Display Strategies for Program Browsing: Concepts and Experiment

Ben Shneiderman, Philip Shafer, Roland Simon, and Linda Weldon
University of Maryland

The new, larger display screens can improve program comprehension—if the added space is used for more effective presentation, not just more code or larger type.

Software maintenance is an important part of a programmer's work and a product's life cycle, yet it remains one of the most troublesome of tasks. Even existing, newly developed techniques are not of much use, since only time can determine their value. Thus, instead of presenting another new maintenance tool or management technique, we focus on strategies for improving the presentation of information — specifically, on the new, larger display screens.*

User interfaces in the software environment are much like spices in good recipes; the right arrangement must be found or the food will not show its full flavor. Factors such as data availability and complexity and the size of the display must be carefully weighed and accounted for in the design of any software environment.

For example, in maintenance tasks, many types of information must be easily available. A study of programmers doing maintenance tasks[1] shows that programmers want more information than is currently available on the display, but they are not sure what exactly would be most helpful. Today, faster and larger displays are becoming more readily available, and we have the opportunity to add more information. The immediate problem then is what information to add and how to display it.

One way is to provide multiple views of the program, as the Pecan[2] system has done. Our attention is directed at discovering which views are most beneficial.

However, bear in mind that large displays are not simply extensions of small displays, any more than large programs are simply extensions of smaller ones. Writing large programs requires new techniques and management schemes to control complexity. The same is true for large displays. Large displays must be designed for economical information presentation — information that has now become more complex, since the size of the available work area presumably interacts with many other aspects of the display. Thus, like large programs, large displays need new strategies for sound and efficient display formats.

In this article we deal with coordinated window systems, in which the windows and their contents appear and scroll automatically as a result of user activities. In most other window systems, the user creates, positions, and manipulates windows and their contents. In coordinated window systems, the user is freed from such tedious chores. We present four strategies for coordinated window systems:

Fusion — shows continuous text over multiple windows.

Synchronized scrolling — shows object-to-object relationships and provides linked views.

Embedded selection — shows detailed information and provides a microscopic view.

Hierarchical browser — shows design information and provides a macroscopic view.

To analyze these strategies and show their benefits for maintenance tasks, we use a model taken from Rombach, Basili, and Chang.[3] Error, fault, and failure have the meanings defined in *Standard Glossary of Software Engineering Terminology*.[4] This model partitions maintenance tasks into five phases:

Detection — detecting a failure or a fault of the program.

Isolation — finding the related fault(s) in the program responsible for the failure.

Design changes — designing the necessary changes to correct the fault(s).

Correction — implementation of the changes.

Validation — validate whether the changed product meets the specification.

The examples given in the following discussion were implemented at the Univer-

*An earlier version of the material in this article was presented at CSM-85. Recent experimental results have been added.

0740-7459/86/0500/0007$01.00 © 1986 IEEE

sity of Maryland's Human-Computer Interaction Laboratory for use in experimental research aimed at determining the usefulness of these and other techniques. The equipment used consisted of an IBM 3290 plasma display workstation, which displays 62 lines of 160 characters, but the strategies themselves can be applied to any large display. We have also implemented many of these strategies on small personal computer displays.

Fusion

The simplest coordination strategy is fusion, in which many lines of code are displayed in sequence on multiple windows. We can better understand the concept of fusion if we think of the windows as one logical screen. Let us look at an implementation of fusion on an IBM 3290 workstation. The 160-column, 62-line screen layout was vertically bisected to give two 80-column, 62-line windows, shown left and right for a total of 120 lines. The programmer could then turn one or two pages of code at a time, either bringing up the next two pages or bringing up one page and moving the other page over to keep the pages in left-to-right order. Fusion allows for normal editor commands to move partial pages or perform string searches.

Fusion may be most helpful when working with large portions of code that must be considered as a whole to fully comprehend the meaning. When viewing large routines on a normal 24-line screen, programmers waste time switching contexts and get lost in turning pages back and forth. The strength of fusion comes in its ability to present entire routines for consideration so that the viewing context is kept constant.

If the routine is larger than fusion can display, the number of context switches is still reduced. Fusion lessens the need for tedious navigational and page turning commands. Programmers can concentrate on comprehending the program without learning new commands. All they need is a larger screen.

A study by Boehm[5] shows that the size of the modification is proportional to the chance of making an error during the modification. When programmers view the entire routine, they can make corrections more easily and with greater accuracy. Seeing the full area in which the correction is made allows them to see the impact of many modifications.

Synchronized scrolling

Synchronized scrolling allows the display of two or more files of related information and automatically positions all files whenever the user moves any file. As the user scrolls one file, all other files scroll as well. No new commands are needed to use synchronized scrolling. The complexity of the relationship between the files determines the complexity of the synchronization program. Synchronized scrolling can be useful in viewing two versions of a program or in evaluating test cases and their results.

Fixed ratio. The simplest synchronization method is based on a constant ratio between the number of lines that should be scrolled in each window. Perhaps the comments and code for a program are in separate files. Blank lines may be used to keep a one-to-one ratio between the files, so a synchronized editor can be called with that ratio. In the same manner, a file containing a description of maintenance changes to a program could be viewed along with the code and design specification for that code.

Unique objects. A more flexible and complex synchronization technique is required to define a relationship between two or more files. If the relationship can be stated as a rule, then it can be made into a routine to facilitate synchronized scrolling. For example, a text formatter transforms its input using the rules of the format language. A synchronizing routine can be written to link two windows, one containing the source document and the other displaying the formatter output. Scrolling one window moves the other so that the same information appears in both.

Synchronized windows are also a logical extension of an environment for an interpreted language, such as the Cornell synthesizer.[6] The windows show the input, output, related data objects, and the code manipulating them. Programmers can have the interpreter step through the code one statement at a time. They can then see the input being read, the data objects being manipulated, and the output being produced. Use of such an environment allows programmers to see precisely where the data is incorrectly modified. Synchronized scrolling is also helpful in isolation and detection, since it can present multiple views of the same object.

Another use of this approach is to have the same comment appear in a flowchart, its PDL, and the related code. Synchronized movement is proportional between these token comments. Comprehension becomes much easier, since we have multiple levels of complexity for context. This particular use is reflected in the hierarchical browser, which is discussed later.

Embedded selection

When viewing source code, programmers are not simply reading the text; they are also attempting to understand its structure. Only a limited number of unique symbols appear in the code, and these are either user-defined or system or language keywords. User-defined symbols include routine names and variables and type and constant identifiers. System- or language-defined symbols are predefined elements, built-in variables, types, constants, and library routine names.

Each symbol, whether user- or system-defined, has properties associated with it, such as what it is, what it does, and what importance it has. For variables and types, these properties would include the specification of composite types. For routines, these properties would include a description of the parameter list and what type of value is returned. With library items, these properties might include the location of the original source, if possible.

All this information must be easily available to the programmer through direct selection of the appropriate symbol, or direct embedded selection. The programmer simply moves the cursor onto the symbol in the context shown on the screen and

The Bettmann Archive

presses a select key. This selection causes the properties associated with that symbol to be presented in a second window. The symbol is selected in context so that only the proper definition of the symbol is considered, regardless of its definition in another context.

Identifiers. When a user-defined variable, type, or constant is selected, the declaration of that identifier surrounds the point at which it is declared (Figure 1). The declaration should include some comment on usage, but this inclusion is at the discretion of the programmer. If the Ada language is used, a second type of selection could be implemented to allow the programmer to see the attributes available for use with an identifier.

When a language-defined variable, type, or constant is selected, a brief explanation of that identifier is shown, perhaps the manual entry.

Let us look at an implementation using Pascal. If the select key is pressed while the cursor is on a user-defined identifier, a new window is opened and the declaration of that identifier is displayed. If the select key is pressed a second time, the window is closed.

User-defined routines. As with identifiers, when a user-defined routine is selected, the declaration of that routine is presented, including formal parameters and a header comment describing the purpose and function of the routine. Again, such comments are assumed to be written with the code.

System-defined routines. If built-in or system routines are selected, the manual entry for that routine is displayed. This language support helps to eliminate the need for interrupting work to locate documentation. As system information becomes more readily available, the programmer is freed to concentrate on the problem and is no longer required to remember possibly arcane search commands.

Keywords. Each keyword has a wealth of language-specific information associated with it. If a keyword is selected, the manual entry for the keyword selected may be displayed, as was done with system routines. However, other displays may result. For example, if a symbol from Ada's Accept statement is selected, the basic syntax with notes on its usage is shown. A complex language such as Ada makes this information most valuable, and its availability is helpful to programmers who work with many languages.

Another approach is to stress the structure of the statement selected. When an element of an If statement is selected, that statement's If, Then, Else, and EndIf symbols are highlighted. The programmer can then determine the exact extent of the statement. For example, if the EndIf highlighted is not the one that logically goes with the If statement, the error can be more easily identified. This technique connects

```
  Const
    MNum = -1000;          { Equals - ( Number of numbers + 10 ) }
  Type
    MNType = Array [ MNum .. -10 ] of Boolean;
  Var
    Mn : MNType;           { Markp Array for Number Table }

Procedure MagicMarker( I : Integer );

{ This procedure recursivelychecks to see if there is something that
  should be saved from collection. }

  Begin { MagicMarker }

    If ( I < = -10 ) Or ( I > 0 ) Then Begin
      If ( I < = -10 ) Then Mn(.I.) : = True
      Else Begin
        If ( I < = MSym ) Then Begin
          If ( S(.I.).T < Undefd ) Then MagicMarker( S(.I.).V )
        End Else Begin
          If ( Not M(.I.) ) Then Begin
            M(.I.) : = True;
            MagicMarker( A(.I.) );
            MagicMarker( D(.I.) );
          End
        End
      End
    End

    End; { MagicMarker }

    Begin { Rubbish }

      Write( 'Please excuse the delay. Garbage collection' );
      Writeln( ' is required and must be performed.' ); Writeln;
      For I : = 1 to MSym Do Begin
        If ( S(.I.).T < Undefd ) Then MagicMarker( S(.I.).V )
      End;
      For I : = 1 to Ap Do Begin
        MagicMarker( Assoc(.I.).F );
        MagicMarker( Assoc(.I.).T );
      End;
      For I : = 1 to Tp Do MagicMarker( T(.I.) );
      { Time to reconstruct the various free space lists }
      Fc : = 0;
      For I : = LowB to HighB Do Begin
        If ( Not M[ I ] ) Then Begin
= = = = >

= = = = >  : Command line
```

Figure 1. An example of embedded selection. The highlighted variable (gray box) has been selected, and all elements composing its declaration are displayed at the top of the screen.

the symbols to the objects they represent, making it easier for the programmer to check for semantic mistakes.

Operators. When operators are selected, some information is provided, but the nature of that information varies according to the language. In Pascal, a programmer might want to learn the resulting type of an expression. In Ada, where operators can be overloaded, the declaration of the operators should be shown, since the operators are really functions. Selecting the assignment operator might tell you whether the assignment is valid. This information is especially valuable when floating point precision is needed, as in scientific calculations.

Remarks. An extension of embedded selection offers a convenient way of viewing a program. The first view shows the main program body. The programmer inspects it and selects a routine to display. The programmer is then shown the declaration of that routine, and is able to inspect its code. A second routine may also be easily selected and inspected. Having a question about a type, the programmer selects it and its declaration is displayed. The programmer can check on the exact syntax of the Do statement by selecting it and is able to see both the manual and the source at the same time. A wealth of information is easily available, with no tricky interface or additional syntax to master.

With direct selection, details can be found easily, permitting the programmer to concentrate on the higher level structures. On-line help for rarely used statements relieves the programmer from hunting for documentation. All these benefits aid the programmer in detecting and correcting faults.

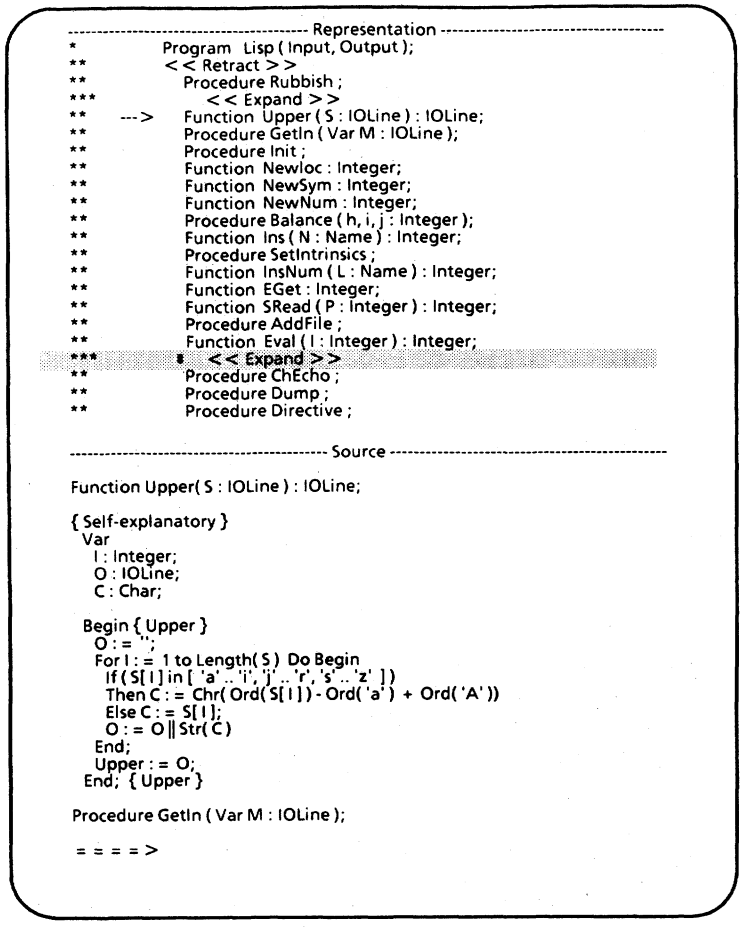


Figure 2. An example of a hierarchical browser. The program displayed is a Lisp interpreter written in Pascal. The source window displays the last routine selected (— >). The cursor is on a special line that provides access to the internal routines.

Hierarchical browser

Browsers are already in use in such systems as SmallTalk-80,[7] Cedar,[8] and MagPie.[9] The browser described here is a hierarchical browser that makes the program's structure more visible.

A hierarchical browser is a representation of the high-level information structure that may be used to access the source code of a program or other text. (This definition is a generalization of one given in the description of the SmallTalk-80 environment.) The table of contents of a book is an excellent example of a hierarchical browser. It provides a structured image of a book by dividing it into chapters and sections. The page number associated with each chapter or section is a short path to each element of the structure.

The editor we developed based on the hierarchical browser has two windows: a representation window and a source window. The source window displays the program code in a normal editor window — in our case, VM/CMS's XEDIT. The representation window displays the headings of the program routines and their nesting level. The programmer sees only the highest level routines at first. If a routine is local to a global routine, it is not shown in the initial display. This structure may be expanded to include lower levels of nesting. Figures 2 and 3 show the hierarchical browser and an expanded structure. Figure 4 shows an implementation on the IBM

The Bettmann Archive

3290. The representation window allows the programmer to select the routine to be displayed in the source window. Note that this approach clearly supports the top-down design methodology, because the representation is one level closer than the source code to the program design.

Isolation and design changes. The hierarchical browser strategy helps the programmer understand the program in two ways:

(1) It facilitates comprehension of the program by showing structured information and the underlying design.

(2) It supports working methods related to standard software design techniques.

Structured information is easier to understand, and high-level information aids comprehension.[10-12] The hierarchical browser acts as a table of contents for the program, thus showing a structured view for the reader. It is hard to think of reading a reference or system manual without a table of contents, but a program never has such a tool.

Studies conducted on reading text have shown that knowledge of the structure is important to text comprehension.[13,14] We think the same principle should work strongly in program comprehension. It is much easier for a programmer to find the design scheme in the structured elements than in the bare source code. Green[15] has observed an inverse correlation between the amount of information shown and comprehension.

Most modern software design techniques would benefit from access to high-level structural elements via a representation window. Normally, to find an object in a program, the programmer must rely on paging, string searching, or using line numbers. As programs grow larger, paging starts to be difficult, line numbers are forgotten or changed, and the results of string searches are unpredictable. Locating the answer to a simple question such as, "Where is the input line converted from lower case to upper case?" requires time and guessing for someone who did not write or design the program. To get the answer, the programmer must interrupt his primary task of understanding the program to find a small portion of code.

When the source code is directly accessible through the representation window, the programmer is freed from the tedious manipulation and switching of contexts. The question becomes, "Which element of the design is responsible for converting an input line to upper case?" and the answer is a candidate routine. The programmer displays the source code for that routine and reads it. Even if the wrong routine is chosen, information is learned because the programmers move to known routines rather than meaningless line numbers.

```
---------------------------------- Representation ----------------------------------
*     Program  Lisp ( Input, Output );
**     << Retract >>
**       Procedure Rubbish ;
***        << Expand >>
**   --->  Function  Upper ( S : IOLine ) : IOLine;
**       Procedure GetIn ( Var M : IOLine );
**       Procedure Init ;
**       Function  Newloc : Integer;
**       Function  NewSym : Integer;
**       Function  NewNum : Integer;
**       Procedure Balance ( h, i, j : Integer );
**       Function  Ins ( N : Name ) : Integer;
**       Procedure SetIntrinsics ;
**       Function  InsNum ( L : Name ) : Integer;
**       Function  EGet : Integer;
**       Function  SRead ( P : Integer ) : Integer;
**       Procedure AddFile ;
**       Function  Eval ( I : Integer ) : Integer;
***      ▮   << Retract >>
***         Function  Equal ( C1,  C2 : Integer ) : Boolean;
***         Function  BuiltIn ( P, QBase : Integer ) : Integer;
***         Function  Uval ( I, QBase : Integer ) : Integer;
**       Procedure ChEcho ;
------------------------------------------ Source ------------------------------------------

Function Upper( S : IOLine ) : IOLine;

{ Self-explanatory }
 Var
   I : Integer;
   O : IOLine;
   C : Char;

 Begin { Upper }
   O : = '';
   For I : = 1 to Length( S ) Do Begin
     If ( S[ I ] in [ 'a' .. 'i', 'j' .. 'r', 's' .. 'z' ] )
     Then C : = Chr( Ord( S[ I ] ) - Ord( 'a' ) + Ord( 'A' ))
     Else C : = S[ I ];
     O : = O || Str( C )
   End;
   Upper : = O;
 End; { Upper }

Procedure GetIn ( Var M : IOLine );

= = = = >

= = = = >  : Command line                    *  : Nesting level
--->       : Last  Routine selected          ▮  : Cursor
```

Figure 3. Expansion in the representation window of a hierarchical browser. The line pointed at by the cursor in Figure 2 has been selected. The internal routines of the function Eval are revealed. The special line has now changed from Expand to Retract to allow the user to remove the information. Note that the nesting level (*) of Retract/Expand lines corresponds to the level of the internals.

The hierarchical browser strategy provides two important features: a structured presentation of the program to the reader and access to the source code through this structure. These two features reduce the complexity of the program, delineate the design, and support working methods related to software design techniques.

Methods of representation. Choosing how to represent the program is clearly critical to the hierarchical browser strategy. In our implementation, we used routine headers for simplicity. Our decision was based on a number of reasons.

- Programmers are familiar with this representation.
- Routine headers map the representation in the order of the paper listing.
- The amount of information is greatly reduced compared with the amount in the program. A 1500-line Pascal program has a representation of about 30 lines.
- Routine headers are very simple to implement. No graphic capability is required from the terminal.

These characteristics, while particular to our environment and experimental goals, reflect common-sense objectives that apply to any environment: The representation should be sensible, and it should strongly reduce the information displayed.

Several levels of representation might be needed. For example, in a 100,000-line program with 1000 routines, seeing only the routines would be better but not sufficient. These routines would have to be clustered into modules to reach a manageable size for the programmer, and then each module would contain a reasonable number of routines. The better the program design, the better the language can support the design and the better this strategy will work.

Obviously, we cannot define a representation that every individual can call sensible. A programmer who has never used a PDL will not find any immediate benefit from seeing such a representation. The representation needs to be defined according to the environment and background of the programmer. Moreover, representations need to evolve with the programmer and his environment, so multiple representations should be available, such as data flow or execution flow.

Integration of tools. Hierarchical browsers help promote the smooth integration of tools and techniques in a software environment. The representation window can display information obtained from other tools, such as a cross-reference table, a debugger, or an interpreter.

Many tools provide information that can be best understood only in context. For example, a programmer needing to verify where a variable is referenced commonly uses a cross-reference table. A cross-reference table gives the programmer the list of line numbers on paper or a way to jump from location to location when used interactively. But it provides little context, and the programmer must remember many things.

If a hierarchical browser showing high-level information, such as routine headers, is used, all routines containing an occurrence of the desired variable can be highlighted, giving a general overview at a glance. The programmer can see if the variable is widely used or is just mentioned in a single routine. Correlating this information with the current focus of attention helps the programmer to understand the purpose of that variable. The programmer can then select the most interesting routine, whether highlighted or not, in any order requested.

If execution fails, the programmer calls the debugger for help. One of the first actions is to trace the routine calls to find the history of execution. A hierarchical browser with an execution flow representation displays that information instantly, providing the necessary context for the programmer. Thus, information is naturally presented, freeing the user from tedious manipulation and allowing full concentration on the problem.

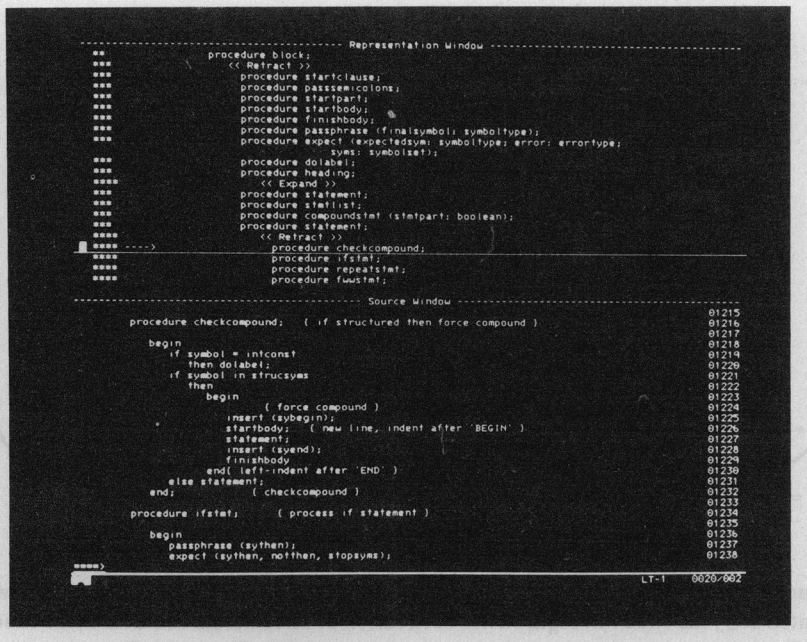


Figure 4. Photograph of the hierarchical browser on the IBM 3290 Plasma Display.

Empirical test

We conducted a preliminary experiment to determine the efficacy of a hierarchical browser display. The display had 24 lines in the representation window and 24 lines in the source window. We then compared

The Bettmann Archive

the browser mode to a listing display mode of 48 lines of program text on the screen in a simple editor. Under both treatments subjects scrolled the program and did string searches. We used a split-plot factorial design to test the differences between the browser and listing modes.[16] With this design, we tested subjects in both conditions, using two sets of experimental materials and four orders of presentation.

Two Pascal programs were used. One program, called Pretty Printer, did Pascal program formatting. It had 1467 lines in 67 modules with a maximum nesting level of four. The other program was a Lisp interpreter, which had 1416 lines in 27 modules with a maximum nesting level of three. Ten comprehension questions, such as "Which routine removes trailing blanks from the input buffer and updates the buffer index?" and "Which variable contains the symbol table?" were asked for each program.

The four orders of presentation required by the experimental design were

- browser mode/Lisp interpreter followed by listing mode/Pretty Printer,
- browser mode/Pretty Printer followed by listing mode/Lisp interpreter,
- listing mode/Lisp interpreter followed by browser mode/Pretty Printer, and
- listing mode/Pretty Printer followed by browser mode/Lisp interpreter.

Subjects and procedure. The subjects were 16 graduate and senior students at the University of Maryland who had written Pascal programs longer than 500 lines. Each was tested individually and was paid for participation in the experiment. Four subjects were randomly assigned to each of the four presentation orders.

For each treatment, subjects received training and a practice session (with another Pascal program) before they began work on the comprehension question. Answers for each question were typed on line, and the computer kept accurate time measures for each question. When the experiment was over, subjects were given a subjective evaluation questionnaire.

Results and discussion. A total of 320 questions were answered by 16 subjects for two programs. Table l shows the means and standard deviations of the times to answer the questions per subject in minutes.

There was a statistically significant interaction at the .05 level between the display mode, programs, and order of presentation. A post-hoc analysis showed that the difference could be attributed to the difficulty subjects had in the listing mode when they were given the Pretty Printer program first.

A review of the questions indicated a difficulty with one question that required using the browser to locate a module at the fourth level of nesting. Some subjects did not fully comprehend how to open the browser to get to the internal modules. The result was very long search times. When we removed the troublesome question and analyzed the results based on nine questions, we got a statistically significant difference for the main effect of display strategy at the .05 level. This difference indicates that performance is faster with the browser display mode.

On the average, Browser users made 5.3 string searches, while listing users made 32.8 string searches, again underlining the efficacy of the representation window in locating desired information.

When asked, "Which editing mode do you feel best enabled you to find the answers to the questions you were given?" subjects favored the browser 13 to 3. When asked, "Of the two editing modes used which do you prefer?" subjects favored the browser 15 to 1. Finally, when asked, "Which editing mode do you feel is more consistent with the way in which you view the structure of a Pascal program?" all 16 subjects chose the browser.

These uniformly positive comments are surprising considering that the subjects had never before seen the browser. The browser's wide acceptance, even in a first-time use, proves that it is helpful in studying 1400-line Pascal programs, and possibly others, to answer comprehension questions. Because the performance speed of the browser was faster and the subjective response more favorable than the listing strategy, we believe that users prefer hierarchical views of programs, databases, knowledge bases, text, and visual information such as maps.

Unfortunately, in spite of these results there were some problems. Browser users had difficulty locating information in routines that were several levels down in the nesting structure. Subjects may need more experience to develop familiarity with the browser. These results need to be replicated with other programs and tasks. However, the positive outcome was encouraging, and the subjects' comments were helpful for refining our future research efforts.

Maintenance applications

The strategies described here focus on browsing programs, but many more issues must be considered when integrating a true maintenance workstation into the management and design practices of an organization. Let us briefly look at one scenario.

The maintenance programmer has two large-screen displays, each capable of showing two 60-line by 80-column windows (Figure 5). The first window contains a log of the 20 to 30 maintenance suggestions that have been made in the past two weeks. By pointing at six to eight of the suggestion titles, the MP produces the full

Table 1.
Mean time (and standard deviation) in minutes to answer comprehension questions using the browser or listing display strategy with each of two Pascal programs.

	DISPLAY STRATEGY	
PROGRAM	Browser	Listing
Lisp interpreter	12.47 (2.66)	13.27 (3.66)
Pretty printer	15.22 (3.80)	19.38 (5.14)

text of each. After reading these, the MP reorganizes the suggestions to display the four or five that relate to the layout of information in a key application screen layout. The MP executes the relevant program on the second window to view the indicated screen and the input data or database contents used. The third window automatically displays the hierarchical browser for the program code, the design information, and a structure chart. The fourth window automatically displays the log of previous maintenance changes to this piece of code and a data dictionary of the program variables. After studying this code for a while, the MP recognizes that one of the variables in the code is incorrect and that the screen layout should be redesigned to improve readability. The MP specifies the program changes and reexecutes the program to view the redesigned application screen. Satisfied that the changes are accurate, the MP adds comments to the maintenance log, sends a report and acknowledgment to the people who submitted the suggestions, and finally sends the full set of information on to an inspector who reviews all maintenance changes before they are applied to the production system. The MP returns to the log of suggestions to choose a new task.

The display strategies we have presented should help programmers to accomplish maintenance tasks on large-screen or multiscreen displays. The strategies can be adapted to smaller displays on mainframe or personal computers.

The strategies offer several advantages. Failure detection is improved by synchronized scrolling of input and output. Change design and isolation is easier with the general view provided by the hierarchical browser. With the detailed information obtained through embedded selection, corrections are made more accurately. Routines are easier to comprehend when shown in their entirety in fused windows.

These strategies do not change the functionality of systems, but allow the programmer to use them more efficiently. They can be applied to existing systems and bring improvements in the work environment. They match the user's needs, provide contextual, general, and detailed information, integrate the tools, and free the user from tedious manipulation and context switching. □

Acknowledgments

We thank Victor Basili, Dieter Rombach, Marvin Zelkowitz, Judd Rogers, and Ruly Arifin for their helpful comments in preparing this article. We also thank Michael Dorsey for his administration of the experiment.

We gratefully acknowledge the support for this research from IBM Federal Systems Division, Bethesda, Maryland.

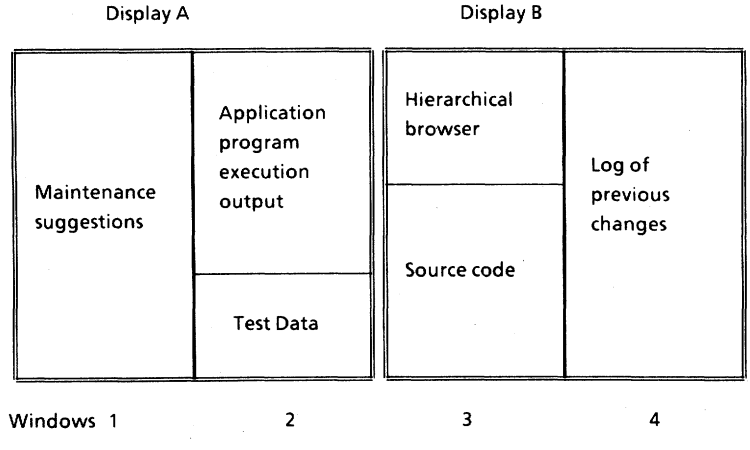
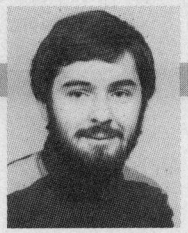


Figure 5. Possible layout for a maintenance workstation using two large displays. Each display allows two sections 60 lines by 80 characters.

References

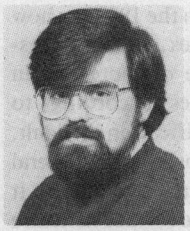
1. C.E. Grantham and B. Shneiderman, "Programmer Behavior and Cognitive Activity: An Observational Study," *Proc. 23rd Ann. Tech. Symp.*, Washington, DC, Chapter ACM, June 1984.
2. S.P. Reiss, "PECAN: Program Development Systems That Support Multiple Views," *Proc. Seventh Int'l Conf. Software Engineering*, Mar. 1984, pp. 324-333.
3. D.H. Rombach, V.R. Basili, and S. Chang, *Methodology for Improving Life Cycle Support by Technique and Tools,* Technical Report, Computer Science Department, University of Maryland, College Park, Feb. 1986.
4. *Standard Glossary of Software Engineering Terminology,* IEEE Press, New York, Feb. 1983.
5. B.W. Boehm, "Software and Its Impact: A Quantitative Assessment," *Datamation*, May 1973, pp. 48-59.
6. T. Teitelbaum and T. Reps, "The Cornell Synthesizer: A Syntax-Directed Programming Environment," *Comm. ACM*, Vol. 24, No. 9, Sept. 1981, pp. 563-573.
7. A. Goldberg, *SmallTalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Mass., 1984.
8. W. Teitelman, "A Tour Through Cedar," *Proc. Seventh Int'l Conf. Software Engineering*, Mar. 1984, pp. 181-195.
9. 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 Environment*, May 1984, pp. 49-56.
10. J.D. Bransford, *Human Cognition*, Wadsworth, Belmont, Calif., 1979.
11. A.L. Glass, K.J. Holyoak, and J.L. Santa, *Cognition*, Addison-Wesley, Menlo Park, Calif., 1979.
12. B. Shneiderman, *Software Psychology: Human Factors in Computer and Information*, Little, Brown, and Co., Boston, 1980.
13. D.L. Horton and C. Mills, "Human Learning and Memory," *Ann. Rev. Psychology*, Vol. 35, 1984, pp. 361-394.
14. R. Lachman, G.L. Lachman, and C.E. Butterfield, *Cognitive Psychology and Information Processing: An Introduction*, Lawrence Erlbaum Associates, Hillsdale, N.J., 1979.
15. T.R. Green, "Programming as a Cognitive Activity," *Human Interactions with Computers*, Academic Press, New York, 1980.
16. R.E. Kirk, *Experimental Design: Procedures for the Behavioral Sciences*, 2nd ed., Brooks/Cole, Belmont, Calif., 1982.



The authors' address is Dept. of Computer Science and Human-Computer Interaction Laboratory, Center for Automation Research, University of Maryland, College Park, MD 20742.

Ben Shneiderman is an associate professor in the Department of Computer Science and head of the Human-Computer Interaction Laboratory, both at the University of Maryland, College Park. His technical interests include interactive systems design, human factors research in programming, database management, and computers in education.

Shneiderman is the author of *Software Psychology* (Little, Brown and Co., 1980), several other books, and over 100 technical research papers. His next book, due out in June, is *Designing the User Interface: Strategies for Effective Human-Computer Interaction,* (Addison-Wesley).



Philip Shafer is an undergraduate research assistant for the Human-Computer Interaction Laboratory at the University of Maryland, where he is pursuing a BS in computer science. His research interests include programmer workstations, programming language design and implementation, and operating systems.

Roland Simon is a faculty research assistant in computer science at the University of Maryland. His research interests include software engineering, distributed systems, algorithms, and human-computer interaction.

Simon received a diploma in mathematics from the Ecole Polytechnique Fédérale de Lausanne (Swiss Federal Institute of Technology of Lausanne), where he subsequently worked as a research assistant in computer science.

Linda J. Weldon is on the research faculty of the Center for Automation Research at the University of Maryland. Her research interests are in engineering psychology and human factors.

Weldon received a BA and an MA in psychology from California State University, Chico, and a PhD in experimental psychology from the University of Maryland.