Smalltalk and Exploratory Programming*

David W. Sandberg
Computer Science Department
Oregon State University
Corvallis, Oregon 97331

88-60-8

Smalltalk and Exploratory Programming*

David W. Sandberg
Oregon State University

Using Smalltalk-80, programmers can produce prototypes much faster than with C or Pascal. What techniques do Smalltalk-80 programmers use to produce these prototypes? What is special about Smalltalk-80 that enables them to uses these techniques? Can these techniques be used with conventional languages such as C? In an attempt to answer these questions we interviewed experienced Smalltalk programmers and asked how they approach programming in Smalltalk. Such introspective interviews that are conducted after completion of a project are known to be somewhat unreliable, but not enough is known to use any other methodology. What follows is a summary of the interviews, followed by an explanation of the results. Finally we discuss some of the weaknesses of Smalltalk and some possible solutions.

The interviews

We interviewed seven Smalltalk programmers at Tektronix. Most of the programmers were not directly involved in developing the Smalltalk system, but were using the Smalltalk system to solve their problems. All of these programmers had successfully completed some Smalltalk project. Three of these programmers were working on products that Tektronix is considering marketing. Another was working on a product that is expected to receive widespread use within Tektronix. All of these projects involved user interfaces. Two of the programmers had about 6 months experience with Smalltalk. The others had a year or more experience. All had extensive experience developing software before turning to Smalltalk.

The interviews were conducted informally and were from an hour to an hour and a half long. We were looking for what makes programming in Smalltalk different from programming in other languages. We did not poll the programmers on specific issues, because we had no idea what the important issues were.

To summarize what the Smalltalk programmers said, we have constructed a composite monologue. Most of the themes in this monoluge are common to all of the interviews although a few of the points were brought out by only one programmer. This monologue is in the first person singular:

> I was attracted to Smalltalk because I can experiment, particularly with user interfaces. I can try out an idea much faster in Smalltalk than in other languages. There are several reasons for this. The automatic garbage collection of Smalltalk frees me from dealing with storage management issues. The browser and debugger are much better tools for understanding code than a program listing is. These tools make listings obsolete. I can test bits and pieces of my code in a workspace. I can save everything just as it is and

[1]Distributed as technical report number TR-88-06 of the Computer Research Laboratory of Tektronix, Inc. and as technical report number 88-60-8 of the Department of Computer Science at Oregon State University.

come back to it later. I can share more code by using inheritance and because I have access to a body of code to reuse.

Programming by experimentation is exemplified by a Beck cycle which is a week long. A Beck cycle starts Monday morning with an idea. Without any planning, a programmer starts coding to see if the idea can be implemented. By the end of Monday, he will have convinced himself of the idea's feasibility. Tuesday, and Wednesday are spend finishing a prototype implementation. Thursday, and Friday are spent showing the prototype to his colleagues, and fixing any bugs that are found. If a colleague likes the idea, he will pick up the prototype, modify it, and use it in his next project.

When the idea is no good, the Beck Cycle is aborted. This could happen when the programmer discovers that he doesn't know how to implement the prototype, or when the prototype uncovers some basic flaw. In conventional environments, projects are seldom aborted because the effort invested is high. In Smalltalk, the effort invested is much less, and I can discard the code and try something else instead of trying to make the idea work by adding patch after patch. Since I throw away my failures, and show off the good ideas, I have acquired a reputation as an very good Smalltalk programmer.

Programming in Smalltalk is different than it is in Fortan, Pascal, or Lisp. The exact difference is hard to determine. The Smalltalk Red[4] and Blue[3] book helped me with the mechanics of syntax and using the system, but did not telling me how to write a Smalltalk program. Although the Tektronix Smalltalk course is excellent, it did not tell me how to write a program either. Beginners tend to write code that looks like C, not Smalltalk. I think assembly language is better preparation for Smalltalk than Pascal or Lisp, because fewer restrictions are placed on the assembly language programmer.

The Smalltalk user interface is different from that of conventional operating systems. The interface has features designed into it that I overlooked for a long time. However, even though the user interface is much different, learning it is not the major difficulty in learning of Smalltalk.

Programming in C focuses on the dynamic aspects of a system, that is, the algorithms. In Smalltalk, I first focus on the state of the system when nothing is happening. I divide this state into classes and then assign the functionality of the system to the different classes.

Smalltalk separates the data representation from the algorithm. Thus, either can be changed with little effort. In Fortran, changes to the data representation are difficult to make since most of the code directly depends upon it.

Much of the code in the Smalltalk-80 system is hard to understand, partly because the overall way pieces fit together is not documented, but must be inferred from the code. In spite of this, code is often reused anyway. For example, the paragraph editor is difficult to understand, but has been successfully modified by many Smalltalk programmers.

Exploratory programming techniques encourage code that is hard to read. It is tempting to make fix after fix to a piece of code until it is impossible to understand. Hence, rewriting code is essential for producing reusable code. In rewritting a piece of code, the programmer brings a better understanding of the problems and writes a more easily understood piece of code. I consider rewriting to be a labor of love, not as a useless task. I plan to have each Smalltalk programmer in this project rewrite their code.

Smalltalk-80 does not support multiple programmer projects well. This lack of support is not as serious as it first seems, since other languages do not provide it either. This lack is more noticeable in Smalltalk since Smalltalk provides a good environment for a single user but provides a much poorer environment for multiple programmer groups. Most system provide poor environments for both the single user and multiple programmers.

Speed has not been a serious problem. In one project we are using Smalltalk to design an oscilloscope and are planning to recode the final system in Objective-C to improve speed and allow us to put the code in ROM.

Exploratory Programming

A *product environment* is the environment in which a piece of software will be used. The standard software engineering practice is to study the product environment and produce a specification for the software. After the specification is complete, the software is written and tested. This technique works when the product environment is stable and well enough understood to produce a specification. However, in many cases the product environment is poorly understood. Applying standard software engineering practices in these environments is likely to produce unsatisfactory software. In such environments exploratory programming is a better approach[11].

Exploratory programming involves producing a piece of software that attempts to meet the known, basic requirements of the system. This software is then tested in the product environment to find out how it fails. These failures lead to more requirements. The software is modified to meet these requirements, and tested again. This process is continued until the software performs adequately in the product environment. The requirements are not usually explicitly expressed by programmers, but are embodied in the code under development.

Standard software engineering uses programming to *implement* a given specification. In contrast, exploratory programming is *writing* the specification. This specification need not be for a piece of software. In one case we studied, the specification was for a piece of test equipment.

To have exploratory programming be successful, the cost of experimentation must be low. The time to write the code for an experiment must be short enough that the code can be discarded if the idea fails to produce the desired result. Discarding the bad ideas results in a much better product since only the best ideas are used.

The cost of experimentation is influenced by two important factors: the cost of making changes and the amount of code that can be shared and borrowed.

Making Changes.

One factor that influences the cost of changes is the length of time it takes to complete the edit-compile-link-run-test cycle. Under conventional operating systems, this cycle is usually on the order of a few minutes. In Smalltalk, it is a few seconds. This is about as fast as a programmer can produce meaningful experiments. It is unlikely that speeding up this cycle further will increase programmer productivity, since the thinking time will become the dominate factor. The edit-compile-link-run-test cycle time has been similarly shorten in many other software development environments such as Magpie[2] and the Cornell Program Synthesizer[12].

Another factor that influences the cost of changes is how the code is organized and written. A good Smalltalk programmer will organize his code so that changes are easy to make. Code sharing is important because changes are easier to make to a single piece of shared code than to many pieces of code that are scattered throughout the system.

In conventional programming, the data representation and algorithms are closely intertwined. This makes it very difficult to change the data representation without changing nearly all of the code. In Smalltalk, it is possible to separate the data representation issues from the algorithms. Thus changes to the algorithm are more independent from changes to the data representation.

The ability to mutate objects in Smalltalk makes the representation of data easier to change. Adding another instance variable to a class is a common change. All instances created before this change must be mutated by adding another storage slot for the new instance variable. In Smalltalk this mutation is transparent to the user.

Sharing Code

Another very important factor in lowering the cost of experimentation is the sharing of code which can take place within an application and among programmers. An existing library of software relevant to the experiment can reduce the cost of experimentation. Three steps must occur if software is to be shared:

- The shared abstraction must be expressed and placed in an accessible location.
- The code must be found by the programmer.
- The code must be understood in sufficient detail to be reused.

The cost of finding and understanding the existing code must be less than that of rewriting it from scratch.

Expressing the abstraction

The ability to express an abstraction in code determines whether or not it can be shared. Some form of code sharing takes place in every system. The ability to express math functions as subroutines has allowed large libraries of math subroutines to be shared. In Unix, many programs are reused through the use of filters, pipes, and text files. The single universal data structure of Lisp has allowed many tools to be shared in the Interlisp-D environment.

Sorting is an important concept, yet it cannot easily be expressed with a subroutine unless the type of data being sorted is known. In environments where there is a dominant data structure, sorting routines are found in the libraries. For example, in Unix there is a program to sort records in a file. In Lisp there is a routine to sort lists. In Fortran and Pascal, there is no predominant data structure and the concept of sorting cannot be expressed without specifying a particular data structure. Because of this, sorting routines are not shared.

In Smalltalk more kinds of sharing can be expressed than with other languages. Smalltalk can express the concept of sorting without specifying the data type, which allows sorting routines to be shared.

The main purpose of inheritance and the class hierarchy in Smalltalk is to permit the sharing of code. Sometimes the class hierarchy is used for organizing classes or methods, but these are not the primary functions of inheritance and the class hierarchy.

Locating Code

A user must be able to find the abstractions relevant to the current task. This has been a problem in Interlisp-D, Unix, and Smalltalk-80, but has not been seriously addressed in any of these environments. Unix does offer some tools for finding things. The manuals include a permuted index and the *man* command can search by keyword. The *man* pages have a 'See also' section as well. Experienced users of these systems seem to remember the approximate location of what they need. This method could certainly be improved, but this issue does not seem to be the current limiting factor of Smalltalk.

Understanding Code

It appears that the most serious limitation in sharing software is the inability of the user to understand the software so that he can apply it to his application. Both the Interlisp-D and Smalltalk-80 environments provide tools that help in understanding code. These tools include windowing, code browsers, debuggers, and inspectors. Windowing allows relevant information to be placed side by side. The code browser aids in understanding the static state. The debugger and inspectors help to understanding the dynamic aspects of the code. The term 'debugger' is a misnomer, since this tool is used for more than just debugging code. The utility of these tools is indicated by the ability of programmers of these systems to understand code that is poorly documented and often poorly written. Information can be obtained faster using these tools than by flipping through a program listing. Hence Smalltalk programmers do not use program listings.

Code in the Smalltalk-80 environment is easier to understand than in the Interlisp-D environment. This is supported by the fact that programmers share code at the source text level in Smalltalk-80, whereas this is rare in Interlisp-D. Code sharing at the source text level occurs when existing source text of another programmer is modified and reused. The precise reason for this difference between Smalltalk-80 and Interlisp-D is unclear. It may be because of Smalltalk's better sharing mechanisms.

Deficiencies of Smalltalk-80

This section discusses some of Smalltalk-80's deficiencies and the prospects for correcting them. Since Smalltalk has already had a long history, the remaining deficiences are the ones that require the most effort to correct or are inherent in the Smalltalk philosophy.

Smalltalk-80 is difficult to teach effectively, primarily because the process of programming in Smalltalk-80 is poorly understood. Our teaching ability will improve as our understanding of programming improves.

Smalltalk-80 currently excels at experimenting with user interfaces, but does not have adequate libraries to excel in other areas. It should be possible to extend Smalltalk-80 to include a library for experimenting in some other field. Care would be needed to include abstractions that are truly useful rather than ones that the library designer thought would be useful.

The Smalltalk-80 support for teams of programmers is clearly deficient. Some support is given[8], but this support has not reached the same level of maturity as for a single programmer. These mature tools do not exist for other programming environments either. Further research is needed before a good system for multiple programmers can be built.

Many attempts have been made to increase the speed of Smalltalk-80 code. Much work[13,6] has been done on trying to increase the speed of the interpreter. Further work on the interpreter is unlikely to produce dramatic increases in speed. Another approach is to optimize the Smalltalk-80 code. This approach is hampered by the lack of compile-time type information. Inferring types from Smalltalk-80 code is difficult[5] and more research is needed before inferring types would be practical.

Yet another approach to increasing the speed of Smalltalk-80 code is through user-coded primitives. A user-coded primitive is a time critical routine, which is written by the user in assembly language or C, and can be called from Smalltalk. Although this approach is awkward from a user viewpoint and does not totally solve the speed problem of Smalltalk-80, it seems to be the only technically feasible approach at this time.

The code sharing mechanisms of the Smalltalk language are clearly limited. However, adding better mechanisms for sharing requires great skill. The attempts at multiple inheritance have not achieved the elegance and simplicity of single inheritance[1]. Multiple inheritance may make Smalltalk code harder to read, rather than easier, and diminish Smalltalk's usefulness as an exploratory programming tool.

It has long been recognized that Smalltalk has no way to separate a product from the development environment. No one has yet come up with a workable solution.

Smalltalk and Lisp have been prevented from enjoying wider use because of problems with speed and separating the product from the development environment. These problems arise due to the run-time binding used in these languages. It is often argued that exploratory programming cannot be done without run-time binding. Smalltalk-like user interfaces have been built for languages that use compile-time binding. Potentially better user interfaces can be built with compile-time binding since more information is available through static analysis. On the other hand, no widely known language has been able to achieve the sharing possible in Smalltalk without using run-time binding. The author has built an exploratory programming environment called X2[9] which uses compile-time binding but achieves a degree of code sharing comparable to that of Smalltalk. To achieve the sharing, X2 uses an extension of the ideas of parameterized types in CLU and Alphard. Standard compiler technology can be applied to X2 so that X2 can potentially compete in terms of speed with C or Fortran. Also, enough information is present in the X2 environment to make separating the product from the development environment easy. Thus a language based on parameterized types and compile-time binding has more potential of replacing existing programming languages than do languages that use run-time binding.

There appears to be overwhelming agreement among Smalltalk programmers that the present tools in Smalltalk-80 for understanding code are not adequate. The overall way the pieces fit together is difficult to ascertain in the current system. This problem may be the greatest weakness of Smalltalk-80. It is also among the most difficult weaknesses to correct. All we can do is discuss some factors that influence the readability of code.

Producing separate documentation for the code does not appear to be the answer. This kind of documentation is very difficult and time consuming to produce. It is often incorrect and, in many cases, may be just as difficult to understand as the original code. Past experience with program documentation has not met with great success and it is unlikely that Smalltalk-80 documentation could break this pattern.

A more feasible solution may be to provide examples which demonstrate the usage of the abstraction. These examples should be easier to produce than complete documentation. I do not believe that this approach has been tried in any large environment.

Including type information in the source code makes it easier to understand. Whether these types are inferred by the system or explicitly included by the programmer does not really matter. Previous work with types in Smalltalk has not met with much success. Furthermore, type information would not address overall organization issues.

How well the code is written plays a larger part in understanding the code than does any of the above factors. One way to make any piece of code easier to understand is to have the original author rewrite it. The author will almost always produce a better piece of code the second time he writes it, especially when the temptation to add more functionality is resisted. Encouraging programmers to rewrite their code may be a partial solution to making Smalltalk-80 code easier to read.

The easiest abstractions to use are those for which the user has to make the fewest decisions in order to use it correctly. Only the essential features of the abstraction should be presented to the user. Special cases should be addressed by the implementor, not the user. The more that can be hidden from the user, the better. Any decision that can be made by the implementor should be made by him and not by the user of the abstraction.

There is a tension between making an abstraction specific to an application so that it is easy to use and making it general enough so that it can be used in many places. Different levels of abstraction might alleviate this tension. A simple view can be provided for most users with more general and complex levels of abstraction available when needed.

The distance between the programmer's mental concept and the code that embodies the concept is a factor in how easy that code is to understand. For example, the concept of merging two lists needs many details added before it can be executed by the computer. Some examples of these details are: how the lists are represented, how storage management is done, which machine instructions are useed, and which registers are used. The more of these details that are present in the source code, the more difficult the code is to understand.

Discussion

The Smalltalk-80 environment is currently the best environment available for exploratory programming. The tools in the Smalltalk environment could be constructed for C or Pascal, but this would still not provide an environment for exploratory programming. The Smalltalk language provides more powerful mechanisms for sharing of code than do most other languages. Without extensive sharing, experimentation is too costly for exploratory programming to be feasible.

The Interlisp-D environment has long been used for exploratory programming. The sharing of code in Lisp is obtained through the use of a common data structure. Usually, whole tools are shared in the Interlisp-D environment, whereas in Smalltalk-80 bits and pieces of code as well as tools are shared. This difference indicates that Smalltalk-80 may provide a better environment for exploratory programming in a general sense. On the other hand much of AI research involves the writing of interpreters. Lisp may be superior for this task. The hardware for the Interlisp-D environment is painfully slow compared to modern hardware, yet Interlisp-D is still being used. This suggest that speed is not the primary concern.

Exploratory programming environments are still in their infancy. Interlisp-D and Smalltalk are still very poorly understood. Until we understand the programming process better, progress in exploratory programming environments is likely to be very slow.

Acknowledgement

Don Birkley, Dale Henrichs, Verna Knapp, Ward Cunningham, Mike Miller, Eirik Fuller, and Norm Delisle willingly shared their Smalltalk experience with me. Ritchey Ruff and Tom Dietterich shared their InterLisp-D experience. Without the generous support of Tektronix, this work would not have been started.

References

1. A. Borning, and D. Ingalls. Multiple Inheritance in Smalltalk-80. *Proc. of the National Conference on Artificial Intelligence*, 1982.

2. N. Delisle. Viewing a Programming Environment as A Single Tool. *SIGPLAN Notices* 19(5), May 1984.

3. A. Goldberg, and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

4. A. Goldberg. *Smalltalk-80: The Interactive Programming Environment.* Addison-Wesley, 1983.

5. R. Johnson. Type-Checking Smalltalk. In 7.

6. G. Krasner, ed. *Smalltalk-80: Bits of History, Words of Advice.* Addison-Wesley, 1983.

7. N. Meyrowitz., ed. *Object-Oriented Programming Systems, Languages, and Applications Conference Proceedings, 1987.* Published as SIGPLAN Notices 21(11).

8. S. Putz. Managing the Evolution of Smalltalk-80 Systems. In 6.

9. D. Sandberg. An Alternative to Subclassing. In 7.

10. E. Sandewall. Programming in the Interactive Environment: The Lisp Experience. *ACM Computing Surveys*,10(1), March 1978.

11. B. Sheil. Environments for Exploratory Programming. *Datamation*, February, 1983.

12. T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. *Comm. ACM* 24(9), Sept. 1981.

13. D. Ungar. *The Design and Evaluation of a High Performance Smalltalk System.* MIT Press, 1987.