwise remain as specified in [8].

6.1 Metaclasses

In our empirical studies, metaclasses were regarded as the most significant barrier to learnability by both students and teachers. We propose that they be eliminated. We have explored various alternatives to metaclasses, such as the use of prototypes [1]. However, for Deltatalk we simply propose that the language revert to the situation in Smalltalk-76. Every class would be an instance of class Class. New instances would be created by sending the new message, and come with fields initialised to nil. Thus, one could create a new point using the expression

```
Point new x: 10 y: 20
```

(Of course, the language would continue to support infix creation of points using e.g. 10@20.)

Two of the reasons that metaclasses were included in the Smalltalk-80 language were to avoid uninitialised instance variables, and to have a convenient place for messages that access or initialise information common to all instances of a class. (Examples of the latter include the message pi to Float, and messages to initialise class variables.) Regarding uninitialised instance variables, some evidence indicates that these errors aren't that frequent, and when they do occur, they are very easy to track down. There is in fact some advantage to having nil in the x field of a point that one has neglected to initialise, rather than say 0, since the error will be discovered earlier. For novices, a protocol that allows them to inquire what parts need to be initialised could be provided. Regarding messages that access or initialise shared information, sending a class variable initialisation message to 0.0 instead of Float is admittedly odd, but isn't hard to understand.

6.2 Variable Length Classes

Variable length classes are another source of complexity, although the novice need not encounter them as soon as metaclasses. We recommend that the language be simplified by removing the option of creating classes with both named and indexed fields. Instead, one could have classes with only named fields, or only indexed fields. (To achieve the effect of a class with both, one would instead create an ordinary class with named fields, with one field pointing to an object with indexed fields.)

6.3 Super

The reserved word super is confusing to novices. (Apparently self is confusing to some as well. However, others have an "Aha!" experience when they finally understand self, and we regard it as essential to object-oriented programming.) In addition to the data regarding novice confusion around super, it does have some peculiar properties. For example,

```
super zork.
```

behaves differently than

```
temp ← super.  temp zork.
```

The problem is that super does not denote an object in the same way as other identifiers in Smalltalk. Rather, it denotes self, and at the same time causes the interpreter to go into a special mode. To prevent the sort of anomaly illustrated above, we recommend that super be allowed only as the receiver of a message. This is not a complete solution to the problem, since super remains different from other variables in the system. However, all of the other alternatives that we explored had disadvantages of their own.

6.4 Multiple Inheritance

As noted above, multiple inheritance is a thorny issue, on which we went back and forth several times. In the Deltatalk spirit, we propose that it be omitted, in favour of tried-and-true single inheritance.

6.5 Blocks

As also noted above, there are a number of problems with blocks: their small differences from method bodies, the lack of local state for block activations, and the general problem of too many kinds of variables and scope rules. We sought a comprehensive solution to all these problems[4], but decided that we were moving beyond Deltatalk. Here, we take a conservative view and suggest that blocks be retained, with three modifications. First, block arguments should be stored local to the block activation. Second, temporary variables in blocks should be supported, and should again be stored local to the block activation. A reasonable syntax for block temporaries is to include the temporaries between a pair of vertical bars; if no temporaries are used, the bars could be elided. (This is exactly analogous to the syntax for temporary variables in methods.) Third, the language should support arbitrary nesting of blocks, with block arguments and temporaries lexically scoped.

6.6 Naming Conventions

Capitalisation should not be significant in either variable or selector names. Underscores should be allowed as a way of building long names.

6.7 Modules

A module mechanism should be included in the system; we suggest that a relatively simple mechanism be used, based on some proven design from e.g. Modula-2. The global name space should consist just of module names. There would be no nested modules or submodules. Within a module are classes and other kinds of objects. Names can be exported or hidden (including selector names). Names must also be explicitly imported, although there would also be a mechanism for importing all the names that a given module exports. The module mechanism should subsume pool variables, and should be well supported by the programming environment.

6.8 Literals

As noted in the previous section, some additional literals would be useful. We recommend that bitmap literals be fully supported by the environment. To handle the other kinds of literals, we

[4]Two schemes showed some promise. In the first, method bodies would be made exactly the same as blocks; arguments for both methods and blocks would be labelled with keywords. (A method's selector would be separate from the keywords.) In the second, blocks would be replaced with inline definitions of objects that understood a single message.

suggest that the construct

{ ... an expression ... }

instruct the compiler that the expression in brackets be evaluated at compile time, and that its value should then be included as a literal at that point. To take advantage of this construct, we should devise compact message protocols to create new anonymous classes, and to build various sorts of parse tree fragments.

6.9 Programs as Data

Finally, we recommend some changes to the parse tree classes to make it easier to manipulate programs as data. Historically, these classes were developed for use by the compiler. Later, they were used as well by the decompiler, and then by other applications that manipulate programs as data. They should be cleaned up to make them more suitable as general vehicles. One change is to devise some more compact message protocols for parse node creation. Another is to remove the dependence of parse nodes on a particular method and literal dictionary. Instead, a literal node would just contain the literal; the indexing of literals and construction of the literal table would take place during the code generation phase.

7 Conclusions

This paper has reported a *gedanken* experiment in which we have sought to improve the learnability and usability of the Smalltalk-80 system with a package of deletions and small tweaks. Ideally, any version of Deltatalk could be described by a version of [8] with a few sections blanked out and a few small inserts. We believe that a version of Smalltalk that was only a "delta" change could be usefully implemented, and see no reason why it would not be used by all the current types of user of the Smalltalk-80 system. The main virtue of the proposal sketched out here is that modest effort would be followed by a fairly certain gain in learnability and hence in the size of Smalltalk programmer community. The enthusiasm exhibited by learners for the key ideas of the language and the quality of the large systems that have been implemented in it to date suggest to us that a collection of improvements in overall learnabilty could have a dramatic positive effect.

Acknowledgments. We gratefully acknowledge the financial and intellectual help of Xerox PARC during the course of the research which led to this paper. We are particularly grateful to David Robson, Randall Smith, and Frank Zdybel for ideas, helpful advice and constructive criticism. The authors are, of course, entirely responsible for any inconsistencies or errors in the views expressed in this paper.

References

[1] A. H. Borning. Classes versus Prototypes in Object-Oriented Languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference*, pages 36–40, ACM and IEEE, Dallas, Texas, Nov. 1986.

[2] A. H. Borning. The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, Oct. 1981.

[3] A. H. Borning. *ThingLab—A Constraint-Oriented Simulation Laboratory.* PhD thesis, Stanford, March 1979. A revised version is published as Xerox Palo Alto Research Center Report SSL-79-3 (July 1979).

[4] A. H. Borning and D. H. H. Ingalls. Multiple Inheritance in Smalltalk-80. In *Proceedings of the National Conference on Artificial Intelligence*, pages 234–237, American Association for Artificial Intelligence, Pittsburgh, Pennsylvania, Aug. 1982.

[5] H. I. Cannon. *Flavors.* Technical Report, MIT Artificial Intelligence Lab, 1980.

[6] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1984.

[7] A. Goldberg and A. C. Kay. *Smalltalk-72 Instruction Manual.* Technical Report SSL-76-6, Xerox Palo Alto Research Center, 1976.

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

[9] A. Goldberg and J. Ross. Is the Smalltalk-80 System for Children? *Byte*, 6(8):348–368, Aug. 1981.

[10] L. Gould and W. Finzer. *Programming by Rehearsal.* Technical Report SCL-84-1, Xerox Palo Alto Research Center, May 1984.

[11] L. Gould and W. Finzer. Programming by Rehearsal. *Byte*, 9(6):187–210, June 1984.

[12] D. H. H. Ingalls. The Smalltalk-76 Programming System: Design and Implementation. In *Proceedings of the Fifth Annual Principles of Programming Languages Symposium*, pages 9–16, ACM, Tucson, Arizona, Jan. 1978.

[13] A. C. Kay and A. Goldberg. Personal Dynamic Media. *Computer*, 10(3):31–42, March 1977.

[14] B. J. MacLennan. *Principles of Programming Languages.* Holt, Rinehart and Winston, second edition, 1987.

[15] T. O'Shea. The Learnability of Object-Oriented Programming Systems. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, page 502, ACM, Portland, Oregon, Sep. 1986.

[16] R. B. Smith. Experiences with the Alternate Reality Kit: An Example of the Tension between Literalism and Magic. In *Proceedings of CHI+GI 1987*, pages 61–67, ACM, Toronto, Canada, Apr. 1987.

[17] R. B. Smith. The Alternate Reality Kit: An Animated Environment for Creating Interactive Simulations. In *Proceedings of the 1986 IEEE Computer Society Workshop on Visual Languages*, IEEE, Dallas, Texas, June 1986.