Draft for publication

March 29, 1990

Object-Oriented Programming Language Issues for Human Interface Development, the case for Smalltalk.

By
Edward Klimas

The proportion of software associated with human interface has been estimated to be between thirty and eighty percent [Case 89] [MITRE 86]. This software is often of a very high complexity level requiring disproportionate development effort when compared to the rest of the application code. Recent advances in personal computer hardware technology, cost and performance have opened up new opportunities for the practical application of object-oriented programming (OOP) languages to solve this problem. These languages have significant potential for manipulating larger and more complex software applications than current technology. The software industry trade journals are routinely documenting notable software productivity opportunities using object-oriented programming.

The issue of which object-oriented language is best for a given application has been of significant management concern recently. In many applications there is no question that Smalltalk, C++ and Objective-C have distinct features that clearly delineate which language is the best choice. **In the area of embedded real-time firmware for control and device I/O, future software development will probably continue to be handled by C and possibly its object-oriented extensions. The area of human interface intensive software development, however, requires much closer analysis of the issues.** In general, excellent human interfaces have been developed with low level languages such as C and even assembly code for applications ranging from CASE tools to process control among others. Therefore one should not assume that object-oriented languages are necessary for development of good human interface software. Rather, OOP provides a potentially good paradigm for managing the inherent complexity of such applications in a more productive fashion. An empirical test of identical large programming tasks in multiple object-oriented languages is required to test the true limits of the various object-oriented technologies bounding envelopes. Although a preliminary comparative analysis of some OOP language issues has been performed for the development of a simple game [Love 90], to this author's knowledge, no such large-scale replicated testing has been documented.

This document will try to review some of the issues that management should be aware of before committing to a particular object-oriented programming language. This document also tries to promote the concept that **human interface intensive software development needs a high level**

object-oriented language such as Smalltalk to complement the low level language capabilities of C and its object-oriented extensions.

The current debates are centered around two major camps, the C dialects such as C++/Objective-C and Smalltalk. A number of other object-oriented languages also exist, but are not discussed here because they typically do not have the same market momentum or are not amenable to supporting commercial group development efforts.

Issues for consideration:

C++/Objective-C are not alternatives to Smalltalk.
There is a general feeling in the object-oriented programming community that C++'s and Objective-C's founding premises are quite different from Smalltalk's and that they should not be considered as equivalent alternatives [**Stroustrup** 90]. The C language was originally developed as a low level language [**Kernighan** 88] that could substitute for assembly code, with the primary concern for a highly portable, structured language with good execution efficiency, developed using standard college textbook compiler technology. Smalltalk was developed with the emphasis on improved software productivity, reuse of code, and an integrated development environment. The computing resources required to support Smalltalk were of secondary concern during its development, with the vision that one day there would be cost effective hardware to support the language. The recent advent of acceptable performance, low cost personal computers have now made the Smalltalk language a viable cost effective technology. One might consider that the object-oriented C dialects are "better" C's that happened to include a number of solutions to the evolutionary computer science "lessons learned" since the language was initially introduced. Fortunately, the improvements to C++ included object-oriented functions that had evolved from the developments in Simula and Smalltalk. In the ongoing industry improvement of C compilers, one can expect all major C tool vendors to soon support one of the object-oriented extensions to C as a matter of course, but the fundamental principles of C will still prove to be a limitation to achieving the full benefits of the OO paradigm. **Although limited, these object-oriented extensions to C could prove beneficial to some software product lines by permitting greater functionality to be embedded in time critical products such as device drivers and high performance databases.**

Execution Speed: Objective-C and C++ are faster than Smalltalk.
Current literature indicates that Smalltalk is between a factor of two to ten times slower than C++ and Objective-C for traditional applications [**Ungar** 89]. However, as applications become more object-oriented, C++ and Objective-C rapidly lose their performance advantage until, in some cases, Smalltalk will actually outperform the C dialects [**LaLonde** 89]. Some windowing systems developed with Smalltalk/V286 have demonstrated acceptable performance even when operating at XT speeds. In general, computationally intensive algorithms are executed more quickly in the C dialects. However, through various improvements in Smalltalk interpreters and compilers, the differences in speed will be less of an issue in the future. **For human interface applications, the speed of the language is not an issue with current 80286 and 68000 personal computer technology, as the application program normally spends a significant amount of its time waiting for the user to respond.**

Code Size: For Large Systems, C Libraries, Objective-C and C++ consume significantly more memory than Smalltalk.
For large applications, Smalltalk is anywhere from a factor of 5 to 10 times more compact than equivalent C code [**LaLonde** 89]. Smalltalk-80 requires approximately 3Mb of memory and Smalltalk/V286 requires a little over 1Mb memory for minimal functionality. This must be compared with the source and object code for UNIX which has less functionality and requires more than 10 times more space [**Cox** 84]. As another example consider X-Windows which alone consumes more memory and is slower than all of Smalltalk/V. This significant difference in memory size is not appreciated until a paging-memory based system's performance is compared with an entirely memory resident windowing system's performance.

Productivity: Objective-C and C++ are not as productive as Smalltalk.
Although there are a number of studies showing that productivity is primarily impacted by the way people are managed and treated, there are some technical issues related to how OOP productivity is also impacted, as follows:

-**Long compilation times:** Smalltalk, being an interpretive language (not unlike BASIC), can almost immediately respond to changes in the code during development. The ability to make changes and see the results immediately, has been shown to dramatically improve productivity because the code developer can continuously maintain a high level of concentration on the work at hand. The current long compile times required for the C dialects significantly impede productivity as the user waits from several minutes to several hours for results. Productivity is especially a major concern for human interface development because of the historically large number of minor changes that are required during development. Incremental compilation is a recognized benefit, and Parc Place Systems and Glockenspiel International have developed incremental C++ compilation facilities.

-**Future software development should include an integrated environment:** Another productivity factor that is often overlooked is the benefit of an integrated environment that permits developers seamless and rapid access to multiple editing sessions, development tools and immediate compilation and display of results. This concept, originally envisioned by Smalltalk, is currently being duplicated in some C++ and Objective-C product offerings to varying degrees. It is premature to conjecture on the long term viability of these "me-also" functions at this time. However, to improve current software development productivity, **one must adopt an integrated software development environment that permits development of large software packages without the constraint of long compile times for viewing changes.**

-**Support libraries for C++ and Objective-C must be standardized:** Although there are currently no significant standard human interface libraries for C++, one can anticipate more standardized interfaces into IBM/Microsoft Windows, Macintosh and X-Windows to become available (these windowing environments are already supported in various Smalltalk dialects to varying degrees). Objective-C and C++ support different libraries for different platforms and have the associated compatibility problems. For comparison, applications developed under Smalltalk-80 are directly portable to any platform that supports it without change in look and

feel. Applications developed under Smalltalk/V286 and Smalltalk/V-Mac can be modified to run under Smalltalk/V-PM for OS/2 as a Presentation Manager window.

-High level languages are better suited to complex programming tasks: Windowing environments and other advanced human interfaces are generally considered to be complex software systems [**Case** 89]. The fact that C was developed as a low level "standardized assembly language" [**Kernighan** 88] is often overlooked by developers, and significant effort is expended on interfacing the C dialects into suitable windowing libraries (hence, the long development times for DEC's X-Windows and MicroSoft Windows as just two examples). Industry experts have also taken positions:

> "A multi-year study by Arthur Andersen on software productivity led to a conclusion that the object-oriented Smalltalk language is the most productive programming language because it significantly reduces program complexity through its inheritance mechanisms" [**Case** 89].

-Automatic memory management can not be handled by C dialects: This is probably **the most serious shortcoming of the C dialects and is a significant problem for large, complex software systems.** Windowing environments and many other software functions determined at run-time, require careful and efficient allocation and deallocation of memory for the complex data structures involved. Per Brad Cox, the developer of Objective-C, [Cox 87],

> **"The benefit of automatic garbage collection is not small, because it eliminates a whole class of truly nasty bugs.** It eliminates the dangling pointer problem, in which invalid object identifiers (produced by freeing the object they point to) can lie dormant for arbitrarily long periods and then cause hard-to-diagnose problems (crashes) when they are finally accessed. And it prevents the equally dangerous problem in which a long-running application strangles from lack of memory because unneeded objects have not been freed. If the programmer is responsible for freeing objects when they are no longer needed, he must be aware of the entire system to ensure that each object is freed only once during its lifetime and never accessed afterward. This increases program complexity for the programmer tremendously, since it forces every programmer to understand the entirety of the application, not just the interface to the appropriate routines."

Commercial software quality assurance problems with C++ have also been referenced in the literature [**Auer** 89].

> "Objective-C and C++ do not provide automatic garbage collection. The allocation and deallocation of memory for the objects is under control of the developer. In a large and highly interactive application, this burden can be significant. I have heard from one developer, at a company which is working at creating a significant class library in C++, that the majority of their QA effort is spent figuring out where and why memory is not being deallocated."

An associated issue is that Objective-C and C++ developers' training must include significant sections on the intricacies of manual heap management and leak detection.

Large systems need what are called "comprehension avoidance techniques", i.e. minimize what the developer and maintainer need to know to effectively do their jobs. Requiring the developer to manually allocate and deallocate a data structure places the burden of knowing exactly how that data structure is used throughout the system at all times. For small real-time systems this is not an unreasonable situation, but for large programs it is a potentially unrealistic expectation that significantly impacts productivity and quality.

Smalltalk has automatic garbage collection facilities which have been optimized to consume less than 4% of the total CPU resources. This functionality can not be expected to be incorporated into C or the 100% compatible dialects of C due to inherent limitations in the definition of the C language [**LaLonde** 89]. (It is impossible to perform automatic garbage collection with C's unconstrained ability to change pointers to arbitrary data types i.e. coercion.) **If programmers are not expected to develop their own code for storing, managing and retrieving disk files, they should not be required to perform comparable manipulations with memory management!**

These productivity issues should manifest themselves in the final development costs of an application. Some uncontrolled embryonic test results are emerging in the OOP community that do show sizeable differences in the cost of delivered systems based upon the OOP language being employed. Smalltalk developments can cost between $6 to $1 per source line of code depending upon the amount of code reused, while $50 per source line of C++ code can also be expected [**Weiss** 90]. There are a number of issues that need to be investigated to ensure a fair comparison for this data and it is not realistic to draw firm conclusions from this data at this time, however the trend seems to corroborate some of the points previously mentioned.

A rule of thumb in new software development is that it usually takes three attempts (major revisions) of the software to achieve acceptable results. The current argument is, that since Smalltalk is estimated to be 5 times more productive than traditional programming languages [**Barry** 89] and twice as productive as C++ on a source lines of code basis [**Love** 90], one can save up to 12 times the development cost over traditional languages using Smalltalk versus 3 times the development costs with the C dialects.

With market windows continuing to become shorter in the future and increasing demand for robust software, **one might consider a strategy that would promote initial human interface product development and delivery in a high level language like Smalltalk, and then, once the performance bottlenecks are identified, rewrite only that necessary functionality as C** or assembly code primitives interfacing into the existing software framework. If the market windows of opportunity continue to shorten, the latter approach may in fact be the only viable avenue for market competitive human interface developments.

Run-time issues and tools: Smalltalk-80 was initially developed with single user support in mind. This led to criticism of its lack of team programming support and lack of support for commercial

deployment of the resulting code. Smalltalk/V developers have responded to these needs with mature tools to provide the following solutions:

-Object Technology International is marketing a number of tools for supporting Smalltalk development for commercial products. These include:

ENVY/MANAGER, a code management system that supports team programming with Smalltalk/V286 over a local area network tied into a server. The system works with NOVELL and DEC/VAX-PCSA systems and is in beta test at a number of sites.

tools for stripping out unnecessary parts of the Smalltalk environment and converting the result into executable (.EXE) files or even "ROM"able code.

a common image of Smalltalk that will be portable across several platforms.

-Smalltalk/V /V286 /V-Mac and /V-PM support interfacing into other languages such as native assembly language, C, C++, Fortran, etc.. through a function called "user primitives". For /V & /V286 each primitive can be up to 64Kb in size. Smalltalk/V-PM permits much larger size primitives through the use of OS/2's Dynamic Link Library (DLL) functions. This feature provides a powerful interface to time critical code for real-time hardware drivers and existing applications.

-Smalltalk-80 supports interfacing into C programs developed under the METAWARE High C compiler rev 1.4 or higher.

-The Smalltalk source code can be totally hidden from the user or partially supplied with the run-time to permit users to seamlessly "hook in" their own Smalltalk applications if appropriate. Within the limitations of the primitives, users can also include foreign language device drivers.

-Digitalk Inc. offers several run-time licenses for their versions of Smalltalk. The low end Smalltalk/V can be run-time licensed for a flat $500 per year per product. The more advanced Smalltalk/V286 and /V-Mac can be run-time licensed for a flat $50 per copy with an unlimited copy, $200,000 royalty buy-out cap (other limited royalty agreements can be negotiated). Smalltalk/V-PM for OS/2 and the presentation manager will compile the Smalltalk code into an OS/2 executable (.EXE) file which can be distributed royalty free. The Smalltalk/V /V286 and /V-Mac run-time licenses include tools for stripping out unnecessary Smalltalk code as well as a tool to permit display of user defined information at start up.

-Parc Place offers similar Smalltalk-80 run-time support to Digitalk's for all of its platforms. Runtime licenses are available that range from $150 per copy for the 80386 MS-DOS and Macintosh to $595 per copy for all of their other platforms.

All major hardware platforms have support: All of the object-oriented languages are well represented on a number of platforms although only Smalltalk-80 claims to be totally platform independent and not require any porting of source code to different platforms. Smalltalk/V286 is currently the only Smalltalk dialect that supports true team program development and sharing of

code through a third party code management system. Although currently, C++ and Objective-C have some restrictions on the availability of libraries across platforms as well as some dependence upon operating system versions, these limitations should be rectified as the C++/Objective-C market increases.

Language	Platforms supported
Smalltalk/V	MS-DOS, IBM-XT, AT, 386
Smalltalk/V286	MS-DOS, IBM-AT, 386, 486
Smalltalk/V-PM	OS/2, IBM-AT, 386, 486
Smalltalk/VMac	Apple Macintosh, SE
Smalltalk-80	386 MS-DOS PC's with MS-Windows, DECstation, UNIX SUN, Macintosh, Apollo, HP9000
Objective-C	UNIX SUN, MS-DOS, IBM-PC's, NeXT
C++	UNIX, ULTRIX, SUN, MS-DOS, OS/2, IBM-PC's, IBM-RT, Macintosh, VAX/VMS

Industry leaders have recognized Smalltalk's merits: The benefits of Smalltalk for graphical user interfaces have been recognized by industry leaders. For example:

IBM's Visual Languages for Interfaces group in the User Interface Institute of IBM T J Watson Research Center has been involved in object-oriented programming since 1983 and currently has a group of Smalltalk programmers involved in user interface research [OOPSLA 89].

Alan Kay, Apple Fellow, Apple Computer, the driving visionary behind the Apple Lisa and Macintosh graphic interfaces, has publicly endorsed Smalltalk/V as the Smalltalk he uses [Digitalk 88].

Arthur-Andersen is promoting the use of Smalltalk/V-PM as the object-oriented language of choice and has developed several commercial CASE tool packages using it [SCOOP 89].

Microsoft Chairman, Bill Gates is publicly promoting Smalltalk/V-PM as "the right way to develop applications for OS/2 and Presentation Manager. OS/2 PM is a tremendously rich environment which makes it inherently complex. Smalltalk/V-PM removes that complexity, and lets you concentrate on writing great programs." [Digitalk 89].

At least one industry trade journal has given Smalltalk/V-PM the highest rating of all of the current development tools for the notoriously complex OS/2 Presentation Manager [Rosch 89].

Training

Training both formal and on-the-job is important. Smalltalk has been recommended as the best way to learn object-oriented programming irrespective of the final implementation language because it is a completely object-oriented environment and doesn't permit the programmer any way to subvert the OO paradigm. Smalltalk also supports mature libraries for browsing exemplary code as a means of learning OOP. Although Smalltalk is now being adopted as the introductory computer science programming language in several universities, many universities are still only offering C++ or Objective-C. This approach has been criticized because it gives the student's a paucity of object-oriented code from which to learn good practices and does little to prevent students from subverting the language and continuing to program in non-object-oriented fashion.

An acceptable alternative appears to be a path that has been followed by some larger companies, where periodic in house Smalltalk courses are offered and video-tapes and course materials are available for self paced training periodically supplemented by recognized outside consultants (i.e. consultants with proven track records in delivering commercial OOP based applications). If the personnel have had formal computer-science or engineering and software backgrounds, this approach coupled with on the job training with experienced personnel, appears to produce productive results after about 60 to 90 days. Eventually in-house experts will emerge that can play a crucial role in further supporting the internal staff. Preliminary results show that the OOP **retraining experiences are not much different than with "traditional" languages** (i.e. as with Cobol programmers learning Fortran, 10% immediately adapt, 80% vacillate back and forth for a while, 10% never adapt). This would argue for a number of small low visibility projects (e.g. rapid prototypes and in house tools for testing) to be initially pursued in house, to develop a critical mass with the proper skills, before large projects are attempted.

There is an unfounded fear of significant culture shock associated with converting C programmers over to Smalltalk programming and hence the perception that C++ or Objective-C should be used because they would be less disruptive. Initial experience seems to indicate that this is not a well founded concern and that in fact, developers with a few years of C programming experience, readily adapted to the OO paradigm and Smalltalk. Those who have had previous experience interfacing into C based windowing libraries quite readily adapt and appreciate the integrated nature of the Smalltalk environment and window management system.

Another misperception is that Smalltalk is more difficult to learn than the hybrid languages. In general this is not true. **The syntax of Smalltalk is much simpler than C++.** To be productive in any language, one must learn the libraries. The larger the libraries, the more potential exists for software reuse and greater reliability. The Smalltalk "learning curve" must be put in perspective against the equivalent training for C, where a programmer must learn an editor, a graphics kernel, a windowing system, a file interface system, a database system and numerous development and debugging tools. When analyzed in this light, Smalltalk is easier to learn than the hybrid languages because all of the features are integrated into a common seamless environment. Smalltalk has much larger libraries (classes) than the hybrid languages. For comparison, Smalltalk-80 supplies over 240 classes, Smalltalk/V over 110 classes and Objective-C 20 to 80 classes. Currently C++ does not typically come with any standard class libraries. (It is difficult to create a truly reusable class in C++ because the type checking prevents general classes, like stacks of anything.) The

larger the libraries, the longer it will take to learn a language. **Therefore although C++ is claimed to be easier to learn than Smalltalk, there is much less learned [Auer 89].**

Conclusion

Although C and its object-oriented dialects are expected to continue to be well suited for the development of low level, speed critical, embedded real-time control software, they have significant productivity, quality and cost issues associated with their use for development of the large complex software systems. Modern window based graphical user interface environments although not computationally intensive, are relatively complex software systems, even when supplied as libraries. A high level language such as Smalltalk compliments the computational affluence of current 80*86/680*0 personal computers for the rapid development of robust complex graphical user interface based systems. Because of the trend for increased software content related to human interface, Smalltalk can be expected to fit well with many future product development requirements.

References

[Auer 89] Auer, K., *Which Object-Oriented Language Should We Choose?*, HOTLINE on Object-Oriented Technology, Nov. 89, Vol. 1, No. 1.

[Barry 89] Barry, Brian. *Prototyping a Real-Time Embedded System in Smalltalk.* Proc. of OOPSLA 89, New-Orleans. La., ACM SIGPLAN.

[Case 89] Case Outlook, *The Case State-of-The-Industry Report:*, v.89, No. 3., May/June 1989.

[Cox 84] Cox, B., *Message/Object Programming An Evolutionary Change in Programming Technology*, IEEE Software, January, 1984.

[Cox 87] Cox, B., *Object-Oriented Programming An Evolutionary Approach*, Addison-Wesley, 1987.

[Digitalk 88] Digitalk, *Smalltalk/V286 Tutorial and Programming Handbook*

[Digitalk 89] Digitalk, *Smalltalk/V-PM Tutorial and Programming Handbook*

[Kernighan 88] Kernighan, B. W., Ritchie, D. M., *The State of C*, BYTE, August 1988, pp. 208.

[LaLonde 89] LaLonde, W., et. all., *The Real Advantages of Pure Object-Oriented Systems or Why Object-Oriented Extensions to C are Doomed to Fail*, Proceedings of IEEE COMPSAC 89, pp. 344-350.

[Love 90] Love, T., *GREED: A Complete Example*, HOTLINE on Object-Oriented Technology, February 1990, Volume 1, Number 4.

[MITRE 86] MITRE, *Guidelines for Designing User Interface Software*, MITRE report number ESD-TR-86-278.

[OOPSLA 89] *Object-Oriented Programming Systems, Languages, and Applications Advance Program*, 1989

[Rosch 89] *PM Developers Kits Offer Different Paths to Similar Results*, PC Week, December 11, 1989, pp. 79, 81, 83.

[SCOOP 89] Digitalk, *SCOOP Newsletter*, August 1989.

[Stroustrup 90] Stroustrup, B., *On Language Wars*, HOTLINE on Object-Oriented Technology, January 1990, Volume 1, Number 3.

[Ungar 89] Ungar, D.,Chambers, C., Lee, E., *An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes*, OOPSLA 89 Proceedings.

[Weiss 90] Weiss, R., *NEXT-GENERATION DESIGN ENVIRONMENT TAKES SHAPE, Mentor details 8.0*, Electronic Engineering Times, February 26, 1990.

Animating Programs Using Smalltalk

Ralph L. London and Robert A. Duisberg
Computer Research Laboratory, Tektronix, Inc.

An animation kit can be used to explain how a program works by creating graphical snapshots, animations, and movies correlated with the program's actions. Such a facility could play an important role in program design, development, and testing.

The availability of today's powerful personal workstations with high-resolution bit-map displays and pointing devices makes possible the creation and display of drawings containing a wide assortment of characters, fonts, icons, and figures, all of which can be continuously moved for realistic animation. We are currently involved in using such animation to visualize programs and algorithms by creating graphical snapshots and movies correlated with the programs' actions. Such a facility we hope will provide programmers or computer users in general with an understanding of what the programs do, how they work, and why they work. It also will give users visual feedback as a program and its parts are being executed. This animation system will provide pictorial representations of those data structures, at the proper level of abstraction, which are used by a program. Standard representations of internal data structures, such as linked lists or arrays with separate index variables, are often insufficient because the viewer must mentally transcribe such representations to the abstractions involved in the use of those structures. We use the type of diagrams or sketches a programmer draws at a desk or wallboard, or the kinds of schematic figures found in a programming or data structures text; fortunately, we do not need pictures with exquisite shadings that re-create photographs. Such figures change to reflect the changes during the execution of the program. People's apparent tendency to understand by visualizing spatially the abstractions that constitute the intention, or "meaning," of a program is exploited by the system. For example, one visualizes in two dimensions the trees or matrices manipulated by a program, whereas the code is always linear and sequential.

Value of animating programs

There are numerous reasons for taking this approach. We expect such a facility to be useful, probably even important, for designing and developing programs, for debugging them, for monitoring their performance, for documenting and describing them, for showing them to colleagues and to technical and managerial visitors, and for directing and interacting with executing programs. Furthermore, such a facility should serve to recall to the program's author and others the inner workings of a program after months of nonuse, permitting, for example, the enhancement and changing of the program. New project members and

0018-9162/85/0800-0061$01.00 © 1985 IEEE

users could benefit from animation showing the programs with which they will be working. We believe that animation has a role to play in the transfer of technology from the laboratory to areas of further development.

Animation has also been used successfully in the classroom as an important teaching technique for courses in introductory programming, algorithms and data structures, and other areas.[1] Our experience suggests the usefulness of providing students with an "animation kit"—a set of easily learned and easily applied tools—and asking them to study an algorithm by creating animations themselves, thus demonstrating what they feel to be important in the algorithm. They would be graded by the quality of their animation and the number of facets of the algorithm that are shown. We were led to this after animating an algorithm (for finding longest common subsequences) that was and still is of importance to a colleague, but about which we, the animators, initially knew very little. From the animation experience alone, we acquired real understanding of the algorithm's workings. Finally, we expect to apply related or similar techniques to such objects as requirements, designs, specifications, and partial programs.

We recognize that an effective animation system for use by others can require no more than an acceptable level of their effort, much like a novice system does. In order to achieve this ease of animation, our work has shown the importance of isolating as much as possible the graphics code from the actual code of the algorithm and thus promoting the modularity and portability of the graphics routines. Suitable isolation was achieved with a refinement of Smalltalk's Model-View-Controller constructs, but we soon encountered some limitations in this method. Constructing animations is still a programming-intensive task. These limits have prompted us to explore more general techniques for the implementation of the animation kit, in particular the use of constraint languages to express and maintain relations between views and the objects they represent. We say more about this in the final section.

Related work

Previous examples of animated programs may be found in the superb (and costly to create) color sound film *Sorting Out Sorting* by Ronald Baecker[2] and in the impressive and effective Brown University Algorithm Simulator and Animator, or Balsa, software system by Brown and Sedgewick,[1] which has been used for many examples. Additional instances, one as early as 1966, may be found in their discussion and its references.[1] Graphical displays of static pictures of data structures are provided by the Incense system of Myers.[3] Constraint satisfaction in ThingLab[4] produced restricted animations in response to user requests to update a figure. Simulations of harmonic motion of a spring, of vehicular bridges, and of the use of an abacus may be found in the Electronic Encyclopedia work by Weyer and Borning.[5] Another related project is the Programming by Rehearsal World by Gould and Finzer.[6] This system is a Smalltalk-based visual programming and design environment in which "performers" can be moved around on "stages" and taught how to interact by sending "cues" to one another.

From the animation experience, we acquired a real understanding of the algorithm's workings; we expect to apply related techniques to requirements, designs, and specifications.

Like Balsa we emphasize customized diagrams for particular programs and multiple views of programs in action rather than just general-purpose, static displays. We differ, however, by emphasizing the creation of smooth transitions between graphical images to help viewers follow the changes rather than rely on updates alone. We also tend to emphasize small examples with more detail, an appropriate complement to the impressions gained from Balsa's larger examples. We are using an object-oriented approach and a different programming environment that, among other things, provide a convenient viewing framework. Finally, our work is aimed primarily at industrial prototyping and simulation.

Other work related to ours involves programs that can explain their own actions. We cite two representative examples. First, medical consultant programs, such as the Digitalis Therapy Advisor, have an ability to explain why a question is being asked, i.e., essentially to explain the context in which the program is operating and value of the question[7]; see also the rule-oriented part of Loops.[8] Second, the Smalltalk environment can explain variable names or message selectors in methods, generally by providing a comment, but sometimes also by providing an expression to be evaluated in order to obtain further information perhaps through a browser.[9] The explanations are textual although diagrams and animations are certainly possible. Our emphasis is on the latter two. These can be realized by using the facilities of the Smalltalk Graphics Kernel to produce graphical explanations.

Moving to Smalltalk

Our original animations were constructed using Pascal with character graphics and some ability to highlight, shade, underline, and boldface. Ordi-

nary compilers were used without any special environment. Very exploratory in nature, these attempts confirmed the value of animation (if such were necessary) but quickly exposed the limitations of character graphics and the need for animation system support.

Magpie,[10] an interactive environment for Pascal, has several capabilities of interest to animation. With Magpie's event monitor mechanism, variables can be marked so that a variable-specific procedure can be executed when the variable is accessed or changed. This provides a capability similar to the active values of Loops. While this facility is often useful, there are many times when updating at each variable change is inappropriate, and it is better to update at selected points with a more global outlook. Event monitoring also allows procedure-specific routines to be called at procedure entry and exit. These facilities of Magpie would allow a convenient separation between the program being animated and the animation code. It is still necessary to invoke animation code as, say, a procedure call at certain points not covered by these capabilities, i.e., to have an equivalent capability to the interesting events of Balsa. With just event monitoring and the displaying of the procedure call stack, the Magpie authors were able to construct interesting demonstrations and animations of changing data structures and computational progress. Examples included simple sorting, the towers of Hanoi, and binary search.

While Magpie could, in principle, have provided us access to the same graphics it uses, it could not support the size of programs we would soon develop. However, Smalltalk* is a powerful alternative that we are able to exploit to obtain a more general and productive programming environment plus superior graphics and some system-supplied animation tools.[11] The Balsa designers, instead of using a system such as Smalltalk, "chose to build a tailored special-purpose system [mainly because] the real-time dynamics of the programs in operation is of fundamental importance: we were not prepared to pay the performance penalties inherent in a general-purpose system."[1] We do not criticize their choice; were we in their position, we would likely have chosen similarly. However, we are fortunate that our colleague Allen Wirfs-Brock has written suitably fast Smalltalk interpreters[12] executing on our Magnolia* and Tektronix 4404 workstations.† For the examples we have run so far, we have always been able to provide animations with appropriate real-time dynamic properties. Indeed, at times it is necessary to include programmed delays so that a viewer sees enough. And, of course, the animations must often pause until the user signals (usually with the mouse) to proceed. We should note that while we have made effective use of Smalltalk in our work, there is much more for us to learn about its best and proper use, optimal style, and efficient program organization techniques. We have not had to perform deep optimizations to gain the necessary speed; just avoiding obvious inefficiencies and poor practices have so far sufficed.

Object-oriented programming lends itself to program animation; still additional layout information must be provided or generated by default.

Even without the graphics, Smalltalk is attractive as a programming language and environment for animation. The philosophy and discipline of object-oriented programming in general, and Smalltalk in particular, lend themselves quite well to the task of program animation. There is a certain naturalness in representing a data structure as a self-contained object, and an algorithm working on that data structure as a method executed within/upon that object; this approach has been widely advocated. Further, a displayed representation of that object is naturally thought of as a "view" thereon. If such a representation dynamically reflects the changing state of the object as it evolves during the execution of the algorithm, all the basic elements of an animation are in place. Still, a great deal of additional layout information must be provided or be generated by a default procedure in order for the view to display the object state. The view must also maintain some kind of spatial map between display objects at which the user may point and their corresponding parts in the object being animated, because the user must communicate with the executing algorithm through what is presented. Even without user pointing, the spatial map is necessary just to display a view incrementally.

Smalltalk for controlling views

Smalltalk currently provides what is called the *Model-View-Controller*, or MVC, system that, though presently undocumented in the three Smalltalk books, pervades the system implementation of the display interface. For example, mouse menu messages, "what-you-see-is-what-you-get" text and

*Specifically, Smalltalk-80, which is a registered trademark of Xerox Corp. In this paper Smalltalk means Smalltalk-80.

*Magnolia is an internally developed, single-user, 68000-based workstation providing a high-resolution bit-map display and computation resources comparable to a DEC Vax 11/750.

†For benchmarks for the 4404, see *Smalltalk-80 Newsletter*, No. 4, Sept. 1984, p. 19. Without changing a single character, all of our work runs on this workstation at least as fast as on the Magnolia.

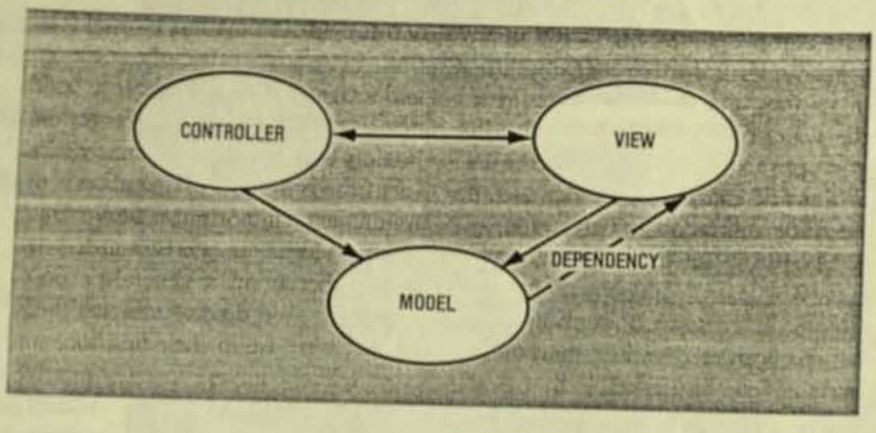


Figure 1. The typical Model-View-Controller structure.

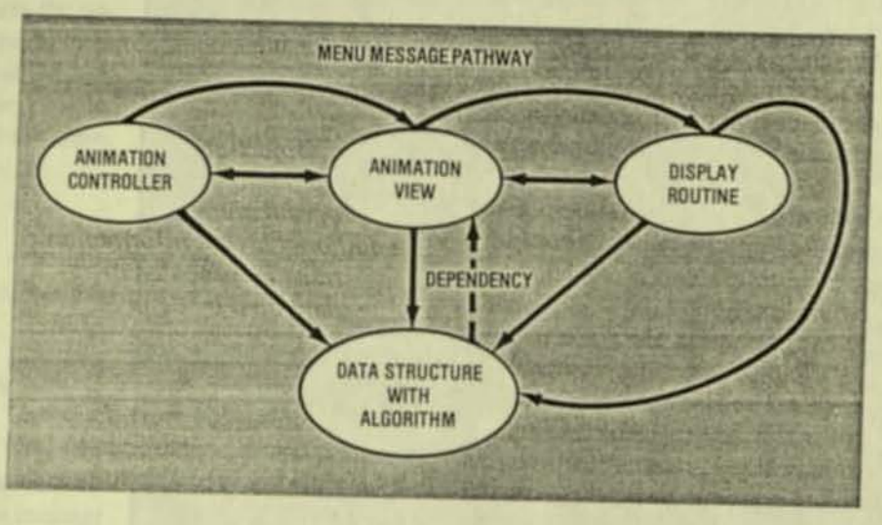


Figure 2. The animation viewing structure with the added particular display routine and circuitous menu message passing.

code editing, scrolled windows, and process scheduling are all in the Model-View-Controller paradigm. MVC seemed to be a reasonable tool for building an animated view on algorithm execution. In any case, much may be learned from the exercise of augmenting MVC to accommodate animation by discovering what structures prove necessary to the task.

In the MVC scheme, the model may be any Smalltalk object. A view is taken thereon by creating particular instances of classes View and Controller and connecting the pointers as shown in Figure 1. The view object takes care of such things as framing, labeling, scrolling, bordering, and transformations from local view coordinates to display coordinates. The controller handles the mouse menu interface and window scheduling; the set of all controllers is polled by the process scheduler to learn if any window wants control (typically when the mouse has been clicked within some window). It is of interest that in this structure, the model has no direct pointers back to any part of the viewing structure. Rather, a model's view is accessed through the "dependency" structure inherited from class Object, which is a list of all views open on an object. The model may then broadcast a message to all of its dependents, without any knowledge of how many or what kind of views may be open on the model. It is by using this built-in mechanism of Smalltalk that we have implemented the equivalent of Balsa's interesting events and Loops' active values. The sequential activation of the views in the dependency list precludes simultaneous updates of multiple views on a single object.

The principal refinement of this structure for animation views is the addition of a particular display routine to the view; see Figure 2. This object contains the specific methods for creating the displayed image of the model representing its current state. Further, the menu message receiver to which the controller passes messages is no longer the controller itself as in most system views. Rather, a menu message is first passed to the display routine that knows the spatial map between images and model parts, so that the user can select the part to be changed and the

display routine interprets the user's action and passes a message on to the model itself.

In our implementation the broadcast of an InterestingEvent* within a method typically takes the form:

```
self broadcast: #update:
    with: (InterestingEvent
        of: # operation with: value
        on: actor).
```

Here an instance of the class InterestingEvent is simply a package to bundle up whatever information the display routine may need to perform the update efficiently. To insert a probe onto an ActiveValue one opens an AnimationView onto some instance variable, say x, within the animated structure. Then instead of writing

x — newValue

when assigning to that variable, one simply writes

x changed: (x — newValue).

The "changed:" message, like the "broadcast:" message, broadcasts to all of x's dependent views the message "update:," sending as an argument the value of the assignment statement, which is newValue. Other syntax that is even less intrusive has been suggested to us.

Figure 3 shows, as an example of the virtue of active values, our animation of Hunt and Szymanski's algorithm [13] to find the longest common subsequence in two strings. The algorithm maintains a "threshold array" defined as $T[i, k]$ = the least j such that the two strings $A(1{:}i)$ and $B(1{:}j)$ have a common subsequence of length k. (This definition seems remarkably obscure to intuition, but its meaning becomes much clearer when one is able to watch the animated array grow.) In the animated view there are three dif-

*The typography follows Smalltalk conventions.

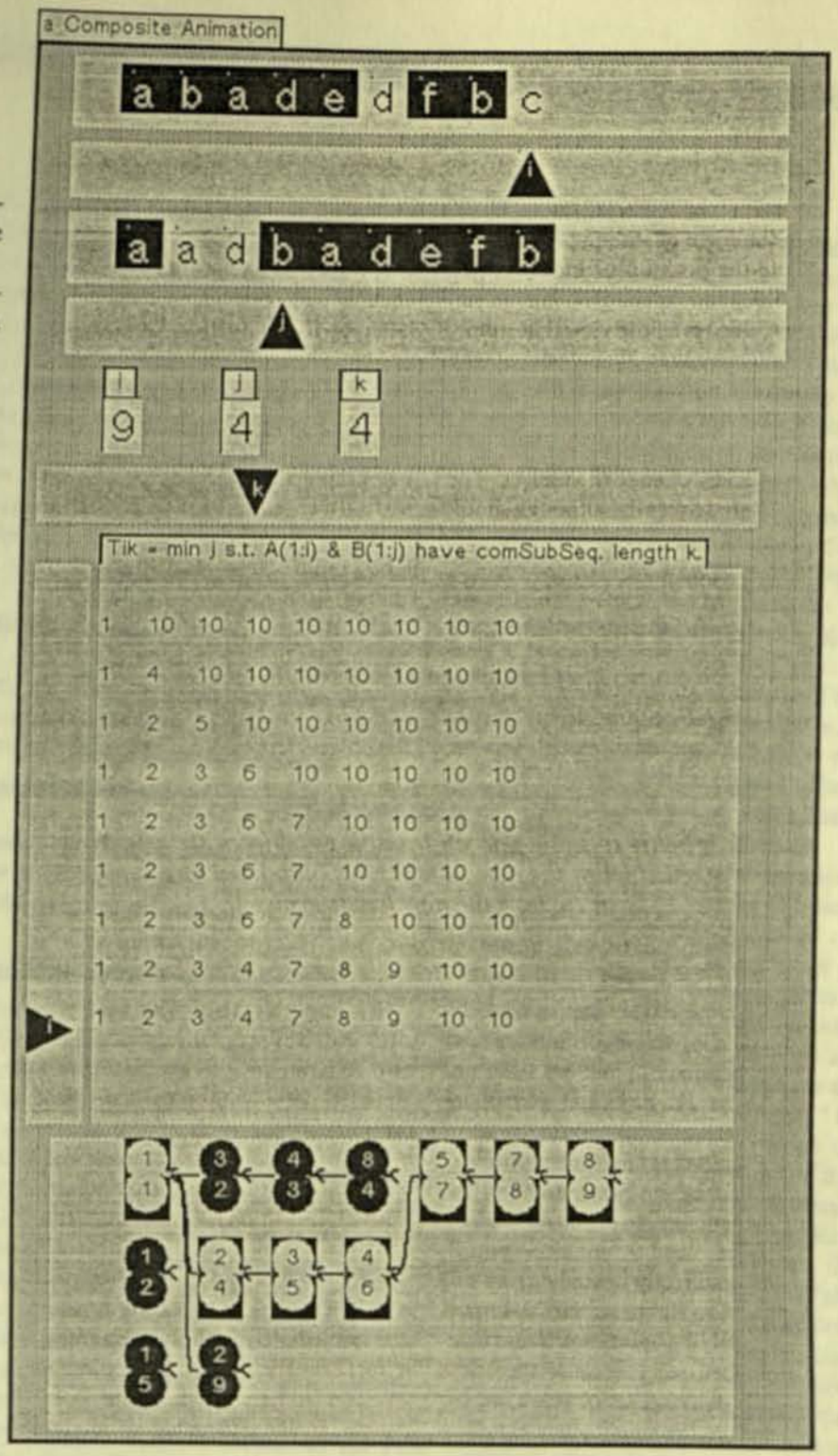


Figure 3. Animation of Hunt and Szymanski's algorithm for finding longest common subsequences in two strings. The algorithm consists of a pair of nested loops in which the index *j* into the second string scans to the left in the string looking for a match with that character pointed to by the index *i* in the first string. If a match is found, the matched pair of characters video-reverses, and the value *j* migrates in a data lozenge from its value window down to the growing threshold array *T[i,k]* and from there to the linked list view at the bottom of the display, where it is joined by its matched *i* value. The longest linked list through such bonded index pairs allows the retrieval of the common subsequence, indicated finally by video reversal.

ferent views open on the index *i*: two are in the form of sliding pointers that indicate the position of index *i* in the first string and in the growing matrix, and the third is a little view that simply shows *i*'s value. The code of the algorithm reflects none of this, but by assigning into *i* in the manner described above, all of these views on *i* automatically change themselves. The updates are sequential although simultaneous updating would be preferred.

Creating individual animations

Creating an animation starts by coding the algorithm cleanly just as one would an unanimated version. There may be advantages in developing and coding the algorithm together with its animation, but we have not done this yet. In any case, we then insert a few appropriate broadcasts of InterestingEvents and probes onto ActiveValues. In this way, much of the viewing structure, in particular the AnimationView and AnimationController, will be directly portable from one animation to the next. This is important in applying animation to aid in algorithm discovery because there is a generally constant set of views, as, for example, in the open "Pancake Flipping" problem.[14] The entire viewing structure can stay intact because, with the minimal effort involved in inserting probes and InterestingEvents, one can install a new algorithm to see what it does, just as one would install a new slide under a microscope.

What will be viewed is a graphical representation of the essentials of the algorithm, which means one or more pictures of the central data structure must be created and updated as the execution of the algorithm proceeds. If enough details are suppressed in the latter or if it is sufficiently different, one can view the graphical representation as an abstract representation of the internal state. In many cases it is possible and appropriate to present just graphical images with little or no textual representation; i.e., certain necessary but secondary information is best omitted to avoid distraction and clutter. It may be desirable for individual visual parts to have some capabilities built into them, for example, the possibility that the user point at them to get a menu of commands to the objects represented.

Often it is better to show smooth transitions between states; viewers are not startled when the new state flashes onto the screen and they can see how the new image evolved.

It is important to provide before and after views of program states, either as two separate images or, better, as an updated view with only the altered parts changed, i.e., without the distraction and cost of a complete redisplay. Often it is even better to show smooth transitions between states; if a structure changes and the new state simply flashes onto the screen, the viewer is typically startled and cannot see immediately without some mental effort how the new image (could have) evolved from the previous one. For example, in the animation of the Producer-Consumer-RingBuffer system in Figure 4, production of a data element is shown by the creation of a circular data lozenge,* a black dot with a symbol printed in it, which moves smoothly from the producer to the monitor and then down into the next open slot in the buffer where the symbol is deposited. (If the buffer is full, the monitor sends the lozenge back to the producer, which then videoreverses to indicate that it is blocked, awaiting a notFullSignal; attempted consumption from an empty buffer causes analogous behavior.) Similarly, when the pointers in the buffer advance, they do so smoothly. Likewise, in the longest common subsequence animation in Figure 3, values of the indices are packaged into lozenges, which then migrate from the index counters into the appropriate place in the growing matrix and then into linked lists of pairs of indices. This effect was produced efficiently not by the usual animation technique of redrawing the updated display offscreen and then redisplaying, but rather by creating a class called Lozenge.

Each instance of class Lozenge contains internally two forms (bit maps), one of which is an instance of class OpaqueForm that shows what the lozenge looks like. The other stores the background that the lozenge, when displayed, conceals. A lozenge also keeps track of its current position and keeps a pointer to its "referent," the object being animated of which the lozenge is the graphical image, so that the actual data object may be accessed through the position of its graphical representation. Upon receipt of the message "moveTo: newPosition" the lozenge divides the vector from its current position to the newPosition into ten parts and successively displays itself along the path while restoring the background. This special treatment of moving elements results from the rigidity of subview placement within MVC. In a more uniform treatment, one would like to think of such moving lozenges as views on the data element in the algorithm whose position is constrained to correspond in some way to where the datum is stored, i.e., in a "port" in the producer or monitor. This approach is being explored in the development of the animation kit.

*Which we call an "MnM," close to a registered trademark.

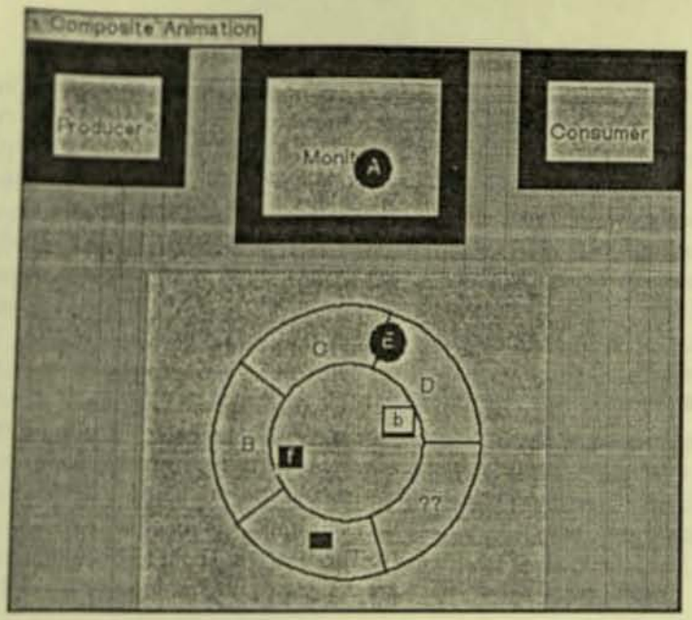


Figure 4. The Producer-Consumer-RingBuffer example. The moving lozenges are caught in stop-action: A has been removed from the front of the queue and is on its way to the consumer through the monitor, and E is on its way to the newly available slot (denoted by ??) at the back of the queue. The front pointer and the currently unavailable cell have already moved, but the back pointer has yet to move.

The particular form of the animating routine may be built directly from Smalltalk primitives and supplied methods, or it may be built from a library of components of sliders, data lozenges, pointers, etc. Discovering, invoking, composing, modifying, and packaging components to meet our exact needs are no different from other programming applications. We began to accumulate experience and "reusable" components applicable in later examples from one example. These include the moving lozenges, expanding and contracting regions, AnimationView and AnimationController, and user control of an animation.

Viewers can intermix single-stepping through the events of an animation and proceeding without pauses by using the mouse as "brake and accelerator pedals." We have not yet used views of code being executed, for example, single-stepping through statements, because we believe such views are usually unnecessary. There are even examples of recursive programs that can be animated effectively without explicitly showing the recursive control stack. Thus, in quicksort, say, each recursive call shows (animates) only the subarray involved in that call, but the subarray is positioned properly with respect to the full array. Parsing and tree walking algorithms can be animated similarly.

Interesting events are closely related to invariant assertions. It should not be too surprising that the places to put interesting events include the beginning and ending of a routine, the initialization and exit of a loop, and at least one point within every loop. Thus, it turns out that interesting event locations are essentially the same locations at which one might place invariant assertions, were one to verify an algorithm to be consistent with its specifications. We can think of animating a program as illustrating the invariants and how they are maintained. The design of an animation is often influenced by considering the task to be one of maintaining the visual invariants, especially those between the internal concrete representation of the program state and its graphical representation. In turn, program development is simplified when programmers see new ways to maintain program invariants. This happened to us in a small way, which we describe in the Dutch national flag example in the next section.

As we proceeded in creating the animation, we usually encountered the experimental nature of the process. What seemed to be a good approach would need to be substantially modified because the animation would reveal less than we expected or because just seeing the animation would suggest better ways to us or to other viewers. Because the purpose of an animation is to use visual cues to communicate an understanding of a process, it must necessarily involve complex psychological and aesthetic issues, including whether the animation creates the illusion of what it purports to represent. Such issues are not easily delineated, or even well understood, so this experimental and iterative approach seems unavoidable for now. Style, taste, and artistic creativity are all important here for the same reasons that some user interfaces are better than others.

Animations can, of course, mislead users, or outright lie, about the workings of an algorithm. What an animation should or might show may depend on the expected audience. A visitor

once asked us if we had any experience with naive users; in particular, were there concepts obvious to experienced programmers that novices surprisingly misunderstood? Because our audience has been veteran programmers and because we usually talked about the algorithm during its animation, we have had no conceptual problems. However, in an animation of a binary search where the number being sought was highlighted all the time (and the active region of the search would shrink at each iteration), one novice viewer did ask, in effect, "If you already know where the number is, why are you doing the binary search at all?"

Further examples

One of the first examples we did in Smalltalk was a replication of the selection sort that is done in the movie *Sorting Out Sorting*: the elements of the array of integers being sorted were represented as a row of varying-length sticks, each proportional in length to the element and each extending upward from a common horizontal line. As two elements (sticks) were compared, they were highlighted. At the end of each search, the two elements to be swapped were highlighted. Then the two elements would simply reappear in their new locations. Later we made the two elements move continuously from original to new locations. This was all accomplished straightforwardly with messages to the array object to invoke its methods for displaying itself and its elements in this representation; no MVC mechanism was involved.

We also animated an abstract queue or ring buffer and later included it, as noted above and in Figure 4, as a subpart of the Producer-Consumer-Ring-Buffer system. The internal representation is an array made circular by modular arithmetic with pointers to represent the front and back of the queue. The graphical representation was two concentric circles with the space in between divided into slots for elements. As we queued elements into this circular structure and removed elements from it, the two pointers continuously moved inside the smaller circle. To distinguish the full and empty queues internally, one array location was always unavailable; this changing location was specially identified. From this animation we were able to answer the question, "When, if ever, do the two pointers cross?" Our prior experience in several contexts with this well-known data structure had not provided the correct answer.

One of our first examples was a sort of an array of integers represented as a row of sticks of varying length: as two elements were compared, they were highlighted.

The Dutch national flag problem[15] led to an interesting series of animations. The problem may be briefly stated as follows: For a row of buckets, each containing one pebble whose color is either red, white, or blue, rearrange the pebbles in the order of the Dutch national flag, i.e., first the reds, then the whites, and finally the blue pebbles (one, two, or all three colors may be absent). Only swaps involving two pebbles are permitted, the color of each pebble may be determined only once, and only a very limited amount of memory is available, independent of the number of buckets, so that no arrays may be used by the program (beyond the bucket array, of course). A reader who has not seen this problem before may wish to attack it before reading the next paragraph. It is not necessary to see Dijkstra's discussion and solution to understand the animation of this example, but his discussion is illuminating.

The animations (see Figure 5) represent the buckets and pebbles as a flag of adjacent, vertical stripes in three shades: gray, white, and black. Underneath the flag is a band that represents the four different zones of pebbles: established red, established white, established blue, and as yet uninspected. There are pointers that show the boundaries of each of the three established zones. The band of zones and the pointers together illustrate the invariant that is the key to discovering the solution and to understanding it. There is an "eye" icon that shows a pebble having its color determined, after which the corresponding stripe contains a "punch mark" to indicate the expiration of the one-time capability. Those stripes about to be swapped are designated by flashing a small portion of the stripes. Over the time we developed the series of animations, each of the four changing parts (eye locations, swaps, pointer changes, and zone expansions) went from instantaneous change to smooth motion. In particular, the smooth swaps of two stripes were first accomplished by wiping, i.e., gradually and simultaneously overwriting (a small vertical stripe of) the color of each with the color of the other. Later we changed with little effort the swapping to be done with smoothly moving lozenges. We first used four steps: each stripe leaving its original position, stopping, and then going to its final position. This was next changed to three steps that, as it turns out, closely resemble a three-statement swap with one temporary. To avoid ghostly images when moving a white stripe, we then added a border to each stripe of the flag. Most importantly, watching the animations led to a simple, new way (at least to us) for avoiding the previously known, unnecessary swaps in the case of, for example, an all-red flag. The new program, which may be viewed as an optimization in the case of seeing and then placing a red pebble, is different from the program sketched by Dijkstra for dealing with the unnecessary red swaps. The

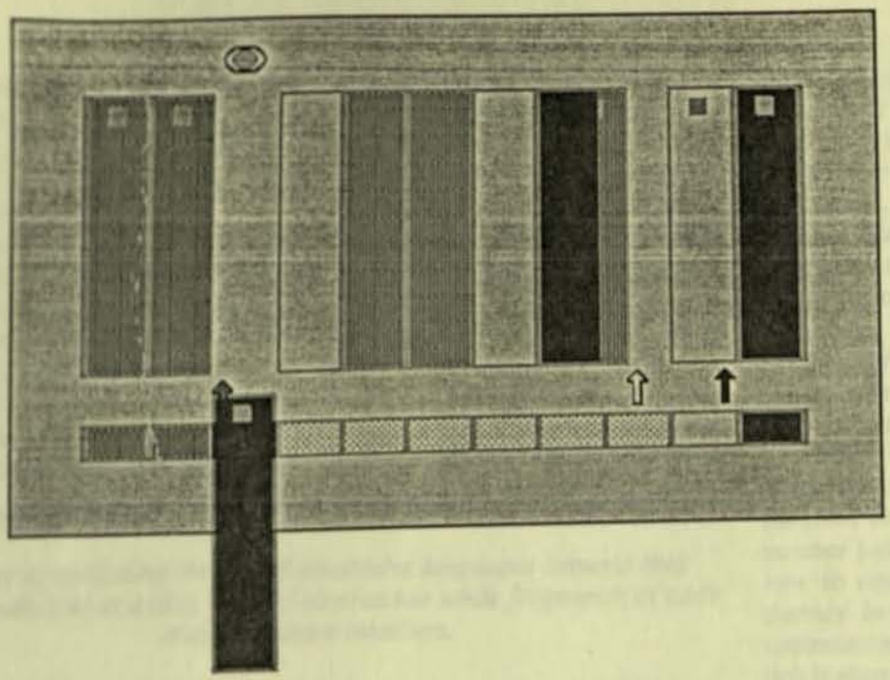


Figure 5. The Dutch national flag example with a view from the new program in midswap. Having seen a red (gray) pebble (which is partially visible in this view) above the white arrow, the eye checks the pebble above the red arrow. It finds a blue (black) pebble. In this case, it is necessary to swap the two pebbles. Had the eye found a red pebble, no swap would be made.

new program results from adding to the original program and, accordingly, is almost immediately animated. The meaning of the "as yet uninspected" zone must now include "or not yet placed in an established zone." In all instances, it is clear from the animations that no pebble is color-determined more than once, a fact that can be made clear by various other considerations.

Finally, because a colleague asked us why we had not animated the eight queens problem,* we soon animated a program doing backtrack search. Systematically, queens moved continuously from one square to the next; if a queen was unsafe, a ray was drawn between the queen and one of its attackers. While watching it, a visiting colleague essentially noted, "So that's how backtracking works for the eight queens problem."

The field of program animation is an application area involving a wide range of issues from the technical to the psychological and the rich interface between the two. Since Smalltalk is an open system designed with much attention to the human interface and with many elegant facilities at our disposal (modifiable, if necessary), it is not surprising that we have been able to use it successfully in our exploratory and prototyping research. What encourages us, however, is the relative ease of prototyping in spite of our not being Smalltalk experts. We are also encouraged by viewers who suggested new uses for animation in their own work, for example, interactive compilers, application accelerators, distributed object managers, and dynamic programming language environments such as Prolog. We have discovered that the number of potential applications, discussions, and issues to which animation may contribute is far broader than we initially imagined.

So far we have relied on the animations themselves and our own verbal remarks to explain the symbolisms employed and the meanings of the icons. Obviously, a view could incorporate explanatory messages or running commentary, using text or even auditory messages or musical accompaniment. An especially intriguing technique to augment explanations would be to incorporate Ward Cunningham's set of iconic "robots," which can be taught to deliver and comment on a demonstration.[16]

In the course of creating these animations, directions for further research have become evident. The planned animation kit must include a library of reusable and connectable animation routines for creating new views. It is impossible to anticipate every need, of course, but the goal is to allow the views to be composed in various ways and to be used as templates for more specific uses if need be, in keeping with much Smalltalk programming practice. We must also specify and abstract the effects of combining movable pictorial elements as well as their potential interactions and interferences. As an example of a kit component, Richard Wagner has pack-

*Place eight queens on a chess board so that all queens are safe from capture.

aged a general method for moving several objects simultaneously along separate paths. The paths may intersect and need not contain the same number of steps.[17]

It is clear that a direct application of the Model-View-Controller scheme as it now exists can be quite awkward, particularly when we attempt to combine a number of views into one composite. (For example, recursive sending of the message "displayView" through the subview hierarchy causes the display to redraw itself repeatedly upon the activation of a window. To avoid redrawing, a special case must be made at some level in the hierarchy to cut off the recursion.) The arrangement of model, view, controller, and associated display routine shown in Figure 2 seems too complex. A cleaner implementation might be achieved by the subsumption of view, controller, and display routine into a single mediating object whose function is to maintain consistency between the object viewed and the graphical image, however that consistency may be specified. At least the functionality of the associated display routine might be better achieved by distinct subclasses of AnimationView. Simplifications by such data abstraction should facilitate composition of numerous views.

We are investigating the use of constraint languages because they express relations at a high level of abstraction while fragments of code maintain those relations.

To address these problems, we are currently investigating the use of constraints to specify and maintain relations between views and their referents. A constraint language such as that supported in ThingLab is attractive for a number of reasons. Constraints express relations at a high level of abstraction while containing fragments of code to maintain those relations. Ordering the interactions of these fragments in the network of objects and constraints, and then actually compiling detailed update and graphics code are the jobs of constraint satisfaction algorithms. A constraint system could support the graphical style of a novice programming interface that one expects from a kit in which new objects are constructed from the building blocks provided. ThingLab's ability to compile incrementally new methods in response to user requests supplies the potential to download a set of such compiled methods as a "packaged" animation that may run independently from the constraint specification and satisfaction mechanism. Common difficulties that arise in constraint systems are not expected in animation since here constraint networks have a low branching factor and generally lack circular dependencies among constrained values. (For example, a constraint that a view be consistent with the thing it represents may always be satisfied if we update the view, which in turn would rarely change the model.)

Thus our first implementation of an animation kit will be built on top of ThingLab. A fundamental required extension to current constraint languages is a mechanism for stating temporal constraints. All existing constraint systems specify consistency of static state, but time requires special treatment and cannot simply be inserted as another variable in a constraint relation. This is largely due to the disjunction between the inherently discrete, time-slice character of our display hardware and the kinds of continuum statements one would like to make in describing smooth motion and rates of change, e.g., $v = dx/dt$. In a discrete approximation, during the evolution from one consistent state to the next, some constraints are explicitly not satisfied, and the constraint satisfier must not insist that they be lest the computation bog down in error relaxation. Furthermore, constraints like to satisfy themselves in any possible direction depending on computational circumstances, but time marches on rather unidirectionally, and it would not do to have a constraint set the clock back to some large negative number just because it did not know how to resolve otherwise the inconsistency. In the system currently being implemented[18] the temporal specification is abstracted from the constrained object and placed into the above-mentioned "mediating object," which maintains versions and histories of the constrained object and owns the temporal constraints that relate one version to the next. This mediator also owns the constraints between view objects and the current version's state. Since ThingLab constraints between objects are always owned by a mutual ancestor, this mediator is not implemented as a filter between view and object but as a parent of both that manages versions and clocks and owns all the relevant constraints among them. □

Acknowledgments

We are grateful to Alan Borning and Ward Cunningham for directing us to the MVC construct and for explaining its inner workings to us. Mayer Schwartz has helped us in several ways: by suggesting the longest common subsequence example, taking photographs of the animations, and, most importantly, by giving continued encouragement and support. We appreciate the helpful discussions with,

and interest shown by the Smalltalk community, especially Roxie Rochat, in our laboratory and at the Oregon Graduate Center. Richard Wagner provided some of the code for displaying linked lists in Figure 3. James Bailey helped us to obtain the printed bit maps in Figures 3 through 5. We are indebted to Thomas Standish and Richard Taylor for important, early discussions.

References

1. M. H. Brown and R. Sedgewick, "A System for Algorithm Animation," *Computer Graphics*, Vol. 18, No. 3, July 1984, pp. 177-186. See also *The Electronic Classroom: A Progress Report*, sound videotape, 17 min., Jan. 1984.
2. R. Baecker, *Sorting Out Sorting*, 16mm sound film, 25 min., Siggraph 1981.
3. B. A. Myers, "Incense: A System for Displaying Data Structures," *Computer Graphics*, Vol. 17, No. 3, July 1983, pp. 115-125. Also *Displaying Data Structures for Interactive Debugging*, Xerox PARC Report CSL-80-7, June 1980, xii + 97 pp.
4. A. H. Borning, *ThingLab—A Constraint-Oriented Simulation Laboratory*, PhD thesis, Stanford University, March 1979. A revised version appeared as Xerox PARC Report SSL-79-3, July 1979, vii + 100 pp.
5. S. Weyer and A. Borning, *A Prototype Electronic Encyclopedia*, University of Washington Computer Science Dept. Technical Report 84-08-01, Aug. 1984, ii + 22 pp.
6. L. Gould and W. Finzer, *Programming by Rehearsal*, Xerox PARC Report SCL-84-1, May 1984, v + 133 pp. A short version appears in *Byte*, Vol. 9, No. 6, June 1984, 187, 188-210 (even pages only).
7. W. R. Swartout, "Explaining and Justifying Expert Consulting Programs," *Proc. Seventh Int'l Joint Conf. Artificial Intelligence*, A. Drinan (ed.), Vancouver, B. C., Aug. 1981, pp. 815-822.
8. M. Stefik et al., "Knowledge Programming in Loops: Report on an Experimental Course," *The AI Magazine*, Vol. 4, No. 3, Fall 1983, pp. 3-13.
9. A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Mass., 1984.
10. N. M. Delisle, D. E. Menicosy, and M. D. Schwartz, "Viewing a Programming Environment as a Single Tool," *Proc. ACM Sigsoft/Sigplan Software Engineering Symp. Practical Software Development Environments*, April 1984; *Software Engineering Notes*, Vol. 9, No. 3; and *Sigplan Notices*, Vol. 19, No. 5, both May 1984, pp. 49-56.
11. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass., 1983.
12. A. Wirfs-Brock, "Design Decisions for Smalltalk-80 Implementors," in G. Krasner (ed.), *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, Reading, Mass. 1983, pp. 41-56. Benchmarks are in Tables 9.1 and 9.2, pp. 165-171.
13. J. W. Hunt and T. G. Szymanski, "A Fast Algorithm for Computing Longest Common Subsequences," *CACM*, Vol. 20, No. 5, May 1977, pp. 350-353.
14. S. B. Akers and B. Krishnamurthy, "Group Graphs as Interconnection Networks," *Proc. 14th Int'l Conf. Fault-Tolerant Computing*, June 1984, pp. 422-427.
15. E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1976, pp. 111-116.
16. H. G. Cunningham, personal demonstration, July 1984.
17. R. M. Wagner, *Program Animation, Tools and Techniques for Object-Oriented Programs* (working title), SM thesis, Massachusetts Institute of Technology, in preparation (at Tektronix Computer Research Laboratory).
18. R. A. Duisberg, *The Design and Implemenation of Animus: A Constraint-Based Animation Kit* (working title), PhD thesis, University of Washington, in preparation.

Ralph L. London is principal scientist in the Computer Research Laboratory at Tektronix, Inc. in Beaverton, Oregon and adjunct professor of computer science and engineering at the Oregon Graduate Center. His research interests include program animation and visualization systems, personal programming and design environments, and specification techniques for designs and programs. He has been a professor and research scientist in computer science with the University of Wisconsin, Stanford University, Information Sciences Institute of the University of Southern California, and University of California at Irvine. He has published extensively in the field of program verification and was leader of the project that developed the Affirm Verification System at ISI.

London received his MS and PhD from Carnegie-Mellon University.

Robert A. Duisberg is a research assistant completing a PhD in computer science at the University of Washington, working with constraint languages and animation and computer music. He has consulted for Atari and is a research intern at Tektronix.

Duisberg holds a MS in physics and DMA in music composition from the University of Washington and a BA from Williams College.

Questions about this article can be directed to either author, Computer Research Laboratory, Tektronix, Inc., PO Box 500, MS 50-662, Beaverton, OR 97077; or to Duisberg, Department of Computer Science, FR-35, University of Washington, Seattle, WA 98195.

AG

Simon
Functional Description and Architectural Design
Revision 1

R.J. Steiger

ParcPlace Systems

DRAFT. CONFIDENTIAL, NOT FOR DISTRIBUTION

1. Introduction

1.1. Purpose

This document defines the functionality and architectural design of a SIMPLE demo prototype called *Simon.*

1.2. Related Documents

[1] *SIMPLE Product Requirement Specification* R. Steiger, March 20, 1987

[2] *LOOM - Large Object-Oriented Memory for Smalltalk-80 Systems* Ted Kaehler, Glenn Krasner, in *Smalltalk-80: Bits of History, Words of Advice,* Glenn Krasner, ed.

1.3. Revision History

[1] first draft.

2. Overview

2.1. Simon's Purpose

Simon is intended to be an experimental testbed to better understand the functional, technical, and performance issues involved in integrating database and Smalltalk technology.

There are two broad motivations for such an integration (see reference [1]). The first is to provide a better information management application development and delivery environment to organizations using existing commercial databases, leveraging off of Smalltalk's general benefits -- portability, productivity, powerful and consistent user interfaces, rich functionality, and application integration. The second is to augment Smalltalk's transient, private objects with persistent, sharable objects. In both cases, there is a strong requirement that the resulting capability be simple to apply and easy to use.

2.2. Phasing

It is expected that Simon will evolve through several phases. The overall strategy is to build each phase on top of the previous one, minimizing unnecessary future rework where possible through principled design.

This document describes the initial version called *Simon-1* in considerable detail, and anticipates its successor *Simon-2* in some of the design choices.

Simon-1's primary focus is on achieving:

(A) *transparency*- relatively seamless integration of Smalltalk and a relational database, resulting in radical application simplification relative to non-transparent systems, through the following means:

(1) *object-oriented representation* - data is represented in the form of Smalltalk objects that are accessed uniformly through message-passing, as opposed to tuples or some other non-object model;

(2) *location independence and automatic data migration* - objects may be referenced independently of their location, and may be accessed without having to explicitly locate and transport them; the control of data migration between Smalltalk and the database is thereby rendered as invisible to the casual user as feasible;

(3) *automatic object storage* - under fairly general circumstances, objects are automatically stored in the database when doing so is required to maintain consistency;

(4) *semi-automatic selection of representation* - class definitions for Smalltalk objects and table definitions of associated database records may be automatically generated from each other, with intelligent defaulting;

(5) *flexible representation* - more knowledgeable users may override default representation choices in order to tune the system's performance;

(B) *information integrity* - basic data integrity guarantees via value constraints and concurrency controls;

(C) *end-user familiarity* - user interfaces oriented toward a look-and-feel familiar to current commercial databases users; and

(D) *portability* - the power of underlying databases is brought out in a generic (database-independent) form, and which applies equally well to accessing objects residing within the Smalltalk image or the database.

Simon-2's primary primary focus is on extending Simon-1 by achieving:

(E) *sharing*- allowing multiple users to interactively and safely share the same set of objects, including providing update coordination and notification;

(F) *conceptual modeling* - providing a higher-level design and query capability based on the Entity-Relationship model;

(G) *relational completeness* - providing views, joins, and derived attributes, thereby matching or exceeding the power of SQL as a query and application language;

(H) *user interface kits* - providing a more powerful and flexible set of building blocks for developing domain-specific applications and tools;

(I) *graphical design aids* - providing direct manipulation of schemas;

(J) *persistent storage reclamation* - extending garbage collection to the database; and

(K) *schema evolution* - integrating classes and change management into the database, securely linking persistent objects to their definitions.

As critical as it is, developing sharing is defered to Simon-2 because (a) the underlying DBMS doesn't support it, and (b) sharing relies heavily on the more fundamental transaction and persistence management mechanisms in Simon-1.

We use the generic term "Simon" when describing an aspect of the whole framework, and a more specific version name "Simon-i" when discussing an aspect specific to that version.

2.3. Remaining Document Structure

Section 3 describes Simon's high-level configuration and architecture.

Section 4 describes the Persistent Object Manager's functionality as seen by applications.

Section 5 describes Simon's application functionality as seen by end-users.

Section 6 describes the design of the Persistent Object Manager.

Section 7 describes the design of the Server Interface Module.

3. Simon Architecture

3.1. Major Subsystems

Figure 1 is a block diagram of Simon's overall configuration. Simon comprises four major subsystems:

- *applications* - tools and utilities for manipulating persistent and transient objects;
- *Persistent Object Manager* - adds database-independent persistent object capabilities to the basic Smalltalk system;
- *Server Interface Modules (SIMs)* - translate between the generic persistent object and operation world, and various servers' (typically very different) data and operation worlds; each type of server has its own SIM; and
- *servers* - underlying storage subsystems (DBMSs, file systems, etc.).

Figure 2 is a more detailed block diagram illustrating the *managers* within each of the above subsystems, and their constituent classes.

3.2. System Configuration Phasing

Simon is written primarily as an ordinary Smalltalk program, plus a few user primitives linked into the VM.

Simon-1 runs in Smalltalk-80 Version 2.2, Sun Release 1.1. It accommodates only a single server type, Sun Oracle, Version 5.0.20.4. Since this version of Oracle supports local-only access, it can't provide sharing, and because it must run on the same workstation as the Smalltalk image, considerable swapping overhead between the two systems is to be expected. The interface to the Persistent Object Manager will be formalized in this phase.

Simon-2 is expected to also have only a single server, Sun Oracle Version 5.1. Since this version provides remote access from multiple workstations, it can support sharing, and can be run on a central server, thereby reducing user workstation swapping overhead.

Simon-3 is expected to add at least one other kind of server and associated SIM. It is at this stage that formalizing a generic SIM interface is to be undertaken.

In general, SIMs consist of two parts, one running in the Smalltalk Virtual Image, the other a set of user primitives that in turn call functions in the server's function library.

AG

Figure 1: Basic Simon System Configuration 6/16/87

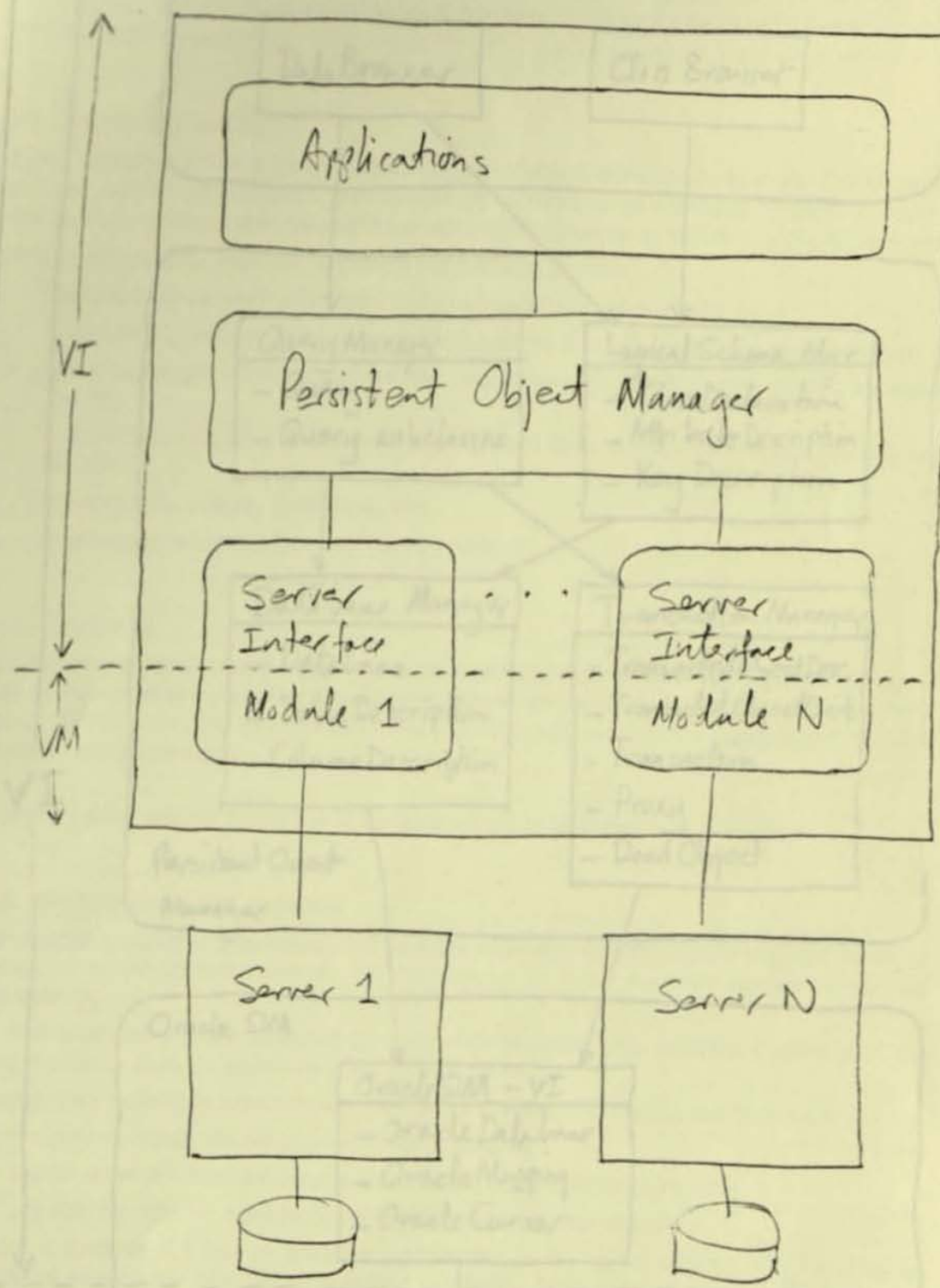
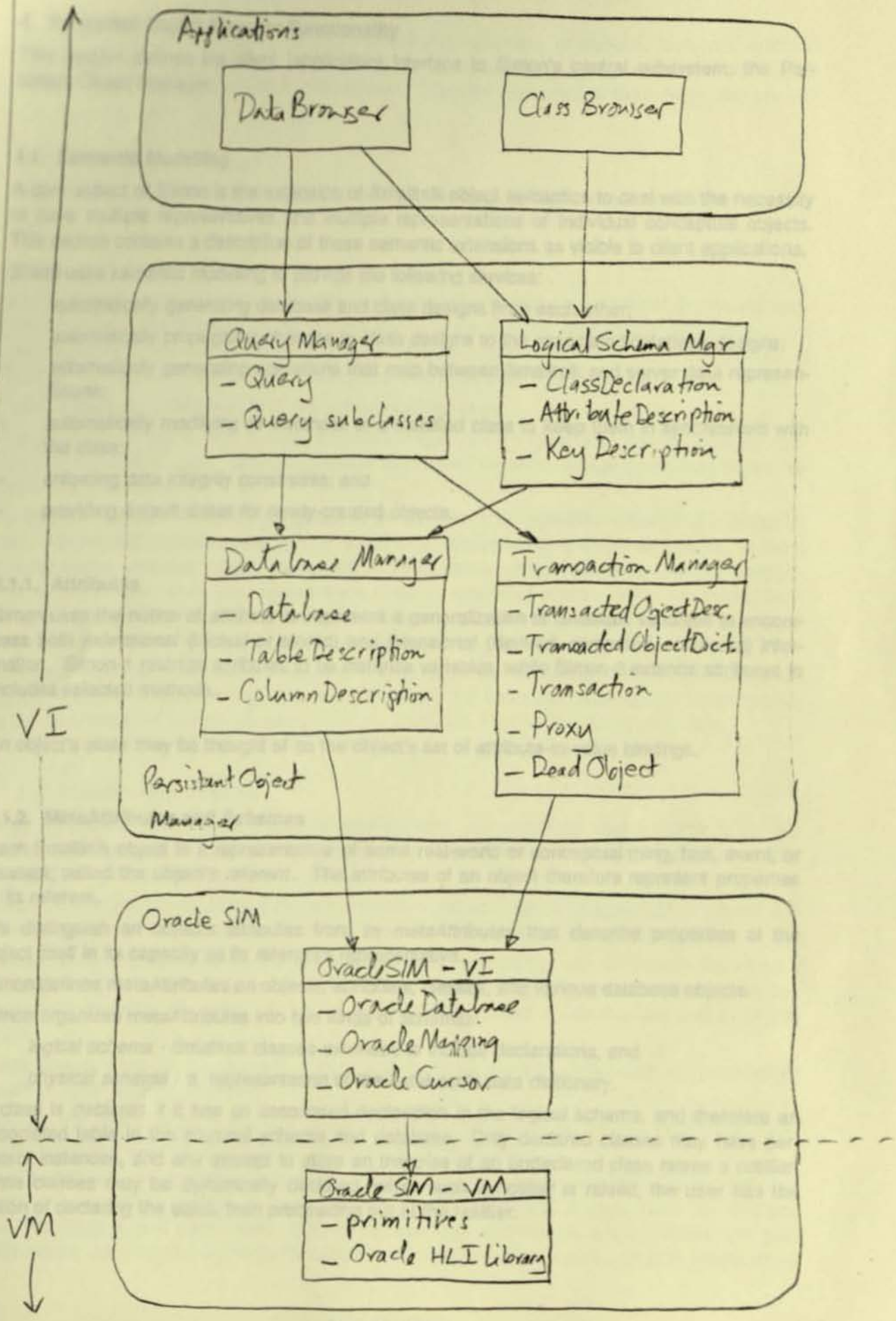


Figure 2: Simon - 1 Architecture 6/16/87



4. Persistent Object Manager Functionality

This section defines the client (application) interface to Simon's central subsystem, the Persistent Object Manager.

4.1. Semantic Modelling

A core aspect of Simon is the extension of Smalltalk object semantics to deal with the necessity to have multiple representitives and multiple representations of individual conceptual objects. This section contains a description of these semantic extensions as visible to client applications.

Simon uses semantic modelling to provide the following services:

- automatically generating database and class designs from each other;
- automatically propagating changes to class designs to the associated database designs;
- automatically generating operations that map between Smalltalk and server data representations;
- automatically modifying all instances of a modified class to keep them in synchronism with the class;
- enforcing data integrity constraints; and
- providing default states for newly-created objects.

4.1.1. Attributes

Simon uses the notion of *attribute* to represent a generalization of Smalltalk variables to encompass both *extensional* (factual or stored) and *intensional* (derived, computed, or infered) information. Simon-1 restricts attributes to be instance variables, while Simon-2 extends attributes to includes selected methods.

An object's *state* may be thought of as the object's set of attribute-to-value bindings.

4.1.2. MetaAttributes and Schemas

Each Smalltalk object is a *representative* of some real-world or conceptual thing, fact, event, or concept, called the object's *referent*. The attributes of an object therefore represent properties of its referent.

We distinguish an object's attributes from its *metaAttributes* that describe properties of the object itself in its capacity as its referent's representative.

Simon defines metaAttributes on objects, attributes, classes, and various database objects.

Simon organizes metaAttributes into two kinds of schemas:

- *logical schema* - Smalltalk classes extended to include declarations, and
- *physical schema* - a representation of the database's data dictionary.

A class is *declared* if it has an associated declaration in the logical schema, and therefore an associated table in the physical schema and database. Only declared classes may have persistent instances, and any attempt to store an instance of an undeclared class raises a notifier. Since classes may be dynamically declared, when such a notifier is raised, the user has the option of declaring the class, then proceeding out of the notifier.

Protocols exist for creating, modifying, and querying schemas. In addition, protocols exist for creating one kind of schema from the other, and for propagating changes made in the logical schema to the physical schema and the server. These are discussed in more detail later on.

4.1.3. Object MetaAttributes

Simon recognizes two fundamental object-level metaAttributes - *persistence* and *identityType*.

4.1.3.1. Persistence

Every object has a persistence metaAttribute whose value is in the enumeration (**transient persistent pendingPersistent**), controlling the scope of the object's accessibility and lifetime. (In general, we shall use metaAttribute values as adjectives, so shall speak of "transient objects" in place of the more verbose "objects whose persistence metaAttribute is transient".)

The granularity of persistence is at the individual object level.

Transient objects are local to a single Smalltalk image; current Smalltalk systems generally contain only transient objects. Transient objects are named within an image by an oop, which we call a *local reference*.

By contrast, persistent objects are global to all images. Each persistent object has a *principal* copy living in the database, and zero or more *local* copies living in Smalltalk images, at most one per image. Principal copies are named by some kind of *global references* that are independent of any image, while local copies are named by local references. Within an image, local and global references for the same object are therefore in one-to-one correspondence. In this model, persistence and sharability are equivalent.

An object is pendingPersistent if it has only a local copy, and is reachable from some persistent object. When a Smalltalk image is synchronized with the database (described in detail below), its pendingPersistent objects are stored into the database, thereby becoming persistent.

4.1.3.2. GlobalReferenceForm

Every persistent object has a globalReferenceForm metaAttribute whose value is in the enumeration (**unique keyed value**), controlling the representation of its global reference. The granularity of globalReferenceForm is at the class level.

Unique objects are identified via a system-generated *uniqueId* that is independent of the object's state. Unique objects may therefore have any possible state without losing their identity.

Keyed objects are identified via the value of some *key* comprising one or more of the object's attributes. Unlike unique objects, there can be at most one keyed object having a given key value, and changing the value of the key can effectively alter or destroy the identity of the object.

Value objects have no identity, only value, and therefore can't be referenced, only stored as values of other objects' attributes.

Because the relational data model is based on keyed references, all objects defined by traditional relational database schemas are restricted to being keyed.

Objects defined by Smalltalk application programmers may have any globalReferenceForm. Objects representing entities (conceptual things) are generally unique, such as classes, methods, images, and documents. Objects representing relationships among entities are generally keyed, such as part-whole relationships and interdocument linkages. Objects representing

simple values as opposed to conceptual tokens (such as numbers and times) are given value globalReferenceForm.

4.1.4. Attribute MetaAttributes and Declarations

Every attribute has the following metaAttributes:

- *domain* - declaring the set of allowable values for the attribute;
- *referenceForm* - controlling how the attribute's values are represented; and
- *defaultValue* - the attribute's initial value in new instances.

In Simon, the class definition interface is extended to include such metaAttributes, by reinterpreting a class's **InstanceVariableName** string as a specification written in a declaration language. In this language, attribute declarations are separated by commas; each declaration consists of the attribute name, followed by the domain specification enclosed in angle brackets, and zero or more keyword-value pairs for the remaining metaAttributes.

4.1.4.1. Domains

Domains constitute a simple type system, comprising the following elements:

- an instance of **Class**, written "class": the domain consists of all instances of the class or any of its subclasses;
- an instance of the **Union** Collection subclass, written "class1|class2|...": the domain consists of the union of all constituent classes' instances;
- an instance of the **Enumeration** collection subclass, containing an arbitrary ordered set of objects, written "(object1 object2 ...)": the domain is this set;
- an instance of **Interval**, written "min to: max": the domain is the elements of the interval; and
- an instance of the **CollectionType** class, written "collectionClass on: indexDomain of: elementDomain", or "collectionClass of: elementDomain": the domain is the set of all instances of collectionClass having elements in the given elementDomain, and indexed over the indexDomain if given, else SmallInteger.

For example,

<IdentityDictionary on: Class of: (ClassDeclaration|UndefinedObject)>

defines the domain of identityDictionaries that map classes to classDeclarations or nil. (Notice the use of parentheses for subdomain grouping; the angle brackets are used to delimit a group of statements constituting a domain declaration.) We shall use these declarations in the remainder of this document.

4.1.4.2. Attribute ReferenceForm

Recall that objects have a globalReferenceForm controlling how their global names are represented, in essence how the "head of the pointer" to the object is encoded. Similarly, objects containing the "tail of the pointer" also have a say in how references are encoded. The actual representation used is therefore negotiated between the referring and referred objects.

On the referring side, the attribute holding a reference has a metaAttribute referenceForm whose domain is <(**reference value**)>. Reference attributes contain a global reference to some autonomous persistent object, either a uniqueId or a key (somewhat like call-by-reference or call-by-name procedure parameters, respectively). Value attributes contain the object itself

(somewhat like call-by-value procedure parameters), packed into the refering object. Such packed "objects" have no separate identity, and therefore can't be shared. In return for being unsharable, they are much more efficient in space and time than objects having identity, so are an important form of representation.

For any domain, all of its values must have the same globalReferenceForm, so as to allow static binding of the attribute's representation. We may therefore extend the globalReferenceForm metaAttribute from a domain's value set to the domain itself, and can therefore speak of "value domains" and so forth.

Attributes with value domains are forced to be value attributes. For example, any attribute with domain SmallInteger automatically embeds its values.

For attributes with non-value domains whose values won't be shared by other persistent objects, the designer is free to select value referenceForm, thereby asking Simon to pack the object referenced by that attribute's value into the same database record as the refering object. Such packing provides a several-fold improvement in storage and retrieval speed, and a significant reduction in storage space. For example, in a CAD application, circuit blocks may have displayBox rectangles that have points that have coordinates. The designer will most likely choose to embed all these objects into a single **Circuit** object, since sharing is unnecessary for all but the top-level circuit.

4.2. Database Interface

Database servers are represented within Simon by instances of subclasses of the abstract **Database** class. Simon-1 has only one concrete subclass of Database - **OracleDatabase**, and permits a maximum of one instance to exist. (We shall use the term "the database" to refer to both the server and its representitive object.)

OracleDatabase class protocol is as follows:

- **new**
  returns the unique instance database.

Database instance protocol is as follows:

- **open: userId**
  opens database for access, where userId contains the user's name and password;
- **isOpen**
  returns true if database is open, else false;
- **store: anObject**
  immediately makes anObject a persistent object in the database; and
- **close**
  closes the database.

4.3. Object Lifetimes

Simon-2 is expected to have the same object lifetime semantics for transient and persistent objects: objects continue to live if and only if they are reachable from some *root* object.

The implementation of this semantics for persistent objects is expected to be sufficiently difficult that Simon-1 takes a shorter-term approach, allowing explicit persistent object deletion. However, this opens the door to dangling references, ones that point to non-extant objects or which *alias* themselves to other extant objects. When Simon encounters a reference to a non-extant object, it maps it to nil. Since unique objects' references are truly unique and never reused,

they will never alias. Keyed references have an inherent danger of aliasing.

4.4. Queries

Simon-1's query capability is very rudimentary. It allows for retrieval of collections of objects all of the same class, based on various Boolean filters, and sorted in various ways. The most basic aggregate functions (yielding a single value for the entire collection such as sum and average) are provided. Query syntax and semantics are native to Smalltalk, not to SQL or some other foreign formalism. Simon-1 provides no joins; in Simon-2, the plan is to provide joins through *virtual* objects (called *views* in normal database parlance) whose attributes are indirect references to one or more other objects' attributes.

Queries are represented by the OrderedCollection subclass **Query**. In Simon-1, query element domains are restricted to a single class, thereby greatly simplifying their implementation. (An attempt will be made in Simon-2 to generalize query domains to encompass subclass hierarchies and possibly unions).

The basic idea of queries is that they represent their elements intensionally ("virtually"), as an *input* collection, plus some *transformation* on the input to produce an *output* collection. Transformations include

- filters - predicates that must be true of their output elements;
- sorters - which reorder the collection based on the value of some attribute; and
- aggregate functions - generally numerical functions of the entire collection.

Since query representation is intensional and highly-structured, it is straightforward to compile equivalent statements in servers' query languages, such as SQL, so as to access persistent elements. Queries also operate locally to obtain transient elements.

Queries may be cascaded, each new query representing a refinement of its input query. Since any number of queries may share the same input without mutual interference, trees of queries may be created and explored, forming the basis for query-by-refinement style user interfaces (ala ISL's Rabbit and Intellicorp's InfoScope), to be developed in Simon-2.

Class has the following additional instance protocol

- **select**
  returns a new query whose domain is the class, and whose elements are the class's instances.

Query instance protocol is as follows:

- **with: anAttributeName** relation **aValue**
  elements are restricted to those whose named attribute stands in relation to aValue, where relation is one of **equalTo:, lessThan:, matching:, notMatching:**, etc.
- **orderAscendingBy: anAttributeName**
  orderDescendingBy: anAttributeName
  elements are sorted according to values of anAttribute;
- **+ aQuery**
  elements are the union of those in both source collections (where both collections must have equivalent domains);
- **- aQuery**
  elements are those in the first collection and not in the second collection (where both collections must have equivalent domains);

- **count**
  the number of elements in the collection;
- **maxOf: anAttributeName**
  minOf: anAttributeName
  averageOf: anAttributeName
  sumOf: anAttributeName
  maximum, minimum, average of, and sum of named attribute of collection; and
- **executeIn: aTransaction**
  executes the query, making its elements available via the query's normal Collection protocol, doing so within aTransaction's environment.

As an example query, the following will return a query of all employees whose names contain the string 'Smith' with salary in the range $35,000 to $50,000, sorted first on name (ascending) then on salary (descending):

```
(((((Employee select)
    with: #name matching: '*Smith')
        with: #salary greaterThanOrEqualTo: 35000)
            with: #salary lessThan: 50000)
                orderAscendingBy: #name)
                    orderDescendingBy: #salary
```

Given an object retrieved by a query, applications are then free to navigate from the object along "links" formed by attributes. Any persistent objects encountered along such a link is automatically fetched from the database, if not already present in the image.

4.5. Transactions

Transactions provide environments in which groups of querying, storage, updating, and deletion actions may be performed on a set of (transient or persistent) objects as a single atomic action.

Transactions are orthogonal to processes: any number of processes may execute inside each transaction, and each process may execute inside any number of transactions. Each transaction is bound to some database.

Transaction class protocol is as follows:

- **openOn: aDatabase**
  returns a new transaction bound to aDatabase; and
- **scavenge**
  scavenges the Transaction Manager's internal tables, removing objects that are no longer accessed from anywhere else in the image.

Transaction instance protocol is as follows:

- **remove: anObject**
  effectively removes anObject from the transaction and any current query, releasing any lock;
- **add: anObject**
  add anObject into the transaction;
- **commit**
  commits all updates, additions, and removals to the database, and releases any locks held; returns only after all changes are secured in the database; and

- **abort**
  undoes all updates, additions, and removals, restoring the state to the beginning of the transaction, and releases any locks held.

Object instance protocol is augmented to control object locking as follows:

- **exclusivelyLockIn: aTransaction waiting: aTime**
  locks the object for exclusive access in aTransaction; returns true if and only if lock was successfully obtained; if the object is locked by another transaction, waits up to aTime before failing, returning false;
- **unlock**
  releases the lock on the object, if any; signals any processes waiting to lock the object to reattempt locking;
- **lockingTransaction**
  returns the transaction currently holding the lock on the object.

Objects may be freely updated in the usual way by assignment to their instance variables. On commitment, only those objects whose states have changed are updated in the database. Locking doesn't protect objects from unwarranted access; it is provided purely as a way for well-behaved applications to correctly coordinate their access to shared objects. (Enforcing such protection would require deep VM changes.)

Whenever an object is retrieved or stored through a transaction, we say that the object becomes *enclosed* by the transaction, and say that an object is *transacted* if and only if it is enclosed in at least one transaction.

4.6. Image-Database Synchronization

Whenever an image is suspended, all outstanding transactions are committed, thereby synchronizing the image with the database.

Whenever an image is restarted after being stopped, the states of all persistent objects are resynchronized, in the sense that their local copies are refreshed from their principal copies (this isn't done for snapshotting).

5. Application Functionality

This section defines the external behavior of Simon's application user interfaces.

5.1. The DataBrowser

Simon-1 has only a single application - a generic *dataBrowser*. DataBrowsers give the user the ability to select a collection via a query, then view the collection at a number of levels of detail: the collection as a whole, individual objects, and attribute values. Each dataBrowser has a transaction in which it operates. Menu commands are provided for creating, deleting, and modifying objects, and aborting or committing the transaction. DataBrowsers provide direct manipulation and hot (continuously up-to-date) viewing.

The Data Browser has six panes: class category, class, query, collection, object, and attribute.

The classCategory and class panes are copied from the Smalltalk brower, and allow selection of a class environment in which to perform queries and create new instances. Only declared classes are presented for selection.

The query pane is a standard codeView in which the user enters queries. When the user invokes the *accept* middle menu item, the query pane does the equivalent of prefixing For example, the above example employee query would be executed by selecting the Employee class, and accepting the following text:

```
((((with: #name matching: '*Smith')
    with: #salary greaterThanOrEqualTo: 35000)
        with: #salary lessThan: 50000)
            orderAscendingBy: #name)
                orderDescendingBy: #salary
```

The resulting query is displayed in the collection pane as a table whose rows are the objects in the collection, and whose columns contain the values of the various objects' attributes. The user may select an individual object in the table with the left mouse button. Columns occupy a fixed vertical region within the pane, and each column is labelled with the associated attribute's name at the top. Numeric values are right-justified, all other left-justified, and values too long to fit in the column are truncated. Column widths are defaulted from the associated attributes' domains. This pane provides two-dimensional scrolling, allowing scanning vertically through objects, and horizontally across columns when objects are too wide to fit within the pane. Scrolling in either dimension is controlled by scroll bars, scroll buttons, and by moving the cursor outside the pane during selection. The middle button menu for the collection pane has the following items:

- *delete*: deletes the selected object;
- *accept*: commits the transaction; and
- *cancel*: aborts the transaction.

Having selected an object in the collection pane, the resulting object is shown in more detail in the object pane. Each row shows an attribute, with a name column on the left, and a value column on the right. One may think of the object pane as a "super inspector" in which all attributes are simultaneously shown in the view, with values truncated as necessary to fit on one line each. When an object is selected, the dataBrowser attempts to lock it; if successful, the user is none the wiser. If the locking times out (after some fixed duration), selection is canceled, and a warning notifier is spawned. When an object is deselected, it is unlocked.

An individual attribute row may be selected with the left mouse button, in which case its value is shown in the attribute pane, a standard ParagraphView. The value may be freely edited as usual, and becomes installed in the object when the user invokes the accept middle button menu item. Updates are automatically and immediately reflected in the object and collection panes. On entry, new values are checked against the attribute's domain, and if illegal, the update is aborted with a notifier.

The middle button menu for the object pane provides *insert* and *copy* items. Both of these create a new instance of the selected class and put it in the object pane for the user to fill in. The current object in the collection pane is deselected to indicate that the object isn't part of the collection. Insert initializes the new object's attributes from their defaultValue metaAttributes, while copy initializes them from the previously selected object. When containing a new object, the object pane's middle button menu contains the following additional items:

- *cancel*: delete the object;
- *accept*: validate the object's attributes, and if legal, keep the object, else raise an error;
- *store*: like accept, but also stores the object in the database.

When a dataBrowser is closed, its transaction is aborted.

5.2. Smalltalk Browser Changes

The standard Smalltalk browser is modified to accommodate metaAttribute access as follows:

- new variables *globalReferenceForm* and *tableName* are added to class definitions; and
- the interpretation of the instanceVariableNames string becomes a declaration language, as described above.

5.3. Standard Menu Item Changes

The standard middle button menu acquires a new *store* item, which directly stores the selected object into the database, if it exists and is open, else spawns a notifier.

6. Persistent Object Manager Design

This section describes the internal design of the Persistent Object Manager.

6.1. Object Mapping

Any practical persistent Smalltalk system must map among various data representations. Even if the server provided a Smalltalk-like data model (e.g. LOOM (reference [2])), it would have to map between local and global oops. When employing a non-object-oriented server such as a relation DBMS, additional mapping is required between Smalltalk objects and records. Indeed, this mapping is a major source of complexity and computational overhead.

A core part of the Simon framework is therefore the object mapping mechanism. In order to achieve efficiency, parts of this mechanism are implemented primitively.

We next consider representation and mapping for various kinds of objects.

6.1.1. Simple Datatype Representation and Mapping

Simple datatypes are SmallInteger, Float, Time, Date, Character, and enumerations. Enumerated values are stored as their cardinal numbers. The remaining cases are straightforward and won't be discussed further.

6.1.2. Byte Indexed Object Representation and Mapping

The elements of byte-indexed objects (such as Strings and ByteArrays) are packed into text fields. Such objects may have any of the globalReferenceForms.

Oracle allows either variable or fixed length text fields up to 240 bytes in length, or variable length long fields up to 64 Kbytes in length. Only one long field is allowed per table, however.

On retrieval, symbols are interned in the Smalltalk dictionary.

6.1.3. Keyed Object Representation and Mapping

Every keyed object is generally stored as a record ("tuple") in a table associated with the object's class. Each of the class's attributes has an associated field in the table. Fields for reference attributes contain either a uniqueId or a key, while fields for value attributes contain the entire state of the referenced object. (Notice that fields may be composite, comprising a collection of subfields; for example, multiattribute keys map to composite fields.)

Keyed object tables are indexed on their key fields.

6.1.4. Unique Object Representation and Mapping

Unique objects are represented like keyed ones, with the addition of an internal field containing the value of a system-generated uniqueId object. UniqueIds have the following properties:

- they are unique across databases and users;
- finding a record given its uniqueId is reasonably fast;
- uniqueId generation is fast;
- the object's table and class are recoverable from the uniqueId;

- the object's creation time and date are recoverable from the uniqueId; and
- the uniqueId for any immediate object contains the object.

Uniqueids are stored in the database as a 64 bit integers. In Simon-1, the representation of uniqueids for non-immediate objects is the concatenation of

- a class code,
- a Unix timestamp, and
- a serial number within the timestamp interval (to guarantee uniqueness when more than one uniqueId is created in the same second).

In Simon-2, a unique workstation id is added, as well.

The representation of uniqueids for immediate objects is the concatenation of a class code and the object's value.

Tables containing unique objects are indexed on their uniqueId fields.

6.1.5. Word Indexed Object Representation and Mapping

Word-indexed objects are treated similarly to byte-indexed objects. Each element's oop is converted into its uniqueId, 8 bytes per element, and stored in a raw text field. (For efficiency, each of the major collection classes may require its own mapping strategy, to be determined during more detailed design.)

6.2. Object Killing

Under certain conditions described in subsequent sections, it is necessary to *kill* a local object, thereby destroying its identity. This is done by asking the object to **become:** an instance of **DeadObject**, a root class (i.e. one having no superclass) that responds to essentially all messages by spawning a notifier. The notifier informs the user that the object no longer exists, and doesn't allow the user to proceed.

6.3. Schema Manager Design

6.3.1. Logical Schema Structure

Class is given a new class variable **ClassDeclarationDictionary** containing an identityDictionary mapping classes to **classDeclaration** objects. (A more efficient implementation would add an instance variable to each class pointing to its declaration, but this appears to be difficult, and not worth optimizing now.)

ClassDeclaration has the following instance attributes:

- **declaredClass** <**Class**>
  the class that this declaration describes;
- **classCode** <**SmallInteger**>
  the code of the class used to form its instances' uniqueIds;
- **globalReferenceForm** <{**unique keyed value**}>
  the class's instances' globalReferenceForm;

- **attributes <Collection of: AttributeDescription>**
  describing attribute metaAttributes;
- **keys <Collection of: KeyDescription>**
  describing the instances' keys, if any;
- **tableName <String>**
  the name of the associated database tableDescription (this is moved elsewhere in future systems to allow classes to span multiple databases); and
- **table <TableDescription>**
  the associated tableDescription (same comment).

AttributeDescription has the following instance attributes:

- **name <Symbol>**
  the attribute's name;
- **index <SmallInteger>**
  the attribute's position in the object;
- **domain <Class|Union|Enumeration|Interval|CollectionType>**
  the domain of allowable values;
- **defaultValue <Object>**
  the attribute's initial value as filled in by the dataBrowser on instance creation; and
- **referenceForm <{reference value}>**
  the attribute's referenceForm.

KeyDesclaration has the following instance attributes:

- **name <Symbol>**
- **attributes <Collection of: AttributeDescription>**

<DO WE WANT KEYS TO BE SPECIAL KINDS OF ATTRIBUTES?>

6.4. Physical Schema Structure

In Simon, the abstract class Database has a single concrete subclass - OracleDatabase. (Additional server types would have their own classes.)

Database has the following instance attributes:

- **physicalSchema <OrderedCollection of: TableDescription>**
  describing the database's tables.

TableDescription has the following instance attributes:

- **columns <Collection of: ColumnDescriptor>**
  describing the table's columns;
- **representedClass <Class>**
  the class whose instances are stored in this class;
- **mapping <ObjectMappingDescriptor>**
  specifies object-record mapping, described below.

ColumnDescription has the following instance attributes:

- **name <String>**
  the column's name;
- **type <String>**
  the columns datatype;

- **size <SmallInteger>**
  the number of bytes in the column.

6.4.1. Schema Storage And Retrieval

In Simon-1, logical schemas are simply enriched classes, so are filed out and in in the normal manner (modulo changes to the class storage and parsing machinery as described above). In future versions, classes should be stored in a database.

In Simon-1, physical schemas are extracted from the database by querying its data dictionary, starting from the tableName of each declared class. This is possible since Oracle's data dictionary contains sufficient information to regenerate the schema. (Physical schemas should also be databased in future versions.)

6.4.2. Logical To Physical Schema Translation

When sent the message *generatePhysicalSchema*, a classDeclaration will return a new tableDescription. When sent the message *install*, a tableDescription will in turn issue a set of commands to the database to create the corresponding table. <details TBS>

6.4.3. ObjectMappingDescriptor Implementation

<To be supplied>

6.5. Transaction Manager Design

The Transaction Manager is the main clearinghouse for metainformation required to manage persistence and transactions within Simon.

6.5.1. The TransactedObjectTable

For each object in the system subject to transactions (including all persistent objects), a separate object called a **transactedObjectDescriptor** (or "TOD") is maintained, containing the object's metainformation.

Transaction has the following class variable:

- **TransactedObjectTable <TransactedObjectDictionary>**
  (the "TOT"), a dictionary containing all TODs, indexed under two independent lookup keys - the TODs' local and global object references.

Whenever any object becomes transacted, it is interned (much the way symbols are interned), yielding a unique TOD. Transient become transacted either indirectly when they are retrieved by a query, or directly when stored through a transaction. Persistent objects become transacted when they are retrieved from the database, again by a query. In the persistent object case, a unique local copy is made,; the TOD binds the persistent object's local and global references to each other within each image.

TransactedObjectDescriptor has the following instance attributes:

- **localReference <Object>**
  the oop of the object's local copy;

- **globalReference** <UniqueId|Array|UndefinedObject>
  the object's global reference, if persistent, else nil;
- **persistence** <(**transient pendingPersistent persistent pendingDeleted**)>
  the object's persistence metaAttribute;
- **shadow** <**Object**>
  a shallow copy of the local copy, used for rollback and update detection;
- **enclosingTransactions** <**UndefinedObject|Transaction|OrderedCollection of: Transaction**>
  the set of enclosing transactions (space-optimized for the extremely frequent zero or one transaction cases);
- **owner** <**UndefinedObject|Transaction**>
  the transaction with exclusive access to this object, if any; and
- **semaphore** <**Semaphore**>
  the semaphore enforcing mutual exclusion on the owner attribute.

Transaction has the following instance attributes:

- **database** <**Database**>
  the associated database;
- **transactedObjects** <**OrderedCollection of: TransactedObjectDescriptor**>
  the objects known to the transaction.

Figure 3 illustrates the Transaction Manager's object management state diagram. Note that states may be represented in two dimensions, with persistence along the vertical axis, and "transactedness" along the horizontal dimension. The diagram emphasizes the state machine's symmetries.

Objects are untransacted if and only if they have no TOD in the TOT. If transacted, they are clean if their local copy and shadow have the same states, else are dirty. Hence, updates cause clean-to-dirty transitions.

Transient objects become transacted either when fetched or stored through a transaction. They become pendingPersistent when stored, else remain transient. Persistent objects become transacted when fetched.

Commits bring their shadows into synch with their local copys. In the case of persistent objects, commits ask the database to update their principal copies from their local copies. In the case of pendingPersistent objects, commits create a new principle copy and insert it into the database. In the case of pendingDeleted objects, the database is asked to delete the principal copy, and the local copy is killed to avoid inconsistencies.

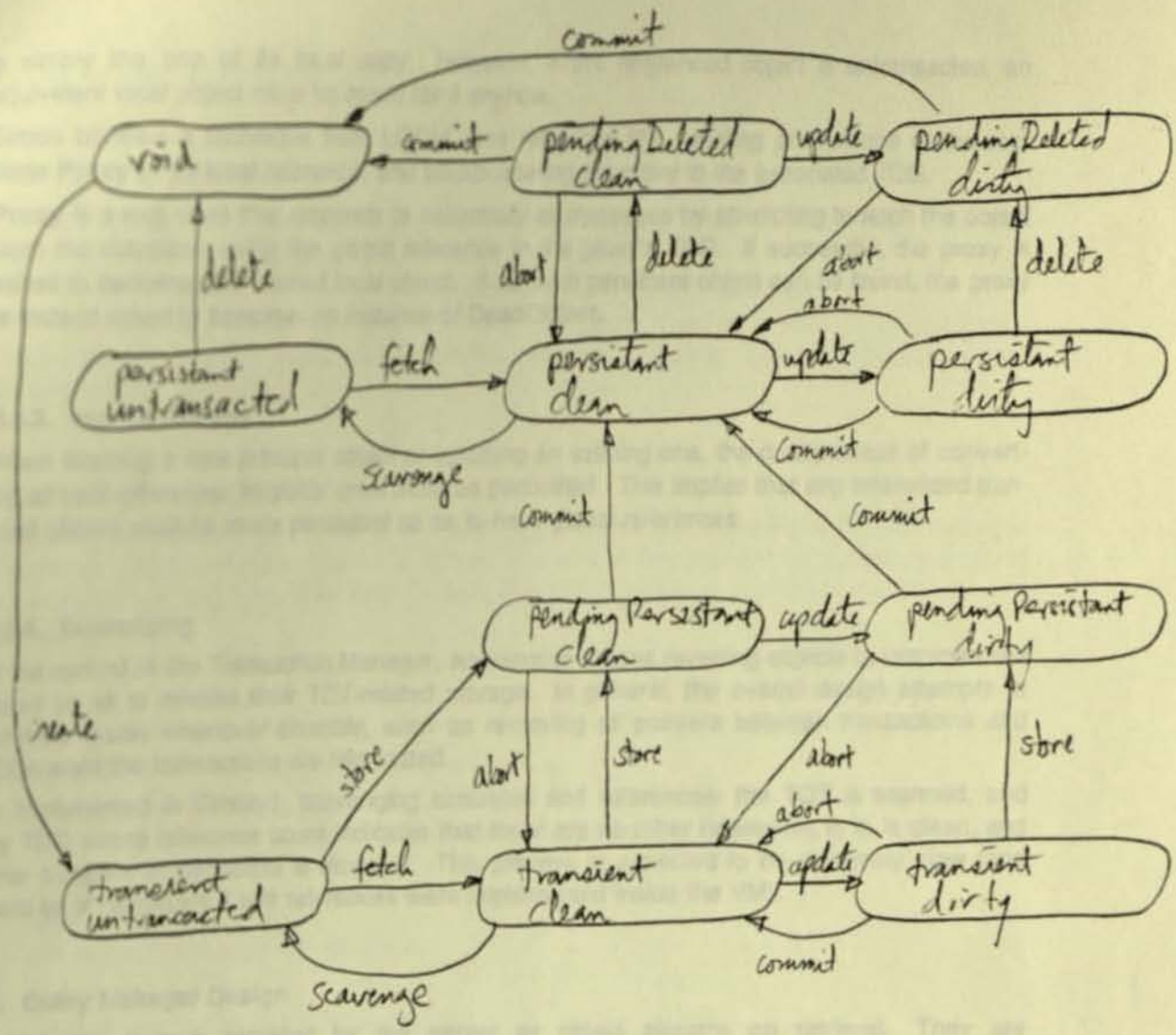
Aborts undo updates by restoring objects' states to their shadows' states, thereby causing dirty-to-clean transitions. In addition, aborts cause pendingPersistent objects to revert to transient ones, cancelling their addition to the database. Similarly, aborts cancel pending deletions, causing pendingDeleted-to-persistent transitions.

In either transaction termination case, the enclosed objects are left in a clean state.

6.5.2. Implicit Retrieval

Whenever a persistent object is fetched from the database, a local copy is made. A design invarient requires that only local references exist in local copies, while only global references exist in principal copies. Hence, when creating the local copy, all global references must be translated to local ones. In the case that the referenced object is transacted, its local reference

Figure 3: Object Management State Diagram 6/15/87



is simply the oop of its local copy. However, if the referenced object is untransacted, an equivalent local object must be made for it anyhow.

Simon borrows a technique from LOOM (see reference [2]), creating an instance of the root class **Proxy** as its local reference, and backpointering the proxy to the associated TOD.

Proxy is a root class that responds to essentially all messages by attempting to fetch the object from the database, using the global reference in the proxy's TOD. If successful, the proxy is asked to **become:** the fetched local object. If no such persistent object can be found, the proxy is instead asked to **become:** an instance of **DeadObject.**

6.5.3. Implicit Storage

When inserting a new principal object or updating an existing one, the dual process of converting all local references to global ones must be performed. This implies that any referenced transient objects must be made persistent so as to have global references.

6.5.4. Scavenging

In the context of the Transaction Manager, *scavenging* means reverting objects to untransacted states so as to release their TOT-related storage. In general, the overall design attempts to minimize cycles whenever possible, such as removing all pointers between transactions and TODs when the transactions are terminated.

As implemented in Simon-1, scavenging simulates *soft* references: the TOT is scanned, and any TOD whose reference count indicates that there are no other references to it, is clean, and either transient or persistent is dropped. This process is expected to be extremely slow (and would be unnecessary if soft references were implemented inside the VM).

6.6. Query Manager Design

Queries use *cursors* provided by the server as object streams on retrieval. They are represented by opaque handles provided by the server primitives.

Each query, being an orderedCollection, cache its elements on retrieval, transacting each element in the associated transaction. The query also maintains an atEnd flag. When asked to retrieve an element beyond the cache, it checks the atEnd flag, and if reset, asks the server for the sufficient elements to fill in the cache up to the requested element, passing the cursor in the request. Attempts to access an element beyond the atEnd limit results in an error.

This design optimizes the query overhead to just those elements requested by the application, such as those actually viewed in the dataBrowser, and allows the dataBrowser to show the first few elements in a bounded time, even though queries are unbounded.

6.7. Image-Database Synchronization

When an image is being shut down, after transactions are committed, the local copies of all persistent objects are asked to **become:** their proxies, so as to force them to be refetched on the first access after image resumption.

June 16. 1987

7. Server Interface Module Design

7.1. Query Compilation

The goal of query compilation is to generate an SQL statement that will perform the equivalent query. The basic template of an SQL retrieval statement is:

select * from tableName where condition orderBy sortingColumns

The compilation is therefore very straightforward: tableName is obtained from the source class's declaration, the condition is assembled from the various filter queries, and the sortingColumns is assembled from the various sorting queries.

7.2. Oracle Schema Translator

The schema translator has the job of converting Oracle-specific data dictionary information into the generic physical schema described above. It also has the dual job of converting physical schema creation and update operations to corresponding SQL commands. Neither is very interesting.

7.3. OracleDatabase Primitives

OracleDatabase has the following primitive protocol:

- **setup: anArrayOfClassOops**
  called after startup or restart from snapshots to pass a set of class oops used by the primitives;
- **connect: userId**
  opens a connection to the database;
- **disconnect**
  closes a connection to the database; and
- **createCursor**
  creates and returns an opaque handle to an Oracle cursor.

7.4. OracleCursor Primitives

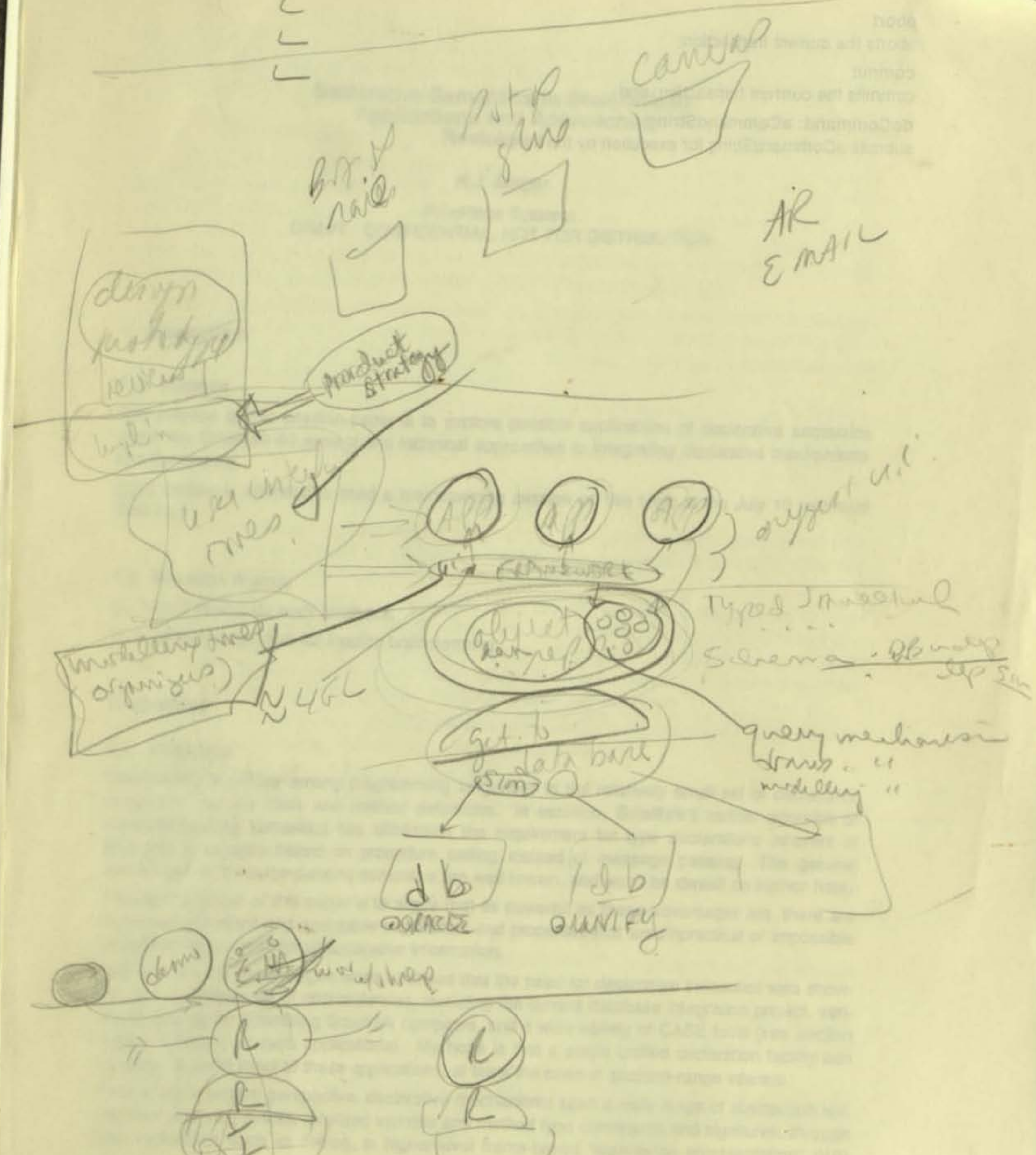
The **OracleCursor** class represents Oracle cursors, and has the following primitive protocol:

- **free**
  frees the internal cursor resource;
- **startQuery: aQueryString withMapping: anObjectMappingDescriptor**
  initiates a query on aQueryString, and establishes the record-to-object mapping;
- **next**
  sequences to the next result object, returning its global reference, or nil if at end of stream;
- **getLocalCopy**
  returns the local copy of the current result object;
- **getErrorMessage**
  returns the Oracle error message string corresponding to the last error;
- **getErrorPosition**
  returns the position of the last error (context dependent interpretation);

June 16. 1987

- **abort**
  aborts the current transaction;
- **commit**
  commits the currrent transaction; and
- **doCommand: aCommandString**
  submits aCommandString for execution by the server.

- draw picture
- dev strategy
- what kind of "user" dev



adele

Declarative Semantics in Smalltalk-80: Applications and Approaches Revision 1.1

R.J. Steiger

ParcPlace Systems

DRAFT. CONFIDENTIAL, NOT FOR DISTRIBUTION

1. Introduction

1.1. Purpose

The purpose of this position paper is to explore possible applications of declarative semantics within the Smalltalk-80 system and technical approaches to integrating declarative mechanisms into the system.

(This version is intended to seed a brainstorming session on this topic at the July 10 technical meeting.)

1.2. Revision History

[1, 7/8/87] sketch for early feedback.

[1.1, 7/9/87] rough draft for input to brainstorming.

2. Overview

2.1. Motivation

Smalltalk-80 is unique among programming languages in the relatively small set of declarative constructs, namely class and method definitions. In essence, Smalltalk's radical adoption of message-passing semantics has eliminated the requirement for type declarations inherent in type-safe languages based on procedure calling instead of message passing. The general advantages of message-passing semantics are well known, and won't be dwealt on further here.

The main purpose of this paper is to argue that as powerful as these advantages are, there are numerous important and desireable capabilities and properties that are impractical or impossible to provide without additional declarative information.

This idea for this paper began when I noticed that the need for declarative semantics was showing up in a diverse set of circumstances, including the current database integration project, various proposals for optimizing Smalltalk compilers, and a wide variety of CASE tools (see section 3 for a summary of such applications). My hope is that a single unified declaration facility can be made to serve most of these applications, at least the ones of shortest-range interest.

From a more general perspective, declarative mechanisms span a wide range of abstraction levels, from relatively simple, localized variable and method type constraints and signitures, through logic formalisms such as Prolog, to higher-level frame-based "knowledge representation" (KR)

systems. While in a brainstorming mode, I suggest that we allow ourselves to look at declarative semantics at all of these levels before focussing on short-term needs, in the hopes that some unification can result.

2.2. The Basic Declarative/Procedural Distinction

The tradeoffs between declarative and procedural approaches to the design of KR systems, programming languages, data models, and user interfaces have been the subject of numerous discussions, papers, and debates over the last 15 years or so. We summarize the main points here to establish common concepts and language for the rest of the paper. (The following discussion is in the terminology of the KR field, since this is historically where most of the attention has been focussed, but should be construed to apply to computing in general.)

An old philosophical distinction exists between "knowing what" and "knowing how". Proceduralists assert that knowledge of a given domain is represented as, and intimately bound into, a set of procedures that operate within that domain. In this view, knowledge is coextensive with knowing how to operationally apply the knowledge.

Declarativists, on the other hand, assert that knowledge is represented as a set of generic (domain-independent) inference procedures, plus a set of domain-specific facts manipulated by these procedures. In this view, knowledge is coextensive with knowing what is true about the domain.

The general advantages of declarative knowledge are:

- flexibility/economy - a given fact may be used in several ways by the general inference engine, such as forward or backward chaining; procedures are more restricted in their applicability to specific contexts;
- understandability/learnability - declarative facts tend to be more loosely coupled to - and hence more independent of - each other, and may therefore be treated "additively"; procedural systems tend to be more tightly coupled and fragile, insofar as seemingly small local changes can have massive global effects (frequently breaking a system altogether); declarative systems are thus more easily understood and changed;
- naturalness - many facts are declarative in nature, and natural language itself is primarily declarative, so seems optimized for encoding declarative facts.

The general advantages of procedural knowledge are:

- behavioral modeling - describing the behavior of systems is often done most naturally in procedural form as a set of activities;
- second-order knowledge - an essential part of knowing is knowing what we know, what we can know, and how to apply what we know; in general, the representation of such second-order knowledge is much easier in procedural terms, typically heuristic.

Winograd summarizes the arguments in the controversy as follows:

- Economy. Procedures specify knowledge by saying how it is used, and every use requires a different procedure. Declaratives require only a single copy for all uses.
- Modularity. Procedures bind knowledge and control in a single package. By keeping facts separate, a declarative approach makes it easier to update and generalize the knowledge base.
- Exception handling. Procedures can do anything, and problems that aren't covered by the formal theory can often be handled by an ad hoc piece of code. Declarative approaches may find handling of unanticipated exceptions difficult to impossible.

One may view much of the work in AI and KR over the last decade as various attempts to rationally reconcile these tradeoffs through the development of frameworks that unify declarative and procedural semantics.

3. Some Applications of Declarative Semantics

Smalltalk declarative information is encoded in its class hierarchy and compiled methods, and is able to put it to good use in the browser, providing a rich variety of information about program structure.

The following is a top-level summary of some additional capabilities that would be rendered more feasible through expansion/generalization of Smalltalk's declarative mechanisms:

- optimizing compilation
- user documentation and annotation to improve readability
- static program checking
- mechanical generation of interfaces to external/remote services:
  - libraries written in other languages
  - databases and file systems
  - graphics presentation services
  - specialized processors (e.g. signal or array processors)
- filter-based user interaction frameworks
- software system structural design, analysis, and display tools:
  - context-specific senders, implementors, accessors relations
  - data structure schema display
  - better modification impact reporting
- improved semantic data modelling
- online documentation, assistance, and learning facilities
- data entry assistance (for commands, information forms, property sheets, etc):
  - completion
  - defaulting
  - explanation
  - integrity enforcement (type/range checking, other constraints)
- more general logical inference (ala Prolog and other logic languages).

4. Competition

The following competitive or potentially-competitive products have some form of declarative semantics, which they use to gain performance, static-checking, and other advantages:

- Common Lisp
- Objective C
- C++
- Actor
- Eiffel

- Object Pascal
- Opal/Gemstone
- VBase (Ontologic's OODB)
- ObjTalk
- QuickTalk

In addition, numerous experimental object-oriented systems employ some form of declarative semantics, such as:

- Typed Smalltalk [Johnson87]
- Field's dataflow type inferencing in Smalltalk [Field87]
- Filter Browser [Borning87]
- SIG [?]
- Incense [?]
- Impulse-86 [Smith86]

5. References

[Bobrow75]
Bobrow, D., and Collins, A. *Representation and Understanding: Studies in Cognitive Science,* Academic Press, 1975.

[Field87]
Field, R., *Data Flow Type Inferencing in Smalltalk: Detection of Generic Classes* Master's Thesis, UCSC, June 1987.

[Johnson87]
Johnson, R., *A User's Guide to Typed Smalltalk* Dept. CS., U. Ill., May 1987.

[Smith86]
Smith, R.G., Dinitz, R., Barth, P. *Impulse-86: A Substrate for Object-Oriented Interface Design* in [OOPSLA86]

[Sowa84]
Sowa, J. *Conceptual Structures: Information Processing in Mind and Machine,* Addison-Wesley IBM Systems Programming Series, 1984.

[Winograd75]
Winograd, T. Frame Representations and the Declarative-Procedural Controversy, in [Bobrow75].

Date

19 FEB

ROUTING AND TRANSMITTAL SLIP

TO: (Name, office symbol, room number, building, Agency/Post)	Initials	Date
1. AC/STB	RH	20 FEB
2. DC/ASG		20 FEB 1980
3. C/ASG	RH	20 Feb.
4.		
5. Angela Coppola last		

	Action		File		Note and Return
	Approval		For Clearance		Per Conversation
	As Requested		For Correction		Prepare Reply
	Circulate	X	For Your Information		See Me
	Comment		Investigate		Signature
	Coordination		Justify		

REMARKS

I found this extract from Berkley's "On-Going Research Projects" interesting & thought I'd pass it along.

The publication is disseminated as part of Berkley's Industrial Liaison Program. The section I copied was from the Dept. of Electrical Engineering & Computer Sciences/Electronics Research Laboratory.

Membership in the Industrial Liaison Program apparently comes with the purchase of Berkley's smallest implementation on a SUN ($10,000).

DO NOT use this form as a RECORD of approvals, concurrences, disposals, clearances, and similar actions

FROM: (Name, org. symbol, Agency/Post)	Room No.—Bldg.
Angela Coppola	1E1508
	Phone No. 5415

5041-102

OPTIONAL FORM 41 (Rev. 7-76)
Prescribed by GSA
FPMR (41 CFR) 101-11.206

Programming Languages and Systems

Our research projects in programming systems seek to facilitate software production by providing advanced interactive systems, improved programming language translation capabilities, and support for program development. Our efforts are increasingly directed toward exploiting the benefits of high-speed personal workstations to enhance programmer productivity.

We are studying interactive systems both for nonprogrammers and for experts. A system for the interactive development of programs by modification of example computations has recently been completed. A high-quality document preparation system is being designed. We are also developing a language-based editor, intended as the user interface both for program design and for preparation of other kinds of structured text.

As part of an integrated hardware-software system for Smalltalk-80 called SOAR, we are developing a compiler, a debugger, a garbage collector, and the operating environment for Smalltalk. Our research on the implementation of Ada* is focusing on the design of the runtime system.

We have an ongoing project to develop tools to automate the production of high-quality compilers. We have just completed a study of techniques for register allocation and intermediate representation. We have developed a new method for automated discovery of low-level target code improvements. Under investigation are a tree transformation system and techniques for the automatic generation of symbol table managers.

An important aspect of software development is the design, construction, and maintenance of large systems. We have implemented an interactive transition diagram editor, used to model parts of the development process. The Evolution Support Environment is being designed to provide a variety of support facilities. In addition, we are designing and building software tools for new forms of computer-assisted collaboration.

*Ada is a trademark of the Department of Defense (Ada Joint Program Office).

Programming by Example

Daniel C. Halbert
(Professor S. L. Graham)

Xerox Corporation

Most computer-based applications systems cannot be programmed by their users. Programming is considered a difficult skill for the average person to learn, so most systems do not provide facilities for ordinary users to write programs that help them do their work.

We believe, however, that ordinary users could program their systems using a technique called "programming by example," which is a way of programming a system in its own user interface. The system user writes a program by giving an example of what it should do. The system remembers the sequence of actions and can perform it again. Succinctly, programming by example is "Do What I Did."

Programming by example has been added to a simulation of a commercial office information system. In addition to a basic programming-by-example mechanism, the facility provides program parameterization, data searching and selection mechanisms, control structure, and a static, readable program representation that can be edited.

Multiple Representations of Documents

Charles L. Perkins
(Professor M. A. Harrison)

NSF Graduate Fellowship and
(DARPA) N00039-82-C-0235

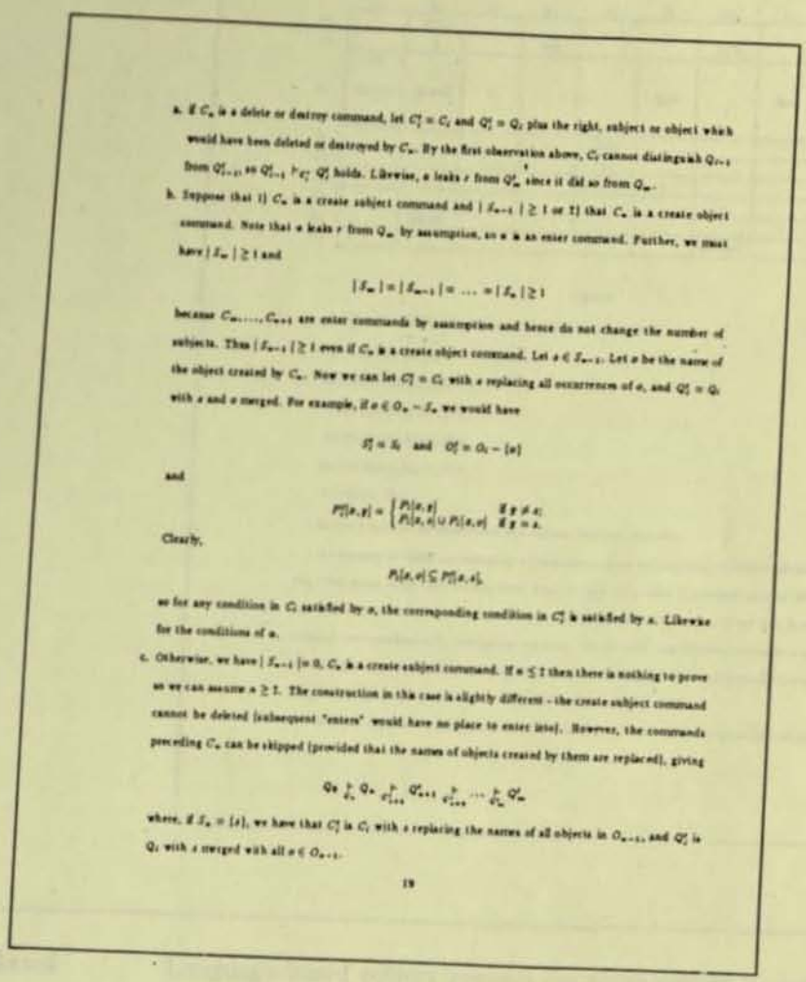
With the now widespread popularity of document preparation systems and with the advent of new technologies, such as local workstations and bit-mapped displays, it makes sense to reexamine these systems and to see what new techniques can be employed. For example, a uniform mechanism for combining two-dimensional media (text, drawings, photographs, cifplots, etc.) will be the center of a new, interactive document system that automatically updates parts of a document when its source has changed. The system would present a page-by-page approximation of the final document on the display, and the user could interactively update those parts of the document for which local editors exist. It may even be possible to have an editor that presents a uniform interface to objects of distinctly different types.

Presently, different editors exist at Berkeley (e.g., vi and EMACS for text, Gremlin for diagrams, and Magic for VLSI designs). An integrated system could use these editors as black boxes, transforming to and from a common representation when needed. The environment thus created is envisioned as being more interactive and incremental than previous batch-oriented systems and would allow the entry of any new two-dimensional data whose format was describable within the system.

This project is working in two different directions toward the system described above. At a high level, techniques for incremental rederivation and for managing multiple representations of two-dimensional data must be found. A user interface, a common representation, and a framework for them both must be designed. Progress here has included identifying the problems involved and researching systems that have tried to solve subsets of them. An interesting formalism that could be used to automatically derive new transformations was discovered.

At a lower level, the editors and transformation programs here at Berkeley must be brought closer together. Some other groups here have unwittingly aided this effort (e.g., Gremlin and Magic have been ported to the SUNs). Also, a simple system for experiments must be set up. Much of the effort has been spent here, exploring project feasibility. The typesetting language TeX, with its notions of boxes and glue, has been adopted as a basis for the system. TeX produces device-independent output files that can be printed on all our local bit-raster printers (and many others). We have recently ported TeX to the SUN workstation, along with most of its related software. A previewer for TeX on the SUNs has been adapted

to run under the window system. These represent the first steps in bringing up a simple prototype of the system. Progress in the future will focus on completing and using this prototype.



a. If C_n is a delete or destroy command, let $C_i' = C_i$ and $Q_i' = Q_i$ plus the right, subject or object which would have been deleted or destroyed by C_n. By the first observation above, C_i cannot distinguish Q_{i-1} from Q_{i-1}' so $Q_{i-1}' \vdash_{C_i'} Q_i'$ holds. Likewise, a leaks r from Q_n' since it did so from Q_n.

b. Suppose that 1) C_n is a create subject command and $|S_{n-1}| \geq 1$ or 2) that C_n is a create object command. Note that a leaks r from Q_n by assumption, so a is an enter command. Further, we must have $|S_n| \geq 1$ and

$$|S_n| = |S_{n-1}| = \ldots = |S_i| \geq 1$$

because $C_n, \ldots, C_{i+1}$ are enter commands by assumption and hence do not change the number of subjects. Thus $|S_{i-1}| \geq 1$ even if C_n is a create object command. Let $s \in S_{i-1}$. Let o be the name of the object created by C_n. Now we can let $C_i' = C_i$ with s replacing all occurrences of o, and $Q_i' = Q_i$ with s and o merged. For example, if $o \in O_n - S_n$ we would have

$$S_i' = S_i \quad \text{and} \quad O_i' = O_i - \{o\}$$

and

$$P'[s,y] = \begin{cases} P[s,y] \\ P[s,s] \cup P[s,o] \end{cases} \begin{array}{l} \text{if } y \neq s; \\ \text{if } y = s. \end{array}$$

Clearly,

$$P[s,o] \subseteq P'[s,s],$$

so for any condition in C_i satisfied by o, the corresponding condition in C_i' is satisfied by s. Likewise for the conditions of a.

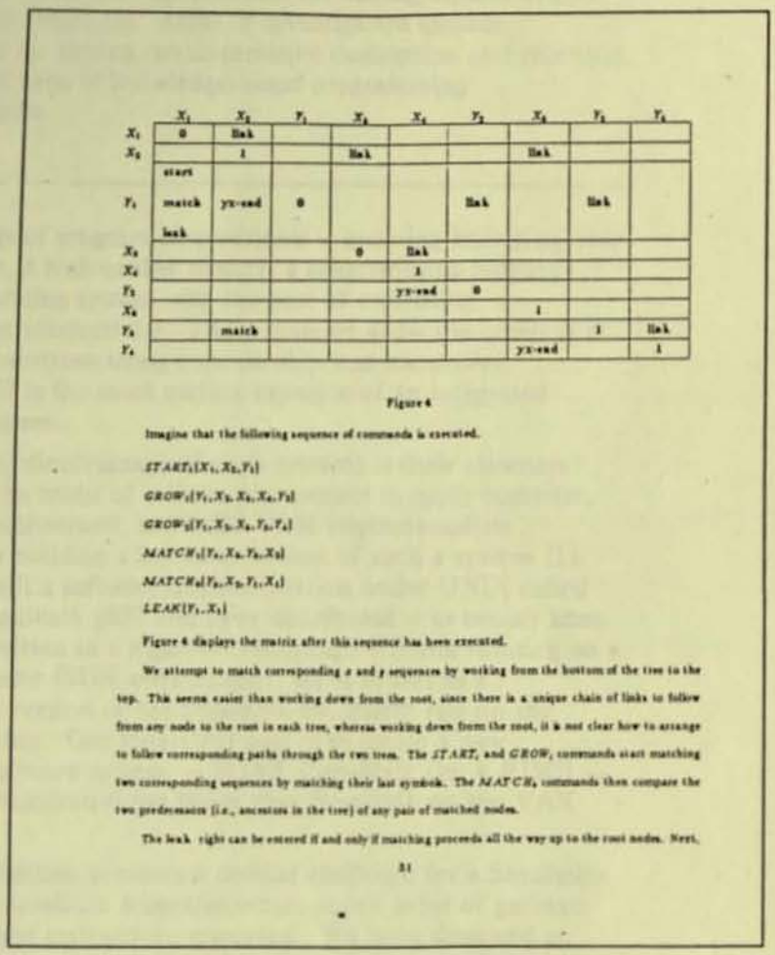
c. Otherwise, we have $|S_{n-1}| = 0$, C_n is a create subject command. If $n \leq 2$ then there is nothing to prove so we can assume $n \geq 3$. The construction in this case is slightly different - the create subject command cannot be deleted (subsequent "enters" would have no place to enter into). However, the commands preceding C_n can be skipped (provided that the names of objects created by them are replaced), giving

$$Q_0 \vdash_{C_0'}^{*} Q_{n'} \vdash_{C_{i+1}'}^{*} Q_{i+1}' \vdash_{C_{i+2}'}^{*} \cdots \vdash_{C_n'}^{*} Q_n'$$

where, if $S_n = \{s\}$, we have that C_i' is C_i with s replacing the names of all objects in O_{n-1}, and Q_i' is Q_i with s merged with all $o \in O_{n-1}$.

19

Fig. 14-1. The two document pages shown on this page and the next are from a TeX document being displayed by dvisun on a SUN-150. Using current fonts, only two-thirds of each page can be shown on the display at one time; commands can move this 'window' in any direction to reach the rest of the page. Although one page involves some graphics and the other many font changes, dvisun can redisplay them in one and two seconds, respectively. Average pages take only about three-quarters of a second. This is five to ten times faster than the local ditroff previewer.



	X_1	X_2	Y_1	X_3	X_4	Y_2	X_5	Y_3	Y_4
X_1	0	leak							
X_2		1		leak			leak		
Y_1	start match leak	yy-end	0			leak		leak	
X_3				0	leak				
X_4					1				
Y_2					yy-end	0			
X_5							1		
Y_3		match						1	leak
Y_4							yy-end		1

Figure 4

Imagine that the following sequence of commands is executed.

$START_1[X_1, X_2, Y_1]$

$GROW_1[Y_1, X_3, X_4, X_5, Y_2]$

$GROW_2[Y_1, X_3, X_4, Y_3, Y_4]$

$MATCH_3[Y_4, X_4, Y_3, X_3]$

$MATCH_4[Y_3, X_3, Y_1, X_1]$

$LEAK[Y_1, X_1]$

Figure 4 displays the matrix after this sequence has been executed.

We attempt to match corresponding x and y sequences by working from the bottom of the tree to the top. This seems easier than working down from the root, since there is a unique chain of links to follow from any node to the root in each tree, whereas working down from the root, it is not clear how to arrange to follow corresponding paths through the two trees. The *START*, and *GROW*, commands start matching two corresponding sequences by matching their last symbols. The $MATCH_k$ commands then compare the two predecessors (i.e., ancestors in the tree) of any pair of matched nodes.

The leak right can be entered if and only if matching proceeds all the way up to the root nodes. Next,

---

Higher-Level Language-Based Editors

Robert A. Ballance
(Professor S. L. Graham)

(DARPA) N00039-84-C-0089

Language-based editors support the programmer by using language-specific information during the editing process. This support includes checking for syntactic or semantic errors, template-based entry of basic structures, and special display algorithms for viewing the program. To date, most systems impose a rigid development methodology on the user.

I am interested in developing "higher-level" language-based editors that support multiple languages and allow users to manipulate programs in terms of the underlying language. This approach subsumes both text- and structure-based editing. Programs and structured text can be manipulated either as text or in terms of their underlying structures. For example, in a program, the user might choose to operate on functions, blocks, statements, expressions, tokens, or characters. At all times, the editor will offer full flexibility between text and structure.

This research is aimed at creating an editor-generating system that accepts a language description as input, creating tables and code for use in a standard front end. The standard front end

provides a consistent user interface for editing objects written in different languages. Areas of investigation include algorithms for parsing, static-semantic description and checking, and connections to knowledge-based programming environments.

Experimental Design of a High-Performance, Object-Oriented Personal Computing System

David Ungar
(Professor D. A. Patterson)

(DARPA) N00039-83-K-0107
and IBM Corporation

A new class of programming systems is evolving that integrates a processor, a high-quality display, a programming language, and an operating system with the goal of enhancing programmer productivity. These systems allow the creation of software prototypes using considerably less manpower. Smalltalk-80 is the most mature example of an integrated software system.

The primary disadvantage of such systems is their slowness. We are in the midst of a three-year project to apply compiler, systems, architectural, and MOS VLSI implementation expertise to building a low-cost version of such a system [1]. We have built a software implementation under UNIX called Berkeley Smalltalk (BS) and have distributed it to twenty sites. Although written in a high-level language (C) and running on a microprocessor (SUN workstation), BS is as fast as a microcoded version of the Smalltalk-80 system run on the Xerox Dolphin. Our long-term goal is to create a new hardware/software system — SOAR (Smalltalk On A RISC) — that runs a hundred times faster than Smalltalk on the VAX 11/750.

Garbage collection presents a serious challenge for a Smalltalk-80 system. Smalltalk programs create seven bytes of garbage for every eight instructions executed. We have designed an algorithm called Generation Scavenging and incorporated it into BS [2]. Pauses disrupt thought and decrease productivity. The pause time for our algorithm is only a fraction of a second. All other Smalltalk-80 systems need indirection to help manage objects. BS, with Generation Scavenging, is the first one with direct object addressing. Our garbage collector also runs in half the time of the best previous algorithm.

[1] D. Ungar, R. Blau, P. Foley, D. Samples, and D. A. Patterson, "Architecture of SOAR: Smalltalk on a RISC," *11th Annual Symp. on Computer Architecture*, Ann Arbor, MI, 1984.

[2] D. Ungar, "Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm," *ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments*, Pittsburgh, PA, April 1984.

A Smalltalk Compiler for SOAR

William Bush
(Professors P. N. Hilfinger and D. A. Patterson)

(DARPA) N00039-83-C-0107

A necessary part of the SOAR (Smalltalk On A RISC) project involved constructing a compiler that would compile Smalltalk programs into SOAR machine instructions instead of the standard Smalltalk virtual machine bytecodes. The primary concern in designing this compiler was the code explosion that could result from using SOAR instructions instead of the much denser bytecodes.

That the semantics of Smalltalk preclude many standard optimizations made the Smalltalk bytecodes themselves an attractive intermediate representation for the compiler. The runtime stack used by the stack-oriented bytecodes is simulated by the compiler at compile time, which converts those operations to register-oriented SOAR instructions. This in practice produces reasonably dense SOAR code, with an average of one SOAR instruction generated for each bytecode. The compiler was written in Smalltalk, and the Smalltalk environment, although sophisticated, requires substantially more resources (primarily in terms of CPU power) than a conventional one in order to provide equivalent programming throughput.

A Debugger for SOAR (Smalltalk On A RISC)

Peter K. Lee
(Professors P. N. Hilfinger and D. A. Patterson)

(DARPA) N00039-83-C-0107

Smalltalk On A RISC (SOAR) is a microprocessor designed to run Smalltalk efficiently. Smalltalk is defined on the Smalltalk Virtual Machine (STVM), and the compiler generates virtual machine instructions, known as *bytecodes*, from Smalltalk methods. On SOAR, bytecode methods are further translated into SOAR machine instructions to be executed by the hardware.

The debugger in Xerox's Smalltalk Virtual Image operates by simulating the semantics of the bytecode instructions, which are no longer available on SOAR. There are also problems with the breakpoint-setting mechanism in the bytecode debugger. Breakpoints are set by inserting "halt" instructions in the instruction stream and recompiling the procedure. This makes setting breakpoints in recursive routines impossible.

The following are guidelines for the design of the SOAR Debugger:

- To provide the basic mechanisms for performing the same functions as the bytecode debugger so that high-level software, like the Debugger Browser, can be reused with minimal change
- To provide a better breakpoint mechanism so that breakpoints can be set and unset without recompiling the procedure
- To allow breakpoints to be taken conditionally, thus allowing breakpoints to be set in recursive routines or routines that are shared with the debugger itself.

Smalltalk On A RISC (SOAR)

A. Dain Samples
(Professors P. N. Hilfinger and D. A. Patterson)

(DARPA) N00039-83-C-0107

Professor Patterson has set out to design a RISC (Reduced Instruction Set Computer) microprocessor that will provide a fast execution vehicle for the Smalltalk-80 programming environment. The Daedalus project under Professor Hilfinger has provided an intermediate step toward implementing Smalltalk on this processor (SOAR). A basic version of Daedalus that allows testing of benchmarks and large portions of the Smalltalk-80 system is now running. Most of the Smalltalk-80 runtime system is written in the Smalltalk language itself; this portion has been translated into SOAR machine code.

We are now in the final stages of designing the RISC/UNIX interface required to run Smalltalk on the hardware. When the chip is fabricated, we will have hardware and software ready to execute Smalltalk using SUN workstations to handle files and graphics. My current tasks are to finish the operating environment specifications and to document the runtime system and virtual machine.

Implementing an Efficient Runtime Organization for Ada

Benjamin G. Zorn
(Professor P. N. Hilfinger)

NSF Graduate Fellowship and
(DARPA) N00039-84-C-0235
and N00039-84-C-0089

Research supervised by Paul Hilfinger was aimed at designing and implementing an efficient and conceptually simple runtime strategy for Ada. The implementation involved using an Ada front end (provided by AT&T Bell Laboratories and modified at Berkeley) that produces DIANA, a proposed standard intermediate representation (IR) for Ada programs. Our so-called "middle end" takes the DIANA representation and produces the lower-level intermediate tree form used by the portable C compiler.

Experiences with the DIANA representation have shown that a normalization pass is necessary to make the representation usable by the middle end. Although the IR of the portable C compiler was not intended as a low-level representation for Ada, we found that it was almost entirely adequate for the task. Furthermore, using the IR allowed us to take advantage of table-driven code-generation tools being researched at Berkeley.

The runtime design stressed efficient application of uniform runtime type representations to implement such features as dynamic arrays and parameterized records. Furthermore, the implementation of these objects did not introduce any distributed overhead on the implementation of objects familiar to the Pascal user. Execution time for a Pascal subset of Ada was comparable to the execution time of code generated by the Berkeley Pascal compiler, as demonstrated by a small set of benchmarks.

Register Allocation and Data Conversion in Machine-Independent Code Generators

Marshall Kirk McKusick
(Professor S. L. Graham)

(DARPA) N00039-82-C-0235
and N00039-84-C-0089

The final goal of our work is to produce a set of tools that will allow the construction of high-quality code generators for Von Neumann-architecture computers in a short time and with a minimum of machine-specific coding.

We have already developed a table-driven code generator using the Graham-Glanville method. We are working on developing formal methods to attack the problems that it fails to address. The result will be a largely table-driven code generator that is retargetable to diverse architectures with a minimum of recoding.

One major goal of this research is to formalize the register allocation of the code generator. We are investigating how to coalesce the optimizations found by a procedurewide data-flow analyzer with the allocation of registers needed during the process of instruction selection. The allocators are driven by a description of the number and types of various registers and some policy description. We are evaluating various coloring techniques as the basis of the allocation policy.

Another major goal of this research is to investigate semantic alternatives to the current syntactic specification of conversions done by the code generator. We are experimenting with different specifications to measure their costs in terms of size, space, and comprehensibility.

Automated Discovery of Machine-Specific Code Improvements

Peter B. Kessler
(Professor S. L. Graham)

(DARPA) N00039-82-C-0235
and N00039-84-C-0089

I am investigating techniques to automate the discovery of machine-specific transformations that will improve the quality of code produced by retargetable compilers. Current retargetable code generators produce provably correct, often optimal code for single statements. However, they fail to take full advantage of target architectures and often use complex instructions only with hand-coding to recognize special cases. The goal of this project is to recognize those special cases automatically, thus making retargetable code generators more robust and easier to retarget.

I distinguish two stages in this process. The first analyzes a description of the target machine when the compiler is constructed and generates tables for the second stage, which transforms each program run through the compiler. This separation allows the analysis of the target machine to be arbitrarily thorough in its attempts to exploit features of the target machine and reduces transformation to a simple pattern match and replacement. The analysis "decomposes" the complex instructions of the target machine, finding sequences of instructions that can be replaced by those complex instructions. The transformation stage uses information derived by the analysis to transform assembler source code. Automating transformation of other representations of programs is possible. A prototype system has been demonstrated and retargeted, and a dissertation describing this work is in progress.

Tree Transformation Systems in Compilers

Eduardo Pelegri-Llopart
(Professor S. L. Graham)

(DARPA) N00039-84-C-0089
and Venezuelan Research
Council (CONICIT)

Trees are convenient representations in many situations largely because of their hierarchical structure, which models many situations, and because of the ease with which they can be manipulated. This manipulation frequently corresponds to transformations between different tree representations to expose or modify some properties of the object being represented.

The goal of our project is to investigate tree transformations, especially in the context of compilation systems, program transformation systems, and programming environments. We will use results and experiences from the areas of term rewriting systems, production systems, and programming languages, among others, to find descriptional mechanisms that are adequate for efficient implementation while also being easy to program, have provable properties, and have an adequate interface to the compiling process, particularly to pattern-matcher-based code generators.

Our approach is quite pragmatic, and we hope to design a tree transformation tool in the tradition of scanner, parser, and code-generator generators. As an application of our research, we expect to fill the gap between the high-level abstract trees that can be obtained from the parser and the low-level trees required by the Graham/Glanville/Henry technique of code generators. In the process, we expect to clarify the relation between program optimization and code generation.

We have implemented a simple tree transformation system, which we have used to gather some first experience. We recently completed an extensive bibliography revision and are evaluating different description mechanisms.

Automatic Generation of Symbol Table Managers from Specifications

Phillip E. Garrison
(Professor S. L. Graham)

(DARPA) N00039-82-C-0235
and N00039-84-C-0089

Symbol Table Managers (STMs) must currently be hand-coded, though some efforts are under way to automate their production. The task of writing an STM can be very difficult for languages such as Ada. Automatic production of an STM from a specification of the relevant parts of the language being processed would have a number of benefits: less work, greater understandability and modifiability, improved faith in the correctness of the generated STM, and automatic optimization. Such a specification would also be useful for formal language definition and for language comparison.

A model of scoping and naming in languages has been developed, and the primitive operations implied by this model have been identified. A functional specification language based on these primitives has been designed as an extension of an attribute grammar system. The extensions will be implemented so that the usefulness of this specification language can be evaluated on real languages.

Because symbol tables are large objects, copying them unnecessarily must be avoided if reasonable efficiency is to be achieved. Elimination of simple copies is well understood, but

elimination of copies necessitated by modification of objects has been studied only in certain cases (e.g., pass-oriented attribute grammars). Two different methods that will handle ordered attribute grammars have been developed.

The Transition Diagram Editor

Charles C. Mills
(Professor A. I. Wasserman*)

(DARPA) N00039-82-C-0235

This project created a highly interactive mouse-and-menu-driven graphical editor for state transition diagrams, along with a generator that provides input to a transition diagram interpreter. The Transition Diagram Editor (TDE) is used to support the User Software Engineering (USE) methodology, an approach to the specification and implementation of interactive information systems. Augmented state transition diagrams are used to model human-computer interaction, with nodes representing system output and arcs (transitions) associated with user input; system operations occur during a transition.

Rather than using a textual language to describe the diagram structure, TDE allows the diagram to be drawn and edited interactively, then generates the textual diagram description for interpretive execution. TDE and its related tools are especially effective for rapid prototyping of interactive systems. TDE runs on the SUN workstation; the other tools run not only on the SUN, but on most other UNIX systems.

*U.C. San Francisco

Evolution Support Environment (ESE)

Kozo Bannai, Atul Prakash, Jaideep Srivastava, Wei-Tek Tsai, and Yutaka Usuda
(Professor C. V. Ramamoorthy)

(BMDSC) DASG60-81-C-0025

Evolution Support Environment (ESE) is an integrated and automatic environment for the software development/evolution process. An ultimate goal for the software development process is to develop an automatic software family generator that, given the specifications for a member of a family, could generate an implementation for the member by reusing as much existing software as possible. Three basic requirements for such an environment include promoting (1) traceability between user requirements, design, and code, (2) reusability of existing designs and code, and (3) compatibility between various phases of the software life cycle. Our next step is to design and develop the Software Engineer Assist System, which would guide the designer's decisions based on metrics (we call that Metric-Guided Design Methodology).

CoLab — Tools for Computer Collaboration

Gregg Foster
(Professor R. J. Fateman)

(DOE) DEAMO3-76500034
and Xerox Corporation

CoLab is a laboratory to experiment with new forms of computer-assisted collaboration. Although networks connect computers and enable electronic mail and sharing of facilities, computer systems aren't usually designed for group activities. When we think of people working with computers, we usually think of them in separate offices working mostly in isolation. To use computers for demonstrations, several people gather around a display designed for a single person. If people decide to work together on a problem, they leave their computers behind and go to a whiteboard. Secondary ideas, arguments,

and random notes are often lost or forgotten when records of a collaborative session must be entered into a computer system as a separate step. When instructors train people to use interactive programs, there is no easy way to interact with several students at once.

Recent technological advances (e.g. Ethernet, the use of the mouse, EvalServer [evaluation of LISP s-exps on remote machines], and bitmap displays) have made possible new software tools and new classes of tools, including tools for group activity mediation and enhancement. There has been little previous study of personal vs. group use of computers.

We are designing software tools for intellectual teamwork (enlightened group problem solving). The software design of these tools addresses the synchronization of shared objects and data space, the network coordination of closely interacting machines, and forms of new primitives for active and interactive displays.

The goals of the CoLab project are as follows:

- To explore the dynamics of group problem solving and interaction
- To explore existing communication devices and paradigms used for collaboration
- To experiment with software, hardware, and social techniques to assist group problem solving
- To build the software foundations and several tools for these experiments and explorations
- To analyze real group use of our system and tools.

CoLab is expected to be an environment in which computers unobtrusively support human interactions, not one in which humans only use computers.

Programming Languages and Systems

Our research projects in programming systems seek to facilitate software production by providing advanced interactive systems, improved programming language translation capabilities, and support for program development. Our efforts are increasingly directed toward exploiting the benefits of high-speed personal workstations to enhance programmer productivity.

We are studying interactive systems both for nonprogrammers and for experts. A system for the interactive development of programs by modification of example computations has recently been completed. A high-quality document preparation system is being designed. We are also developing a language-based editor, intended as the user interface both for program design and for preparation of other kinds of structured text.

As part of an integrated hardware-software system for Smalltalk-80 called SOAR, we are developing a compiler, a debugger, a garbage collector, and the operating environment for Smalltalk. Our research on the implementation of Ada* is focusing on the design of the runtime system.

We have an ongoing project to develop tools to automate the production of high-quality compilers. We have just completed a study of techniques for register allocation and intermediate representation. We have developed a new method for automated discovery of low-level target code improvements. Under investigation are a tree transformation system and techniques for the automatic generation of symbol table managers.

An important aspect of software development is the design, construction, and maintenance of large systems. We have implemented an interactive transition diagram editor, used to model parts of the development process. The Evolution Support Environment is being designed to provide a variety of support facilities. In addition, we are designing and building software tools for new forms of computer-assisted collaboration.

*Ada is a trademark of the Department of Defense (Ada Joint Program Office).

Programming by Example

Daniel C. Halbert
(Professor S. L. Graham)

Xerox Corporation

Most computer-based applications systems cannot be programmed by their users. Programming is considered a difficult skill for the average person to learn, so most systems do not provide facilities for ordinary users to write programs that help them do their work.

We believe, however, that ordinary users could program their systems using a technique called "programming by example," which is a way of programming a system in its own user interface. The system user writes a program by giving an example of what it should do. The system remembers the sequence of actions and can perform it again. Succinctly, programming by example is "Do What I Did."

Programming by example has been added to a simulation of a commercial office information system. In addition to a basic programming-by-example mechanism, the facility provides program parameterization, data searching and selection mechanisms, control structure, and a static, readable program representation that can be edited.

Multiple Representations of Documents

Charles L. Perkins
(Professor M. A. Harrison)

NSF Graduate Fellowship and
(DARPA) N00039-82-C-0235

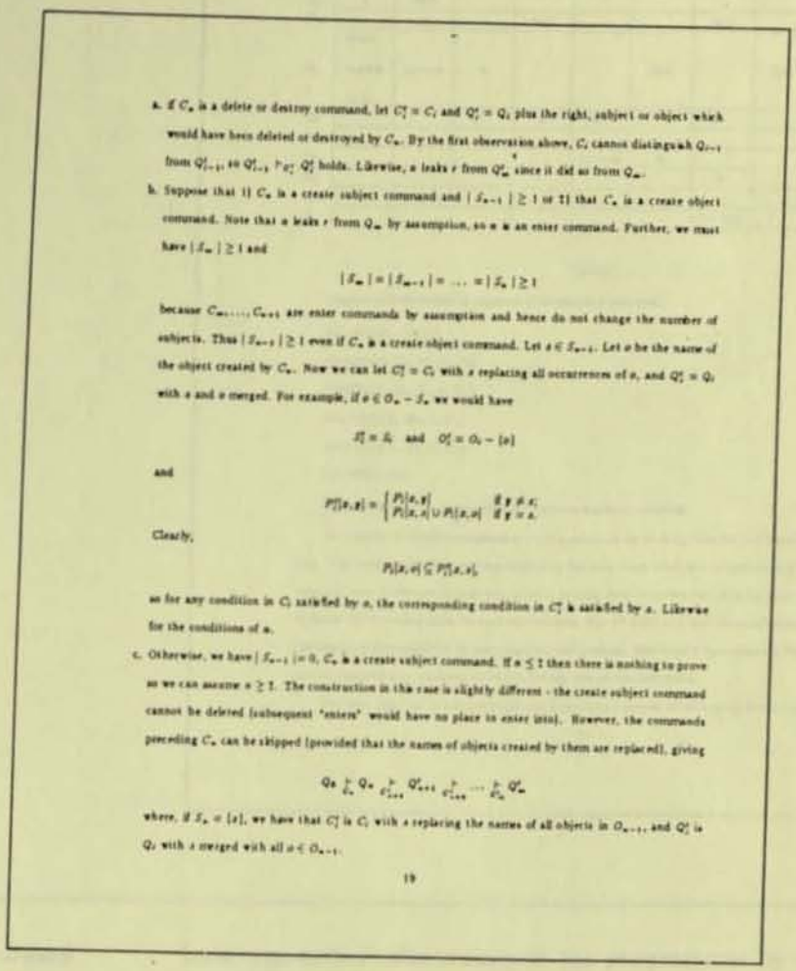
With the now widespread popularity of document preparation systems and with the advent of new technologies, such as local workstations and bit-mapped displays, it makes sense to reexamine these systems and to see what new techniques can be employed. For example, a uniform mechanism for combining two-dimensional media (text, drawings, photographs, cifplots, etc.) will be the center of a new, interactive document system that automatically updates parts of a document when its source has changed. The system would present a page-by-page approximation of the final document on the display, and the user could interactively update those parts of the document for which local editors exist. It may even be possible to have an editor that presents a uniform interface to objects of distinctly different types.

Presently, different editors exist at Berkeley (e.g., vi and EMACS for text, Gremlin for diagrams, and Magic for VLSI designs). An integrated system could use these editors as black boxes, transforming to and from a common representation when needed. The environment thus created is envisioned as being more interactive and incremental than previous batch-oriented systems and would allow the entry of any new two-dimensional data whose format was describable within the system.

This project is working in two different directions toward the system described above. At a high level, techniques for incremental rederivation and for managing multiple representations of two-dimensional data must be found. A user interface, a common representation, and a framework for them both must be designed. Progress here has included identifying the problems involved and researching systems that have tried to solve subsets of them. An interesting formalism that could be used to automatically derive new transformations was discovered.

At a lower level, the editors and transformation programs here at Berkeley must be brought closer together. Some other groups here have unwittingly aided this effort (e.g., Gremlin and Magic have been ported to the SUNs). Also, a simple system for experiments must be set up. Much of the effort has been spent here, exploring project feasibility. The typesetting language TeX, with its notions of boxes and glue, has been adopted as a basis for the system. TeX produces device-independent output files that can be printed on all our local bit-raster printers (and many others). We have recently ported TeX to the SUN workstation, along with most of its related software. A previewer for TeX on the SUNs has been adapted

to run under the window system. These represent the first steps in bringing up a simple prototype of the system. Progress in the future will focus on completing and using this prototype.



a. If C_n is a delete or destroy command, let $C_n' = C_n$ and $Q_n' = Q_n$ plus the right, subject or object which would have been deleted or destroyed by C_n. By the first observation above, C_n cannot distinguish Q_{n-1} from Q_{n-1}', so $Q_{n-1}' \vdash_{C_n'} Q_n'$ holds. Likewise, a leaks r from Q_n' since it did so from Q_n.

b. Suppose that 1) C_n is a create subject command and $|S_{n-1}| \geq 1$ or 2) that C_n is a create object command. Note that a leaks r from Q_n by assumption, so $n \neq 0$ an enter command. Further, we must have $|S_n| \geq 1$ and

$$|S_n| = |S_{n-1}| = \ldots = |S_0| \geq 1$$

because $C_n, \ldots, C_{n+1}$ are enter commands by assumption and hence do not change the number of subjects. Thus $|S_{n-1}| \geq 1$ even if C_n is a create object command. Let $s \in S_{n-1}$. Let o be the name of the object created by C_n. Now we can let $C_n' = C_n$ with s replacing all occurrences of o, and $Q_n' = Q_n$ with s and o merged. For example, if $o \in O_n - S_n$ we would have

$$S_n' = S_n \quad \text{and} \quad O_n' = O_n - \{o\}$$

and

$$P_n'[x,y] = \begin{cases} P_n[x,y] & \text{if } y \neq s; \\ P_n[x,s] \cup P_n[x,o] & \text{if } y = s. \end{cases}$$

Clearly,

$$P_n[x,o] \subseteq P_n'[x,s],$$

so for any condition in C_n satisfied by o, the corresponding condition in C_n' is satisfied by s. Likewise for the conditions of o.

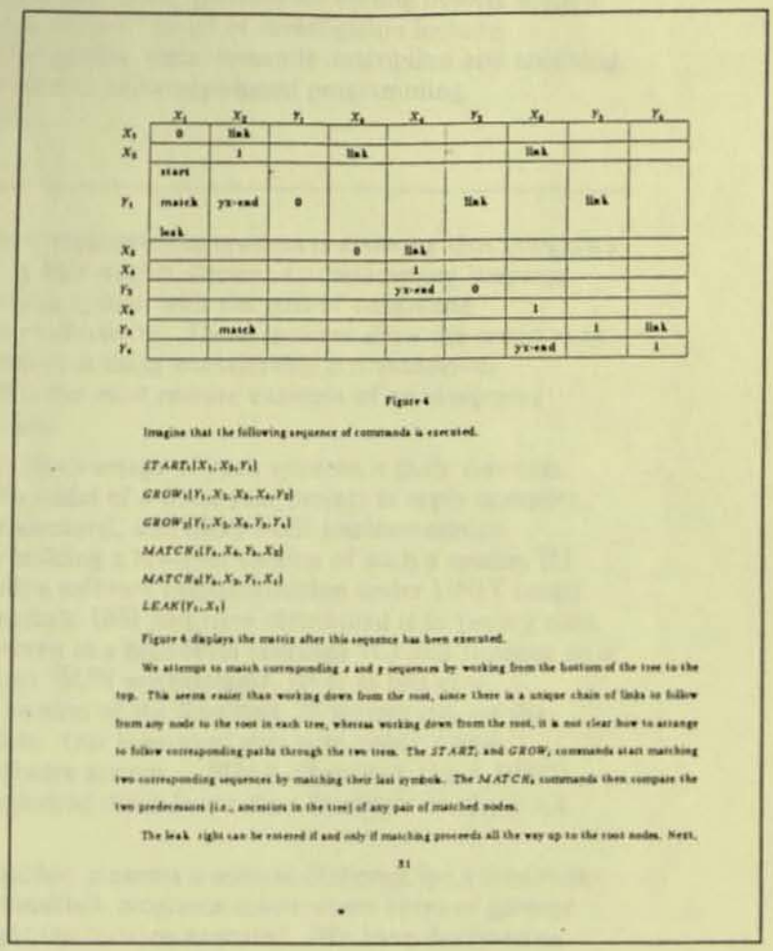
c. Otherwise, we have $|S_{n-1}| = 0$, C_n is a create subject command. If $n \leq 1$ then there is nothing to prove so we can assume $n \geq 1$. The construction in this case is slightly different - the create subject command cannot be deleted (subsequent 'enters' would have no place to enter into). However, the commands preceding C_n can be skipped (provided that the names of objects created by them are replaced), giving

$$Q_0 \vdash_{\alpha}^{*} Q_n \vdash_{C_{n+1}'}^{} Q_{n+1}' \vdash_{C_{n+2}'}^{} \cdots \vdash_{C_m'}^{} Q_m'$$

where, if $S_n = \{s\}$, we have that C_i' is C_i with s replacing the names of all objects in O_{n-1}, and Q_i' is Q_i with s merged with all $o \in O_{n-1}$.

19

Fig. 14-1. The two document pages shown on this page and the next are from a TeX document being displayed by dvisun on a SUN-150. Using current fonts, only two-thirds of each page can be shown on the display at one time; commands can move this 'window' in any direction to reach the rest of the page. Although one page involves some graphics and the other many font changes, dvisun can redisplay them in one and two seconds, respectively. Average pages take only about three-quarters of a second. This is five to ten times faster than the local ditroff previewer.



	X_1	X_2	Y_1	X_3	X_4	Y_2	X_5	Y_3	Y_4
X_1	0	link							
X_2		1		link			link		
Y_1	start match leak	yx-end	0			link		link	
X_3				0	link				
X_4					1				
Y_2					yx-end	0			
X_5							1		
Y_3		match						1	link
Y_4							yx-end		1

Figure 4

Imagine that the following sequence of commands is executed.

$START_1[X_1, X_2, Y_1]$

$GROW_2[Y_1, X_2, X_3, X_4, Y_2]$

$GROW_2[Y_1, X_2, X_5, Y_3, Y_4]$

$MATCH_3[Y_4, X_4, Y_3, X_2]$

$MATCH_4[Y_3, X_3, Y_1, X_1]$

$LEAK[Y_1, X_1]$

Figure 4 displays the matrix after this sequence has been executed.

We attempt to match corresponding x and y sequences by working from the bottom of the tree to the top. This seems easier than working down from the root, since there is a unique chain of links to follow from any node to the root in each tree, whereas working down from the root, it is not clear how to arrange to follow corresponding paths through the two trees. The $START_i$ and $GROW_i$ commands start matching two corresponding sequences by matching their last symbols. The $MATCH_i$ commands then compare the two predecessors (i.e., ancestors in the tree) of any pair of matched nodes.

The leak right can be entered if and only if matching proceeds all the way up to the root nodes. Next,

---

Higher-Level Language-Based Editors

Robert A. Ballance
(Professor S. L. Graham)

(DARPA) N00039-84-C-0089

Language-based editors support the programmer by using language-specific information during the editing process. This support includes checking for syntactic or semantic errors, template-based entry of basic structures, and special display algorithms for viewing the program. To date, most systems impose a rigid development methodology on the user.

I am interested in developing "higher-level" language-based editors that support multiple languages and allow users to manipulate programs in terms of the underlying language. This approach subsumes both text- and structure-based editing. Programs and structured text can be manipulated either as text or in terms of their underlying structures. For example, in a program, the user might choose to operate on functions, blocks, statements, expressions, tokens, or characters. At all times, the editor will offer full flexibility between text and structure.

This research is aimed at creating an editor-generating system that accepts a language description as input, creating tables and code for use in a standard front end. The standard front end

provides a consistent user interface for editing objects written in different languages. Areas of investigation include algorithms for parsing, static-semantic description and checking, and connections to knowledge-based programming environments.

Experimental Design of a High-Performance, Object-Oriented Personal Computing System

David Ungar
(Professor D. A. Patterson)

(DARPA) N00039-83-K-0107
and IBM Corporation

A new class of programming systems is evolving that integrates a processor, a high-quality display, a programming language, and an operating system with the goal of enhancing programmer productivity. These systems allow the creation of software prototypes using considerably less manpower. Smalltalk-80 is the most mature example of an integrated software system.

The primary disadvantage of such systems is their slowness. We are in the midst of a three-year project to apply compiler, systems, architectural, and MOS VLSI implementation expertise to building a low-cost version of such a system [1]. We have built a software implementation under UNIX called Berkeley Smalltalk (BS) and have distributed it to twenty sites. Although written in a high-level language (C) and running on a microprocessor (SUN workstation), BS is as fast as a microcoded version of the Smalltalk-80 system run on the Xerox Dolphin. Our long-term goal is to create a new hardware/software system — SOAR (Smalltalk On A RISC) — that runs a hundred times faster than Smalltalk on the VAX 11/750.

Garbage collection presents a serious challenge for a Smalltalk-80 system. Smalltalk programs create seven bytes of garbage for every eight instructions executed. We have designed an algorithm called Generation Scavenging and incorporated it into BS [2]. Pauses disrupt thought and decrease productivity. The pause time for our algorithm is only a fraction of a second. All other Smalltalk-80 systems need indirection to help manage objects. BS, with Generation Scavenging, is the first one with direct object addressing. Our garbage collector also runs in half the time of the best previous algorithm.

[1] D. Ungar, R. Blau, P. Foley, D. Samples, and D. A. Patterson, "Architecture of SOAR: Smalltalk on a RISC," *11th Annual Symp. on Computer Architecture*, Ann Arbor, MI, 1984.

[2] D. Ungar, "Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm," *ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments*, Pittsburgh, PA, April 1984.

A Smalltalk Compiler for SOAR

William Bush
(Professors P. N. Hilfinger and D. A. Patterson)

(DARPA) N00039-83-C-0107

A necessary part of the SOAR (Smalltalk On A RISC) project involved constructing a compiler that would compile Smalltalk programs into SOAR machine instructions instead of the standard Smalltalk virtual machine bytecodes. The primary concern in designing this compiler was the code explosion that could result from using SOAR instructions instead of the much denser bytecodes.

That the semantics of Smalltalk preclude many standard optimizations made the Smalltalk bytecodes themselves an attractive intermediate representation for the compiler. The runtime stack used by the stack-oriented bytecodes is simulated by the compiler at compile time, which converts those operations to register-oriented SOAR instructions. This in practice produces reasonably dense SOAR code, with an average of one SOAR instruction generated for each bytecode. The compiler was written in Smalltalk, and the Smalltalk environment, although sophisticated, requires substantially more resources (primarily in terms of CPU power) than a conventional one in order to provide equivalent programming throughput.

A Debugger for SOAR (Smalltalk On A RISC)

Peter K. Lee
(Professors P. N. Hilfinger and D. A. Patterson)

(DARPA) N00039-83-C-0107

Smalltalk On A RISC (SOAR) is a microprocessor designed to run Smalltalk efficiently. Smalltalk is defined on the Smalltalk Virtual Machine (STVM), and the compiler generates virtual machine instructions, known as *bytecodes*, from Smalltalk methods. On SOAR, bytecode methods are further translated into SOAR machine instructions to be executed by the hardware.

The debugger in Xerox's Smalltalk Virtual Image operates by simulating the semantics of the bytecode instructions, which are no longer available on SOAR. There are also problems with the breakpoint-setting mechanism in the bytecode debugger. Breakpoints are set by inserting "halt" instructions in the instruction stream and recompiling the procedure. This makes setting breakpoints in recursive routines impossible.

The following are guidelines for the design of the SOAR Debugger:

- To provide the basic mechanisms for performing the same functions as the bytecode debugger so that high-level software, like the Debugger Browser, can be reused with minimal change
- To provide a better breakpoint mechanism so that breakpoints can be set and unset without recompiling the procedure
- To allow breakpoints to be taken conditionally, thus allowing breakpoints to be set in recursive routines or routines that are shared with the debugger itself.

Smalltalk On A RISC (SOAR)

A. Dain Samples
(Professors P. N. Hilfinger and D. A. Patterson)

(DARPA) N00039-83-C-0107

Professor Patterson has set out to design a RISC (Reduced Instruction Set Computer) microprocessor that will provide a fast execution vehicle for the Smalltalk-80 programming environment. The Daedalus project under Professor Hilfinger has provided an intermediate step toward implementing Smalltalk on this processor (SOAR). A basic version of Daedalus that allows testing of benchmarks and large portions of the Smalltalk-80 system is now running. Most of the Smalltalk-80 runtime system is written in the Smalltalk language itself; this portion has been translated into SOAR machine code.

We are now in the final stages of designing the RISC/UNIX interface required to run Smalltalk on the hardware. When the chip is fabricated, we will have hardware and software ready to execute Smalltalk using SUN workstations to handle files and graphics. My current tasks are to finish the operating environment specifications and to document the runtime system and virtual machine.

Implementing an Efficient Runtime Organization for Ada

Benjamin G. Zorn
(Professor P. N. Hilfinger)

NSF Graduate Fellowship and
(DARPA) N00039-84-C-0235
and N00039-84-C-0089

Research supervised by Paul Hilfinger was aimed at designing and implementing an efficient and conceptually simple runtime strategy for Ada. The implementation involved using an Ada front end (provided by AT&T Bell Laboratories and modified at Berkeley) that produces DIANA, a proposed standard intermediate representation (IR) for Ada programs. Our so-called "middle end" takes the DIANA representation and produces the lower-level intermediate tree form used by the portable C compiler.

Experiences with the DIANA representation have shown that a normalization pass is necessary to make the representation usable by the middle end. Although the IR of the portable C compiler was not intended as a low-level representation for Ada, we found that it was almost entirely adequate for the task. Furthermore, using the IR allowed us to take advantage of table-driven code-generation tools being researched at Berkeley.

The runtime design stressed efficient application of uniform runtime type representations to implement such features as dynamic arrays and parameterized records. Furthermore, the implementation of these objects did not introduce any distributed overhead on the implementation of objects familiar to the Pascal user. Execution time for a Pascal subset of Ada was comparable to the execution time of code generated by the Berkeley Pascal compiler, as demonstrated by a small set of benchmarks.

Register Allocation and Data Conversion in Machine-Independent Code Generators

Marshall Kirk McKusick
(Professor S. L. Graham)

(DARPA) N00039-82-C-0235
and N00039-84-C-0089

The final goal of our work is to produce a set of tools that will allow the construction of high-quality code generators for Von Neumann-architecture computers in a short time and with a minimum of machine-specific coding.

We have already developed a table-driven code generator using the Graham-Glanville method. We are working on developing formal methods to attack the problems that it fails to address. The result will be a largely table-driven code generator that is retargetable to diverse architectures with a minimum of recoding.

One major goal of this research is to formalize the register allocation of the code generator. We are investigating how to coalesce the optimizations found by a procedurewide data-flow analyzer with the allocation of registers needed during the process of instruction selection. The allocators are driven by a description of the number and types of various registers and some policy description. We are evaluating various coloring techniques as the basis of the allocation policy.

Another major goal of this research is to investigate semantic alternatives to the current syntactic specification of conversions done by the code generator. We are experimenting with different specifications to measure their costs in terms of size, space, and comprehensibility.

Automated Discovery of Machine-Specific Code Improvements

Peter B. Kessler
(Professor S. L. Graham)

(DARPA) N00039-82-C-0235
and N00039-84-C-0089

I am investigating techniques to automate the discovery of machine-specific transformations that will improve the quality of code produced by retargetable compilers. Current retargetable code generators produce provably correct, often optimal code for single statements. However, they fail to take full advantage of target architectures and often use complex instructions only with hand-coding to recognize special cases. The goal of this project is to recognize those special cases automatically, thus making retargetable code generators more robust and easier to retarget.

I distinguish two stages in this process. The first analyzes a description of the target machine when the compiler is constructed and generates tables for the second stage, which transforms each program run through the compiler. This separation allows the analysis of the target machine to be arbitrarily thorough in its attempts to exploit features of the target machine and reduces transformation to a simple pattern match and replacement. The analysis "decomposes" the complex instructions of the target machine, finding sequences of instructions that can be replaced by those complex instructions. The transformation stage uses information derived by the analysis to transform assembler source code. Automating transformation of other representations of programs is possible. A prototype system has been demonstrated and retargeted, and a dissertation describing this work is in progress.

Tree Transformation Systems in Compilers

Eduardo Pelegri-Llopart
(Professor S. L. Graham)

(DARPA) N00039-84-C-0089
and Venezuelan Research
Council (CONICIT)

Trees are convenient representations in many situations largely because of their hierarchical structure, which models many situations, and because of the ease with which they can be manipulated. This manipulation frequently corresponds to transformations between different tree representations to expose or modify some properties of the object being represented.

The goal of our project is to investigate tree transformations, especially in the context of compilation systems, program transformation systems, and programming environments. We will use results and experiences from the areas of term rewriting systems, production systems, and programming languages, among others, to find descriptional mechanisms that are adequate for efficient implementation while also being easy to program, have provable properties, and have an adequate interface to the compiling process, particularly to pattern-matcher-based code generators.

Our approach is quite pragmatic, and we hope to design a tree transformation tool in the tradition of scanner, parser, and code-generator generators. As an application of our research, we expect to fill the gap between the high-level abstract trees that can be obtained from the parser and the low-level trees required by the Graham/Glanville/Henry technique of code generators. In the process, we expect to clarify the relation between program optimization and code generation.

We have implemented a simple tree transformation system, which we have used to gather some first experience. We recently completed an extensive bibliography revision and are evaluating different description mechanisms.

Automatic Generation of Symbol Table Managers from Specifications

Phillip E. Garrison
(Professor S. L. Graham)

(DARPA) N00039-82-C-0235
and N00039-84-C-0089

Symbol Table Managers (STMs) must currently be hand-coded, though some efforts are under way to automate their production. The task of writing an STM can be very difficult for languages such as Ada. Automatic production of an STM from a specification of the relevant parts of the language being processed would have a number of benefits: less work, greater understandability and modifiability, improved faith in the correctness of the generated STM, and automatic optimization. Such a specification would also be useful for formal language definition and for language comparison.

A model of scoping and naming in languages has been developed, and the primitive operations implied by this model have been identified. A functional specification language based on these primitives has been designed as an extension of an attribute grammar system. The extensions will be implemented so that the usefulness of this specification language can be evaluated on real languages.

Because symbol tables are large objects, copying them unnecessarily must be avoided if reasonable efficiency is to be achieved. Elimination of simple copies is well understood, but

elimination of copies necessitated by modification of objects has been studied only in certain cases (e.g., pass-oriented attribute grammars). Two different methods that will handle ordered attribute grammars have been developed.

The Transition Diagram Editor

Charles C. Mills
(Professor A. I. Wasserman*)

(DARPA) N00039-82-C-0235

This project created a highly interactive mouse-and-menu-driven graphical editor for state transition diagrams, along with a generator that provides input to a transition diagram interpreter. The Transition Diagram Editor (TDE) is used to support the User Software Engineering (USE) methodology, an approach to the specification and implementation of interactive information systems. Augmented state transition diagrams are used to model human-computer interaction, with nodes representing system output and arcs (transitions) associated with user input; system operations occur during a transition.

Rather than using a textual language to describe the diagram structure, TDE allows the diagram to be drawn and edited interactively, then generates the textual diagram description for interpretive execution. TDE and its related tools are especially effective for rapid prototyping of interactive systems. TDE runs on the SUN workstation; the other tools run not only on the SUN, but on most other UNIX systems.

*U.C. San Francisco

Evolution Support Environment (ESE)

Kozo Bannai, Atul Prakash, Jaideep Srivastava, Wei-Tek Tsai, and Yutaka Usuda
(Professor C. V. Ramamoorthy)

(BMDSC) DASG60-81-C-0025

Evolution Support Environment (ESE) is an integrated and automatic environment for the software development/evolution process. An ultimate goal for the software development process is to develop an automatic software family generator that, given the specifications for a member of a family, could generate an implementation for the member by reusing as much existing software as possible. Three basic requirements for such an environment include promoting (1) traceability between user requirements, design, and code, (2) reusability of existing designs and code, and (3) compatibility between various phases of the software life cycle. Our next step is to design and develop the Software Engineer Assist System, which would guide the designer's decisions based on metrics (we call that Metric-Guided Design Methodology).

CoLab — Tools for Computer Collaboration

Gregg Foster
(Professor R. J. Fateman)

(DOE) DEAMO3-76500034
and Xerox Corporation

CoLab is a laboratory to experiment with new forms of computer-assisted collaboration. Although networks connect computers and enable electronic mail and sharing of facilities, computer systems aren't usually designed for group activities. When we think of people working with computers, we usually think of them in separate offices working mostly in isolation. To use computers for demonstrations, several people gather around a display designed for a single person. If people decide to work together on a problem, they leave their computers behind and go to a whiteboard. Secondary ideas, arguments,

and random notes are often lost or forgotten when records of a collaborative session must be entered into a computer system as a separate step. When instructors train people to use interactive programs, there is no easy way to interact with several students at once.

Recent technological advances (e.g. Ethernet, the use of the mouse, EvalServer [evaluation of LISP s-exps on remote machines], and bitmap displays) have made possible new software tools and new classes of tools, including tools for group activity mediation and enhancement. There has been little previous study of personal vs. group use of computers.

We are designing software tools for intellectual teamwork (enlightened group problem solving). The software design of these tools addresses the synchronization of shared objects and data space, the network coordination of closely interacting machines, and forms of new primitives for active and interactive displays.

The goals of the CoLab project are as follows:

- To explore the dynamics of group problem solving and interaction
- To explore existing communication devices and paradigms used for collaboration
- To experiment with software, hardware, and social techniques to assist group problem solving
- To build the software foundations and several tools for these experiments and explorations
- To analyze real group use of our system and tools.

CoLab is expected to be an environment in which computers unobtrusively support human interactions, not one in which humans only use computers.

FINAL REPORT

THE PORTLAND EXPERIMENT IN SCL

XEROX PALO ALTO RESEACH CENTER

Margrethe H. Olson
Associate Professor
New York University

August 30, 1988

INTRODUCTION

This report summarizes my observations of Systems Concepts Laboratory (SCL) and its "remote work site" experiment (to be referred to here as the Portland Experiment) from October 1985 until January 1988. The primary purpose of this report is to document and interpret what was learned from the Portland Experiment during this time.

Background

In the spring of 1985 two significant events occurred: SCL became a full-fledged laboratory and the Portland site began operation. The details of the establishment of the Portland site will not be reviewed here. During the next six months, SCL grew from twelve to eighteen members, with four of the six new members based in Portland. Only one member of the lab transferred from Palo Alto to Portland.

The espoused "vision" of the laboratory under which the Portland experiment played a key role was originally described by Adele Goldberg, SCL lab manager until September 1986, as interpersonal computing. Its roots were in the previous focus, of the group which was the predecessor to SCL, on personal computing. While personal computing supports individuals and involves their interaction with a computer, it does not support person-to-person interaction or work group collaboration. The notion of interpersonal computing is that it supports people communicating and working together through computers. Thus it would include tools to support face-to-face interaction and meetings as well as interaction separated by time and/or space. One lab member articulated it this way: "There is a vision of an environment in which it is easy to work with anyone you want to in space and time. That requires the ability to interact, to get in touch, to share resources."

The evaluation project began with a pilot consisting of two

visits, in October 1985 and December 1985. The project was subsequently funded for one year and then extended for a second year. The researcher made eight more visits over the two year period. Each visit consisted of two full days at each site, interviewing (on an individual basis) as many of the lab members as possible. All interviews were open-ended and unstructured. The lab members discussed a wide range of issues, many of which were not directly related to the Portland experiment. However, through these interviews the researcher was able to learn about the overall social process and culture of the lab as well as specific projects; these issues inform in an important way the evaluation of the Portland experiment.

The researcher produced an interim report in November 1986 and a final report, including recommendations for further research, in November 1987. The Portland site was closed three months later. The purpose of this report is to reexamine the Portland Experiment in light of the fact that it is now complete, and the body of knowledge it generated can be summarized. This report does not address the rationale for either opening or closing the Portland site, nor does it deal specifically with whether the Portland Experiment was a "success" or a "failure".

The Central Research Question

The Portland Experiment was designed to be a forcing function for the lab to focus on the issues of collaboration in a geographically distributed organization. The central research question is thus two-fold. What did the Portland Experiment teach PARC in terms of:

* the process of collaboration in a distributed organization;
* the definition and implementation of tools to support collaboration in a distributed organization.

Outline of the Report

The next section of the report briefly describes the environment and technology of the Portland experiment. Then, some background to understanding the research contribution, in terms of work group collaboration, socialization, management control, and physical place versus "social place", is provided. The research contributions of the lab during this period are then reviewed and analyzed in view of the central research question defined above. Finally, some conclusions about the contribution of the Portland Experiment to PARC research are given.

For readers who are unfamiliar with SCL or the Portland experiment, a discussion of the evolution of lab process and

culture in SCL and the two sites over the two year period is contained in Appendix A.

THE PORTLAND SITE AND TECHNOLOGICAL ENVIRONMENT

The dominant espoused research goal of the Portland experiment was to be a forcing function for the lab to develop tools to support collaboration in a distributed environment. The fact of the Portland site created two types of barriers to communication and collaboration: geographical and cultural. The assumption was that these barriers would have to be overcome, forcing the lab to develop tools to help overcome them. For this reason, a second site geographically closer to PARC, where the barriers could be overcome easily (i.e., by travel between sites) was ruled out.

The Portland site was designed to support up to eight full-time researchers plus one technical and one administrative support person. For most of the two year period of the evaluation, there were seven full-time researchers, two support people, and at various times a consultant and a graduate (summer) student. The facility was designed to be similar to the lab in Palo Alto, with a large "commons" area for meetings and informal interaction, individual offices on the periphery, and a conference room for more formal or private meetings.

The key technology providing a "link" between the two sites was an open communication channel to support (at 56kb) interactive video and audio at all times. It became commonly known as "the link" or "the Widcom", referring to the first system installed for the video connection. Originally, the video (cameras and monitors) and audio equipment was installed in the commons areas at each site, and used for informal interaction as well as group meetings. Later, experiments were done with moving all or part of the equipment into individual offices for private meetings. Eventually, video switches were installed at each site and many of the offices were equipped with monitors and cameras. At each site the equipment was linked through the video switch and also linked to the computing environment, so that a person in an office could establish a video link with another office at the same site or, through the single channel between Portland and Palo Alto, to an office at the other site. This environment, with assorted other features, became collectively known as the "media space".

BACKGROUND TO UNDERSTANDING THE RESEARCH CONTRIBUTION

In this section, some research areas that help put in perspective the Portland Experiment are reviewed.

The Nature of Work Group Collaboration

Recently, computer science and information systems researchers have begun to pay attention to technological support for work group collaboration (Greif, 1988; Olson, 1988). In the academic fields of management as well as social psychology and sociology, there is a considerable body of research on the nature of work groups. While research in social psychology has tended to focus on group process, particularly in decision making (e.g., Kelley & Thibault, 1968), research in sociology has focused more on official work units (i.e., departments) and their interdependence in terms of organizational structure (e.g., Pugh, et al; Hage and Aiken). Research in management theory has focused recently on "teamwork" and participative management (e.g., Tjosvold, 1986). Most of this research has attempted to demonstrate the benefits of teamwork in terms of employee motivation and productivity. Very little research in any field has specifically focused on how work is performed in and managed by groups. Computer scientists interested in building tools to support groups at work have not found a useful framework in any of this research to define what happens in work groups and how information technology might improve work group process and output.

The computer science community addressing this issue has come to be known as the CSCW community, after the first conference on Computer Supported Cooperative Work in December 1986. There has been much debate about the name; some object to the term "cooperative" since work groups are not necessarily so and might be in direct conflict. The author prefers the term "collaboration"; according to Webster's Dictionary, to "collaborate" means "to work together, particularly in an intellectual effort."

The type of work group collaboration dealt with in this report is strictly of the "intellectual" variety, as opposed to work groups assigned to assembly, manufacture, or construction of physical artifacts. This type of collaboration has several distinguishing features:

* There is at least one common goal shared by all group members, although subordinate goals may not be shared by all members or may even be in conflict;

* The primary "resource" required to carry out the activity is information or ideas; thus there must be some facility for work group members to share information;

* The coordination of effort required to accomplish the task primarily requires knowing what other work group members are thinking and/or doing;

* Work group members are thus interdependent in the long run, although for periods of time they may be able to work independently.

The nature of interdependence of collaborative work groups is best defined by Thompson (1967). Thompson defines three types of interdependence of resources in any work process:

* Sequential interdependence, where resources are consumed sequentially, as in an assembly line;

* Pooled interdependence, where the resource to be consumed may be accessed simultaneaously by multiple facilities requiring it, as in access to a centralized database for an airline reservations system;

* Reciprocal interdependence, where each of two facilities also has resources required by the other, and coordination of the two is required.

It should be clear that when work is organized with reciprocal interdependence, more resources are required for coordination of the work than for the other two types of interdependence. In intellectual work, where the primary resource to be coordinated is information, reciprocal interdependence in a work group implies that each facility (in this case, people) needs to know what the other members of the work group are doing and/or thinking. This knowledge must span time (how did they arrive at this solution? What did the group do yesterday when I was out of the office?) and space (What are the other members doing now? Is it necessary for us to meet face to face to resolve this disagreement?).

Alternative Goals for Development of New Tools

Different tools have different effects on work group process. Some alternative goals leading to development of different tools are examined here.

1. To make two separate physical environments "more like" a single environment. This has traditionally been the goal of video teleconferencing: to be as much as possible like a face-to-face meeting.

2. To improve the accessibility of more information. This implies moving from reciprocal interdependence to pooled for at least information that can be "shared". Videotape recordings that provide a "sense of the past" would be an

example, as well as image processing. The notion of an object service as an underlying technology supports this design goal.

3. To improve the efficiency of reciprocal interdependence. For a task where reciprocal interdependence is necessary, generally a great deal of time is spent "informing" other members of the work group. Tools that focus on improving the efficiency of this process might reduce the amount of information required to be shared by increasing specificity. A common example today is the substitution of electronic mail for telephone because it eliminates unnecessary "social" conversation. An example of a new tool is an electronic mail system which imposes a structure on the dialog, such as the Coordinator by Action Technologies Corporation.

4. To increase the capacity of reciprocal interdependence. Tools of this type increase the amount of information sharing among work group members, either in order to overcome barriers of space and time (e.g., interactive video in offices) or to make face-to-face interaction more effective (e.g., meeting augmentation, group decision support systems).

The Nature of Socialization

Another aspect of work group collaboration which has been neglected in research to date is the process by which work group members learn and act out their roles. This has to do with the nature of contracts: who determines who should do what and how is commitment from work group members elicited? There may be different models, from highly authoritarian (the manager dictates task assignment and demands commitment) to highly participative (all work group members negotiate together and agree on tasks). Prior to task assignment, understanding of the expertise and knowledge each work group member brings to the project is a more subtle aspect of work group process that is particularly important if the work group wants to be cooperative and foster trust among members. For instance, if a person takes on a particular task voluntarily, the other group members should have some a priori belief that the person is competent to do the task and can be trusted to deliver as promised.

The primary type of organization of which SCL is a prototypical example is defined by Henry Mintzberg (1979) as an adhocracy. The dominant form of coordination of work in an adhocracy is mutual adjustment, which refers to "coordination of work by the simple process of informal communication". Furthermore, the adhocracy is fairly flat, with few layers of management. Roles and organizational responsibilities are fairly loosely defined and highly

ambiguous, with individuals given a considerable amount of leeway to choose how to prioritize their time. The process of adjusting to an organization of this sort involves learning what are the appropriate "projects" to work on without any explicit direction offered, and establishing and/or demonstrating competence and trustworthiness in order that other work group members seek out the new member for projects.

The Nature of Management Control

The final aspect of learning and doing roles in the work group process is the nature of control. With certain types of tasks, milestones and deliverables may be highly specific and measurable so that individual performance can be easily determined. In many work groups, the only real control by either management or other work group members is pure observation ("He is never in his office. No wonder he isn't going to meet the deadline -- he is never working.").

An adhocracy tends to support an egalitarian, frequently participative, management style. Thus the organization members themselves may set policy and direction. In many such organizations, the implicit culture strongly indicates what is acceptable behavior (including such mundane things as dress, punctuality, etc.) without any explicit rules or policies. Individuals are "expected to figure it out", and those who do not are either misfits (working on the "wrong" things) or feel uncomfortable in the environment and choose to leave.

In a work group where group members themselves determine task assignments and roles, it is more likely that at least some control is held by group members themselves. Thus if one group member is "slacking off", the most effective control process might be peer pressure.

Physical Place versus Social Place

Many of the phenomena regarding work group process and socialization are, at least traditionally, highly dependent on physical place and physical (i.e., face-to-face) interaction. Indeed, virtually all of the research reviewed above assumes that a work group is colocated. In the literature on socialization, strong emphasis is placed on observation. Anything which is not face to face is "less than" and therefore necessary but not as good.

Most representative of this point of view is the considerable body of research on teleconferencing (Johansen et al, Short et al), which emphasizes specifically how it is less than face to face. For instance, Short and his colleagues operationalize the notion of "social presence" of a media and measure it relative to face-to-face interaction.

In the socialization process, physical place plays a much more subtle but possibly more important role. People learn how to act in organizations by watching other people. The "culture" of the organization is reflected in its physical environment (Deal and Kennedy). Roles and status are reflected in a very formal way in the size of an office, the number of windows, the type of desk (in one organization, the "wooden desk people" are the only ones who make decisions), even the color of the carpet. On a more subtle note, who talks to whom in the elevator, who goes to lunch together, etc. are all cues that are carefully observed by other organization members.

Furthermore, the physical place of the organization exerts a direct, if not very efficient, form of control over individuals. In essence, when an employee enters the facility, his or her time is "owned" by the organization. Even if employees are not being very productive, their "time in" is the basic metric on which their performance is determined. This fact was brought home to the author in extensive research on "telework", where employees worked at home instead of going to the office. The primary obstacle to telework as an employee work option was management's discomfort with not being able to "see" that their employees were working, and furthermore that the employee was in an environment, the home, which was explicitly outside of the organization's control (Olson, 1987).

These notions of physical place are challenged in a very important book (Meyrowitz, 1985). Meyrowitz argues that "social place" is becoming a dominant factor in society today. If we examine the role of electronic media (particularly television) in our understanding of the world around us, it is apparent that assuming that physical and social place are equivalent is inadequate. In the case of work groups embedded in an organizational culture, this means that the traditional research and practice assumptions of physical place (epitomized by the face-to-face meeting) are inadequate to understand the impact of electronic media. In essence, electronic media present a new set of roles and meanings that "undermine the traditional relationship between physical setting and social situation" (p. 7).

THE CULTURE OF SCL AND ITS WORK

In this section, we will examine the work group and socialization culture of SCL in light of the research discussed above. The primary purpose of this section is to define the SCL culture relative to other types of organizations, so that the research findings can be placed in the proper context.

The Social Organization

SCL's management structure, while often debated, was relatively participatory. There were two levels of management (area managers and lab manager), who took seriously the responsibility to handle administrative matters so that their subordinates could concentrate on research. There are several important characteristics:

* There were no explicit task assignments. Lab members decided themselves what to work on, and thus new members had to figure out what was appropriate by learning what others did.

* Being a competent researcher was highly valued and rewarded but the standards of competence were not well defined. While one way of demonstrating competence was to build an artifact and the emphasis was on this rather than producing papers, once someone had gained respect as a competent researcher they had more latitude to do what they wanted. This made "learning what to work on" by new members even more difficult.

* There were only highly subjective measures of performance. Programming is an excellent example; it is very difficult to detect programming performance and programming output can vary by orders of magnitude. A few members of the lab were remarkably proficient at programming but visually they looked like they worked just as hard as some other members who were "slacking off" on programming tasks.

The Nature of Lab Research Work

What was the nature of the work actually performed? Although some members worked alone, the common mode was to work together, and the common thread across many projects was collaboration in design. (See next section for descriptions of specific projects.) While the Design Methodology group worked on very different things than, for instance, the group developing Amber, they were both nevertheless doing collaborative design. Several characteristics of the design process in SCL generally hold across projects:

* The design process is "research" rather than engineering. The artifacts to be produced are generally evolutionary prototypes and vehicles for exploring possibilities rather than finished products.

* The mode of operation is reciprocal interdependence, with the primary resource to be shared being information or ideas. In order to work together, everyone needs to know at all times what everyone else in the work group is doing and how they got where they did.

Furthermore, the nature of the collaborative design process has the following characteristics:

* It is highly interactive, requiring dialog in real time;

* It generally requires a shared workspace as the focus of that interaction, to create a visual record or representation of the interaction;

* It often requires additional reference materials -- notes, documents, manuals, etc.;

* It requires some record of past interaction, even though it may be highly informal -- memory of work group members (e.g., Do you remember why we came up with that solution?).

Activities in Design

The term "design" in this context is used broadly to refer to an overall process which can be broken down into different observable activities. The activities defined below, and used subsequently to classify research projects, are based primarily on the discussions between the researcher and SCL members with respect to "What are you working on?".

Definition: There is an identifiable subactivity of the design process which generally involves defining and scoping the problem. Some projects (i.e., architecture) only engage in this activity; subsequent activities are performed by other parties. In a narrow definition of the term "design" this stage is the real "design process".

Implementation: This refers to actual production, i.e., of code, an artifact, an equipment installation. This activity generally has a tangible output.

Experiment: Some activities involved gathering data and analyzing it. This may be directly related to the design of something (e.g., statistics on volume of disk accessess to inform the design of the Object Service) or more indirect (e.g., the "Day in the Life" experiment).

The definition of projects was done by the author based entirely on interview notes, rather than project documentation or individual progress reports. In some cases a large project is broken down into smaller projects or stages; for instance, "straw proposals" and "core samples" are two sub-projects of the "new language" effort and are defined as two separate projects. Appendix B shows the list of projects, the number of people involved at each site, the classification of activity, and the output. The name of the project is often a combination of different names; the same project might be listed as different projects at different stages with different activities and outputs (e.g., "SCL villages" and "media space"). In total, seventy separate projects were identified.

Appendix C lists the lab members over the two-year period, and the number of projects on which they worked alone, in collaboration at a single site, and in collaboration across sites. Since being associated with a project has very little do with a person's relative contribution, and since the sizes of the projects varied greatly, this list in no way reflects individual members' contributions to the lab. It does show a few interesting things. First, virtually everyone in the lab worked on some projects collaboratively; two work alone predominantly but even they work with others on occasion. Second, all except one lab member and three contract employees worked on at least one project across site. Some, particularly, those in Portland, worked predominantly in collaboration across sites.

Table I shows the projects categorized by activity. It shows that projects (or project stages) focusing on definition are the most numerous, followed by programming. Although programming is predominantly done individually or at a single site, far more definition projects are done collaboratively and most of these involved both sites. Thus collaborative design (i.e., definition) across the two sites was a predominant activity in the lab over the two-year period.

Table II shows projects classified by output. It shows that the dominant output of cross-site collaboration is specifications (the primary output of definition). It is also worth noting that nearly half of the projects done collaboratively across sites (nine) resulted in no tangible output, far more than projects done collaboratively at a single site or individually. Having no tangible output does not necessarily mean the project failed. Many of these projects were high-level definitions (e.g., early Object Service, New Language) that produced documents reflecting members' thought process but no operational output in the sense of specifications or code.

AN INTERPRETATION OF THE RESEARCH CONTRIBUTION

In this section, the research contribution of SCL, in terms of articulating the collaborative design process and building tools to support it, is examined.

Articulating the Collaborative Design Process

The Design Methodology group articulated a vision early on which, although not explicitly shared widely with the rest of the group, implicitly fit the work of the entire group. The Design Methodology group tended to focus specifically on architectural design with a heavy visual orientation; as should be clear below, their notions of supporting design extend to other collaborative design activities, particularly systems design, as well.

As was noted in a previous section, SCL members view themselves as researchers rather than engineers. It is this underlying assumption about design which holds them together. In the words of one member, "There is a recursive process of evolving the system and allowing the system to support the process." This is the overriding model of design that was similar across architectural and system design.

Furthermore, this type of design is not highly structured, and does not benefit from tools to help structure the process. As defined above in terms of alternative goals, the tools required to support this type of collaborative design must increase the availability of information and increase the capacity for reciprocal interdependence (Goals 2 and 4), rather than simulating face-to-face (Goal 1) or improving the efficiency of the process (Goal 3) through structuring tools such as, in system design, CASE tools are purported to do.

One lab member described the research theme in the following way: "It is about the theory of the process of design, how people interact with design. To support the process of design, we need infrastructure tools -- video, computing, social process. You also need to be able to carry in your head what everyone else is doing. It is not about structuring, not hierarchical. It is about expanding the capacity of the infrastructure."

The Nature of Interaction in Design

Two dimensions of supporting this design process are thus: supporting communication in real time and recording the process over time as an artifact of the process. We shall call the first synchronous (or supporting interactivity) and the second asynchronous (or creating a record). A second dimension deals with focus of the interaction: we shall

distinguish open process from focused process. The four alternatives, with exemplary projects done within the lab, are summarized in Table III. Each alternative or quadrant may be thought of as a separate design environment for which support tools can be developed.

The activities of the group over the period (see Appendix B) can be classified into understanding and/or building tools for each of the four quadrants. It is important to note that they did not only focus on collaboration over a geographical barrier (i.e., remote collaboration), but also, and probably more important, on making the design process more effective even without that constraint.

The work in each quadrant is briefly discussed below.

Quadrant I: Open, Synchronous Activities

General problem definition takes place in this type of design environment. The primary product of this aspect of design is talk. Examples of this type of design process are the straw proposals and core samples of the New Language effort, as well as the pre-Amber object service work. The forums are frequently open meetings (with many "kibitzers") for "kicking around ideas", also informal spontaneaous meetings, and many informal "drop-in" conversations. They take up a significant amount of time with the only tangible output being an occasional document of user needs (as in the Object Service scenario paper) or "straw proposal". The primary purpose of these exercises is to provide clarity of ideas for the writer, not to inform the reader, although they are generally read. As the general design process progresses, the documents produced along the way become out of date.

One general problem with this aspect of design is keeping it sustained. According to one member involved with the new language project, "High-level goal issues have not stimulated alot of interaction. There is alot of interaction on fairly low-level language issues." According to another lab member, "Normally in the lab people spend thirty seconds thinking about what to do and almost no planning; then they just start coding."

In the lab, tools for "making connections lightweight", which were often labeled as supporting informal interaction, supported this type of collaboration. The primary "tool" for this support is of course the link between sites, that permitted both formal and informal interactions to occur. An artifact designed to support connection is CONTACT.

Quadrant II: Open, Asynchronous Activities

This aspect of design deals primarily with capturing the "thought processes" generated by the high-level design activity described above. In informal meetings or spontaneaous interaction, a formal record is rarely kept and the primary source of recall for later, more focused design, is human memory. Yet many new research ideas are generated in just such an "open", informal interaction.

The lab did no specific work on capturing informal interaction for later recall. However, the constant presence of the "media space" in offices was beginning to generate a sensitivity to the need to be able to record and recall this type of interaction.

Quadrant III: Focused, Synchronous Activities

A focused design process resulting in operational design specifications (written, graphic, verbal, etc.) takes place here. Support for this aspect of design requires intensive dialog (audio) support and a shared, focused workspace for drawing. Overwhelming, regardless of the type of design, lab members expressed this as their greatest need in cross-site design: the ability to have a shared workspace. In architectural design it is clear that drawing capability would be essential, but the same need seems to hold true for collaborative systems design as in the Amber project. According to one member, "This project pushes some things with respect to ambiguity in collaboration. When people collaborate they need to manipulate the things in their world."

The primary tool developed in the lab to support this aspect of design is media space. According to one lab member:

> "Two things came together to form media spaces -- the Portland link and the recognition that design requires support for not just goal-oriented activity but the process of design itself."

The Office/Design Experiment explicitly examined what happens if the environment is controlled in such a way that only the focussed interaction, without the potential distractions of a more open environment is available. As documented in [Stultz], the experiment revealed some important insights about the focused design process. A major theme which emerged was the relationship between working together and privately at the same time. The participants in the experiment acted as if they were working privately -- not taking breaks, not chatting, feeling compelled to work, and they were amazed at the volume of work produced. Yet it was indeed an intense collaboration

through shared dialog (as well as live video) and a shared video drawing space.

> "The tapes show the discovery of moving through the video space....One guy wanted to draw on the other person's screen, and he figured out a way to use tracing paper to essentially do that. It helped that we had very visually oriented people."

Another study that specifically focused on use of a shared workspace, comparing face-to-face, video and audio, and telephone only, is reported in (Bly, 1988). An interesting observation from this study is that "the process of creating drawings may be as important to the design process as the drawings themselves."

Quadrant IV: Focused, Asynchronous Activities

The focused design process frequently takes place over time rather than a single intensive session. Some problems requiring a record of the process are the need to review assumptions, and the need to bring a new person "up to speed" on a design. Early work in Design Methodology demonstrated how a rich record of the design process, primarily based on video recording and selective videodisk access, could be used to bring a new member into a design group in the middle of the process. A second major theme of the work/office experiment was recording the process so that it could be recreated selectively for the client (user) as well as the designers. The fundamental notion of needing a repository of "things" for sharing -- and thus the Object Service -- fits into this aspect of collaborative design. CORAL was a prototype for providing this sharing, as well as some simple videodisk server implementations and the "sense of the present" database prototype. In general, any applications which are built on the Object Service would support this aspect of design.

The Design Methodology group did not pursue this aspect of tool building as much as media space. It was a much greater technical problem that could not go far without an Object Service in place. It was also a difficult operational problem; without effective tools for indexing and selecting materials, a massive amount of material could be accumulated. In the office/design experiment, human labor was used to simulate this support, but this was not very feasible for more experiments.

One member of the lab articulated the relationship between recording the process over time and supporting the process across space: "We want to say that if you put communication media and ritual into place you can make it easier to participate in the present moment. The present moment has a past.....up to two weeks. There are fewer

interruptions....it is about getting rid of overheads that the group demand-driven activity requires."

In the Amber project, there were many complaints about misunderstandings among group members which were generally attributed to the two locations. According to one member, "We have discussions here and they have the same discussions there. Sometimes we make decisions about the same things they do and they conflict." In another case: "There is a small project to develop a database in Palo Alto. Independently I thought it was a good idea and wanted to do it. I didn't know they were already doing it." While it is easy to dismiss these problems as lack of face-to-face interaction, having adequate tools to record the process may have solved most of these kinds of problems.

Summary of Research Contribution

In general, the group started with a preoccupation with I, but steadily moved in the direction of supporting more focused collaborative design in III and IV. They did not tackle problems that were easier but less interesting and less critical to their work, such as programming together or enhanced electronic mail. They focused on the job that for them was both hardest and most rewarding: collaborative design. With media space they made significant progress in supporting focused, synchronous design and were moving toward real progress with support of asynchronous design. The underlying technology of the Object Service is a critical component primarily of Quandrant IV, but to some extent of all four quadrants. The group had also designed and was in the process of implementing GEAR, which provides the underlying technical infrastructure (equipment access) for tools to support all four quadrants. If the work had continued, once Object Service and Gear were in place the technical developments to support all four aspects of collaborative design would have taken off.

THE LAB EXPERIENCE

What did it "feel like" to work in this environment? Certainly the Portland Experiment was not a well-articulated, goal-oriented project. The experiences and frustrations of the lab members working in a distributed environment were probably a more important contribution (although more difficult to detect) than the activities or outputs of specific projects.

The Nature of Work Group Collaboration

It is clear that, in terms of the espoused research goal of understanding and supporting collaboration in a distributed organization, the lab became preoccupied with activities in the collaborative design process rather than other sorts of

projects. In doing so, they used the same model which was successfully with Smalltalk -- building something which they could use in their own work, which is of course, collaborative design. A significant number of prototypes (artifacts) were built during the period but few members focused on supporting collaboration in the building of a prototype (although there are some interesting examples). Although at first "programming together" was felt do be necessary and supportable across sites, the needs of the lab over the period did not push them in this direction either.

As shown in Table I, collaborative definition across sites was a dominant activity. Dealing with their frustrations in coping with the limitations of the "media space", the group was able to gain valuable insights related to this particular type of collaborative activity and the tools required to support it. The greatest frustration was expressed, for instance, in Amber, in the stage of definition; when they began to implement code, they were able to work effectively with much less communication between group members.

One example of the expression of frustration is the following, describing the definition stage of the Amber project:

> "Sometimes things were in such a state that the link worked well because we HAD to communicate....We agreed we wouldn't change it any more but it didn't work. There were alot more changes....It shoved you over the barrier to communications [i.e., the link]. The telephone became more important....It was good there was more than one person at each site. It provided local support for sanity testing. For some reason the more bandwidth the easier it is to do perspective shifting."

There are many examples that show the lab members' sensitivity to the process of design and the need to support that process rather than simply providing more information:

> "The other labs think of knowledge as a base you can draw on. I think knowledge is also a process....there is knowledge implicit in the process; e.g., knowledge engineering. The design process and knowledge of how different people approach the problem is part of the knowledge base. It is knowledge about process or procedure, not just about content."

> "This project [Amber] has no central management. That is not a problem. The whole group has to come to the revelation of a problem when there is one. It is up to the group to manage itself."

> "It is very nice to have them [the other group members] up there in a way. They don't come in and see where I'm at all the time."

The second goal of the research agenda was the definition and implementation of tools to support distributed collaboration. It is clear from Table IV as well as the list of projects in Appendix A that the lab made considerable progress toward this goal. The lab had a very clear understanding of the infrastructure required to support collaborative design and was progressing on that infrastructure with Gear and Amber. In support of specific focused design interaction (see Table III), there was also considerable progress with projects related to interaction in the media space (e.g., the Work/Office experiment, the "Janaia" study).

The Nature of Socialization

As described earlier, socialization is the process by which work group members learn and act out their roles. In SCL, this process was affected by the Portland Experiment in dramatic but subtle ways.

Many problems, such as those described in the Interim Report, were exacerbated if not caused by the two-site split. This is particularly an issue because so many of the Portland members were new to PARC, and their "cues" (i.e., what to work on, what is research versus "play", how much is acceptable, what are acceptable hours, etc.) were mostly provided in Palo Alto. As in a typical adhocracy (Mintzberg), there was no formal orientation; roles and norms, including even what to work on, were primarily learned by observation. As already discussed, the standards of competence were not well defined and the indicators of performance were highly subjective. These issues were all exacerbated by the distributed organization.

It would be presumptuous to point to any particular personnel problems and attribute their cause to the fact that the person was not in Palo Alto and thus was not properly "socialized" (although it is certainly tempting). However, this is a rich area for further investigation, particularly in the light of Meyrowitz's insights regarding "social place".

Meyrowitz emphasized the role of television in allowing formerly "private" spheres to become "public". By contrast, SCL was beginning to experiment with a "social place" extended by video and audio that changed the relationship between "private" and "public" workplaces. It was not at all unusual for a person's office to be "tuned in" to another office with video as well as audio and for this to be treated as unobtrusive background noise and not regarded

as an invasion of the other person's privacy. This type of extension of private workspaces offers whole new possibilities for patterns of socialization which remain unexplored.

Some of the experiences with using media space are particularly enlightening:

> "This is more like a window than a workstation or a microphone. It is more like an open office with shared acoustic space."
>
> "It doesn't intrude; it is there and you can pay attention or consider it background. Also you don't need to leave your workspace to interact with others in your group. You can choose when and how to participate."

Some of the most interesting insights in terms of social process had to do with defining the etiquette of the media space. Certainly the media space technology could be used for much more intensive but unobtrusive monitoring of workers. In SCL, the goal was to build media space so that it encouraged the status quo of shared control. For instance, a person should always be able to know if someone else was looking at them; such a feature was built into Contact. Another notion was that a person should always know what the other part is seeing; this leads to a scheme of relating a single camera and a single monitor.

The media space was not yet at the point where lab members could easily move in and out of each other's "spaces", particularly cross-site. It is possible, however, that had the experiment gone on, many more insights into the diffuse nature of socialization in "social places", and the design of tools to support them (etiquette of two-way interaction versus unobtrusive monitoring, etc.) would have been gained.

<u>The Nature of Management Control</u>

What can be generalized from SCL and the Portland experiment regarding the nature of control over work? Systems that are designed to improve the efficiency of coordination or to move from reciprocal to sequential or pooled interdependence are also implicitly about the control process. The work done in SCL is differentiated from these approaches specifically because it emphasized sharing control by increasing the capacity of reciprocal interdependence.

Cross-site reporting was often debated and the area managers, while admitting it made sense as part of the "experiment", generally felt uncomfortable with it. Since there were few specific objectives or deliverables, individual performance was often determined in a fairly

subjective ad hoc way. This is not unusual, but managers tended to discount their intuitions when the subordinate was not on-site. Managers in Palo Alto made frequent trips to Portland to "confirm a hunch" about a problem (e.g., a subordinate slacking off) and then to deal with the problem face-to-face.

Alternative Goals for Development of New Tools

In terms of the four goals of tool development, to which did SCL make a contribution? The group began by focusing on (1), making the two separate environments more like a single one. Many continued with a preoccupation that it was not "as good as" face-to-face and thus the split site was a frustrating obstacle. An example was in the definition process of Amber, where there were frequent expressions of frustration with the limitations of the media. In this case in particular, the group did not seize on th eopportunity to ask "What is the real problem?" and thus articulate the group's real need -- i.e., a shared work space. In other efforts, such as the Work/Office experiment and the "Janaia" study, the group did take advantage of the opportunity to learn what is different about collaboration in a media space rather than simply measuring it against the metric of face-to-face interaction.

Much of the ongoing work of the lab focused on (2), improving accessibility of more information which is needed for collaboration in design. The underlying infrastructure of Object Service and Gear supported this goal, as did the aspect of Design Methodology having to do with creating and accessing a record of the design process.

Some members of the group thought (in hindsight) that the direction they expected of the group was in support of (3). For instance, they might have developed the next-generation electronic mail system or group authoring system. Instead, the group turned to (4) and, in so doing, made a contribution which is unique to the CSCW community. It is significantly different from work on meeting augmentation, such as CoLab, because it opens up many possibilities for interaction across space and time while keeping the interaction at least as effective as, and possibly more effective than, face-to-face interaction with its geographical and timing limitations.

CONCLUSIONS

Table III and Appendix B demonstrate a significant body of work directly or indirectly related to the Portland Experiment. What does this mean relative to the stated goals of the experiment, as well as to the collective body of research known as CSCW?

The focus on collaborative design emerged rather than was defined a priori. It is clear that in fact, in the process of defining and tackling different problems in the design process, the lab was better able to articulate that process. More important, it was able to articulate the tools required, both for an underlying infrastructure (i.e., Amber, Gear), and for specific aspects of design (e.g., a shared workspace). It is interesting to speculate whether such progress could have been made on any individual project without the overall driving force of the Portland experiment.

There is a considerable amount of ongoing work on collaboration support that treats work groups generically. A major contribution of the SCL work is that it focuses on a specfic type of work group collaboration which evolved to be identified as the design process. Their work, however, can be generalized to other types of collaborative work with the following characteristics:

* Collaboration in an intellectual effort;
* The primary resource required is information;
* Reciprocal interdependence of group members for information;
* Control primarily through example and peer pressure.

The cumulative work of the two years of the Portland experiment is good work by itself. This report should clearly demonstrate that the nature of the Portland experiment did indeed act as a "forcing function" to produce this rich and well-differentiated body of work that will be a significant contribution to the body of research on technological support for work group collaboration.

REFERENCES

Bly, S.A., "A Use of Drawing Surfaces in Different Collaborative Settings," Proceedings, Conference on Computer Support for Cooperative Work, Portland Oregon, September 1988.

Deal, T.E., and Kennedy, A.A., Corporate Culture: The Rites and Rituals of Corporate Life, Reading, MA: Addison-Wesley, 1982.

Greif, I., Computer-Supported Cooperative Work: A Book of Readings, Boston: Morgan Kaufman, 1988.

Hage, J.T., and Aiken, M., "Routine Technology, Social Structure, and Organizational Goals," Administrative Science Quarterly 14(3):366-377, 1979.

Johansen, R., Vallee, J., and Vian, K., Electronic Meetings, Reading MA: Addison Wesley, 1979.

Kelley, H.H. and Thibault, J.W., "Group Problem Solving," in Lindzey, G. and Aronson, E., (Eds.), Handbook of Social Psychology, Reading, MA: Addison-Wesley, Volume 3: 1-105, 1968.

Meyrowitz, J., No Sense of Place: The Impact of Electronic Media on Social Behavior, New York: Oxford University Press, 1985.

Mintzberg, H., The Structuring of Organizations, Englewood Cliffs, NJ: Prentice-Hall, 1979.

Olson, M. H., An Investigation of the Impacts of Remote Work Environments and Supporting Technology, (NSF Grant No. IST-8312073, 9/83-8/85), Working Paper #161 (GBA #87-80), New York: Center for Research on Information Systems, New York University, August, 1987.

Olson, M.H. (Editor), Technological Support for Work Group Collaboration, Hillsdale, NJ: Lawrence Erlbaum, 1988.

Pugh, D.S., Hickson, D.J., and Hinings, C.R., "An Empirical Taxonomy of Work Organizations," Administrative Science Quarterly, 14: 115-126, 1969.

Short, J., Williams, E., and Christie, B., The Social Psychology of Telecommunications, New York: John Wiley and Sons, 1976.

Stefik, M., Foster, G., Bobrow, D., Kahn, K., Lanning, S., and Suchman, L., "Beyond the Chalkboard: Using Computers to Support Collaboration and Problem Solving in Meetings," Communications of the ACM, 30 (1), 32-47.

Stults, R., "The Office Design Project and Other Experimental Uses of Video to Support Design Activities," Xerox Corporation, 1988.

Thompson, J.D., Organizations in Action, New York: McGraw-Hill, 1967.

Tjosvold, D., Working Together to Get Things Done: Managing for Organizational Productivity, New York: Lexington Books, 1968.

APPENDIX A
EVOLUTION OF SCL PROOCESS AND CULTURE DURING THE PORTLAND EXPERIMENT

In this section, changes in process and culture of the laboratory, as they relate to the Portland Experiment, will be briefly reviewed. Four distinct stages of evolution of the relationship between the two sites since the establishment of the Portland site have been identified. Each stage is described below.

Stage I: Simulating Being There

In the early period after the Portland lab was established, a considerable amount of experimentation focused on simulating "being there". The link was used a great deal for meetings and there were a number of informal experiments designed to make the two sites feel more like one group. One, for instance, involved remote control of the camera. There were also a number of informal experiments with informal interaction and establishment of contact with people at the other site.

The transition to a laboratory and accelerated growth of the lab were felt in Palo Alto rather strongly, in that there was considerable discussion of the need for a "vision" to drive the lab's work. There was some concern about the myth perpetuated not only by some longer-term members of the lab, but by stories told by others even outside of PARC of the vision which drove the early development of Smalltalk. In Palo Alto, there were strong feelings of ambiguity and lack of community awareness of a shared vision.

In Portland, most of the members were new and knew less about the myth surrounding SCL than even the newer members in Palo Alto. The part of Palo Alto they experienced was primarily through informal interaction, constantly encouraged or championed by the Portland side, and the sense of ambiguity felt in Palo Alto did not come across.

In this stage, there were four area managers under the lab manager. Only one member of the lab located in Portland was a cross-site report to an area manager in Palo Alto, and that did not occur until six months after the Portland site was established.

In terms of work, the Collaborative Systems group was nominally established in Portland with no collaborators in Palo Alto. One project, remote control of the camera, involved one person from each site. The Object Service project had been defined at that point; there was work at both locations but it did not overlap, so that the need for communication on technical issues was low. Design

Methodology was a distinct group with all members in Palo Alto.

For most of the lab members, knowledge related to skills and technical competence was not generally transmitted across the link; it was not unusual for a person to comment that he or she had heard someone in the other site was doing similar work but had no idea what it was.

In summary, Stage I involved experimentation with using the single audio and video link to simulate being in the other place. Most of the cross-site activity was around informal (generally non-work-related) interaction or meetings. Beyond the norm of keeping others informed by electronic mail, little cross-site collaboration took place.

Stage I lasted from the time of establishment of a critical mass in Portland (early summer 1985) until early 1986.

Stage II: Separate Entities

The transition to the second stage took place with the Portland Pow-Wow in February 1986. The Pow-Wow brought all lab members together for two days in Portland. After the Pow-Wow, most lab members had a strong impression of the unity of the lab in terms of vision and consensus on research goals. The word they used to refer to this least common denominator of consensus was sharing.

After the Portland Pow-Wow, a number of lab members made attempts to establish cross-site collaborative relationships. The word "kibitzing" came up often, as in "I am kibitzing on the new language project". The New Language project began formally, and lab members at both sites attended meetings, some as kibitzers. The role of the Collaborative Systems group and/or the Portland lab members as users of the new language, who therefore should have significant input into its design, was identified.

With the establishment of the New Language project and the continuation of Object Service as a separate project, the lab settled down into project-oriented groups. The other two groups were Collaborative Systems, exclusively in Portland, and Design Methodology, exclusively in Palo Alto. The role of project leader, with technical responsibility for a project, emerged more-or-less officially. For a short time, there were four designated project leaders who met regularly (the "project managers' lunch") in that role. After the initial flurry of "kibitzing", attempts to make cross-site contacts trailed off. A second Portland member, newly hired, became a cross-site report to a Palo Alto area manager.

Over the link, there were fewer attempts to simulate "being there" in informal interaction. Members became accustomed to using the link and seeing themselves on video and began to rely on it for everyday use. With many meetings for both the New Language and Object Service projects, the lab began to use the link extensively for technical meetings.

In general, the lab settled into as close a routine as can be expected in a research organization. The attempts to push on the connection were reduced, and the lab began to operate more as two separate entities. With the decreased emphasis on being "together", Portland began to develop a somewhat separate and distinct culture. After a time, there was some strain between the two sites focused on serious lapses of communication and misunderstandings in projects. In particular, the Object Service project and the Collaborative Systems effort experienced minor crises based on miscommunication and misunderstandings across sites. Many of the problems were attributed to the lab manager's personal style and her inability to adapt it to remote supervision. These incidents are discussed in the interim report.

This stage lasted until approximately late summer 1986.

Stage III: Remote Management

The third stage of lab culture was precipitated by the change in management of both PARC and the laboratory in September 1986. The lab went through a period of major readjustment, which was experienced differently in the two sites. In particular, Palo Alto had the physical presence of the new entity, Parc Place Systems, for the next six months. Portland members expressed concerns about the relationship between SCL and PPS but did not feel it in terms of everyday presence. There were, of course, many misgivings about the new management structure and the lab's survivability under the new structure of PARC. With time, however, these fears subsided.

More important, with the reorganization almost all of the keepers of the "myth" of how SCL operates left PARC. Several other significant changes occurred. Much of the decision making on day-to-day administrative matters shifted to the lab and area managers with selective input from lab members. Most important for the Portland experiment, the reorganization provided the opportunity for the vision of interpersonal computing held by the former lab manager to be redefined. According to one lab member, "Can we take this as an opportunity to make Portland and Palo Alto come to a more positive set of working relations? We could not do that before because Adele identified the Portland / Palo Alto link as her personal research agenda."

Over the next few months, three more members changed management and reported cross-site. In the new organization, there were three area managers, all of whom had at least one cross-site report. There was an increase in research activity specifically addressing remote collaboration.

In general, this was a period of adjustment to management and articulation of new reseach agendas under a redefinition of the research vision. The period lasted until February or March 1987.

Stage IV: Consolidation and Focus

The lab then moved into a stage of consolidation and focus around key projects. The most significant change was the consolidation of the New Language and Object Service groups and the creation of the Amber project. This project had a core group of people with relatively well-defined work roles (i.e., no kibitzers). The project gained significant momentum quickly, with the successful delivery of a feasible design on April 15. Most important for this evaluation, the project was a true exercise in cross-site collaboration, with three members in Portland and two, later three, members in Palo Alto.

Other projects also gained momentum and independent focus. The Design Methodology group executed its Work/Office experiment in this stage. The Gear project was designed and began implementation. The Collaborative Systems group began to explore alternative methodologies, including involvement in an Interaction Analysis Laboratory with members of ISL and a Collaborative Readings group. One of the Palo Alto lab members became more active in these activities, signaling the first time the Collaborative Systems group had some cross-site collaboration.

This stage was winding down in September 1987. At that time it was not clear what direction the lab would take next; it was a particularly crucial time for the Collaborative Systems group. There was beginning to be a considerable amount of discussion of how to position and focus the Collaborative Systems research agenda, so that its third year would produce some more concrete results. At this time, I wrote an evaluation report which included recommendations for the Collaborative Systems research agenda.

Termination of the Portland Experiment

In December, 1987, PARC management announced that there would be a reorganization of the existing labs. In particular, SCL and ISL would be reorganized along some other project lines and probably be divided into three labs.

Proposals for reorganization were left to lab members, and this topic monopolized their time for the next two months. In early January 1988 management announced that the Portland facility would be closed. All the Portland employees were offered positions in Palo Alto, but, with the exception of the administrative support person, they all declined. The reorganization of the labs in Palo Alto was completed by the beginning of March.

APPENDIX B
SUMMARY OF SCL PROJECTS

PROJECT	PARTICIPANTS PA	PTLD	ACTIVITY	OUTPUT
Early Object Service	3	1	Design	----
Conversion of Smalltalk to Sun	2	3	Implement	Working System
Control of video from computing environment	2	0	Design	Prototypes
Recording design process	2	0	Experiment	?
Remote camera control	1	1	Implement	Prototype
Meeting on meetings	1	3	Talk	----
Weather map	2+	0	Implement	Prototype
Servo design	3	0	Definition	----
Stable storage (OS)	0	2	Program	Code
Audio solutions	1	1	Implement	----
CORAL	0	3	Define/Impl	Prototype
SCL Villages	2	1	Define	----
Planning the Pow-Wow	3	1	Define	Specs
Object Service Stage II	5	1	Define	----
New Language	2	0	Define	----
Illustrate design process	2	0	Document	Report (Video)
Video space	2	0	Implement	Prototype
Video server	2	0	Implement	Prototype
Media space	2	1	Define	Specs
Transparent forwarding (OS)	0	3	Program	Code
Straw proposals (New language)	5	1	Define	Specs
Media space implementation	4	0	Implement	Prototypes

ARK experiments	2	0	Experiment	Report
Scenario paper (OS)	4	0	Define	Specs
Core samples (New language)	3	1	Define	Specs
OS Requirements	1	2	Define	Specs
"A Day in the Life of SCL"	2	1	Experiment	----
Contact	0	3	Define/Impl	Prototype
Amber design	3	2	Define	Specs
Gear design	2	0	Define	Specs
Relational DBMS Impl.	2	0	Implement	Prototype
Shoptalk III	2	0	Document	Report (Video)
Office/design experiment	4 (+2)	0	Experiment	Report
Amber virtual machine	0	2	Program	Code
Amber compiler and cloner	0	2	Program	Code
Amber image	2	0	Program	Code
Amber simulations	2	0	Program	Code
OS "design"	1	1	Define	----
ARK extensions	1 +2	0	Define/Impl	Prototypes
Office/design documentation	4	0	Document	Report (Video)
ARK conversion	1	1	Program	Code
Audio solution work group	0	3	Talk	----
Portfolio of collab. studies	1	1	Define	Specs
Collaborative readings	3	4	Talk	----
Trans lan implementation	2	1	Implement	System
ARK beasts	1	2	Program	Code
"Janaia" study	1 (+1)	0	Experiment	Report

SINGLE PERSON PROJECTS

ARK	1		Define/Imp/ Experiment	Prototypes
Babar	1		Implement	System
Knowledge representation	1		Define	Specs
Viewers	1		Implement	System
Screen sharing		1	Program	Prototype
Opus		1	Define/Impl	Prototype
Digraph browser	1		Program	Code
Print spooler		1	Program	Prototype
Sense of the present database	1		Program	Prototype
Screen saver		1	Program	Prototype
MVC interface	1		Program	Code
Color coding algorithms	1		Program	Prototypes
Frame grabber	1		Implement	Prototype
ARK experiments	1		Experiment	Report
Shared ARK	1		Define	----
Chinese temple	1		Implement	Prototype
Videodisk interface	1		Program	Code
SOUP		1	Define	Specs
Tanga painting	1		Define	Specs
Gear implementation	1		Implement	Prototypes
Mail sorter		1	Program	Code
Videodisk data	1		Experiment	Report
Video switch	1		Implement	Prototype

APPENDIX C
LAB MEMBERS AND COLLABORATION

PERSON	SINGLE	COLLAB ONE SITE	COLLAB CROSS-SITE
PALO ALTO			
Bay			2
Bly	3	1	3
Deutsch	1		2
Flegle	1	3	4
Godreau	3	3	1
Harrison	1	10	2
Hibbert	1	3	4
Horton			1
Krasner			2
McCall		1	2
Minneman		2	
O'Shea		1	2
Putz	1		1
Ranjit	2	1	
Robson		1	2
Smith	3	2	3
Stultz		10	2
Trow			1
Weber		2	
Zybdel	1	2	5
PORTLAND			
Abel	1		4
Axel		1	
Ballard	3	2	3
Darlington		2	
Goodman		3	2
Larsen	2	3	2
McCullough		3	4
Merrow		2	4
Purdy		1	4

IDIOMATIC ILLUSTRATORS

William Bowman

Robert Flegal

November 1975

XEROX USER SCIENCES GROUP
PALO ALTO RESEARCH CENTER

INTRODUCTION

This report describes an effort to design and implement a set of computer-based graphic tools that enable people, unskilled in either Graphic Arts or Computer Science to easily illustrate technical ideas and information. The basic notion explored was: is it possible to break down the world of technical graphics into 'idioms' (constrained environments) such that the computer could provide both mechanical and aesthetic aid to the non-professional user.

In order to test this concept, we divided technical graphics into four basic environments:

1. **quantative**
2. **ideographic**
3. **isomorphic**
4. **volumetric.**

Each of these basic environments was then further subdivided into graphic 'idioms'. For example, *Piecharts* and *barcharts* are **quantative** idioms while *exploded views* and cutaways are examples of **volumetric** idioms.

From the wide spectrum of possible idioms we choose to examine three of them: a typographic idiom, block diagrams and piecharts. This report is primarily devoted to a description of the 'idiomatic' approach to computer graphics as we experienced it within the context of working with these three idioms.

1. IDIOMATIC ILLUSTRATORS

OBJECTIVES

The aim of this project was to provide Alto-based graphics tools that would enable people unskilled in either computer science or the graphic arts to easily construct articulate graphic statements. This was a six-month project, begun in February, 1975 and concluded in August, 1975.

METHOD OF APPROACH

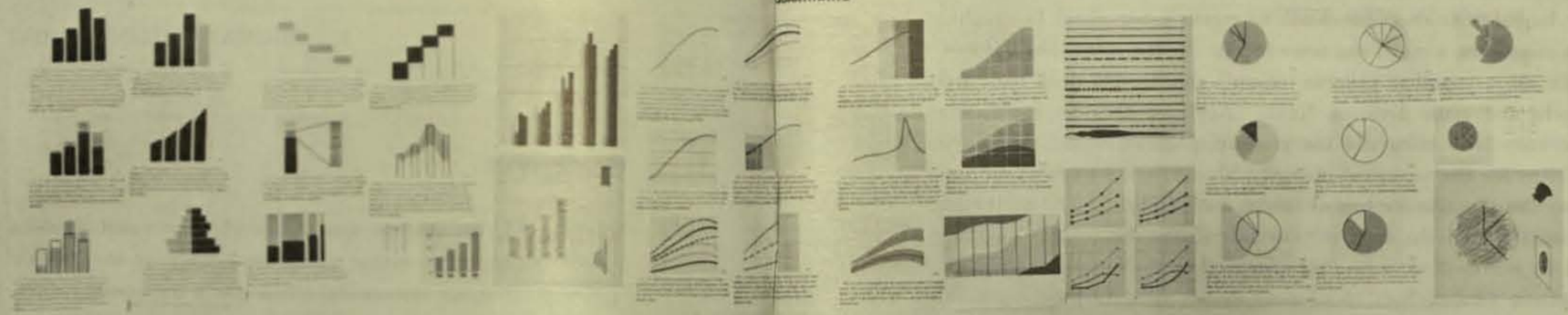
We conceived a research plan for creating a series of special-purpose subsystems, called illustrators, to deal with graphic problems on a specific rather than a general level. The design of these special-purpose illustrators was driven by an attempt to conform to conventional notions about graphic 'idioms' which are commonly understood and used in the working world. To establish a comprehensive frame of reference for this approach we reviewed a wide variety of illustrations, and constructed a graphic mural (reproduced on the following page) which represented four basic graphic environments:

1. quantitative
2. ideographic
3. isomorphic
4. volumetric

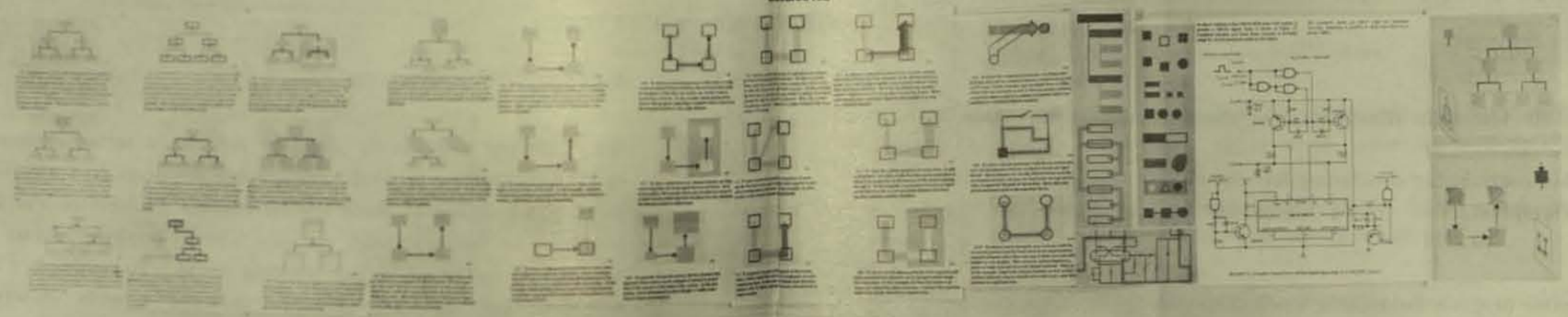
Quantitative figures dealt with visual translations of numerical data. Ideographic figures symbolized conceptual information. Isomorphic figures communicated through abbreviated versions of real forms. Volumetric figures represented objects as they appear or might appear. In each environment we subdivided illustration types in terms of communicative aim, and displayed various particular occasions of each aim. Each of these specific aims, along with its associated occasions, we called an 'idiom'; and it is on this basis that we built the idiomatic illustrator project.

GRAPHIC ENVIRONMENTS

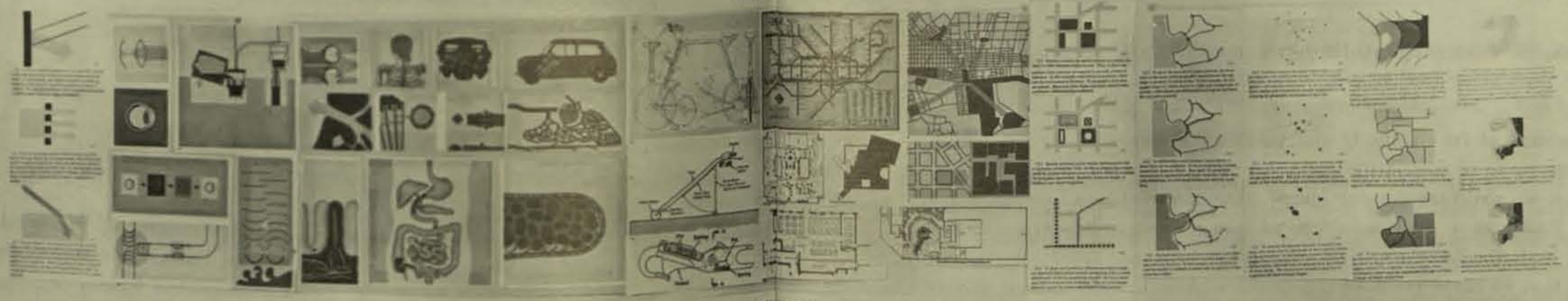
QUANTITATIVE



DIXGRAPHIC



ISOMORPHIC



VOLUMETRIC

The reason for choosing the idiomatic approach is that one does not need to have the whole world of graphic language at one's command to create a bar chart (a graphic 'idiom'); all one needs is some bars, a scale, and some labels. By the same token, if one would rather make a pie chart (another graphic 'idiom') one doesn't need bars and scales; one needs a circle and some dividing lines. Applying this approach, the barchart program would only draw bars, and the piechart program would only draw pies. Too constrained? Not for the unskilled user who simply wants a bar chart *now* without having to master the illustrator's bag of tricks, both technical and aesthetic. For the unskilled user constraint means support: the user enters the graphic world at an idiomatic level, and so can deal with his/her ideas using the specific secondary forms which represent them (scales, bars, pies, etc.) rather than the more general primary form vocabulary (line, shape, texture, etc.) of the professional illustrator.

SCOPE

From the range of possible idiomatic illustrators we chose to work with three:

SIGN - A typographic program for simulating letraset type in making headlines, poster-notices, view-graphics, etc. (This idiom was not represented in the graphic mural.)

BLOCK - An illustrator program for making block diagrams, organization charts, process charts, etc.

PIE - A program for visualizing tabular data automatically in the form of a pie chart.

SIGN was chosen because of its simplicity and because it was needed by the PARC video communications group to make titles for their videotapes. This meant a set of real users with whom we could try out our ideas. BLOCK was chosen because of its potential value to PARC as a communication tool, and because it offered us an opportunity to deal with the basic graphical problem of form and space interaction. PIE was selected so that we could get some experience with an automatic table-driven illustrator.

All of these programs were written in SMALLTALK, with much help from people in LRG. The following three sections of this report describe in detail the basic features of SIGN, BLOCK, and PIE. The last section presents research conclusions drawn from this project.

2. THE SIGN PROGRAM

SIGN is a modest typographic program originally designed to produce hard copy text titles for use in PARC's videotape projects, but it is equally useful for creating bulletin-board notices, small posters, identification labels, view-graphs, and other kinds of 'social-style' office communications. SIGN distinguishes itself from other text systems in that it is environmental: that is to say, it can be used to create word 'pictures' that catch the eye in the physical world of competing visual objects, such as the PARC office scene.

The basic design criteria for SIGN were:

1. A minimum 24 point font size, bold, and sans serif to insure readability in the video medium. A 24 point helvetica bold face was chosen.

2. Exact compositional control on the ALTO screen and identical hard copy by SLOT - so that what you see is what you get.

3. A simple operating procedure that enables people not skilled in computer science or the graphic arts to create professional headline text a-la-letraset.

SIGN is also a step toward solving the graphical problems associated with text headings. Currently, it lacks a coherent scheme for dealing with margin justification, color, changeable leading, inter-character spacing, etc. Much interesting design remains to be done in this area.

Two details about SIGN deserve mention: the spatial gridding and the ease with which a user can obtain hard copy. Vertical gridding is always enforced between lines. There is a grid of 1/2 of the inter-line spacing in the horizontal direction when a line of text is first specified. This aids centering along a vertical guideline. After the initial placement of a line of text, the horizontal gridding is relaxed. This allows for subsequent margin justification. The output is obtained through the use of command files (lots of crocks) which eventually send a press-format file of the screen image to LPT. The important point about output is that the program owes much of its popularity to the ease with which one can obtain it...with the 'push of a button'.

The command language for SIGN is menu-driven and 'modeless'. The menu itself looks like a sign so as to relate aesthetically with the text on the screen. The SIGN program is the most complete of the illustrators discussed in this report; the design criteria were met, and the program has found much use. SIGN is a particularly interesting idiom in that it lies graphically between 'run-on' text and illustration. Simply stated, SIGN deals with text pictorially. This is a very common procedure in the graphic design world in the production of headline materials for magazines, books, brochures, etc.

How to use SIGN:

1. Obtain an ALTO disk labeled 'SIGN'.

2. Load that disk into an ALTO, then push the boot button.

3. Type 'start sign' (+ carriage return) and the ALTO screen will appear like the image on the following page:

The command language for SIGN is menu-driven and 'modeless'. The menu itself looks like a sign so as to relate aesthetically with the text on the screen. The SIGN program is the most complete of the illustrators discussed in this report; the design criteria were met, and the program has found much use. SIGN is a particularly interesting idiom in that it lies graphically between 'run-on' text and illustration. Simply stated, SIGN deals with text pictorially. This is a very common procedure in the graphic design world in the production of headline materials for magazines, books, brochures, etc.

How to use SIGN:

1. Obtain an ALTO disk labeled 'SIGN'.

2. Load that disk into an ALTO, then push the boot button.

3. Type 'start sign' (+ carriage return) and the ALTO screen will appear like the image on the following page:

TEXT MOVE

4. Touch the word "TEXT" (located in the black portion of the alto screen) with the mouse cursor, then touch a place on the white screen area (which corresponds to your expected 8 1/2 x 11 paper output) that you want to be the center for your line of text:

5. Then type your desired line of text, followed by a carriage return. Your text will appear, centered on the designated point:

FIRST WORD

6. Try a second line of text, touching about a letter height underneath the first line. You will notice tht the second line automaticaly spaces itself 3/8" below the first line (30 Alto CRT scan lines) for normal text placement. Extra vertical spacing may be added by increasing the distance of the next mouse touch:

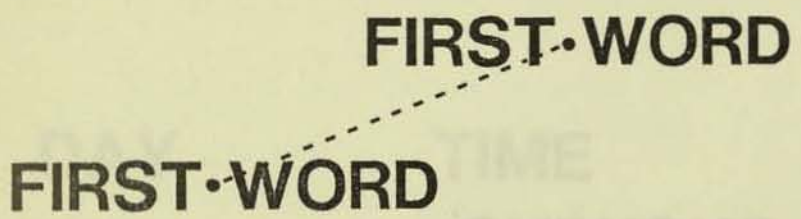
FIRST WORD
SECOND

THIRD

FOURTH

FIFTH

7. If you want to change the position of a line of text, first touch the word 'MOVE' with the mouse cursor. Then touch the text line to be moved. Third, touch the new center for the text line. The text will move as you have indicated:



8. If you want to throw away a line of text, touch 'MOVE'; then touch the text line to be eliminated, and then touch the 'trash can' area (black) at the bottom of the Alto screen. That line of text will disappear.

9. To print a SIGN image type 'PRINT' (+carriage return). This will cause much flashing and nonsense on the Alto screen. When you see a MAXC logout message at the bottom of the screen, push the boot button and you will return to your SIGN image. Then walk down the hall to the SLOT machine and you will find the hardcopy of your SIGN.

10. To save a SIGN image for possible printing or modification at a later date type 'SAVE' + a file name.

11. To recall a previously saved SIGN image type 'RECALL' + a file name.

12. To clean the image area of unwanted debris (location points, etc.) type 'CLEAN'.

13. To begin a new SIGN with a clear screen type 'NEW'.

The following examples illustrate some possible uses for SIGN.

REMINDER

SSL SHOWING
OF XIP VIDEOTAPE

WEDNESDAY
APRIL 23
1:30 PM
SECOND FLOOR COMMONS ROOM

LRG STUDENT SCHEDULE

DAY	TIME	NUMBER
MON	9:00 - 11:30	5
TUE	3:00 - 4:30	3
WED	----------	-
THU	1:30 - 3:00	10
FRI	9:00 - 11:30	5

SEMINAR

BY: ROBERT KAHN

TITLE: "PACKET RADIO --
A MICROPROCESSOR
BROADCASTING NETWORK"

TIME: 11:OO AM

PLACE: CSL COMMONS ROOM

SIGN: SUMMARY EVALUATION

1. Videotape title applications are very successful, and the program is now used regularly for that purpose by PARC's video communication group.

2. Totally inexperienced users in the video group were able to operate the program immediately, as were secretaries, researchers, and others in the PARC community.

3. The volume of general (non-video) office applications has been much larger than we expected, and has proved the program to be a useful multi-purpose workhorse. A dribble-file associated with the program has recorded this volume of use.

4. SIGN's single-font (caps only) capability is far too limited for most practical applications. Currently, we have no easy answer for this deficiency.

5. The program is essentially an elegant hack, and consequently some users have experienced frustrating breakdowns.

6. The move function is still crude, and offers inadequate support for the variety of alignment and spacing situations which commonly occur in graphic design.

7. Conceptually, SIGN offers considerable promise as a headlining device for graphic design work, particularly in the areas of magazine, book, and brochure production. The main reason for this is that it treats word forms as graphical objects, and consequently relates to the graphic designer's methodology.

3. THE BLOCK PROGRAM

BLOCK is designed to deal with graphic problems in the idiom of block diagrams; including organization charts, process charts, and other rectilinear figures. This program is a specialist. It does not attempt to take on the whole world of graphic needs, although a few interesting by-products such as 3-D perspective are possible. BLOCK takes a view of graphic language that emphasizes design grammar (spatial dynamics, composition, etc.) rather than form vocabulary (gray scale, sophisticated detail, etc.). It is intended to help the ordinary (non-illustrator) user construct an articulate graphic figure without having to learn the illustrator's profession. Basic aesthetics as well as manual skills are supplied by the program.

The design criteria for BLOCK were:

1. A basic form vocabulary of lines and rectangles for building the structural elements necessary to block diagrams. Secondary requirements included word and arrow forms.

2. A spatial grammar for composing form elements on the ALTO picture plane with respect to aesthetics of planar design (visual relationship and differentiation).

3. A capability for visual editing, including move and copy functions. Later, an area move/copy function was added to the criteria.

4. A set of graphic processing utilities, including such functions as clean (refresh), file (save and get), print (xgp), and reset.

BLOCK, like SIGN, has been developed to the point that it is a usable SMALLTALK subsystem for making illustrations. The essence of the BLOCK program lies in its gridding scheme which spatially organizes its graphical forms (box, line, arrow, text) in an aesthetically related manner. During the design of BLOCK it became clear that no existing font was suitable for diagrammatic purposes, so we designed and executed a new font. The design criteria for the font (BLOCKFONT) were:

1. That it be a condensed font to maximize horizontal space on the ALTO screen, which is a major constraint in making diagrams.

2. That it have the smallest bold (2-bit thick) face possible on the ALTO screen, and still remain readable.

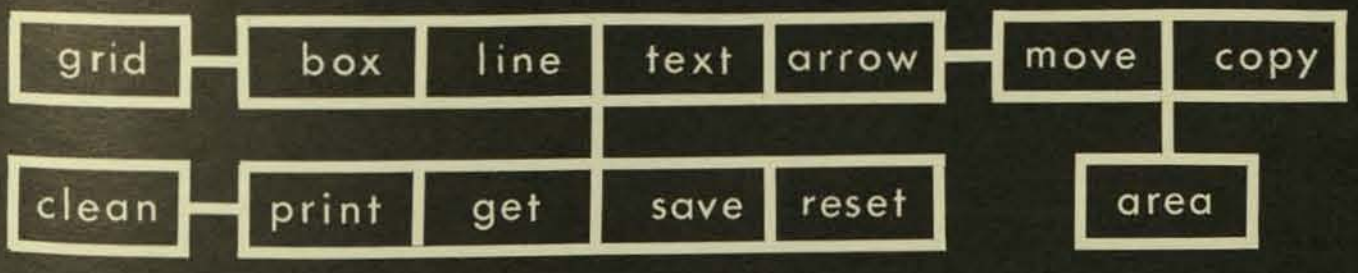
3. That the font relate aesthetically to the rectilinear forms generated with the BLOCK program.

First, an ALTO font satisfying these criteria was designed. Its dimensions are 6 x 10. Subsequently a coordinated spline outline version was constructed. This font should find wide usage in PARC terminal displays where horizontal space is at a premium.

The command language for BLOCK is menu-driven and 'modeless'. The thirteen commands are divided into four logical groups:

1. form vocabulary (box, line, arrow, text)
2. space control (grid module)
3. editing functions (area, move, copy)
4. memory commands (print, save, get, reset, clean)

The menu itself is presented in the form of a block diagram for (1) aesthetic relevance and (2) to enable the visual presence of a number of command options without creating a sense of visual confusion. It appears on the ALTO screen like the image on the following page:



Individual command functions for BLOCK are as follows:

BOX:

Draws boxes, any size or shape. The command requires two mouse inputs: upper left and lower right box coordinates. The box corners are positioned at the nearest points on a 32-unit grid. This aligns boxes automatically, provides consistent spacing, and allows the user to be rough in his/her manual command executions.

LINE:

Draws lines, any length or direction. The command requires two mouse inputs: beginning and ending points of the line. The line endpoints are positioned at the nearest points on a 16-unit because of the grid, lines will automatically split spaces between boxes, and provide centering and exact box contact when used as connecting links. In addition, lines built at right angles to each other automatically form a perfect corner. Again, the user may be somewhat rough in manual execution without problem.

ARROW:

Draws lines with arrowheads attached to the point designated by the second mouse input. Arrow lines may be any length, vertically or horizontally. In all other respects this command functions like line.

TEXT:

Prints a line of text as objects anywhere in the figure, an 8-unit grid. The text automatically centers itself within boxes. Inputs are typed sequences (terminated with carriage return); and mouse points (center location for text).

MOVE:

Moves any of the above objects anywhere in the image, in terms of its assigned grid. Move can also be used to dump unwanted objects into the garbage can at the bottom right of the screen, causing them to disappear. Two mouse inputs are required, corresponding old and new locations.

COPY:

Copies objects anywhere in the image, in terms of assigned gridding. Like move, two mouse inputs are required, to indicate form selected and the desired position of its copy.

AREA:

Selects a form area rather than an object, for moving or copying. As in box, two mouse inputs are required to indicate upper left and lower right corners of the rectangular area selected. In addition, third and fourth mouse inputs are required corresponding to old and new locations for the forms included within the rectangular area selected. The rectangular area selected will then be moved or copied in the new location.

GRID:

Permits the user to change the assigned grid spacing for any particular form.

PRINT:

Creates an XGP file for hard copy. Input is a filename (one word terminated with line-feed). The file created may then be transmitted to a NOVA with an XGP, and the command 'XPLOT filename' given to the NOVA operating system.

CLEAN:

Refreshes the entire image, restoring forms damaged by moving, etc.

SAVE:

Allows images to be saved for future display, printing or modification.

GET:

Allows previously saved images to be recalled.

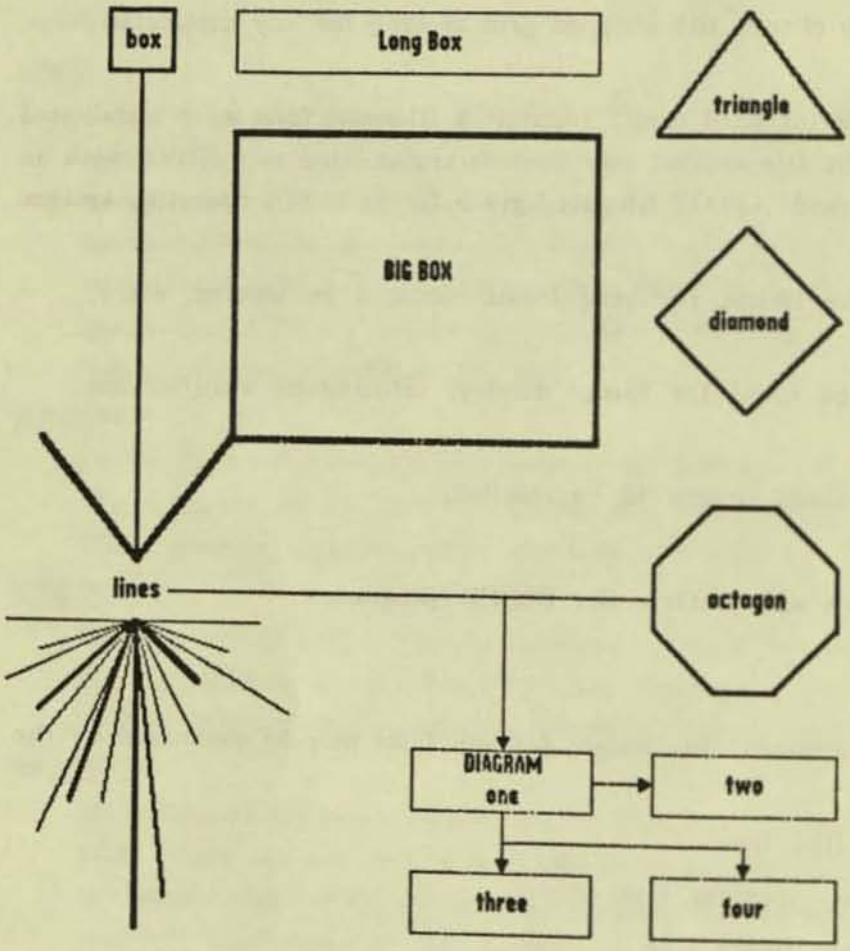
RESET:

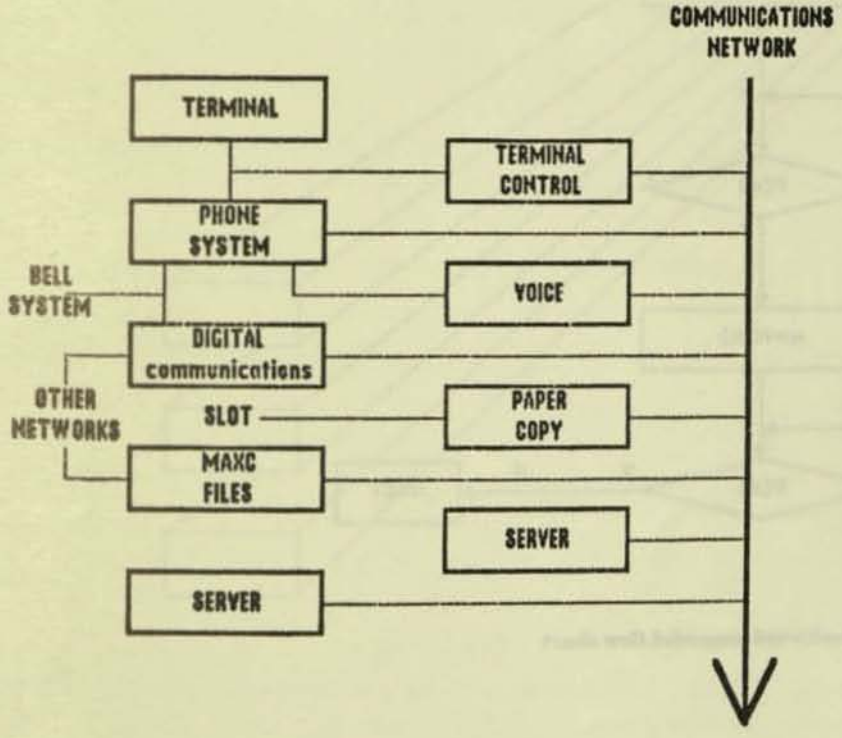
Erases entire screen and restarts the BLOCK program.

In addition to menu commands, line weight for any form may be controlled by the mouse button pushed:

1. top button: fine line
2. middle button: medium line
3. bottom button: heavy line

We have included a group of illustrations which describe BLOCK's range of capabilities, and suggest how the program might be used.





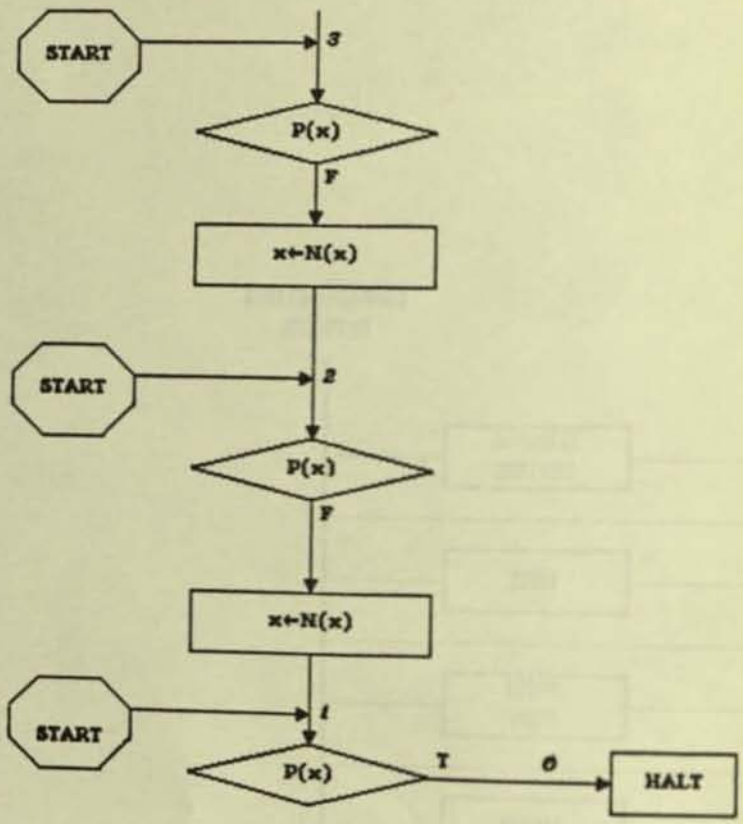
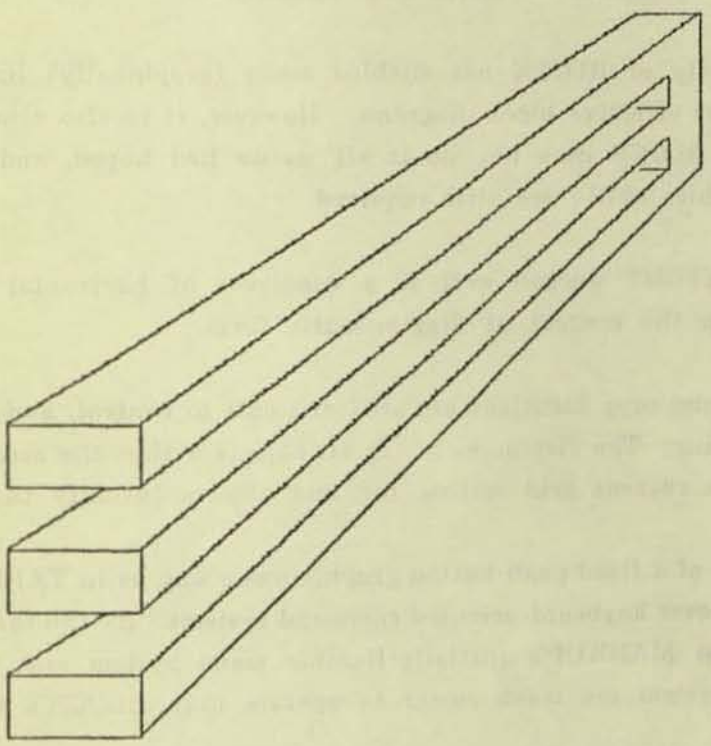


Figure 6. A backward-expanded flow chart



BLOCK: SUMMARY EVALUATION

1. The most successful aspect of this program is its spatial control of form. The notion of 'invisible' gridding as a strategy for the management of form/space interaction (design grammar) worked well, and has since been used with equal success by other programs at PARC (e.g. MARKUP).

2. The simplicity of BLOCK has enabled many (graphically) inexperienced users to construct effective block diagrams. However, it is also clear from the work done that BLOCK does not 'do it all' as we had hoped, and that some elementary graphics skills are still required.

3. The BLOCKFONT worked well as a conserver of horizontal space, and competes well in the context of diagrammatic form.

4. Area move and copy functions are still difficult to control, and require too much visual editing. The displacement for all objects within the area is gridded according to the current grid setting for text objects (usually the smallest).

5. The concept of a fixed push-button graphic menu was, as in TAPE, felt to be an improvement over keyboard-oriented command systems. By the same token, it now appears that MARKUP's spatially-flexible menu system and TOOLBOX's keyset control system are much easier to operate than BLOCK's fixed menu.

6. BLOCK lets the user know where his/her cursor is in relation to the 'invisible' grid spacing by moving the cursor to the nearest grid point (according to the form being created) when the mouse button is depressed. As long as the button remains depressed the cursor "hops" from grid point to grid point when the mouse is moved, and a point is specified when the mouse button is released.

7. It is a demonstrable fact that infinite variations on the 'block diagram' theme can be created with relative ease using this program. However, *exactly* where BLOCK ends and FLOW, or PERT, etc., begin is not yet clear. Further exploration with other related idioms would help to answer this question.

8. Some fairly sophisticated illustrations can be created through line vocabulary alone.

9. Mouse buttons work well as a tactical means for controlling line weight.

10. In an illustrator context, it helps to be able to deal with words as graphic form objects (like lines or boxes).

4. THE PIE PROGRAM

PIE is an experimental effort to create an 'automatic illustrator'; that is, a program that puts the 'illustrator' entirely within the machine and thus allows the user to get a professional-level illustration without having to perform any graphical tasks. The graphic idiom of pie charts was chosen for this experiment because, as a data-based idiom, it lends itself naturally to mechanical graphic translation. The basic graphic design decisions in making a pie chart are quantitative: not only the spatial division of the pie into its component segments, but also the placement of labels in relation to the available space resulting from those segments. Therefore, all that PIE requires of the user is a table of items (labels) and their associated numerical values (segments). The program (1) makes the pie, (2) translate the numbers into percent values and cuts the pie into corresponding pieces, and (3) attaches item labels to the segments. User interaction takes place entirely within the context of creating and/or editing the tabular data, a familiar and ordinary office activity.

Basic design criteria for PIE were:

1. A form vocabulary comprised of a single fixed-diameter circle, straight radial lines within that circle, and text labels.

2. A spatial grammar that translates a set of numbers into degree equivalents, and represents those equivalents as pie segments using radial lines.

3. Automatic/aesthetic label placement, with respect to spaces and positions of pie segments.

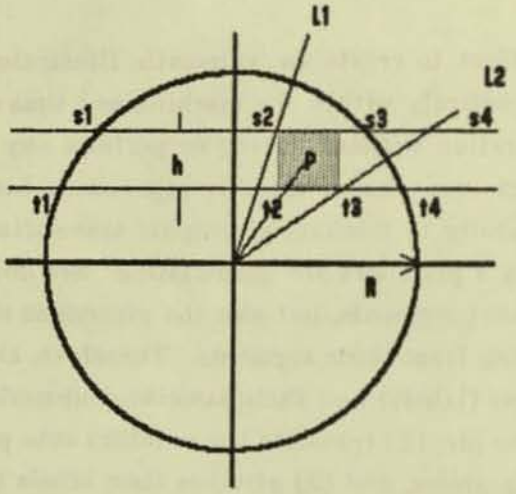
4. A system for tabular data entry that permits interactive user editing.

In satisfying the design criteria for an automatic piechart-maker the most difficult problem was that of label placement. The strategy for this part of the program was as follows: If a pie segment had adequate size and/or an advantageous position for (horizontal) text, then (1) the label would be placed internally and generally centered within the available space. If the segment was small and/or vertically oriented, then (2) the label would be placed externally, and related to its segment by a connecting link.

The space available for a text label within a slice of the pie was computed as follows:

(a) point p is chosen so that it lies on the bisector of angle $\frac{\theta + \varphi}{2}$

and is located 3/5R from the center of the circle.



(b) the four points S_1, S_2, S_3, and S_4 are computed by finding the intersections of the line $y = \frac{(Py + \frac{1}{2}h)}{Px} x$ with lines l_1, l_2 and the circle.

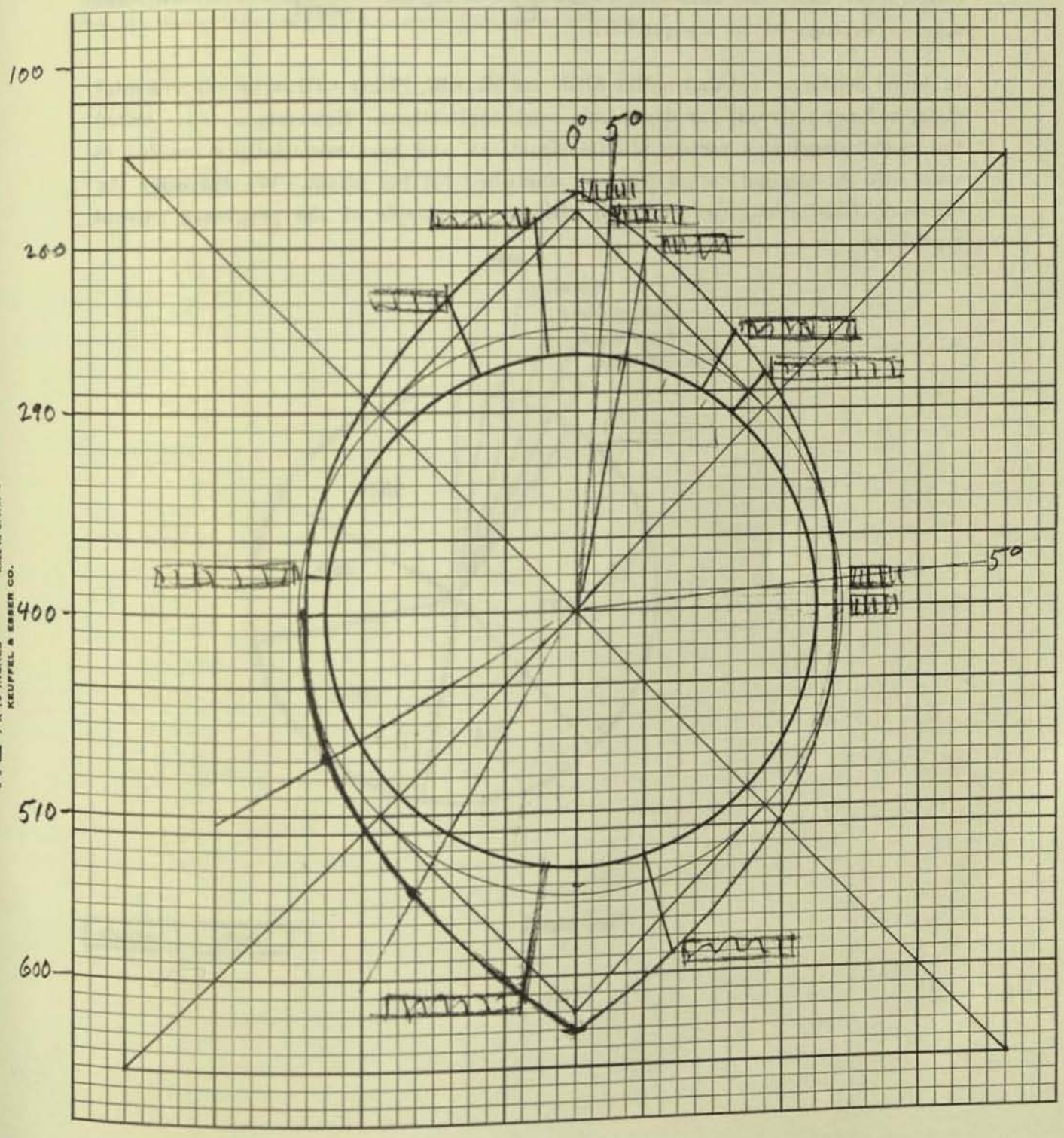
(Note: h=font height)

(c) next the four points t_1, t_2, t_3 and t_4 are computed by finding the 4 intersections of the line $y = \frac{(Py - \frac{1}{2}h)}{Px} x$ with lines l_1, l_2 and the circle.

(d) finally, if s_i and s_j $\in\{s_1,s_2,s_3,s_4\}$ are the intersection that lie immediately to the left and right of Px and t_x, t_r $\in\{t_1,t_2,t_3,t_4\}$ are defined similarly then the rectangle with upper left corner at max (S_i,t_x) and lower left corner at min (S_j,t_l) is the space that text may occupy and still be inside the slice defined by θ and φ.

It should be noted that the space for text obtained by the method just described does not yield the maximum width rectangle that can lie in a segment of the pie. Originally we completed the maximum width rectangle that can lie in a segment. Placing the text centered in this rectangle caused graphical interference between the text and the radial lines which divide the pie and the links used where text could not fit inside the slice. Hence we chose the algorithm which, in general, produced a smaller space for the text but yielded a more aesthetically pleasing result.

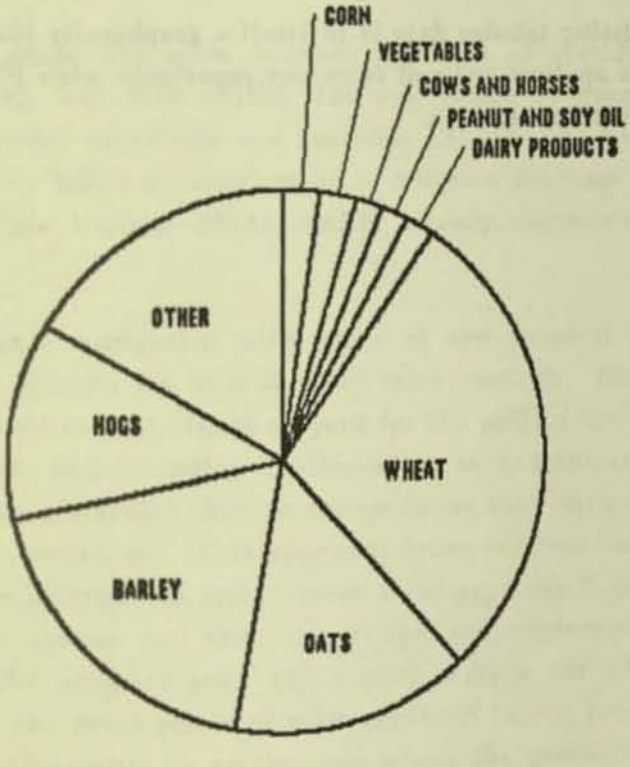
The system for external pielabels sought to maximize the number of possible labels that could be automatically arrayed around the pie, and at the same time make the most economical use of available space on the ALTO screen. It appeared that a parabolic arrangement of labels around the pie produced the most efficient and manageable external pielable system, as illustrated by the following design drawing:



The user interface for PIE is a simple table, into which the user types item names (for labels) and corresponding numerical quantities (for segments). As the user types in items and quantities the table expands downward. This table can be edited: items and quantities can be added, deleted, exchanged, or moved as desired. The order in which items are displayed corresponds to the order in which they are represented in the pie (starting at 'noon' and advancing clockwise). Thus, the user has control over the segment arrangment in his piechart.

It should be emphasized that unlike SIGN and BLOCK, PIE is still in an experimental state, and not yet ready for dependable work applications. However, we have tested the program against a variety of data situations, and have essentially succeeded in satisfying the original design criteria established for the idiom. We can offer the following illustration, executed with PIE, as an example of the program's current capability:

LABELS	DATA
CORN	5
VEGETABLES	5
COWS AND HORSES	4
PEANUT AND SOY OIL	4
DAIRY PRODUCTS	4
WHEAT	60
OATS	30
BARLEY	40
HOGS	25
OTHER	30



piechart text

xplot 'test'?

PIE: SUMMARY EVALUATION

1. For certain kinds of illustrations (particularly quantitative) automatic illustrator programs are quite possible. Essentially, PIE can produce a good pie chart without *any* user participation in the graphic process. Based on our experience with PIE, we believe that bar charts and curve graphs can also be produced in a more or less automatic fashion.

2. Word and number labeling (because of its unpredictable length) is a serious problem for automatic illustrators, and as of now there appears to be no simple solution.

3. Graphic execution time saved in PIE-like illustrators is enormous - much more than in SIGN or BLOCK.

4. Creating and editing tabular data is in itself a graphically idiomatic process (quite aside from its application) and from our experience with PIE looks like a pregnant area for future research.

5. CONCLUSIONS

These conclusions are an attempt to summarize our research findings in relation to BLOCK, SIGN and PIE. We hope these conclusions will be helpful to others involved in the design of interactive picture-making systems.

ON METHODOLOGY

We did not adopt the more common approach of specifying and implementing a graphics system and then writing the application programs. We rather scrounged whatever graphics capability was available (SMALLTALK) and began by simulating the illustrator's habit of building up a 'graphics language' as we worked. We were able to do this because SMALLTALK already contained a rich set of graphics primitives.

We began our investigation with three of the simplest and most commonly used idioms. Our reasons for this decision were twofold: first, about a dozen simple, well-known conventional idioms account for the bulk of technical graphics used in the working world, and secondly, it allowed us to concentrate on user issues such as command languages rather than on system issues that arise when dealing with complex pictorial representation. This approach drove out two insights that we might have missed had we adopted the more conventional approach that involves the development of a graphics system and then the design and implementation of the application programs. The insights are: (1) a very **simple set of programming tools** is sufficient for the development of most graphical idioms for general office use and (2) the **user requirements** in applications where the presentation is 2-dimensional and dynamic are much more subtle and complex than we had imagined. We found ourselves designing form 'processes' rather than form 'products' through which to create pictures. Picture creation takes place in a human time continuum and the 'rhythm' of visualization is as important as the availability of form options.

ON RESOURCES

As mentioned above one does not need much in the way of a graphics system to write useful application programs. The SMALLTALK picture manipulation and drawing primitives are quite sufficient. These include and enable:

1. rectangles, points and grids - SPATIAL GRAMMAR
2. lines of up to seven thicknesses - FORM VOCABULARY
3. text strings - LITERAL IDENTIFICATION
4. turtle delineation - GRAPHIC STATEMENT

For a complete description of this system please refer to the SMALLTALK manual.

ON PROJECT RESULTS

We feel that this project was a success in that we have demonstrated that it is possible to combine conventional graphic idioms and current computer technology to make it possible for ordinary (graphically unskilled) people to create articulate graphical statements. This has been demonstrated by various utilizations of the BLOCK program within the PARC community involving the creation of block diagrams. The simple compositional help that is offered by BLOCK greatly enhanced the aesthetic character of the user diagrams. The piechart program offers a powerful 'machine tool' for the person who wants to represent tabular data in visual form without having to actively engage in the techniques of technical illustration, or in this case, decisions of label placement. Evidence of SIGN's utility can be found in PARC videotapes and on many PARC bulletin boards.

ON FUTURE RESEARCH

We have in the scope of this project only scratched the surface of the idiomatic illustrator concept. There remain many modifications to explore with BLOCK, PIE and SIGN. For example, can one make the stages in picture specification like block-out and touch-up more explicit? Thus offering the non-professional user even more help during the creation of his/her illustration.

There also remain a host of other idioms to explore, such as barcharts, curve graphs, plans, maps, volumetric representations, etc. We believe an understanding of commonly understood graphic communication idioms in the context of a display-based interactive computing system will have large payoffs in office information systems of the future. For such systems (idiomatic illustrators) to be really useful they need to be integrated with a system that includes text. Research in this area is currently underway at PARC (Master-maker Project).

Projecting even further into the future, text/graphics systems should allow for the personalization of graphic programs so that professionals in the fields of graphic design and illustration can incorporate the computer as an effective medium for visual communication.

BIBLIOGRAPHY

Anderson, Donald

THE ART OF WRITTEN FORMS
Holt, Rinehart and Winston, New York, 1969

Bertin, Jacques

SEMIOLOGIE GRAPHIQUE
Mouton, Paris, 1967

William Bowman

GRAPHIC COMMUNICATION
John Wiley and Sons, New York, 1968

Kepes, Gyorgy

SIGN, IMAGE, SYMBOL
George Braziller, New York, 1966

Learning Research Group, Xerox Corporation
Palo Alto Research Center

PERSONAL DYNAMIC MEDIA
Palo Alto, 1975

An Annotated Bibliography of PIE Publications

Ira Goldstein and Dan Bobrow
July 28, 1980

General Abstract

PIE is an experimental personal information environment implemented in Smalltalk. PIE uses a description language to support the interactive development of programs, and to support the office-related tasks of document preparation, electronic mail, and database management. PIE's salient characteristics are:

1. The system employs a network of nodes to represent specific facts (personnel data, appointments, Smalltalk methods), generic information about different kinds of entitites (constraints, defaults), and procedural knowledge regarding the functions associated with different entitites (summarizing personnel data, producing specialized code from abstract descriptions).

2. Each node can be assigned several perspectives. A perspective describes a different aspect of the entity represented by the node, and provides specialized actions from that point of view.

3. The network is layered, that is, the links between nodes are separated into distinct sets. This allows alternatives to be expressed regarding the structure of a design described in the network. It also facilitates cooperative design by separating the contributions of collaborators into distinct layers.

4. Contracts can be created that monitor several nodes whose descriptions must be kept consistent. Contractual agreements are expressible as formal constraints, or, to make the system failsoft, as English text interpretable by the user.

Bibliography

General Discussions

[ivy]<pie>AAAI.Press (10 pages)
Goldstein, I. P. and D. G. Bobrow, "Descriptions for a programming environment", Proceedings of the First Annual Conference of the National Association for Artificial Intelligence, Stanford, Cal., August, 1980, pp. 187-194.

This paper provides a short introduction to the PIE project from the perspective of its application to software development.

[ivy]<pie>Chatham.Press (5 pages)
Goldstein, I. P., "PIE: A network-based personal information environment", Presented at the Conference on Office Semantics, Chatham, Mass., June 15-18, 1980.

This paper provides a complementary introduction from the perspective of its application to office related tasks.

Description Languages

[ivy]<pie>LispConf.Press (10 pages)
Goldstein, I. P. and D. G. Bobrow, "Extending Object Oriented Programming in Smalltalk", Proceedings of the Lisp Conference, Stanford, August, 1980, pp. 75-81.

This paper discusses the extensions required to Smalltalk to support a network-based description language.

[ivy]<pie>NBS.Press (3 pages)
Goldstein, I. P., "Position Paper for the NBS/ACM Workshop on Data Abstraction, Data Bases and Conceptual Modelling", Pingree Park, Colorado, June 23-26, 1980

This paper discusses the relation of our Smalltalk extensions to other work on data modelling and abstract datatypes.

Layers

[ivy]<pie>AISB.Press (10 pages)
Bobrow, D. G. and I. P. Goldstein, "Representing Design Alternatives," Proceedings of the Conference on Artificial Intelligence and the Simulation of Behavior, Amsterdam, July, 1980.

This paper provides a brief discussion regarding layers as a means to represent the evolution of a software system.

[ivy]<pie>Software.Press (30 pages)
Goldstein, I. P. and D. G. Bobrow, "A Layered Approach to Software Design," (submitted for publication).

This paper provides a more thorough discussion of layers.

User Interface

[ivy]<pie>Browsers.Press (17 pages)
Goldstein, I. P. and D. G. Bobrow, "Browsing in a programming environment", to appear in the Proceedings of the 14th Hawaii Conference on Systems Science, Jan. 1981.

This paper focusses on issues related to the system interface for examining and manipulating networks.

Descriptions for a Programming Environment[1]

Ira P. Goldstein and Daniel G. Bobrow
Xerox Palo Alto Research Center
Palo Alto, California 94304, U.S.A

Abstract: PIE is an experimental personal information environment implemented in Smalltalk that uses a description language to support the interactive development of programs. PIE contains a network of nodes, each of which can be assigned several perspectives. Each perspective describes a different aspect of the program structure represented by the node, and provides specialized actions from that point of view. Contracts can be created that monitor nodes describing different parts of a program's description. Contractual agreements are expressible as formal constraints, or, to make the system failsoft, as English text interpretable by the user. Contexts and layers are used to represent alternative designs for programs described in the network. The layered network database also facilitates cooperative program design by a group, and coordinated, structured documentation.

Introduction

In most programming environments, there is support for the text editing of program specifications, and support for building the program in bits and pieces. However, there is usually no way of linking these interrelated descriptions into a single integrated structure. The English descriptions of the program, its rationale, general structure, and tradeoffs are second class citizens at best, kept in separate files, on scraps of paper next to the terminal, or, for a while, in the back of the implementor's head.

Furthermore, as the software evolves, there is no way of noting the history of changes, except in some primitive fashion, such as the history list of Interlisp [Teitelman78]. A history list provides little support for recording the purpose of a change other than supplying a comment. But such comments are inadequate to describe the rationale for coordinated sets of changes that are part of some overall plan for modifying a system. Yet recording such rationales is necessary if a programmer is to be able to come to a system and understand the basis for its present form.

Developing programs involves the exploration of alternative designs. But most programming environments provide little support for switching between alternative designs or comparing their similarities and differences. They do not allow alternative definitions of procedures and data structures to exist simultaneously in the programming environment; nor do they provide a representation for the evolution of a particular set of definitions across time.

In this paper we argue that by making descriptions first class objects in a programming environment, one can make life easier for the programmer through the life cycle of a piece of software. Our argument is based on our experience with PIE, a description-based programming environment that supports the design, development, and documentation of Smalltalk programs.

[1] Published in the Proceedings of First Annual Conference of the American Association for Artificial Intelligence, August, 1980, pp. 187-194.

Networks

The PIE environment is based on a network of nodes which describe different types of entities. We believe such networks provide a better basis for describing systems than files. Nodes provide a uniform way of describing entities of many sizes, from small pieces such as a single procedure to much larger conceptual entities. In our programming environment, nodes are used to describe code in individual methods, classes, categories of classes, and configurations of the system to do a particular job. Sharing structures between configurations is made natural and efficient by sharing regions of the network.

Nodes are also used to describe the specifications for different parts of the system. The programmer and designer work in the same environment, and the network links elements of the program to elements of the design and specification. The documentation on how to use the system is embedded in the network also. Using the network allows multiple views of the documentation. For example, a primer and a reference manual can share many of the same nodes while using different organizations suited to their different purposes.

In applying networks to the description of software, we are following a tradition of employing semantic networks for knowledge representation. Nodes in our network have the usual characteristics that we have come to expect in a representation language--for example, defaults, constraints, multiple perspectives, and context-sensitive value assignments.

There is one respect in which the representation machinery developed in PIE is novel: it is implemented in an object-oriented language. Most representation research has been done in Lisp. Two advantages derive from this change of soil. The first is that there is a smaller gap between the primitives of the representation language and the primitives of the implementation language. Objects are closer to nodes (frames, units) than lists. This simplifies the implementation and gains some advantages in space and time costs. The second is that the goal of representing software is simplified. Software is built of objects whose resemblance to frames makes them natural to describe in a frame-based knowledge representation.

Perspectives

Attributes of nodes are grouped into perspectives. Each perspective reflects a different view of the entity represented by the node. For example, one view of a Smalltalk class provides a definition of the structure of each instance, specifying the fields it must contain; another describes a hierarchical organization of the methods of the class; a third specifes various external methods called from the class; a fourth contains user documentation of the behavior of the class.

The attribute names of each perspective are local to the perspective. Originally, this was not the case. Perspectives accessed a common pool of attributes attached to the node. However, this conflicted with an important property that design environments should have, namely, that different agents can create perspectives independently. Since one agent cannot know the names chosen by another, we were led to make the name space of each perspective on a node independent.

Perspectives may provide partial views which are not necessarily independent. For example, the organization perspective that categorizes the methods of a class and the

documentation perspective that describes the public messages of a class are interdependent. Attached procedures are used to maintain consistency between such perspectives.

Each perspective supplies a set of specialized actions appropriate to its point of view. For example, the *print* action of the structure perspective of a class knows how to prettyprint its fields and class variables, whereas the organization perspective knows how to prettyprint the methods of the class. These actions are implemented directly through messages understood by the Smalltalk classes defining the perspective.

Messages understood by perspectives represent one of the advantages obtained from developing a knowledge representation language within an object-oriented environment. In most knowledge representation languages, procedures can be attached to attributes. Messages constitute a generalization: they are attached to the perspective as a whole. Furthermore, the machinery of the object language allows these messages to be defined locally for the perspective. Lisp would insist on global functions names.

Contexts and Layers

All values of attributes of a perspective are relative to a *context*. Context as we use the term derives from Conniver [SussmanMcDermott72]. When one retrieves the values of attributes of a node, one does so in a particular context, and only the values assigned in that context are visible. Therefore it is natural to create alternative contexts in which different values are stored for attributes in a number of nodes. The user can then examine these alternative designs, or compare them without leaving the design environment. Since there is an explicit model of the differences between contexts, PIE can highlight differences between designs. PIE also provides tools for the user to choose or create appropriate values for merging two designs.

Design involves more than the consideration of alternatives. It also involves the incremental development of a single alternative. A context is structured as a sequence of layers. It is these layers that allow the state of a context to evolve. The assignment of a value to a property is done in a particular layer. Thus the assertion that a particular procedure has a certain source code definition is made in a layer. Retrieval from a context is done by looking up the value of an attribute, layer by layer. If a value is asserted for the attribute in the first layer of the context, then this value is returned. If not, the next layer is examined. This process is repeated until the layers are exhausted.

Extending a context by creating a new layer is an operation that is sometimes done by the system, and sometimes by the user. The current PIE system adds a layer to a context the first time the context is modified in a new session. Thus, a user can easily back up to the state of a design during a previous working session. The user can create layers at will. This may be done when he or she feels that a given groups of changes should be coordinated. Typically, the user will group dependent changes in the same layer.

Layers and contexts are themselves nodes in the network. Describing layers in the network allows the user to build a description of the rationale for the set of coordinated changes stored in the layer in the same fashion as he builds descriptions for any other node in the network. Contexts provide a way of grouping the incremental changes, and describing the rationale for the group as a whole. Describing contexts in the network also allows the layers of a context to themselves be asserted in a context sensitive fashion (since all

descriptions in the network are context-sensitive). As a result, super-contexts can be created that act as *big switches* for altering designs by altering the layers of many sub-contexts.

Contracts and Constraints

In any system, there are dependencies between different elements of the system. If one changes, the other should change in some corresponding way. We employ contracts between nodes to describe these dependencies. Implementing contracts raises issues involving 1) the knowledge of which elements are dependent; 2) the way of specifying the agreement; 3) the method of enforcement of the agreement; 4) the time when the agreement is to be enforced.

PIE provides a number of different mechanisms for expressing and implementing contracts. At the implementation level, the user can attach a procedure to any attribute of a perspective, (see BobrowWinograd77 for a fuller discussion of attached procedures); this allows change of one attribute to update corresponding values of others. At a higher level, one can write simple constraints in the description language (e.g. two attributes should always have identical values), specifying the dependent attributes. The system creates attached procedures that maintain the constraint.

There are constraints and contracts which cannot now be expressed in any formal language. Hence, we want to be able to express that a set of participants are interdependent, but not be required to give a formal predicate specifying the contract. PIE allows us to do this. Attached procedures are created for such contracts that notify the user if any of the participants change, but which do not take any action on their own to maintain consistency. Text can be attached to such informal contracts that is displayed to the user when the contract is triggered. This provides a useful inter-programmer means of communication and preserves a *failsoft* quality of the environment when formal descriptions are not available.

Ordinarily such non-formal contracts would be of little interest in artificial intelligence. They are, after all, outside the comprehension of a reasoning program. However, our thrust has been to build towards an artificially intelligent system through succcessive stages of man-machine symbiosis. This approach has the advantage that it allows us to observe human reasoning in the controlled setting of interacting with the system. Furthermore, it allows us to investigate a direction generally not taken in AI applications: namely the design of memory-support rather than reasoning-support systems.

An issue in contract maintenance is deciding when to allow a contract to interrupt the user or to propagate consistency modifications. We use the closure of a layer as the time when contracts are checked. The notion is that a layer is intended to contain a set of consistent values. While the user is working within a layer, the system is generally in an inconsistent state. Closing a layer is an operation that declares that the layer is complete. After contracts are checked, a closed layer is immutable. Subsequent changes must be made in new layers appended to the appropraiate contexts.

Coordinating designs

So far we have emphasized that aspect of design which consists of a single individual manipulating alternatives. A complementary facet of the design process involves merging two partial designs. This task inevitably arises when the design process is undertaken by a team rather than an individual. To coordinate partial designs, one needs an environment in which potentially overlapping partial designs can be examined without overwriting one another. This is accomplished by the convention that different designers place their contributions in separate layers. Thus, where an overlap occurred, the divergent values for some common attributes are in distinct layers.

Merging two designs is accomplished by creating a new layer into which are placed the desired values for attributes as selected from two or more competing contexts. For complex designs, the merge process is, of course, non-trivial. We do not, and indeed cannot, claim that PIE eliminates this complexity. What it does provides is a more finely grained descriptive structure than files in which to manipulate the pieces of the design. Layers created by a merger have associated descriptions in the network specifying the contexts participating in the merger and the basis for the merger.

Meta-description

Nodes can be assigned meta-nodes whose purpose is to describe defaults, constraints, and other information about their object node. Information in the meta-node is used to resolve ambiguities when a command is sent to a node having multiple perspectives.

One situation in which ambiguity frequently arises is when the PIE interface is employed by a user to browse through the network. When the user selects a node for inspection, the interface examines the meta-node to determine which information should be automatically displayed for the user. By appropriate use of meta-information, we have made the default display of the PIE browser identical to one used in Smalltalk. (Smalltalk code is organized into a simple four-level heirarchy, and the Smalltalk browser allows examination and modification of Smalltalk code using this taxonomy.) As a result, a novice PIE user finds the environment similar to the standard Smalltalk programming environment which he has already learned.

Simplifying the presentation and manipulation of the layered network underlying the PIE environment remains an important research goal, if the programming environment supported by PIE is to be useful as well as powerful. We have found use of a meta-level of descriptions to guide the presentation of the network to be a powerful device to achieve this utility.

Conclusion

PIE has been used to describe itself, and to aid in its own development. Specialized perspectives have been developed to aid in the description of Smalltalk code, and for PIE perspectives themselves. On-line documentation is integrated into the descriptive network. The implementors find this network-based approach to developing and documenting programs superior to the present Smalltalk programming environment. A small number of other people have begun to use the system.

This paper presents only a sketch of PIE from a single perspective. The PIE description language is the result of transplanting the ideas of KRL [BobrowWinograd77] and FRL

[GoldsteinRoberts77] into the object oriented programming environment of Smalltalk [KayGoldberg77, Ingalls78]. A more extensive discussion of the system in terms of the design process can be found in BobrowGoldstein80, and GoldsteinBobrow80a. A view of the PIE description language as an extension of the object oriented programming metaphor can be found in GoldsteinBobrow80b. Finally, the use of PIE as a prototype office information system is described in Goldstein80.

References

Bobrow, D.G. and Goldstein, I.P. "Representing Design Alternatives", *Proceedings of the AISB Conference*, Amsterdam, 1980

Bobrow, D.G. and Winograd, T. "An overview of KRL, a knowledge representation language", *Cognitive Science* 1, 1 1977

Goldstein, I.P. "PIE: A network-based personal information environment", *Proceedings of the Office Semantics Workshop,* Chatham, Mass., June, 1980.

Goldstein, I.P. and Bobrow, D.G., "A layered approach to software design", *Xerox Palo Alto Research Center CSL-80-5.* 1980a

Goldstein, I.P. and Bobrow, D.G., "Extending Object Oriented Programming in Smalltalk", *Proceedings of the Lisp Conference.* Stanford University, 1980b

Goldstein, I.P. and Roberts, R.B. "NUDGE, A knowledge-based scheduling program", *Proceedings of the Fifth International Joint Conference on Artificial Intelligence,* Cambridge: 1977, 257-263.

Ingalls, Daniel H., "The Smalltalk-76 Programming System: Design and Implementation," *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, January 1978, pp 9-16.

Kay, A. and Goldberg, A. "Personal Dynamic Media" *IEEE Computer,* March, 1977.

Sussman, G., & McDermott, D. "From PLANNER to CONNIVER -- A genetic approach". *Fall Joint Computer Conference.* Montvale, N. J.: AFIPS Press, 1972.

Teitelman, W., *The Interlisp Manual*, Xerox Palo Alto Research Center, 1978

Extending Object Oriented Programming in Smalltalk[1]

Ira P. Goldstein and Daniel G. Bobrow
Xerox Palo Alto Research Center

Abstract:
Smalltalk is an object oriented programming language with behavior invoked by passing messages between objects. Objects with similar behavior are grouped into classes. These classes form a hierarchy. When an object receives a message, the class or one of its superclasses provides the corresponding method to be executed. We have built an experimental Personal Information Environment (PIE) in Smalltalk that extends this paradigm in several ways. A PIE object, called a *node*, can have multiple *perspectives*, each of which provides independent specialized behaviors for the object as a whole, thus providing *multiple inheritance* for nodes. Nodes have *metadescription* to guide viewing of the objects during browsing, provide default values, constrain the values of attributes, and define procedures to be run when values are sought or set. All nodes have *unique names* which allow objects to migrate between users and machines. Finally attribute lookup for nodes is *context sensitive*, thereby allowing alternative descriptions to be created and manipulated.

Object oriented programming is a powerful computational framework for many applications, and Smalltalk [Kay72] is a good example of a language that embodies this framework. Smalltalk is especially excellent for simulation, as one would expect from the fact that Simula [Dahl66] is part of its intellectual genealogy. Objects can represent the participants in a simulation; messages can represent their interactions. However, the 1976 implementation of Smalltalk [Ingalls76] lacks a number of capabilities that we believe can extend its power considerably, especially for applications (including simulation) that occur in the context of an overall design process. These capabilities arise from the assignment of different kinds of description to objects.

(1) *multiple perspectives*: the assignment of more than one point of view that allows inheritance of behavior from independent superclasses.

(2) *metadescription*: the assignment of constraints to attributes that allows the system to check new values and propagate their intended effects.

(3) *identification*: the assignment of identifers, unique across an entire computing community that allow multiple users to manipulate a common set of objects.

(4) *context sensitive description*: the assignment of a situation marker to values that allows alternative descriptions to coexist within a common workspace.

Our overall goal is to crossbreed Smalltalk with recent AI representation languages in order to obtain a hybrid that exhibits the strengths of both lineages. We have pursued this crossbreeding with the help and cooperation of Smalltalk's originators, the Xerox PARC Learning Research Group.

[1] Published in the Proceedings of the 1980 Lisp Conference, Aug. 24-27, 1980, Palo Alto, Cal., pp. 75-81.

This paper first reviews Smalltalk, then discusses our implementation of each of the above capabilities within PIE, a Smalltalk system for representing and manipulating designs. We then describe our experience with PIE applied to software development and technical writing. Our conclusion is that the resulting hybrid is a viable offspring for exploring design problems.

Current Smalltalk

Smalltalk-76 is a programming language based on three metaphors: simulation, communication and classification. An atomic element of the language, termed an *object*, simulates a computer. It has internal state and responds to a set of instructions termed *messages*. An object responds to a message in one or all of the following ways: it changes its internal state; it transmits messages to other objects; it reads or writes an I/O channel such as the display. A sender need have no knowledge of the internal structure of a receiver: it need only know the receiver's message set. For example, there exist display objects such as rectangles that store their position and extent, and respond to messages to move, show and erase themselves.

Each object is associated with a single class. The objects associated with a given class are called its instances. The class owns a dictionary that defines methods for a set of messages. When a message is sent to an instance, that instance in turn requests the appropriate method from its class. The method returned by the class is then applied to the arguments of the message. Smalltalk has predefined classes for Rectangle and BitRect, the latter being a class that includes a state variable for storing the display state of the points enclosed by the rectangle. (Rectangle and BitRect define behavior for classes that interact with a BitMap display).

Classes are hierarchical. A superclass is used to describe the behavior common to several classes. Given superclasses, the protocol for retrieving a method is extended as follows: when a message is sent to an instance, the instance asks its class for the method associated with the message. If the class knows this method directly, it supplies it. If it does not, the class asks its superclass. If the superclass responds with a method, this method is passed back to the object. For example, BitRect is defined as a subclass of Rectangle. A method like **blink** is defined only in Rectangle since its definition, a repetitive invocation of **show** and **erase**, applies to instances of both classes. When **blink** is sent to an instance of BitRect, BitRect finds no associated method, and hence passes the buck to Rectangle, which has the desired definition.

The root of the class hierarchy tree is the class Object. If a request for a method associated with a message comes up to Object, and it does not know the definition of the message, an error occurs.

Although one class may have a great deal in common with the behavior of another, they may still differ on some methods. For example, the **show** method of BitRect differs from the **show** method of Rectangle in that BitRect displays the contents of the rectangle while Rectangle only displays the outline. The desired behavior is achieved by redefining the **show** method in the subclass. Since method retrieval is bottom up, the redefinition in BitRect will dominate the definition in Rectangle for instances of BitRect, yet be invisible to instances of Rectangle.

In addition to a method dictionary, each class also owns a list of variable names. The state of an instance is defined in terms of values for variables with these names as well as values for any variables whose names appear in the superclass chain. For example, instances of BitRect store state for *contents*, the instance variable defined in BitRect, as well as *origin* and *extent*, the instance variables defined in the superclass Rectangle. When any method of an instance is activated by passing it a message, that activation can read and change the values of these instance variables.

A message consists of selectors and arguments. For example, the method with selector **move:** has an argument named ***distance***. A particular call to this method might look like **rect1 move: 3**, where **rect1** is an instance of class Rectangle and the argument ***distance*** is bound to 3.

The three classes, Object, BitRect, and Rectangle, appear in Figure 1 with their associated instance variables and some of their messages. The syntax employed in this and other figures of this article is for didactic purposes only, and does not correspond to Smalltalk syntax for defining classes.

The class Object with instance variables {} and methods **{is: *class*, ...}**

> The class Rectangle, a subclass of Object, with instance variables {*origin, extent*} and methods **{show, erase, move: *distance*, blink, ...}**
>
> > The class BitRect, a subclass of Rectangle, with instance variables {*contents*} and methods **{show, erase, ...}**

Figure 1. A class hierarchy in Smalltalk.

Multiple Inheritance

Smalltalk-76 does not support multiple inheritance. Classes are organized into a strict hierarchy and an instance can be associated with only one class, at a single position in the hierarchy. However, there are situations in which one desires greater descriptive power. For example, consider an environment for hardware design. Objects in this environment represent circuit elements -- resistors, chips, wires, etc. There are at least two points of view from which one may wish to examine these objects. The first is as circuit elements with associated electrical behavior; the second is as display objects that know how to draw pictures of themselves. To choose one point of view as primary, i.e., as the class of the object, and copy methods of the other points of view into this class, is clearly unsatisfactory. Equally unsatisfactory is making one class, say DisplayObject, a subclass of another, say CircuitElement. Such subclassing would be erroneous for other display objects that are not circuit elements. One would really like to be able to have multiple superclasses.

We have explored two designs for multiple inheritance. Both are based on the use of class Node, which defines the basic representational unit. An instance of Node represents some entity: a circuit part, a Smalltalk method, a paragraph of a document. Multiple inheritance is achieved by assigning perspectives to nodes. A perspective is an instance of a class that represents the node from a particular point of view. For example, a node representing a part of a displayed circuit design might have a CircuitElement perspective and a DisplayObject perspective. Class Node defines an instance variable *perspectives* that stores each node's list of perspectives.

In our first design for multiple inheritance, the state of the object was represented entirely in the node. Perspective classes carried no state: they supplied method definitions only. This required that perspectives have backpointers to their node, since their methods manipulated the state variables stored directly in this node.

Smalltalk-76 constrains the number of named state variables to be fixed when the class is created. This is an efficiency constraint: it allows compiled code to reference instance variables by their position in a vector of fixed length rather than by their name. However, in our scheme, we prefer that it be possible to assert or delete perspectives at any time. Hence, an instance of Node cannot know all of its state variables at creation time. Our solution was to give class Node a second state variable whose value was a dictionary keyed by variable names. All variable access went through this dictionary and the dictionary could be modified at run time. Flexibility was obtained at increased computational cost. Figure 2 shows a node representing a resistor in a circuit simulation.

R17, an instance of Node, with

state = {ohms = 100; connection1 = wire6; connection2 = wire8; location = (100,100)}

and *perspectives* = {CircuitElement; DisplayObject}

Figure 2. A Node with multiple perspectives and a common set of state variables.

Our first design for multiple inheritance presumed that a state variable such as *ohms* had a meaning independent of the individual perspectives. Hence, it was sensible for it to be owned by the node itself. All perspectives would reference this single variable when referring to resistance. This proved adequate so long as the system designer knew all of the perspectives that might be associated with a given node, and could ensure this uniformity of intended reference.

When we extended PIE from a single user to a multiple user system, we encountered the difficulty that two users might define perspectives that employed a variable of the same name, although they had different purposes in mind for the variable. For example, one user might define a perspective InventoryPart that used the variable *location* to point to the node representing the bin containing the part, while another user might define a perspective DisplayObject that used a variable of the same name to refer to the location of the part on the screen. The result would be an unintentional clash. In our first implementation, both perspectives would be erroneously referencing the same variable in the common pool of node variables.

Our solution was to eliminate the central database owned by the node in favor of local databases owned by each perspective. This new design achieved privacy at the cost of additional space. Furthermore, it required the user to supply functions for coordinating state variables in different perspectives that represented the same data. However, this seemed unavoidable if we were to open the process of perspective creation to multiple users. Figure 3 illustrates our representation for R17 using this second design. There is no longer a common pool of state variables.

R17, an instance of Node, with perspectives =

{A CircuitElement with *ohms* = 100, *connection1* = wire6, and *connection2* = wire8;

A DisplayObject with *location* = (100, 100);

An InventoryPart with *location* = bin101}.

Figure 3. A node with state distributed among the perspectives.

In both implementations, a message sent to a node consists of the message pattern and the class of the intended perspective. Thus, to obtain the resistance, one would execute the following statement: **(R17 as: Resistor) ohms.** The **as:** message to R17 causes R17 to return the perspective of the desired class, in this case perspective 1. Perspective 1 is then sent the message **ohms.**

An alternative to passing the perspective to the node is to require that the node poll its perspectives for any that understand the message. This approach has the advantage that the source code is more concise, but introduces the necessity to resolve cases in which more than one perspective responds to the message. This resolution could be based on a predefined ordering of the perspectives. We have not adopted this approach for two reasons: (1) In most cases, we have found that the sender knows the point of view that the recipient should employ to understand the message. (2) There is generally no good criterion for declaring that one perspective should dominate another. In those few cases where the intended perspective is not known, we have adopted the procedure that the node polls its perspectives for any that understand the message. If an ambiguity exists, a user interrupt occurs.

The use of perspectives for multiple inheritance is not new. FRL [GoldsteinRoberts77] had a scheme very much like our first implementation; KRL [BobrowWinograd77] has multiple perspectives like those of our second implementation. Both of these implementations were based on the assumption that one wants to make it easy to add a new perspective to an existing instance at any time. We have adopted this assumption in PIE.

An alternative approach is available if one allows multiple inheritance for classes, but not for instances; that is, an instance can be associated with one, and only one, class but a class can have more than one superclass. In this case, it is only in the construction of a class that clashes must be resolved between variable names occurring in more than one superclass. This is the approach employed by Thinglab [Borning77], a multiple inheritance, constraint satisfaction system.

To summarize, perspectives differ from ordinary Smalltalk objects in four respects:

- They expect to be part of a closely interacting system consisting of other perspectives and a central node; hence they come with a backpointer to their node.
- They share some of their state with other perspectives in this system, but maintain a private variable pool for their own purposes.
- They are intended to represent a point of view on an entity, rather than the entity itself.
- They can be attached at any time to a node. It is not necessary to assign all perspectives when the node is created.

Metadescription

Perspectives express different descriptions of the entity represented by the node. Changing these descriptions can lead to inconsistencies. We handle this problem by providing the node with various kinds of information about itself. We term this information *metadescription* to distinguish it from the primary description implicit in the node regarding the entity in the world that it represents. For a general discussion of metadescription see [BobrowWinograd77].

The first kind of metadescription we supply is knowledge of the expected type of an attribute. This information is supplied in a *constraint* dictionary. For each attribute, the constraint dictionary supplies an expression that describes the class of the expected value. For example, a value for the *ohms* attribute of the resistor perspective is expected to be of class Integer, while the value of *connection1* is expected to be a node with an associated Wire perspective. This mechanism takes care of simple unary constraints.

Secondly, we supply procedures that are triggered by the retrieval or storage of a value. These procedures typically serve to maintain consistency between dependent attributes. For example, if a change is made by the user in the connectivity of the displayed schematic, then procedures attached to the instance variables being altered can update the circuit element perspectives to correspond to the new display linkages. Similarly, attached procedures can update the inventory perspective as parts are added or deleted from the design.

To take care of less formal cases in which only the user knows what to do, we have dependency notification. A dependency list can be added to the metadescriptions of a node. The user supplies this list for a node or attribute, but does not inform the system of what actions to take if a change is made. Consequently, when the node is altered, the user is reminded of these dependencies by attached procedures, but no automatic actions are taken. For example, the user might place a dependency link between a capacitor and an

inductor to serve as a reminder that the two elements are intended to operate together as a tuned circuit.

A more powerful dependency model replaces the dependency list with a pointer to a node with a *contract* perspective. The contract perspective contains a list of participants and, at a minimum, an English statement of the contract. We plan to formalize this contract progressively. For the electrical world, contracts might include the mathematical formulae that describe the circuit. For the programming domain, contracts would include the expected type of a variable. See [Borning77] for a general study of constraints as the basis of a Smalltalk system and [SussmanStallman77] for a more detailed study of dependency relations in circuits.

Unique identification

The object metaphor suggests that each user of Smalltalk has his or her own unique set of objects. I run on my computer; you on yours. But the description metaphor suggests that you and I may well be working on the same set of descriptions. Hence, we need a way to separate my contributions from yours but, at the same time, to clearly identify that they are being generated to describe the same topic. To solve the first problem, we employ machinery to separate descriptions into contexts. This is discussed in the next section. To solve the second problem, we employ unique identifiers.

Consider the following scenario: I create a set of nodes representing a design and deliver these nodes to your environment for subsequent development. To accomplish this delivery, I generate a set of descriptions that can be used to recreate a set of Smalltalk objects with the same state. This was our first implementation.

However, the following difficulty arises with this scheme. You modify and supplement these nodes, and then generate a new set of descriptions. But when I reread them into my environment, how can I determine which of these descriptions should be added to existing nodes, rather than used to create a new collection of nodes?

Recognizing that two sets of descriptions describe the same intended object is a difficult problem. However, in this special case, the problem can be solved easily. A node is assigned a unique identifier when created. This identifier travels to the consumer when descriptions are generated. The consumer checks to see if a node already exists with the identifer. If so, the descriptions of this node are appended to those already there. If no such node exists, a new node with this unique identifier is created.

The computational cost of this scheme is not excessive, since the consuming environment can maintain a table that associates identifiers with existing nodes within that environment. Hence, in consuming a set of descriptions, it is necessary only to check this table to find

the preexisting node, if any. This is similar to the way Lisp atoms, or Smalltalk unique identifiers are implemented, with the important difference that the identifers are generated by the machine in such a way that two users can never create identical identifers. In fact, the identifiers consist of an encoding of the time and machine of creation.

Contextualization

From a design standpoint, it is important that alternative descriptions be able to coexist in the same environment at one time. Alternatives arise from a designer exploring different plans to achieve his goals; or from the interactions of several designers on a joint project. For example, one designer may propose a particular circuit to realize the specifications of a module; while another designer may propose an entirely different circuit to accomplish the same goals. In a design environment, descriptions are sensitive to who has created them and for what purpose. A user must be able to examine and manipulate such descriptions from different points of view.

To implement context sensitive descriptions, we have altered the behavior of the dictionaries that store the attribute/value pairs of perspectives. In Smalltalk-76, a dictionary is a list of attributes and an associated list of values. We have replaced the value associated with the attribute with another level of dictionary. This level of dictionary associates a *layer marker* with different values. The *layer marker* is a tag for the situation in which the value was supplied. Figure 4 shows a partial view of a layer structured description of R17.

R17, an instance of Node, with perspectives =

{A CircuitElement with *ohms* = [<layer1 100>], *connection1* = [<layer1 wire6>], and *connection2* = [<layer1 wire8> <layer2 wire13];

A DisplayObject with *location* = [<layer1 (100,100)> <layer2 (300, 300)]}

Figure 4. A partial view of the node *R17* with layers indicated. Layer1 stores the original design. Layer2 stores a change in the display location of the resistor and an associated change in the circuit connectivity.

Storage and retrieval is therefore situation dependent. Storage is done with respect to a layer. Retrieval is done with respect to a sequence of layers. The retrieval algorithm checks the layers in order for a value, returning the first value in the layer sequence. This layer sequence is called a *context.* These notions of layer and context are derived from Conniver [Sussman72]. There are minor differences in the implementation, and major

differences in the use of the mechanism. This is discussed in more detail in [BobrowGoldstein80].

Values stored in a layer represent a coordinated set of values. Suppose the connectivity of R17 in a circuit is changed as a display object. An attached procedure (or the user) might make the corresponding change in the circuit simulation. These two changes are meant to be coordinated, and are therefore placed in the same layer. By "coordinated", we mean that one sees either both changes or neither in any view of the circuit. All retrievals in a context will get either both these values (if the layer is included in the context) or neither.

The flexibility to represent alternative descriptions in layers comes at the cost of increased complexity. We have designed several display interfaces to explore different mechanisms for simplifying the presentation of this inherently more complex database. For example, one interface provides a way for a user to view two different contexts simultaneously with differences between the two highlighted. We have also explored the use of metadescription to default some of the contextual choices that would otherwise fall on the user, e.g., selecting the default layer for assertions and the default context for retrieval. Finally, we have supplied commands that suppress the context machinery. The user stores and retrieves state in a context free fashion. This is faster, occupies less space, and has no cognitive overhead for remembering alternative contexts. But the user no longer can explore alternatives or separate his contributions from those of a codesigner. All three of these strategies have proved useful in some circumstances, but it remains an important research goal to make the context machinery available to the user in a convenient fashion.

Use of PIE

The PIE system provides an environment for doing software development. Perspectives are provided for representing Smalltalk classes and methods. A user of PIE is therefore able to build a collection of nodes that represent a software system. Unique identifers and contexts allow users to engage in cooperative design and to explore alternatives. When a design is complete, it can be installed in Smalltalk by generating executable code from the node descriptions. Other designs described in separate contexts remain unaffected by this installation. Metadescription is used to express type knowledge regarding method variables, thereby obtaining the strengths of a typed language while still preserving the underlying flexibility of an untyped interpreter.

The utility of this descriptive base for developing software is illustrated by the following experiments: (1) We have successfully redesigned PIE's user interface within PIE. Ordinarily, such redesigns would clobber the coding environment itself, but the separation between description and installed code prevents such conflict. (2) We are able to describe a method as belonging to multiple classes, despite the fact that the Smalltalk kernel does not allow this. At the descriptive level, a node representing a method may be linked to

more than one class. Within Smalltalk itself, a method is local to a class. For compatibility, all that is necessary is that installation of the description involves placing copies of the compiled code in each class. However, at the descriptive level, the designer can treat the method as a single integral entity; editing it affects its occurrence in all of its classes. (3) Multiple perspectives and metadescription support improved browsing and prettyprinting of code, thereby improving the user's ability to examine his designs. (4) Unique identifiers and contexts provide a mechanism for generating an incremental system release. The new system is created by transmitting a layer with the changes to a consumer and then asking the consumer to do a reinstallation. Separating release changes into layers allows the consumer to examine the alterations of the release and exercise some choice regarding which parts he wishes to accept, before performing the reinstallation.

The same machinery has also been used to support a document design environment. Nodes are used to represent the structure of the document; i.e., the document is a tree of nodes whose root represents the document as a whole and whose terminals are the individual paragraphs. The nonterminals of the tree are chapters, sections and subsections. Again, contexts and identifers facilitate coauthoring and exploring alternative organizations, two capabilities not well supported by present text editing environments. Metadescription can be used to express formatting constraints. Multiple perspectives allow a paper to appear as either an abstract, a citation, a bibliographic reference, the outline for a lecture, or a formatted document, depending on the desired point of view.

The PIE system code occupies approximately 200 kilobytes and 100 pages of listing in a Smalltalk system of approximately 1 megabyte and 1000 pages of listing. Storage space for nodes grows as layers increase, and previous or alternative values for attributes of nodes are stored. Retrieval time increases with the number of layers in the retrieval context. However, neither price has proved exorbitant since PIE has been used largely as an interactive design tool. In this application, time is primarily limited by the responses of the user, i.e. there is more thinking than computing. Space is released when the design is complete and an installed package of code is created.

Conclusion

We conclude by reconsidering Smalltalk's underlying metaphors of simulation, communication and classification in the light of our addition of descriptive machinery to the language.

In Smalltalk-76, objects simulate computers and therefore have a fixed identity. They use a predetermined set of state variables and respond to a fixed set of messages. In PIE, nodes have a flexible set of state variables which can grow or shrink as the attributes of individual perspectives are changed. Furthermore, the message set can change as new perspectives are supplied or old perspectives deleted. Nodes are more analogous to an evolving

biological species than to an inanimate computer. At any moment in time, a member of the species has a fixed anatomy and physiology. Over time, however, both the anatomy and physiology evolve.

In Smalltalk-76, objects have an unambiguous message semantics. A message is sent to an object and that object, in turn, requests the appropriate method from its class. In PIE, nodes have multiple perspectives and more than one perspective may supply a method for a given message. The user must specify the perspective, or allow the node to decide. Communication is still an applicable metaphor, but the complexity of communication has increased as the underlying objects have moved from a monolithic to a pluralistic society.

In Smalltalk-76, objects participate in a simple, hierarchical classification scheme. In PIE, nodes are the locus of a set of descriptions and behaviors, each generated from a different point of view. Classification, with its implication of simple hierarchy, has been replaced by description, with its more open-ended connotation.

Thus, the evolution from Smalltalk to PIE has produced a change in the behavior of the basic computing element. In Smalltalk, objects have a fixed structure and engage in communication based on a simple classification scheme. In PIE, nodes have an evolving structure and engage in a more complex communication based on the use of descriptions. We believe that this evolution yields a more flexible environment for exploring design problems.

References

Bobrow, Daniel G. and Goldstein, I.P. "Representing Design Alternatives". *Proceedings of the AISB Conference*, Amsterdam, 1980

Bobrow, Daniel G., Terry Winograd, and the KRL Research Group. "Experience with KRL-0: One Cycle of a Knowledge Representation Language". *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, August 1977, pp. 213-222.

Borning, A. "ThingLab -- an Object-Oriented System for Building Simulations Using Constraints". *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, 1977, pp. 497-498

Dahl, O.J. and Nygaard, K. "SIMULA--an ALGOL-Based Simulation Language", *CACM* 9, September 1966, pp. 671-678.

Goldstein, I.P. and Roberts, R.B. "NUDGE, A Knowledge-Based Scheduling Program". *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, 1977, pp. 257-263.

Ingalls, Daniel H. "The Smalltalk-76 Programming System: Design and Implementation". *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, January 1978, pp. 9-16.

Kay, A. "A Personal Computer for Children of All Ages". *Proceedings of the ACM National Conference,* August 1972.

Sussman, G. and McDermott, D. "From PLANNER to CONNIVER - A Genetic Approach". *Fall Joint Computer Conference*. Montvale, New Jersey, 1972.

Sussman, G. and Stallman, R. "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Alded Circuit Analysis". *Artificial Intelligence, 9*, 1977, pp. 135-196.

Representing Design Alternatives[1]

Daniel G. Bobrow and Ira P. Goldstein
Xerox Palo Alto Research Center
Palo Alto, California 94304, U.S.A

Abstract:

Artificial intelligence systems are complex designed artifacts. Techniques used in AI systems to describe structures and to represent alternatives can be used to support the design of the systems themselves. PIE is an experimental personal information environment which provides users with descriptive structures for programs and documents. In PIE, alternative designs for programs and documents are simultaneously viewable in the system through the use of a context structured database. This short paper gives an overview of how the use of these facilities improves the design environment for builders of software systems.

Introduction

A major activity in artificial intelligence research is the design of complex systems. Yet most software environments do not support this activity well. They do not allow within the system description of different properties of a design nor the flexible examination of alternative designs. All designers create alternative solutions, develop them to various degrees, compare their properties, then choose among them. Yet most software environments do not allow alternative definitions of procedures and data structures to exist simultaneously; nor do they provide a representation for the evolution of a particular set of definitions across time. It is our hypothesis that a context-structured database can substantially improve the programmer's ability to manage the evolution of his software designs.

Present computing environments support the creation of alternative designs only with file services. Typically users record significant alternatives in files of different names; the evolution of a given alternative is recorded in files of the same name with different version numbers. We contend that this use of files provides both an impoverished structure as well as an inflexible one. The poverty is a result of the fact that file names are simply a limited length sequence of characters, hardly an adequate scheme to describe the purpose and contents of a file, and its relation to other files. It can be an adequate reminder to the originator of the name, but is often opaque to a new reader. The rigidity is a reflection of the fact that one typically cannot use parts of files as part of a new composite design, except by tedious text editing. Finally, the most serious limitation is that files are "off-line" in the sense that the alternative designs are not stored within the computing environment in a form that can be easily manipulated by the

[1] Published in the Proceedings, Artificial Intelligence and Simulation of Behavior Conference, July, 1980, Amsterdam. A more extended discussion of this research can be found in Goldstein & Bobrow [80].

programmer. Although Interlisp [Teitelman, 78] provides some facilities for manipulating pieces of a file (e.g. individual function definitions), it still suffers from the "off-line" limitation.

To ameliorate this software bottleneck, we have constructed a computing environment in which "on-line" descriptions of alternative software designs can be readily created and manipulated. We use a context-structured description-centered database to describe code. Such databases have been explored in artificial intelligence research for over a decade as a mechanism to represent alternative world views. [e.g. Hewitt, 71; Sussman & McDermott, 72].

Our application of this machinery is novel in several respects. (1) Previous applications have focussed on the use of such databases by mechanical problem solvers. We are exploring the use of such databases in a mixed-initiative fashion with the user primarily responsible for their creation and maintenance. (2) Previous applications have always demanded a uniform overhead in space and time for adopting the context machinery. We are exploring configurations for a design environment that allow the programmer to trade flexibility for efficiency, decreasing the system's investment in tracking the evolution of particular parts of a design at the price of not being able to represent alternatives simultaneously in primary memory. Thus, employing the design environment is not an all or nothing choice for the user. (3) Previous applications have been to problems of limited complexity. In our application of context structured databases to software design, we are exploring their utility in a world several orders of magnitude more complex.

To understand the pros and cons of context structured environments for software design, we have implemented a prototype environment and conducted several experiments. The environment is called PIE, an acronym for personal information environment. PIE allows the user to build context sensitive descriptions of code, documents, and, indeed, any object for which a machine representation exists. PIE has been employed (1) to allow a programmer to create alternative software designs, examine their properties, then choose one as the production version, (2) to coordinate the interactive design of two programmers, and (3) to coordinate the documentation and definitions of an evolving package of code.

The Smalltalk environment

To describe PIE further, we must first introduce Smalltalk [Ingalls, 78; Kay, 74], the programming environment in which it has been implemented. Smalltalk is an object-oriented programming language. (See Dahl & Nygaard [66] on Simula and Hewitt et al [73] on "actors" for related work on such programming languages). Behavior arises from the transmission of messages between objects. Each object is, in essence, a

simulation of a computer. It can respond to some number of messages and it maintains its own state between message invocations.

The message set of an object is specified by Smalltalk's class structure. Each object is an instance of a class. When a message is sent to the object, it asks its class for the method associated with that message. The class either contains the definition directly, or if not, passes the request to its superclass. For the object to understand the message, its definition must occur somewhere in this superclass chain. Thus, objects of the same class are analogous to computer products of the same model.

Figure 1 shows a fragment of the definition of a Smalltalk class for **Spaceship**. The fragment shown indicates that instances of **Spaceship** understand messages that simulate motion and collision and that each instance carries its own private state regarding its position and velocity.

```
Class new title: Spaceship
    superClass: Object "class Object is the root of the superClass hierarchy."
    declare: 'allSpaceships' "a class variable --shared by all instances"
    fields: 'position velocity' "instance variables -- each instance has private versions of these"

Moving "methods are divided into 'protocols' -- this one is called Moving"

accelerate: dv "dv is the argument of the method with selector accelerate"
    [velocity←velocity + dv]

move [position←position + velocity. "points understand the message +"
        self crashes => "self refers to this instance. => indicates a conditional expression"
            [↑ self explode] "if condition is true, move returns with value of self explode"
        self display. "done if condition is false -- display is a message this instance understands"]

Collisions "another protocol"

 crashes | ship "ship is a local variable for the activation"
"This assumes that all ships are of unit size, and collide only when at the same point"
[for: ship from: allSpaceships do: [ ship collideAt: position =>[↑true]].↑false]

 collideAt: place
    "a method to test if I collide with another object at place."
[position = place =>[↑true] ↑false]
```

Figure 1: Partial Definition of a Smalltalk class

We chose Smalltalk over Lisp, the usual vehicle for AI research, because Smalltalk has a superior set of interactive display facilities. DLISP [Teitelman, 77] provides enough capabilities we believe, but was not available on the same fast hardware. These interactive display facilities were of critical importance to allow the functionality of the design environment to be delivered to a user. No matter how powerful the design tools,

no experiments would have been possible with an interface based on an inadequate communication channel. Using Smalltalk, however, has required that we reimplement machinery common to such AI languages as FRL [Goldstein & Roberts, 77] and KRL [Bobrow et al, 77]. This has proved straightforward because the object oriented structure of Smalltalk is congenial to the frame-based viewpoint of a AI representation languages.

The PIE environment

To describe Smalltalk code, we created a class of Smalltalk objects called *nodes*. Nodes are analogous to KRL units, or FRL frames: they consist of a set of attribute value pairs with support for attached procedures, the use of defaults, meta-descriptions and inheritance.

PIE provides convenient ways of viewing relationships between nodes, and viewing and changing the properties of nodes. One can automatically create nodes which describe existing pieces of the Smalltalk system, and conversely, make the system congruent with a description of it. Node23 in Figure 2 is a description that might have been been computed from one method of the Smalltalk code shown in Figure 1.

```
Node23
   class          Node17        "Node17 is the node describing the class Spaceship"
   selector       'crashes      "This is a unique string -- like a Lisp Atom"
   localVariables ('ship)       "This is a set of unique strings"
   variablesUsed  ('ship 'allSpaceships 'position 'mySize)
   methodBody     "This is an editable paragraph"
      [for: ship from: allSpaceships
       do: [ ship collideAt: position =>[↑true]].↑false]
   comment
      'This assumes that all ships are of unit size, and collide
      only when at the same point'
```

Figure 2. A node describing the method for crashes

In PIE, changing the values of any of these attributes does not automatically change the object being described by the node. The node describes an intended object in the system, not necessarily the version that exists in the system. This is worth emphasizing as one of the principles characterizing our point of view towards the design process.

> ★ **The Description Principle:** In a system there should exist a descriptive level at which objects can be described without actually affecting the objects themselves.

Representing alternative designs

Using node structure, there are two distinct ways to have alternative descriptions of the same object: coreference and context. We have explored both, with our current preference being for the use of contexts.

Coreference uses separate nodes to describe separate alternatives. In Figure 3, **Node25** is a description of an alternative version of **crashes**. The intended identity of the **Node23** and **Node25** (they are both are describing the same object) is made explicit with the coreferentNodes attribute.

```
Node25
   class        Node18          "Node18 is the node describing the class Spaceship which differs
                                 from Node17 in having an additional instance variable -- mySize"
   selector     'crashes
   localVariables   ('ship)
   variablesUsed    ('ship 'allSpaceships 'position 'mySize)
   methodBody       "a different method body"
      [for: ship from: allSpaceships
         do: [ ship collideAt: position of: mySize =>[↑true]].↑false]
   comment 'Uses mySize for each ship to determine overlap'
   coreferentNodes (Node23)
```

Figure 3. An alternative method for crashes

However, coreference has certain difficulties. The first is that it does not represent the manner in which two descriptions may differ on some attributes but otherwise be identical. The second is that the coordination of the choice of Node23 vs. Node25 and other choices in the system for consistency is not expressed. For this reason we have chosen to explore another way of expressing alternatives.

In this second method, all descriptions (values of attributes) of any node are relative to a context. *Context* as we use the term extends the notion of context as used in Conniver [Sussman & McDermott, 72], and has certain similarities to the vistas of partitioned semantic nets [Hendrix, 75].

> ★ **The Context Principle:** All attribute-values in the system are relative to a context, and alternatives in a system are expressed by alternative contexts.

When one retrieves the values of attributes of a node, one does so in a particular context, and only the values assigned in that context are visible.

Incremental design

Design involves more than the consideration of alternatives. It also involves the incremental development of a single alternative. Every programmer is aware that software has a life cycle: following its birth, it undergoes progressive refinement in response to changing external requirements. PIE supports the incremental modification of a design by providing a fine structure to contexts that we have not, as yet, discussed.

A context is structured as a sequence of layers. It is these layers that allow the state of a context to evolve. The assignment of a value to a property is done in a particular layer. Thus the assertion that a particular procedure has a certain source code definition is made in a layer. Retrieval from a context is done by looking up the value of an attribute, layer by layer. If a value is asserted for the attribute in the first layer of the context, then this value is returned. If not, the next layer is examined. This process is repeated until the layers are exhausted.

Figure 4 shows a layer C containing some coordinated changes to the spaceship class of Figure 1. This layer contains those changes necessary to allow the class to use size information in determining collisions. In a context which contained this layer dominating those containing the information implicit in Figure 1, the changes would be visible. Those attribute-values such as the superclass of Spaceship that are not contained in layer C would be found in less dominant layers.

```
Node17 "the node for the class Spaceship"
   fields:        ('position 'velocity 'mySize)    "a change in a declaration"
   methods  (... Node23 Node27 ...)

Node23 "the node for the method crashes"
   methodBody
      [for: ship from: allSpaceships
       do: [ ship collideAt: position of: mySize =>[↑true]].↑false]

Node27 "the node for the method that tests for a collision"
   selector  'collideAt:of:
   methodBody
      [(position + mySize>place-size)and:(position-mySize<place + size) =>[↑true]
       ↑false]
```

Figure 4. Layer C, containing coordinated changes to use mySize

---

Figure 5 shows several spaceship nodes in which the values of attributes have not been filtered by a context sensitive lookup. Instead, we see the underlying data structure, which is an association list of layers and values. Layer B is the base layer in which all the nodes were presumed to have been originally defined for this example.

```
Node17 "the node for the class Spaceship"
    fields:     LayerB ('position 'velocity)
                LayerC ('position 'velocity 'mySize)
Node23 "the node for the method crashes"
    methodBody
     LayerB
         [for: ship from: allSpaceships
          do: [ ship collideAt: position  =>[↑true]].↑false]

     LayerC
         [for: ship from: allSpaceships
          do: [ ship collideAt: position of: mySize =>[↑true]].↑false]
```

Figure 5. An unlayered view of node structure

Extending a context by creating a new layer is an operation that is sometimes done by the system, and sometimes by the user. The current PIE system adds a layer to a context each time the context is modified in a new session. Thus, a user can easily back up to the state of a design during a previous working session. The user can create layers at will. This may be done when he or she feels that a given groups of changes should be coordinated. Typically, the user will group dependent changes in the same layer.

Given the existence of layers, a complex design developed over many stages can be summarized into a single new layer. The old layers, reflecting past choices, can then be deleted. Thus, the designer, if he wishes, can compress the past, achieving a more compact representation at the price of no longer representing the dynamics of the design.

Coordinating designs

So far we have emphasized that aspect of design which consists of a single individual manipulating alternatives. A complementary facet of the design process involves merging two partial designs. This task inevitably arises when the design process is undertaken by a team rather than an individual. To coordinate partial designs, one needs an environment with these properties: (1) *non-interference*. Two designs may overlap. It must be possible to examine the overlap without the designs overwriting one another. (2) *incompleteness*. It must not be necessary for a design to be complete before it is examined. (3) *merging*. It must be convenient to create a common design from the individual contributions. It was encouraging for us to learn that the context/layer machinery created to manage alternatives lent itself well to meeting these requirements for coordinating partial designs.

Non-interference between the overlap of two partial designs was accomplished by adopting the convention that different designers place their contributions in separate layers. Thus, where an overlap occurred, the divergent values for some common attributes were separated by distinct layers. Handling incomplete designs of software was facilitated by the distinction between intensional node descriptions and the actual code definitions. Since the node descriptions were not installed code, they could be partial and hence non-executable with no difficulty.

Merging two designs can be viewed as a process that creates a new layer into which are placed the desired values for attributes as selected from two or more competing contexts. It is hence very much like the summarization process described earlier, but it is relative to more than one context and requires user interaction. For complex designs, the merge process is, of course, non-trivial. We do not, and indeed cannot, claim that PIE eliminates this complexity. What it does provides is a more finely grained descriptive structure than files in which to manipulate the pieces of the design.

Understanding how to merge two designs is facilitated by examining commentary supplied by the designers regarding the rationale of their choices. But this raises the classic software problem of coordinating documentation with design. Fortunately no additional machinery is required in PIE to address this problem. Commentary such as the rationale of a procedure, or its dependencies on other procedures, can be stored as attribute value pairs within the node describing the procedure in question. A request to be informed of the rationale of some change is answered by fetching this information from the same layer as the one which records the change, thus keeping them coordinated. Figure 4 shows how the rationales of various method definitions are recorded in the layer along with the altered definitions.

Complexity.

We claimed in the introduction that PIE copes with problems several orders of magnitude more complex than those previously represented in AI systems such as Conniver. By complexity we mean both the size of the data base in the system, and the variety of operations done on contexts. The Conniver database was never efficient enough to implement any useable subsystems. McDermott's [McDermott, 74] examination of the Monkey and Bananas problem within Conniver exercised it to its limit.

PIE is able to build a context sensitive description of any class within Smalltalk. Thus, it can be applied to any programming problem that a Smalltalk programmer undertakes. This is analogous to using Conniver to build a programmer's interface to Lisp. Attacking problems of this size is, in part, possible because we have more computational resources than were available in the early 70's. PIE runs as a stand alone job on a processor with at least the power of a KA10. However, it is also possible because we have implemented

machinery to allow the programmer to move between context sensitive and context free descriptions at will. Thus, there is a more congenial marriage between PIE and Smalltalk than there was between Lisp and Conniver. This is discussed in the next section.

An interesting side effect of PIE's ability to describe any code within Smalltalk is that it can and has been used to describe itself. Thus, PIE's present capabilities have passed the test of being sufficiently powerful to support its own development, for example, by allowing us to examine alternative implementations of the PIE user interface within PIE.

Efficiency versus Flexibility

PIE allows the user to trade flexibility for efficiency. At one extreme, the user can employ standard Smalltalk mechanisms for defining new code. If this route is chosen, then no evolutionary history is maintained, and no context overhead is paid. However, if the user wishes to pay the price of some decrease in efficiency of storage and retrieval time, then he can first build a set of nodes describing Smalltalk code, then continue his development in a context structured fashion. From this point forward, the evolutionary history is maintained. If the user reaches the point where he once again prefers efficiency to flexibility, the context definitions can be converted to pure Smalltalk and the layers deleted. If desired, the user can first store the layers remotely, preserving the ability to recreate the context description later. All these facilities are curently implemented.

This discussion suggests how a central design facility can serve as the nucleus of a network of remote servers that provide current packages to users. Periodically, the design server can release new layers to these servers with updates to particular designs. The servers can then generate new Smalltalk versions and release these designs to clients. Clients who wish to know what has changed, can get a description from the new layer.

Interaction

PIE's ability to represent non-trivial alternative designs raises deep problems related to the user interface. How can we make available this power in a useable form? What are the cognitive requirements of the programmer? Presently we are employing an interface modelled on the standard Smalltalk interface for examining and altering code. This interface, called the browser, displays a hierarchy of descriptions of Smalltalk code to the user. The user can examine any method by a process of selection that specifies first a category of classes, then a particular class, then a protocol of methods within the class, and finally a particular method. This scheme of organizing code into a four-level taxonomy has been adopted in PIE to minimize the overhead for a Smalltalk user learning to employ the PIE environment. However, PIE makes this classification context

dependent. As with the standard Smalltalk browser, the user can alter the definitions of any object viewed. But these alterations are made in the dominant layer of the associated context, and do not affect the Smalltalk kernel itself, whereas making changes with the standard Smalltalk browser forces immediate incorporation of any changes.

Research is needed to explore whether this interface is adequate given the increased complexity of a context structured environment. In Smalltalk, the hierarchy of code definitions is the primary structural organization. In PIE, this hierarchy is now context dependent. Has this additional complexity made the Smalltalk organization inadequate? Will we need a classification scheme with more levels of division, or will some other kind of organization be appropriate? Just one of the problems that we will have to consider is that in a design environment, there is no need for a particular method description to be associated with only a single class, even though the actual Smalltalk system requires that the method be separately compiled for each class to which it belongs. Hence, a strict hierarchy is obviously inadequate.

Conclusions

This paper presents only a sketch of the PIE system; our research is reported in greater detail in Goldstein & Bobrow [80]. We have not discussed here issues in the design of the user interface, although a successful interface is critical to delivery of these capabilities to the user. We only suggest here that layered networks are applicable to more than software: an extended example in cooperative writing of a document is given in the larger work. Finally, the system has as yet had only limited use. We do not know which features will be used most, which need to be automated to be helpful, and which may prove to be too complex to be useful. Recording and analyzing this experience is an important part of our research program.

A major theme of Artificial Intelligence research has been the development of languages to describe complex evolving structures. In general, these structures have been the belief structures of an artificial being about some subject matter (e.g., the SRI consultant's [Hart, 75] beliefs about the state of a water pump being constructed, or SAM's [Schank et al, 75] beliefs about what went on in a story it just read). We have been exploring the premise that these techniques can be used to describe the complex evolving structure of a software system, and as such can provide aids to the designer of such a system. One use of artificial intelligence is to amplify human intelligence. We suggest that the (recursive) application of AI techniques to AI can have a powerful effect on the development of the field.

References

Bobrow, Daniel G., Winograd, Terry, and the KRL Research Group, Experience with KRL-0: One cycle of a knowledge representation language, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, MA: 1977, 213-222.

Dahl, O.J., and Nygaard, K., SIMULA—an ALGOL-Based Simulation Language, *CACM* 9, September 1966, 671-678.

Goldstein, I.P. and Bobrow, D.G., *A layered approach to software design*, Palo Alto, CA: Xerox Palo Alto Research Center, Computer Science Laboratory, 1980, in preparation.

Goldstein, I.P. and Roberts, R.B., NUDGE, A knowledge-based scheduling program, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, MA: 1977, 257-263.

Hart, P., Progress on a computer based consultant, *Advance papers of the fourth international joint conference on artificial intelligence*, Tbilisi: 1975, 831-841.

Hendrix, Gary G., Expanding the utility of semantic networks through partitioning, *Advance papers of the fourth international joint conference on artificial intelligence*, Tbilisi: 1975, 115-121.

Hewitt, C., Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot, Ph.D. Thesis, June 1971 (Reprinted in AI-TR-258 MIT-AI Laboratory, April 1972.)

Hewitt, C., Bishop, P., and Steiger, R., A universal modular ACTOR formalism for artificial intelligence, *Proceedings of the Third International Joint Conference on Artificial Intelligence*, 1973, 235-245.

Ingalls, Daniel H., The Smalltalk-76 Programming System: Design and Implementation, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, AZ: January 1978, 9-16.

Kay, A., *SMALLTALK, A communication medium for children of all ages*, Palo Alto, CA: Xerox Palo Alto Research Center, Systems Science Laboratory, 1974.

McDermott, D.V., *Assimilation of new information by a natural language-understanding system*, AI-TR-291 MIT-AI Laboratory, February 1974.

Schank, Roger, and the Yale AI Project, *SAM—A story understander*, Yale University, Computer Science Research Report #43, August 1975.

Sussman, G., and McDermott, D., From PLANNER to CONNIVER—A genetic approach, *Fall Joint Computer Conference*, Montvale, NJ: AFIPS Press, 1972.

Teitelman, W., *Interlisp reference manual*, Palo Alto, CA: Xerox Palo Alto Research Center, Computer Science Laboratory, 1978.

Teitelman, W., A display oriented programmer's assistant, *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, MA: 1977.

Browsing in a Programming Environment[1]

Ira P. Goldstein and Daniel G. Bobrow
Xerox Palo Alto Research Center
Palo Alto, California 94304, U.S.A

Abstract: Programming today takes place in a complex environment containing a large collection of previously defined packages, routines and data structures. A browser is a display based interface which allows a user to examine this complex environment without prior knowledge of its exact structure. The Smalltalk browser allows perusal of a four level information net of code in the system. Our extension, the PIE browser, allows examination of an arbitrarily deep net which can describe many aspects of the programmer's environment, including messages between programmers and design notes, as well as code and documentation. In this paper, we provide a general framework for describing and evaluating browsers, and use it to highlight the strengths and weaknesses of these two examples.

Introduction

A browser is a software development tool that supports the incremental examination of a system by accessing some kind of information network. A user starts at a canonical place in this network, and selects entities that represent parts of the system. This causes the browser to display the substructure of the system connected to the selected entity, and some information about that entity. In this manner, a browser can be employed to engage in a hierarchical examination of a system by proceeding level by level from subsystem to module to sub-module, until the terminal structure—possibly individual procedure definitions—is reached. In addition, the browser allows a user to add or alter structure at any point in this examination process.

Most programming environments allow a user to retrieve and manipulate different parts of a software system, if the programmer knows their exact name and location; but do not support well the examination of structure whose exact description the programmer does not know. In such situations, the programmer will frequently be reduced to examining file directories, hoping that the file names reveal the contents of the file. A browser seeks to ameliorate this difficulty by allowing a user to examine different regions of a software system based on their general classification. Thus, the underlying database imposes an organization on the software system analogous to the organization imposed on a library by the Dewey decimal system. The browser provides an electronic analog of moving from a general classification to the stacks, and then subsequently browsing there.

Browsers were introduced into Smalltalk by Larry Tesler in 1977, and have since become a mainstay of the Smalltalk programming environment. (The general nature and goals of Smalltalk are described in Kay [77]; the 1976 implementation in Ingalls [78]; and the Smalltalk browser in Goldberg and Robson [79].) In recent research, we have extended the simple, hierarchical system model provided by Smalltalk and developed a generalization of the Smalltalk browser to manipulate these richer descriptions [GoldsteinBobrow80a,b,c;

[1] To appear in the proceedings of the 14th Hawaii Conference on System Science, January, 1981.

BobrowGoldstein80]. We have dubbed this extended environment PIE, an acronym for Personal Information Environment.

In the next two sections, we describe the Smalltalk system model and its associated browser. This is followed by two sections that describe the PIE system model and its browser. The following nine questions are used as a framework for comparing the functionality of these two browsers.

1) *Overview*: How much of the information network can the user see at one time?

2) *Path:* What part of his path to the current position is visible to the user?

3) *Presentation:* What should be displayed on the screen for each selection?

4) *Operations:* What operations can be performed on the view for each selection?

5) *Multiple Views:* Can more than one view of the network be seen? Are they all of the same form?

6) *Consistency:* What guarantees of consistency are there between multiple views?

7) *Alternative Access*: Can the user find a known entity in the system without tracking through the network?

8) *Integration*: Is the data environment integrated with the operational environment of the underlying system?

9) *Changeability*: Can the user change the format in which information is displayed?

The Smalltalk System Model

Smalltalk is an object oriented programming system, where behavior arises from the transmission of messages between objects. Objects are grouped into *classes*, all of which have identical internal structure, and respond to the same set of messages. An object is like a simulation of a computer; it can respond to set of instructions, maintaining its state between invocations. Smalltalk generalizes Simula67 [Birtwistle73] and is related to the Actor languages developed by C. Hewitt [Hewitt73].

The Smalltalk information network partitions all classes into *categories* for ease of access. These categories are not mutually exclusive, although multiple category membership is generally avoided. (Since classes are stored in files corresponding to their category, multiple category membership gives rise to redundant storage and possible inconsistencies between versions.) A *method* is the code which implements the class specific response to a message. The set of methods of each class is partitioned into mutually exclusive groups called *protocols*. Neither categories nor protocols has any significance for the Smalltalk interpreter; rather they are artifacts of the desire to browse through the system.

There is a subclass hierarchy in the Smalltalk system that does have semantic significance. A class can inherit behavior and structural description from another class called its superclass. All instances of a particular class contain the fields specified in the superclass. If the subclass has no specialized behavior (method) for responding to a particular message, it will request that its superclass respond to the message. This inheritance is a very powerful way of sharing behavior.

The Smalltalk Browser

Figure 1 shows a sequence of views of a Smalltalk browser as a user selects a path through the network. The browser is a rectangular region on the display screen called a *window* and is built from 6 sub-windows called *panes*. The top pane is the title pane and shows the label 'Smalltalk Browser'. Below it is a row of four *list panes* that display, from left to right, categories, classes, protocols and methods. The lower pane is a *text pane* that displays text associated with the most recently selected item.

Figure 1a shows the browser in its initial state with the leftmost list pane displaying part of the list of categories defining the Smalltalk system. The pane can be *scrolled* to view other categories in the list. The browser enters the state shown in Figure 1b in response to the user selecting the category **Data Structures**. A selection is made by moving a cursor over the item to be selected and depressing a button on the device controlling the cursor. Selections appear in inverted video in the actual system, but are shown in boldface in the figures. The most recent selection is in bold italics. The selection of **Data Structures** causes the classes of this category to be displayed in the second list pane and a template for defining a new class to appear in the text pane. In Figure 1c, the user selects **Set**, a class whose instances provide the behavior of sets by appropriately manipulating an array. This selection causes the class' protocols to be displayed in the third list pane and the definition of the class to appear in the text pane. The user can edit this definition to modify the title, superclass, or fields of the class. In Figure 1d, the user selects the **Access** protocol, causing its methods to appear in the last list pane and a template for defining new methods to appear below. In Figure 1e, the user selects the **has: element** method and its definition appears in the text pane. Figure 2 shows the path that the user has traversed in the system taxonomy. (This particular graphic view is not generated by Smalltalk.)

The **organization** entries under categories and protocols are not actually items of that type, but rather data structures that can be edited to alter the taxonomy. For this reason, the organization entries are not shown in Figure 2. Changing the category organization by selecting it and editing the text that appears below can move existing classes to different categories. The protocol organization serves a similar function for its class.

Overview: The browser shows a slice of the four level system taxonomy that extends through all four levels but is of limited breadth. Figure 2 shows this slice relative to a graphic view of the taxonomy. At his discretion, the user can select any element in the

displayed slice of the taxonomy. To see other elements on a given level, the user must *scroll* that pane, thereby changing the slice of the tree seen in the pane.

Path: Since the hierarchy is only four deep, the user can see the entire path from the root. The user cannot see, and the browser does not maintain, a history of other nodes that have been selected before, but are not on the path.

Presentation: Selection causes text and sub-structure to be displayed. Sub-structure is displayed in the list pane to the right. Text consisting of either templates or definitions is displayed below. For categories and protocols, a template is shown for defining new classes and methods respectively; for classes and methods, their definition appears. The reason for this difference is that categories and protocols have no semantic significance other than grouping a set of subordinate elements.

Operations: For each of the list panes, operations are defined for deleting, printing and filing the selected element. These commands are available from a menu that is not shown.

Insertion is not an explicit menu command. Instead, it occurs in two different ways. New classes and methods are inserted in their respective categories or protocols as a side effect of compiling their definitions. Old classes and methods can be rearranged by manipulating the table that the browser presents when the **organization** entry is selected in the category or protocol pane. Manipulating this table is also the mechanism for creating new categories and protocols.

A limitation is that the browser does not permit the creation of partially defined classes or methods. A class or method must be compilable to be successfully included in a category or protocol; this is a result of the browser assumption that the data structure it is viewing is the one currently installed in the system. This has undesirable consequences for program design when the designer wishes to delay certain decisions. In this respect, the marriage between the browser and the software environment is too intimate.

Multiple Views: Several browsers can be brought to the screen at once and can overlap. Commands are provided to move a browser to a new region of the screen and to view an obscured browser. The result is that the display screen is like a desktop with multiple browsers representing different pieces of paper.

This browser provides a command to spawn additional text windows that display the selected method. These windows maintain a constant view of the method, allowing the user to browse to other regions of the network. They are incomplete views of the method, however, in that they do not display its class or protocol, and hence these attributes of the method cannot be altered through this window.

The hardcopy format of Smalltalk code represents a third view of the system. This view is a depth first listing of the tree. Users occasionally prefer this view to the browser in order to obtain a perspective on a segment of code. The hardcopy format cannot be manipulated within the system.

The browser does not support other taxonomic views of the system such as an examination of the class/subclass hierarchy.

Consistency: The view seen on one browser is almost completely independent of that seen on a second, *even if they are both looking at the same method or class definition.** This means that if a method is changed using one browser, the definition seen on the screen for the other is not altered because that browser is unaware that the underlying model it is viewing has changed since it fetched the definition. Only if an explicit request is made to fetch the definition again is the underlying model queried, thereby ensuring that the view is consistent.

* The exception is that browsers do check whether the list of classes has changed whenever they are reactivated. If a class has been added or deleted from this list, the browsers reenters its initial state. No check is made for changes to the definitions of existing classes, protocols, or methods.

The reason for the inconsistency is two-fold. First, the view in the browser is just that, a computed view, and changes to that view are not reflected immediately in the model. Only when the method is compiled is the underlying system model altered. This is desirable since the user should be able to complete a set of changes to a procedure before it is altered permanently. Otherwise compilation might be attempted on an inconsistent state. Second, when a change does occur to some software object, there is no way for that object to inform the appropriate views since the underlying system model has no knowledge of existing views.

There are at least two solutions to this problem. One is to give each object responsibility for updating views of itself, using a "notification protocol"; for example, a class whose method changes would notify all browsers which have informed it of their current interest. A second solution is to give each view the responsibility for keeping itself updated, and to provide a way for it to check what the last time an object it is viewing changed. Then any time a viewer becomes active, it can compare its last update time with this list to see if updating is required.

Alternative Access: The only means to move through the network is by progressive selection of displayed objects. No browser commands exist to select an object via a partial description or even by specifying its name.

Integration: The browser does not support access to other kinds of data such as manuals, primers, and system specifications nor does it support examination and manipulation of instances of classes.

The browser is integrated in a limited fashion with a history list of changes in the sense that defining or redefining methods affects this list. However, deleting a method has no effect on the history nor can the history list be examined through the browser. No distinction is made between different kinds of modifications such as the difference between adding a breakpoint and making a permanent change made to the code.

Changeability: The user can change the size, number and position of browsers on the display screen by invoking commands supplied by the browser, but no commands are supplied to alter the relative widths of various panes.

The user can alter the behavior of the browser in two ways. He can redefine methods in the browser (using the browser itself), although bugs in these changes could make the interface inoperative. Or he can subclass the classes used to define the browser and make whatever changes he wishes in these subclasses. This is a safer strategy, since old style browsers are unaffected, but all behavioral changes must be programmed in Smalltalk itself. It is equally parsimonious in that subclasses inherit all of the behavior of their superclasses, except for messages that they define directly.

The browser does not support idiosyncratic behavior for particular objects of a given type: all classes, for example, are treated identically.

Summary of Smalltalk browser strengths and weaknesses

Strengths: The Smalltalk browser provides an excellent way of examining and editing the Smalltalk system code as evidenced by its universal adoption within the Smalltalk community and relative stability. Its browsing capabilities and the associated system architecture of a taxonomy of constructs serve a useful documentation role. Users often familiarize themselves with new software by browsing through new categories in a system release. The browser provides a uniform way to examine and manipulate the software, and guides novices with templates for creating new entities.

Weaknesses: The Smalltalk browser keeps no history of its interactions except for the names of methods that have been changed. It only reflects the current state of the world; there is no way to go back and forth between different consistent states. The system does not help a user to maintain any design constraints other than the ones implicit in the programming language. For example, a programmer cannot indicate that two methods in a class are dependent, and that subsequent modifications to one should be checked for compatibility with the other. There is no incremental way of modifying the behavior of the browser by attaching your own procedure to provide a specialized function in the interface; for example, one cannot provide specialized templates for new methods of a particular class.

The Smalltalk browser also reflects deficiencies in the underlying system model. Smalltalk provides for comments for classes and methods but not for categories of classes or protocols of methods. Class comments are separately manipulable from the class

definition; method comments are not. Storing a method comment requires that the procedure be recompiled.

The PIE System Model

PIE was motivated, in part, by the goal of providing a more complete and more integrated representation for Smalltalk systems. It provides a network structured database whose nodes describe all the entities in the system and employs techniques developed for describing entities in knowledge representation languages like KRL [BobrowWinograd77].

Nodes provide a uniform way of describing entities of many sizes, from a small piece such as a single procedure to a much larger conceptual entity. For example, nodes are used to describe code in individual methods, classes, categories of classes, and configurations of the system to do a particular job. Sharing structures between configurations is made natural and efficient by sharing regions of the network.

The uniform use of node structure extends to software documentation. Manuals and specifications can be embedded in the network using nodes representing the chapters, sections and paragraphs of the material and can be cross-linked to the relevant software. Because software and documentation coexist in the same environment, it is easier to develop them in a coordinated manner.

Nodes are distinct from the system objects that they represent. Changing a node does not immediately alter its corresponding software object. For example, the node representing a class can be created and a partial definition supplied. This node can be stored, examined and edited. It does not affect the underlying Smalltalk environment, however, until its description is compiled.

Attributes of nodes are grouped into perspectives. Each perspective reflects a different view of the entity represented by the node. For example, the **structuralSpec** of a Smalltalk class defines the structure of each instance by specifying the fields it must contain; the **proceduralSpec** defines the protocols; the **interfaceSpec** defines the set of messages required by external clients, and the **documentSpec** describes the implementation and its use.

Perspectives may provide partial views which are not necessarily independent. For example, the **proceduralSpec** and the **interfaceSpec** both describe certain methods of the class. Attached procedures are used to maintain consistency between such perspectives.

Each perspective supplies a set of specialized actions appropriate to its point of view. For example, the *print* action of the **structuralSpec** perspective of a class knows how to prettyprint its fields and class variables, whereas the **proceduralSpec** perspective knows how to prettyprint the methods of the class. These actions are implemented directly through messages understood by the Smalltalk classes defining the perspective.

All values of attributes of a perspective are relative to a *context*. Context as we use the term derives from Conniver [SussmanMcDermott72]. When one retrieves the values of attributes of a node, one does so in a particular context, and only the values assigned in that context are visible. Therefore it is possible to create alternative contexts in which different values are stored for attributes of various nodes. For nodes representing software, these contexts typically describe alternative designs. One can compare and test alternatives without leaving the design environment.

Contexts are themselves nodes in the network. This allows a description of the rationale for the set of changes to be stored in the context node in the network, in the same way that descriptions for for a method node contain comments on their purpose.

In any system, there are dependencies between different elements of the system. If one changes, the other should change in some corresponding way. We employ *contracts* between nodes to describe these dependencies. These contracts are themselves nodes with specialized behaviors. These behaviors include installation of procedures to maintain consistency of simple constraints expressed in a formal language, and notification to the user when changes have been made to contract participants. Use of contracts raises a number of questions which we have just begun to explore; e.g. when should one check agreements and still avoid seeing temporary states of inconsistency during the process of change.

Finally, the PIE system provides perspectives which allow the system to describe itself. Perspectives themselves are described in the system, and small modifications to the behavior of a particular perspective can be made by manipulation of the network structure. Nodes can be assigned meta-nodes whose purpose is to describe defaults, constraints, and other information about their object node. Information in the meta-node is used to resolve ambiguities when a message is sent to a node having multiple perspectives.

The PIE Browser

The PIE browser was constructed as a generalization of the Smalltalk browser, in order to minimize the overhead of Smalltalk users immigrating into the PIE environment. It is shown in Figure 3a. Two additional panes have been added in the middle of the browser. The left pane lists the perspectives of the most recently selected node while the right pane lists the attributes of the selected perspective. The title pane shows the node at which the browsing begins and the context from which the network is being viewed.

In Figure 3b, the user has selected the node representing the **Data Structures** category. This causes the two perspectives of this node to be displayed. The first is the perspective describing categories: it includes a **classes** attribute and additional attributes describing the most recent file and modification dates to classes in the category. The second is the **description** perspective, common to many nodes, that specifies a title and

optional text for the node. In this case, the text attribute is employed to store a comment regarding the category, and this comment is displayed in the text pane.

In Figure 3c, the user has selected the **category** perspective and its attributes appear in the attribute pane. In Figure 3d, the classes attribute is selected and its value, a list of nodes representing the classes of this category, appears in the second list pane. The attribute is used as a label for the pane. In Figure 3e, the user has selected the Set node, and its perspectives appear below. Thus, moving from one node to the next in the network requires selection of a node, then a perspective, then an attribute. Figure 4 shows a graphic representation of the PIE network and the path traversed by the user.

Overview: As with the Smalltalk browser, the user can see a slice of the network. In addition to nodes surrounding previous selections, this slice includes the perspectives and attributes of the current selection. We have explored browsers that show the perspectives and attributes of every node in the path, but these trade breadth of view for increasing complexity on the screen.

The labels on the four upper list panes are dynamic and computed from the selection. The Smalltalk browser employed static labels since the same attribute was always displayed in a given list pane.

Path: The PIE network is not restricted to a depth of four. However, the PIE browser contains only four list panes, a constraint derived from the size of the screen. To go deeper into the network, the user can shift the view to the left. In Figure 5, the user has moved the view one to the left. The origin of the browser is now the **Data Structures** category and the rightmost pane is available to show subordinate nodes linked to the **has: element** method. In this case, the user is examining nodes representing constraints on the definition of the method. If the user tried to see substructure which would logically be to the right of the fourth pane, PIE blinks the browser to indicate that it cannot show the requested information in the current browser configuration. The user can then shift the view as described, or spawn a new browser rooted further down the tree, and continue.

The PIE browser does not maintain a chronological history of selections. Hence, it is limited, like the Smalltalk browser, to displaying only four steps in the path to the current selection. An unfortunate consequence of this lack of historical information is that while the view can be shifted to any node in the network, the browser cannot recreate selections made from that node. Hence, a shift to the right, for example, from the **Data Structures** node back to the **Code** node, would require that the user remake his selection choices to again be examining the **has: element** method.

Presentation: To minimize the interactions required by the user, the browser can operate in a mode in which it makes various default decisions on its own initiative. These decisions are based on additional descriptions provided in the network. For example, the

network contains descriptions that specify that the **category** perspective should be selected by default over the **description** perspective and that its **classes** attribute should be displayed. As a result of these default specifications, the selections of Figures 3c and 3d are made by the system and selecting the data structures category in Figure 3b produces the display of Figure 3e immediately. Hence, the user need not engage in any more interaction with the PIE browser than with the Smalltalk browser to conduct similar actions. The user can override these defaults by making explicit perspective or attribute selections.

The specification of the default display behavior of a node is described in *meta-nodes* linked to perspective types and to particular nodes. In the former case, the meta-node applies to all instances of the perspective. In the latter case, its advice is idiosyncratic to a particular node. These meta-nodes can be examined and edited from the browser.

Templates for creating new nodes of a particular type are available upon request and are stored in the meta-node of the perspective. They are shown automatically only if they are specified to be the default display information. Many perspectives, not just those for classes and methods, have templates.

Operations: The PIE browser supplies four standard operations: insertion, deletion, filing and printing. Insertion consists of adding a node to the list and assigning it a perspective. Default knowledge is employed to supply a particular perspective when the list is constrained to be a set of nodes of a particular kind. For example, the **classes** attribute of the **category** perspective has the default description that all of its elements have a **class** perspective assigned. Descriptions of nodes can be stored without having to compile them. Therefore partial descriptions of methods can be left in the network and returned to later.

Insertion of nodes of arbitrary type eliminates the need for an *organization* entry. Categories and protocols are created by adding nodes with those perspectives. Rearranging an old organization is accomplished by moving nodes from one attribute set to another.

The PIE browser also differs from the Smalltalk browser in that special actions specific to perspectives at a node can be invoked by the user through a special menu. This menu is computed from the selected node, using default description that specifies a subset of the messages of a perspective to be user commands. The PIE browser can view nodes with arbitrary perspectives in any pane. Hence, the ability to interrogate the perspective for its associated commands was necessary. Since the Smalltalk browser views only four kinds of objects and these objects are tied to particular panes, this generality was not included.

Multiple views: There are three different senses in which multiple views are available to the user of PIE. The first is similar to that of the Smalltalk browser. There can be more than one instance of a browser on the screen at a time, viewing different parts of the Smalltalk system.

A second kind of multiple view comes from the notions of context embodied in the PIE network. The value of any attribute is context dependent. The user can change the view seen in the browser by changing the context associated with that particular browser. This causes the browser to recompute all fields seen.

The third arises from the fact that the user can request an outline view to be generated of the substructure of the selected node. A portion of the subtree descending from the selected node is shown in an indented outline format. The default perspective and attribute of each node is used to determine which part of the subtree to display. For class Set, this outline would include the Set node, the protocol nodes of its **structuralSpec** perspective, and the method nodes of each protocol. This outline is very close to the standard hardcopy view of Smalltalk code—a fact that is not accidental. The defaults have been chosen to make this view the preferred one.

Consistency: As with the Smalltalk browser, there are no backpointers from nodes to views. This means that a change made to the network through one browser is not reflected in another browser's view computed earlier. One approach to solving this problem is presently being introduced into Smalltalk by providing backpointers from software objects to their views. A separate control process is assigned responsibility for maintaining consistency. Another approach that we are considering is to describe the browser itself in the PIE network in order to take advantage of the contract machinery provided by PIE to maintain consistency between descriptions. However, this is still an unexplored area.

Alternative Access: A browser provides one way to get access to a node in an information network. Sometimes it is useful to shift the point of view of the system to a node which matches a given description without having to browse through one level at a time. This is provided in PIE. A user can specify the perspective type and some distinguishing features of a node. For example, he can search for classes entitled Set, any class that is a subclass of these classes, or even any class whose comment includes the substring 'set'. PIE engages in a a search and causes the view to be shifted to the selected node. If more than one node matches the description, PIE offers the user all matches. Selection of a match causes the view to be shifted to the selected node.

Some indexing facilities are provided to limit the potential candidates for a match: each perspective maintains a list of the nodes to which it has been assigned. This is a very simple scheme, but the present size of the Smalltalk system—consisting of several hundred classes owning several thousand methods—does not require anything more elaborate.

One novelty of our searching machinery with respect to traditional database design is that no general set of indices are maintained. Rather, each perspective has its own matching protocol. Thus, if a perspective receives a description like 'set' without a specification of the attribute of the perspective to which this description must match, the perspective itself decides which attributes can be used as the basis of a match. For

example, the **structuralSpec** perspective checks the title and superclass attributes, but not the field variable or class variable declarations. This is in contrast to most data base environments where entities are matched against a pattern by a standard algorithm which matches the values of attributes, perhaps using range tests. Because PIE is integrated in the Smalltalk system, each entity can run its own idiosyncratic program to test whether it matches a description.

Integration: The PIE browser integrates the examination of data, code, documentation, and system description since all of this information is uniformly described in the network. The browser also integrates the computation of views of the database with the underlying programming language. In most data bases, "views" are supported which compute virtual relations from real ones that exist in the data base. However, the programming language to compute these views is impoverished, usually being restricted to expressions in the relational calculus. The advantage of this language is that it makes the update problem easier by providing an expression calculus with no side effects for specifying how to compute a view each time. In PIE, the full power of the Smalltalk language is available, but we must provide notification and time stamp mechanisms to help with the update problems.

Changeability: In addition to the ways that the Smalltalk browser can be altered, the behavior of the PIE browser is affected by changes to the information network. A user can alter the default display behavior of perspectives by editing the meta-nodes involved. For example, the user can change the meta-node to cause the default text displayed when a class is selected to be the comment describing the class rather than the class definition.

Summary of PIE browser strengths and costs

Strengths: Some strengths of the PIE browser arise from the improvements in the PIE system model over the standard Smalltalk model. The network database that the browser manipulates is arbitrarily deep, allows multiple perspectives and context-sensitive description, integrates the representation of text and software, and supports search and matching behavior. Other strengths arise from the availability in the network of interface-specific description. This includes description of default perspectives and attributes for display, and idiosyncratic behavior of particular entities. This self-description minimizes the user's workload for expected actions.

Weaknesses: The PIE browser shares a number of weaknesses with the Smalltalk browser. For example, it does not maintain a history of user interactions and it does not provide any means to maintain consistency between multiple views. However, the PIE model provides a possible solution to both of these weaknesses. Nodes can be employed to represent the history of a design and to represent contracts between multiple views. This solution has the appeal of building upon existing machinery and maintaining a highly integrated system model. These are current research issues for us.

Another potential weakness common to both the Smalltalk and the PIE browsers is that they do not present the network in a two dimensional graphical notation such as the one shown in Figures 2 and 4. Indeed, since those figures were used to elucidate the network structure being examined by the browsers, one might very well ask why it is not the format actually generated by the interfaces. The answer, of course, is that the pane-oriented structure of both browsers is simpler to implement than a general two-dimensional layout program. However, a research issue is whether this implementation simplicity comes at a serious cost in comprehensibility to the user. Experiments need to be performed with users of different levels of expertise to investigate which graphical metaphors are most useful in clarifying the presentation of a network description of software.

Conclusions

PIE reflects a natural evolution of the Smalltalk system model to provide a more extensive description of an evolving software design. The PIE browser has evolved in parallel. An unexpected result is that the boundaries between the two have become fuzzy as the network describing the software system is employed to describe the desired display behavior. Specifications of system semantics do not usually include such descriptions. However, the availability of more powerful machines, coupled to the increasing complexity of software, makes their inclusion both possible and necessary.

The PIE system and its associated browser is largely independent of the semantic details of Smalltalk. It is based on the existence of a network description of a software system. It could be the basis for programming environments for other software languages, to the extent that those languages supported display facilities and a network database which can hold representations of code easily accessible by the language processors. Experiments reported in [Cattell80] are planned for exploring these ideas in a programming environment for Mesa, a PASCAL-derived systems programming language.

References

Birtwistle, G., Dahl, O.-J., Myhrhaug, B., and Nygaard, C., *Simula Begin*, Auerbach, Philadelphia, 1973.

Bobrow, D.G. and Goldstein, I.P. "Representing Design Alternatives", *Proceedings of the AISB Conference*, Amsterdam, 1980.

Bobrow, D.G. and Winograd, T. "An overview of KRL, a knowledge representation language", *Cognitive Science* 1, 1 1977.

Cattell, R.G.G., "Integrating a Database System and Programming/Information Environment", to appear in a *Joint Issue of SIGMOD, SIGPLAN, and SIGART*, October, 1980.

Goldberg, A. and Robson, D. "A Metaphor for User Interface Design", *Proceedings of the 13th Hawaii International Conference on System Science*, Jan. 1979, pp. 148-157.

Goldstein, I.P. and Bobrow, D.G., "Extending Object Oriented Programming in Smalltalk", *Proceedings of the Lisp Conference*. Stanford University, 1980a.

Goldstein, I.P. and Bobrow, D.G., "A Layered Approach to Software Design", *Xerox PARC CSL-5-80*, 1980b.

Goldstein, I.P. and Bobrow, D.G., "Descriptions for a Programming Environment", *Proceedings of the First Annual Conference of the American Association for Artificial Intelligence, August*, 1980c.

Hewitt, C., Bishop, P., and Steiger, R., "A Universal Modular ACTOR formalism for artificial intelligence", *Proceedings of the Third International Joint Conference on Artificial Intelligence*, 1973, pp. 235-245.

Ingalls, Daniel H., "The Smalltalk-76 Programming System: Design and Implementation," *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, January 1978, pp. 9-16.

Kay, A. "Microelectronics and the Personal Computer" *Scientific American*, September, 1977.

Kay, A. and Goldberg, A. "Personal Dynamic Media" *IEEE Computer*, March, 1977.

Sussman, G., & McDermott, D. "From PLANNER to CONNIVER -- A genetic approach", *Fall Joint Computer Conference*, Montvale, N. J., AFIPS Press, 1972.

Thacker, C. P. McCreight, E. M., Lampson, B.W., Sproull, R.F., and Boggs, D.R. "Alto: A personal computer" in Siewiorek, Bell and Newell, *Computer Structures: Readings and Examples*, 1980.

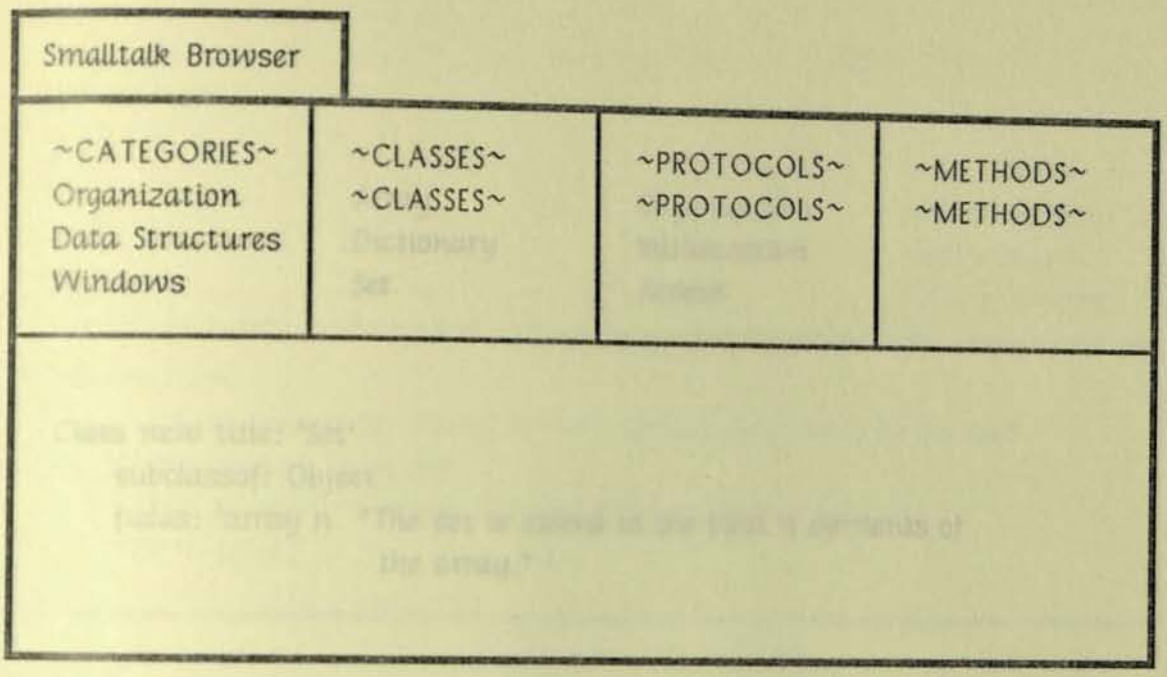


Fig. 1a. The browser is in its initial state, displaying a list of categories.

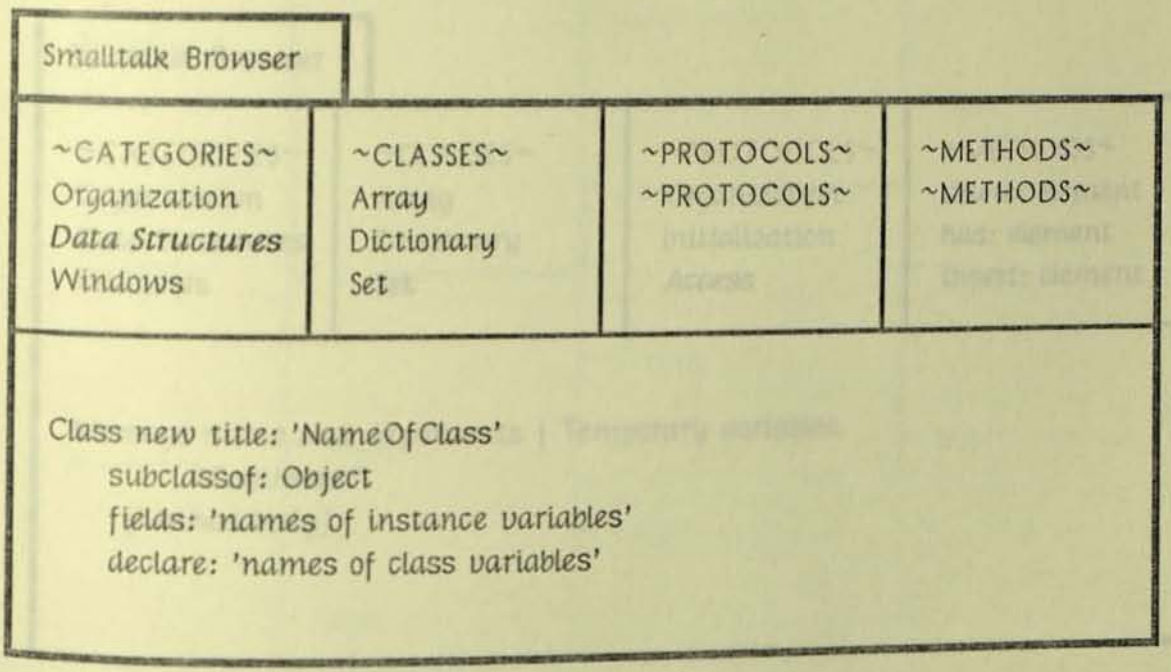


Fig. 1b. The user has selected the Data Structures category.
The classes of this category appear in the classes pane
and a template for defining new classes appears in the text pane.

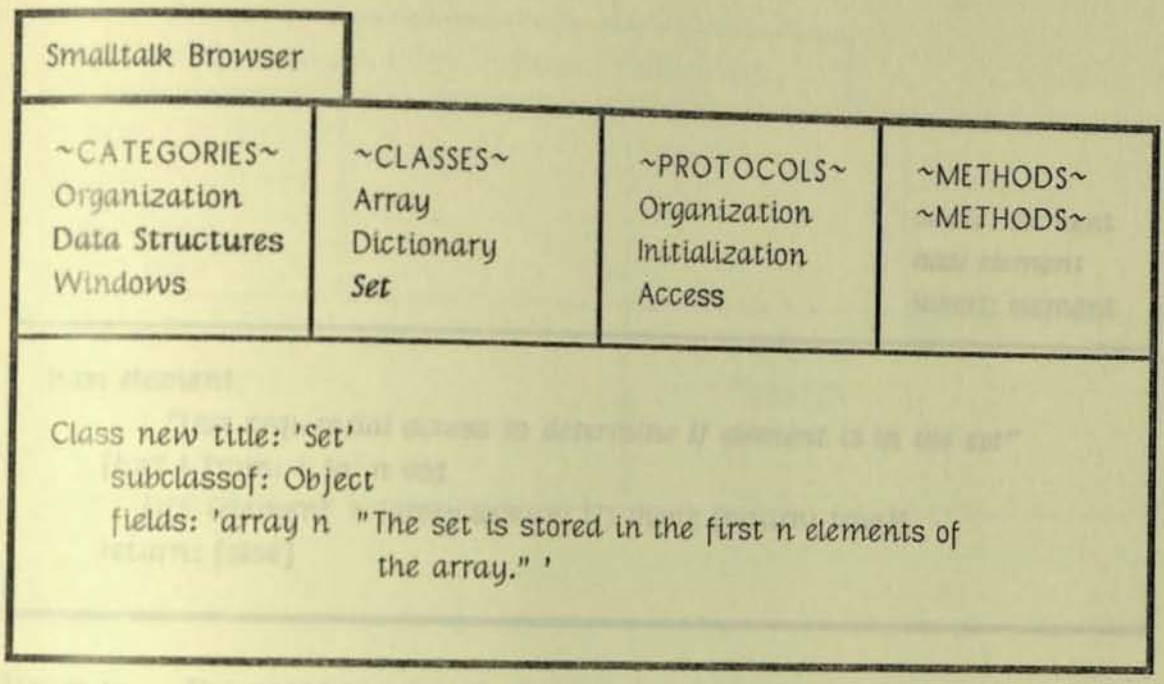


Fig. 1c. The user has selected the class Set.
The protocols of this class appear in the Protocols pane
and the definition of the class appears in the text pane.

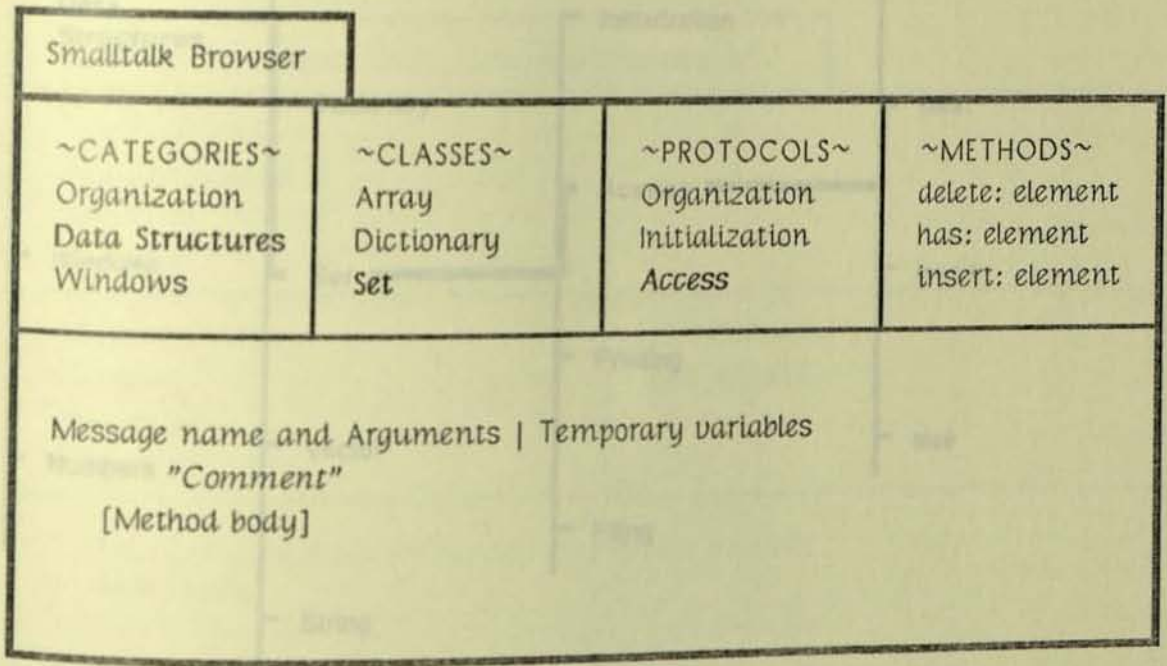


Fig. 1d. The user has selected the Access protocol.
The methods of this protocol appear in the Methods pane
and a template for defining new methods appears in the text pane.

Smalltalk Browser

~CATEGORIES~	~CLASSES~	~PROTOCOLS~	~METHODS~
Organization	Array	Organization	delete: element
Data Structures	Dictionary	Initialization	*has: element*
Windows	**Set**	**Access**	insert: element

```
has: element
        "Use sequential access to determine if element is in the set"
    [for: i from: 1 to: n do:
        [if: (element = (array lookup: i)) then: [return: true]].
    return: false]
```

Figure 1e. The user has selected the has: element method and its definition appears in the text pane.

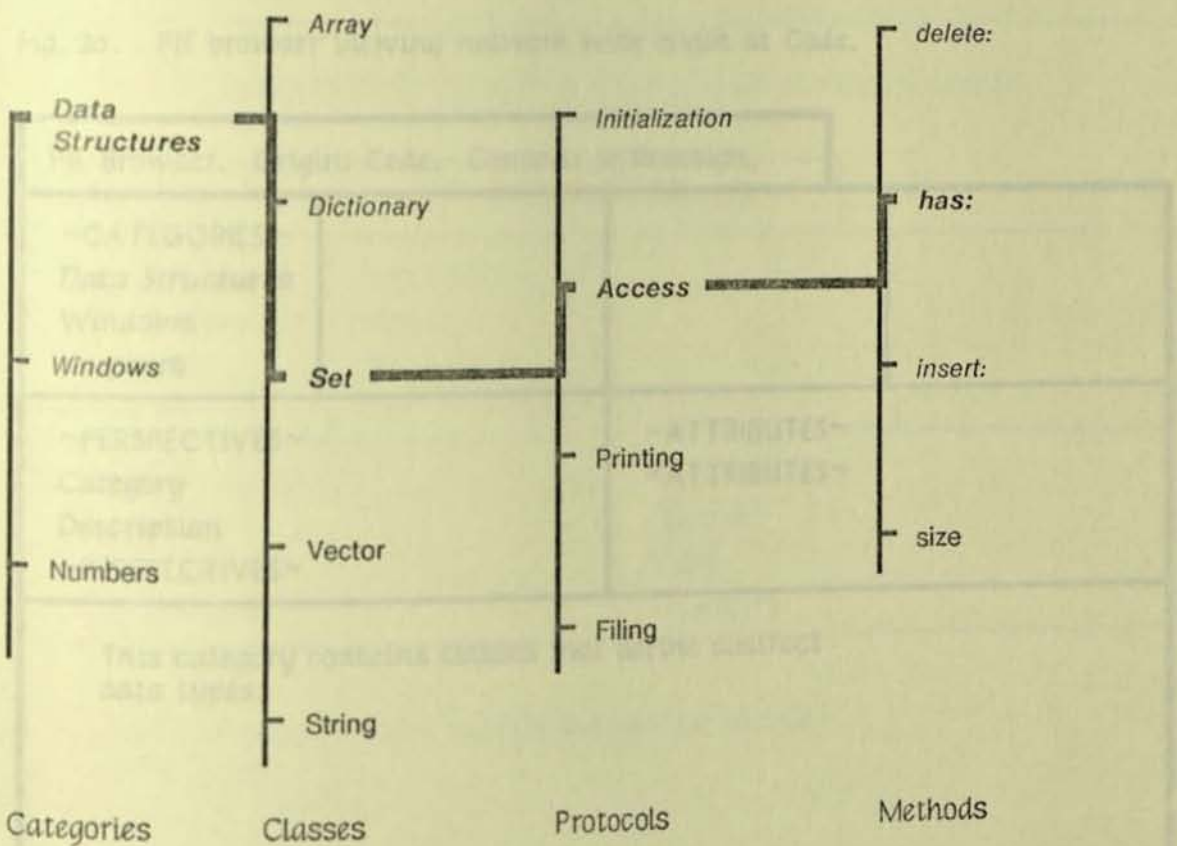


Fig. 2. A tree representation of the Smalltalk taxonomy. The path selected in the browser is shown in boldface. The slice of the taxonomy visible in the browser is shown in italics.

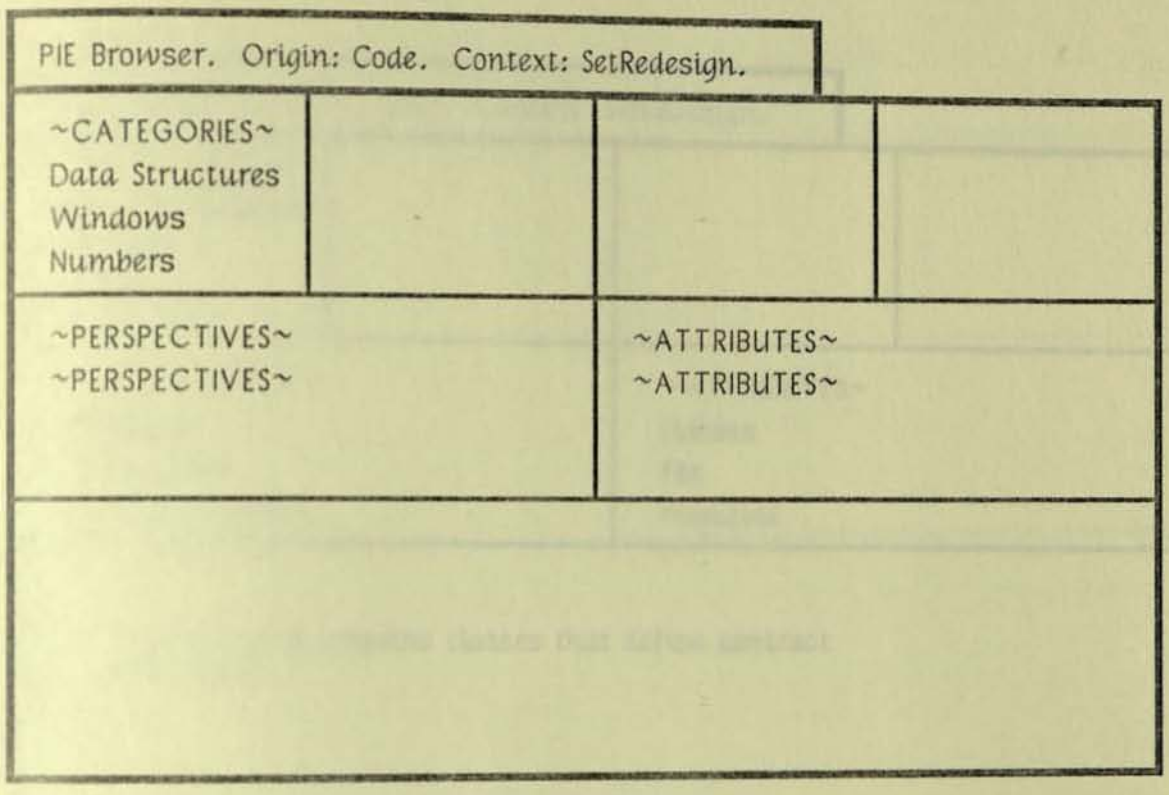


Fig. 3a. PIE browser viewing network with origin at Code.

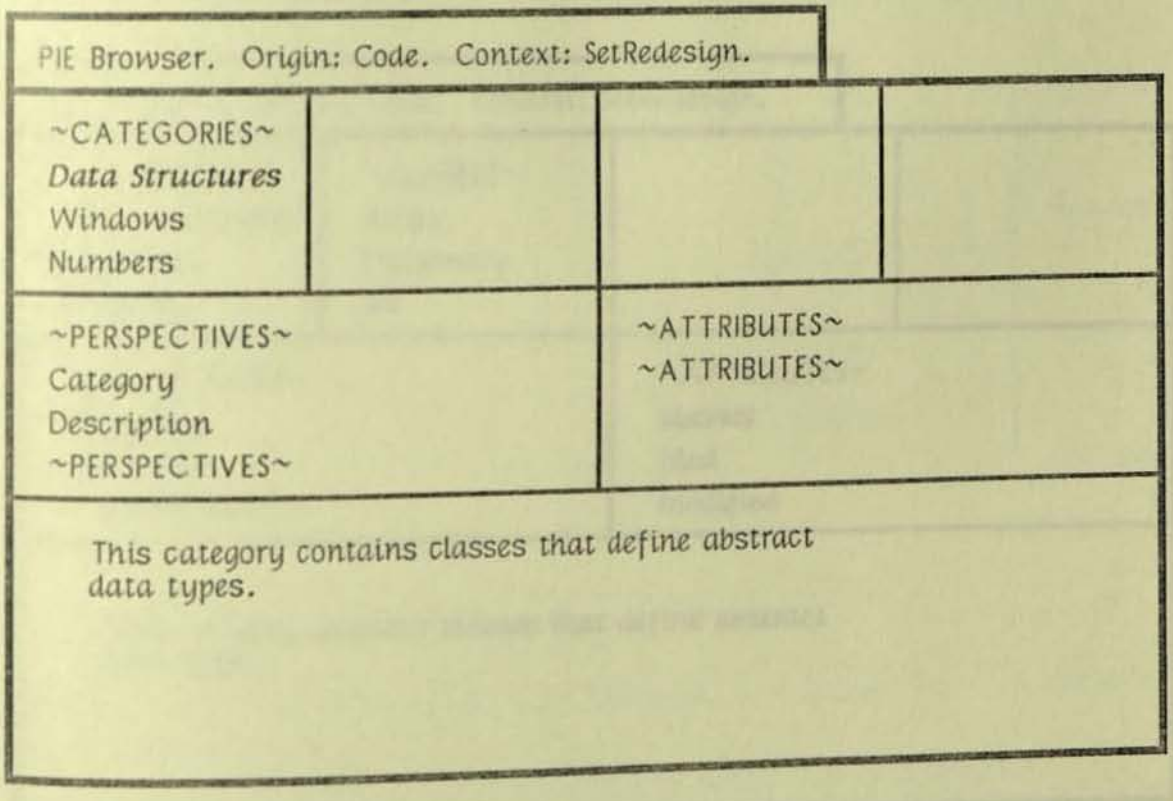


Fig. 3b. The Data Structure node is selected and its perspectives appear. The comment is the text attribute of the description perspective and is displayed by default.

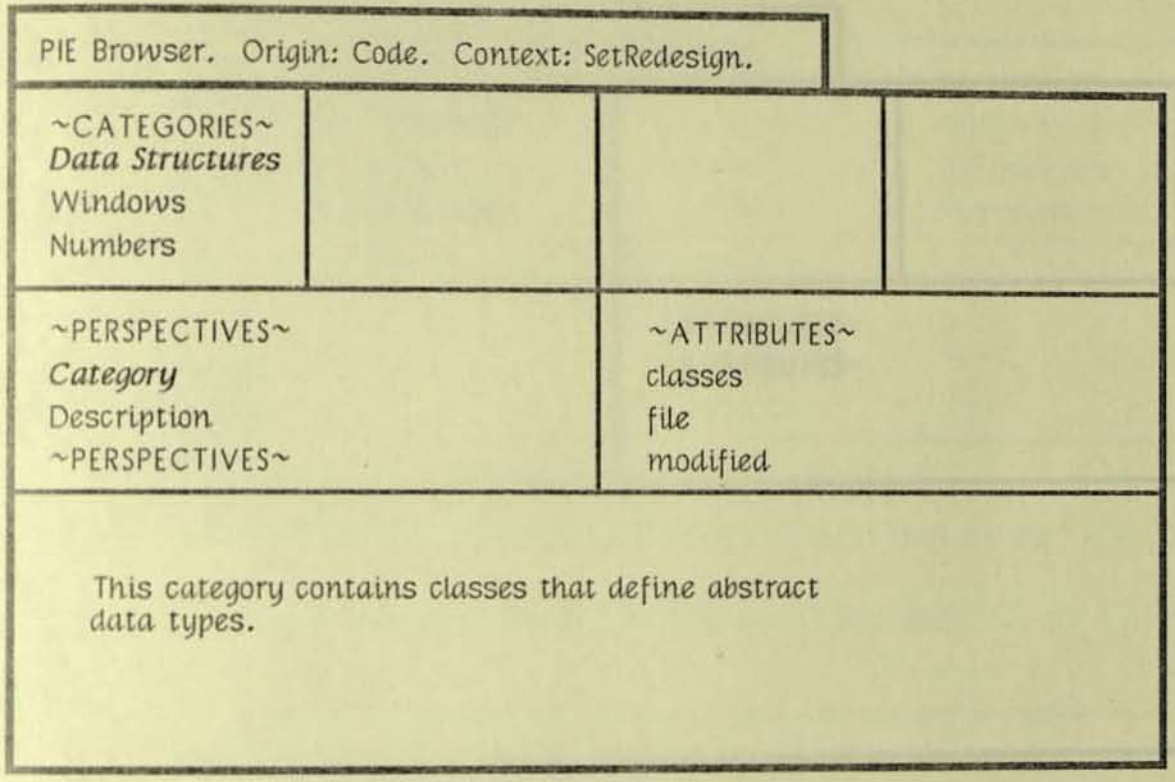


Fig. 3c. The category perspective is selected and its attributes appear.

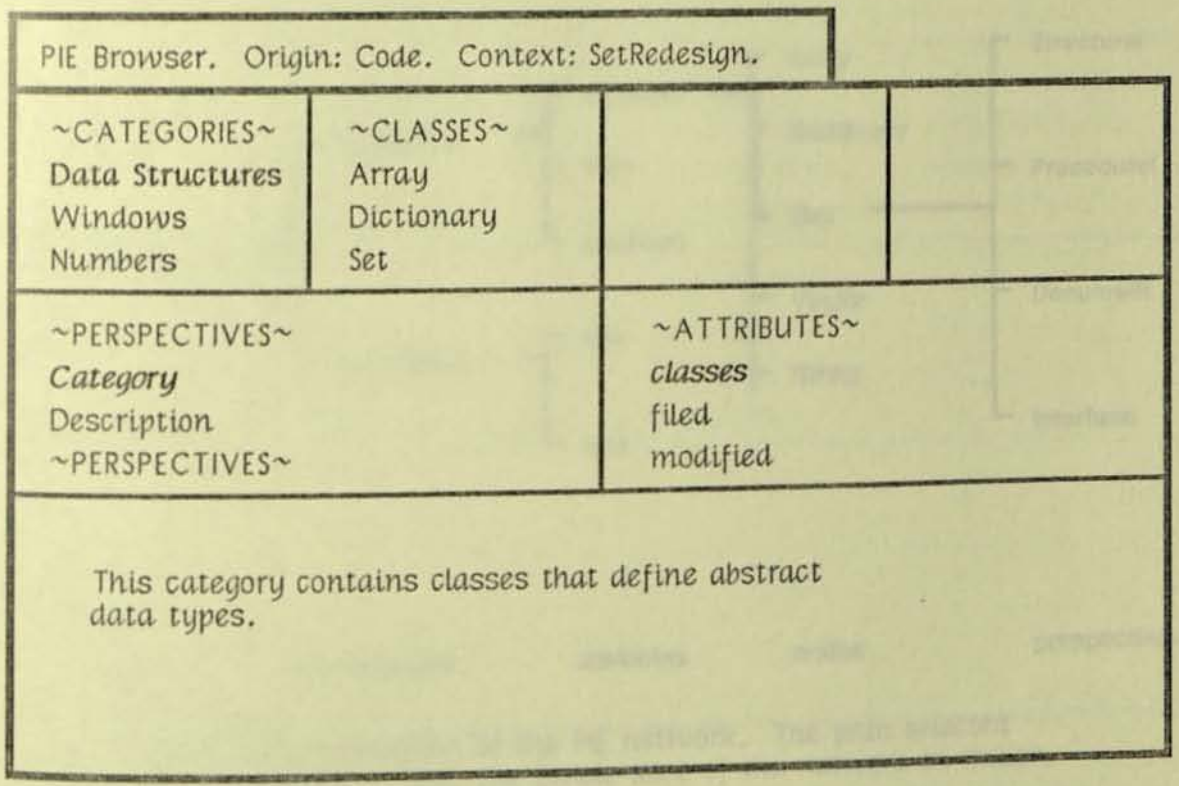


Fig. 3d. The classes attribute is selected and the list of classes appears.

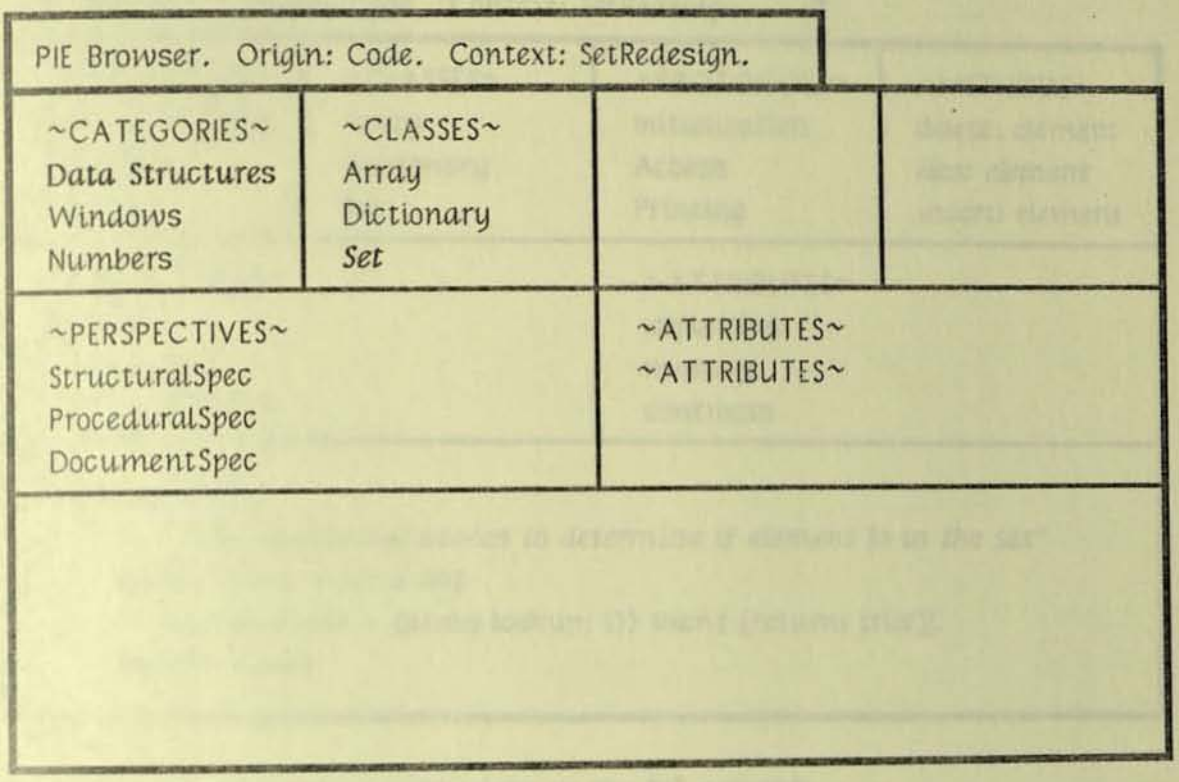


Fig. 3e. The Set node is selected and its perspectives appear.

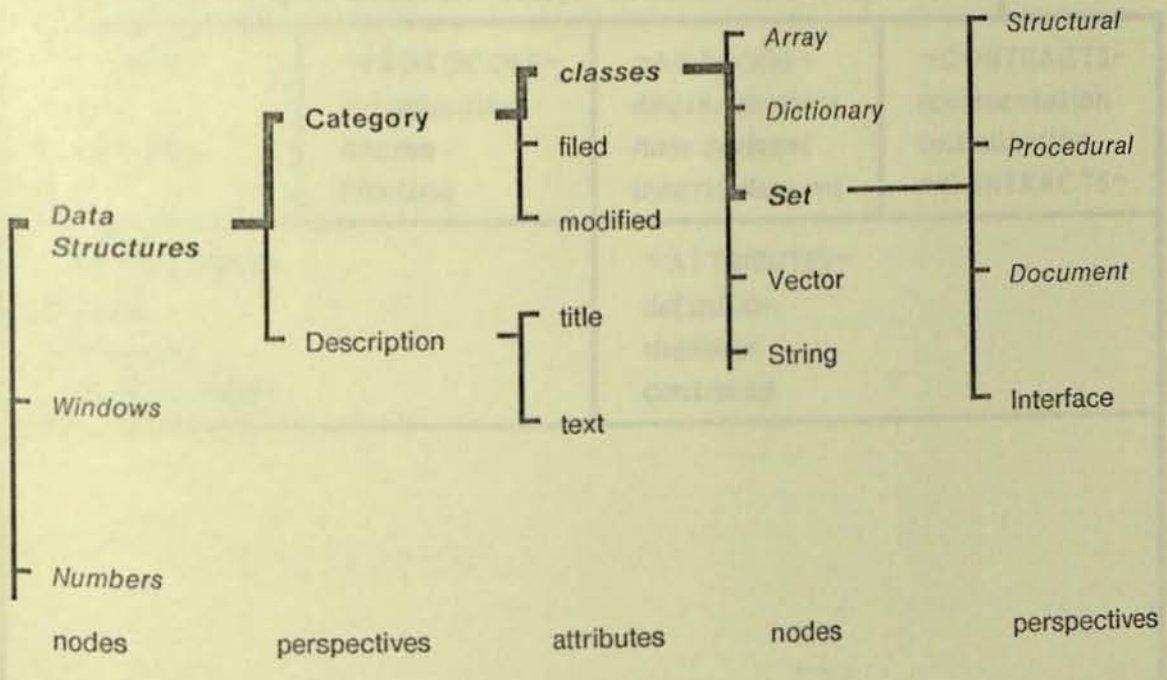


Fig. 4. A graphic representation of the PIE network. The path selected in the browser is shown in bold, the visible slice of the network in italics.

PIE Browser. Origin: Code. Context: SetRedesign.

~CATEGORIES~	~CLASSES~	~PROTOCOLS~	~METHODS~
Data Structures	Array	Initialization	delete: element
Windows	Dictionary	**Access**	***has: element***
Numbers	**Set**	Printing	insert: element

~PERSPECTIVES~	~ATTRIBUTES~
Method	***definition***
Description	*message*
~PERSPECTIVES~	contracts

```
has: element
        "Use sequential access to determine if element is in the set"
    [for: i from: 1 to: n do:
        [if: (element = (array lookup: i)) then: [return: true]].
    return: false]
```

Fig. 5a. The user is four levels deep in the PIE network.

PIE Browser. Origin: Data Structures. Context: SetRedesign.

~CLASSES~	~PROTOCOLS~	~METHODS~	~CONTRACTS~
Array	Initialization	delete: element	representation
Dictionary	**Access**	***has: element***	initialization
Set	Printing	insert: element	~CONTRACTS~

~PERSPECTIVES~	~ATTRIBUTES~
Method	definition
Description	*message*
~PERSPECTIVES~	***contracts***

Fig. 5b. The origin has been shifted to the Data Structures node, allowing the user to view the network one level deeper.