The user interface in a hypertext, multiwindow program browser

Richard H.C. Seabrook* and Ben Shneiderman

The program browsing problem is discussed, with particular emphasis on a multiple-window user interface and its implications for recording acquired knowledge, navigation, and attention-tracking. Hypertext systems are considered as an implementation of browsing techniques for nonprogram text. A classification scheme for text-viewing systems is offered, and then browsing is discussed as a nonintrusive, static technique for program study.

Multiple techniques are synthesised into a coherent plan for a multiwindow program study tool, based on theories of program browsing and the use of hypertext. A test system, HYBROW, emerged from the plan for studying the application of several hypertext multiple-window techniques to program browsing, especially window replacement. HYBROW is a hypertext, multiple-window program browser. This generic tool is applicable to any source language, although certain aspects of the preprocessing and the hierarchical browser presentation are specific to the C language. The tool permits opening an arbitrary number of text windows into an arbitrary number of files, rapid window switching, multiple-window search, placemarking, automatic screen organisation, and services for the creation, maintenance and production of study notes. An informal usability study was conducted.

Keywords: user interfaces, windowing systems, hypertext, browsing systems

Program source-code texts have many characteristics in common with other literatures: they are made up of words and phrases taken from a well-defined common language; they often have a distinct beginning, middle and end; they are often constructed by a single person; and they tend to include author's annotations and explanations. The principal difference between programs and other literature lies in the intended communication. Most programs exist, not for conveying ideas among people, but rather for conveying complex instructions from people to machines, although there is a recent trend towards 'literate

Department of Computer Science, University of Maryland, College Park, MD 20742, USA. Tel: (301) 454 4244

*Currently at Anne Arundel Community College, 101 College Parkway, Arnold, MD 21012, USA

0953-5438/89/030301-37 $03.00 © 1989 Butterworth & Co (Publishers) Ltd
programming' directed at solving the programmer-to-programmer communication problem (Knuth, 1984; Brown, 1988). However, instructions must often be changed to meet new requirements or new hardware/software environments. Unfortunately, such changes often occur long after the creator of the instructions has departed, resulting in a need for others to modify the programs, or use them as a basis for redevelopment.

A further reason to read programs is when the inevitable bug appears. New programmers must understand what the instructions do, their relationship to each other, and their relevance for the current environment. They must then modify the existing programs or develop new ones, after which the previous versions are seldom, if ever, seen again. Studying obsolete programs has not been considered a fruitful endeavour, except for teaching purposes in academic environments, although this situation may be changing due to current emphasis on reuse (Fischer, 1987).

Before microcomputers, program study techniques were few, consisting primarily of paging through source code with a text-editing system and generating listings for offline perusal. With the wider use of computers has come a greater variety of techniques for creating and studying text of all kinds, including program source code. One technique that has been used successfully in other literatures is hypertext browsing with multiple windows. Hypertext comprises a body of techniques originally developed for prose text (Nelson, 1980), and more recently applied to other forms as well (Garg and Scacchi, 1987; Bigelow and Riley, 1987). Hypertext systems provide multiple access (both one studier to many text segments and many studiers to the same text segment), ease of use, preservation of the source material and its ownership, and novel search techniques.

Dynamic program study techniques involve running the programs to collect data or gain overall impressions of what they make machines do. All other activities are static, in that they consider the program as an unvarying body of text, from which an understanding is to be gained. Theories differ on precisely what an understanding of a program means (Littman, 1986), although it involves an ability to predict the behaviour of a machine running the program. It is not clear that dynamic techniques either provide a better ability to predict machine behaviour than static techniques, or provide it sooner in the study process.

Intrusive techniques permit modifying the object under study in some way in order to compare the result with the unmodified version, and thereby study the effect of the modification. The operant word here is 'permit' — a study technique is intrusive if any possibility of modifying the object under study exists, even if such a capability is not normally used.

Being able to view several text segments at the same time in multiple windows is important for program browsing. This is partly due to a program's relative degree of modularity (its division into separate independently compiled code segments), and partly due to the way programmers think. The 'planning model' of programmers' cognitive behaviour (Letovsky and Soloway, 1986; Rist, 1986) recognises programs as organisations of well-defined 'delocalised' fragments, each of which is designed according to a plan. In the planning
model the logical design of programs is a process of selecting the correct set of plans and integrating them into a cohesive whole using a programming language. Such a view implies strongly that understanding a program amounts to breaking the code into many localities and realising the plan under which each was devised. Plans are relatively 'high-level' or 'low-level' depending upon the generality of their actions and goals, and may be 'localised' or 'delocalised' depending on whether they are implemented in a single code segment or several separate ones.

The planning model view of a program as a large collection of individual fragments implies a high degree of locality of access to source code, by which is meant that often relatively small groups of instructions will be of interest in explaining the work the program carries out. Whenever the instructions are part of a delocalised plan, several localities must be examined simultaneously. However, even when all the instructions are together, employing a modern computer language such as the C language implies a need to examine several localities in order to understand a single statement's symbols. For example, a programmer may need to compare local variables used in the statement with their declarations, global variables with their declarations (often in a different file), preprocessor variables and macros with their definitions in several header files, and function-calling sequences with their declarations.

Whenever several program parts need to be examined together, a single-window system implies a great deal of switching among various locales, while a multiple-window system provides several views on the screen simultaneously. Thus, a single-window system demands more cognitive work of the studier than a multiple-window system in two ways: remembering the content and meaning of previously examined segments to compare them with the one presently in view; and remembering their relationships in order to recall how to get back to previous windows for another look.

Moving from window to window in a hypertext system is termed 'navigating' and is supported typically in several ways: at any time the studier may select a link to another window embedded in the current window, search for a word or phrase in another window or several windows, return to the previous window by reversing the effect of the most recent link selection, or employ an external mapping device such as an index to select one from a large number of available windows. Single-window hypertext systems limit navigation to a single step in any direction, while multiwindow techniques assume that several views are active at any time and include features that assist the studier to select a set of them.

Hypertext browsing systems provide a sufficient level of access for studying programs. Littman (1986) described two strategies employed by his experimental subjects during a program maintenance task — systematic, and as-needed. In Littman's view.

'The programmer using the systematic strategy traces data-flow through the program in order to understand global program behavior. The programmer using the as-needed strategy focuses on local program behaviour in order to localize study of the program.'
In short, those who study programs go about it in ways that are similar in many respects to those applied to other bodies of literature: top-to-bottom word-forward study; selective access to particular words and phrases by scanning or using some external device such as a cross-reference listing; and some combination of the two. While the emphasis chosen often depends more on the magnitude of the task (Soloway, 1988), urgency of the situation at hand and the studier’s level of expertise (Littman, 1986) than on features of the program language or programs in general, a program browsing system needs to include a relatively small number of access options in providing support.

Our main concern is to consider the usefulness of existing techniques for a multiwindow hypertext system designed to assist the study of program source code. A hypertext is a literature organisation comprised of a large number of relatively short interrelated text segments, to which readers may refer in any order. It is incumbent on the organiser of the text to install links in each segment to support moving rapidly to other relevant segments, and also ways of avoiding the necessity of such jumps whenever possible by providing summary or abstract information from the destination. Such systems often provide an external device, such as an index of segment titles or key words, that makes possible a non-link-order perusal of the material if the user so desires. Many systems include full-text searching to support access in ways not provided by explicit links (Cognetics Corp., 1987).

A significant byproduct of this study is HYBROW, a research tool that supports nonlinear program text perusal. Using a multiple-window hypertext code browsing tool has another fortuitous advantage: usage patterns may be captured to reveal attention patterns, and gain objective data on the reader’s relative interest in different code segments.

**Previous work**

The design of a program browsing tool ought to reflect what is known about how programmers study source code (Soloway, 1984). The literature in this area is quite varied in approach (Curtis, 1986). The field has attracted the interest of human factors theorists (summarised in Shneiderman (1980)), cognitive psychologists (see, for example, Wiedenbeck, 1986; Letovsky, 1986), computer scientists (Weiser, 1987) and software engineers (Reiss, 1987). While studying program code is something every experienced programmer does, very little is known about preference for or success of one technique over another. Littman (1986) found a clear advantage in success rates of those who used a systematic study strategy over those who used an ‘as-needed’ strategy. Although the sample was small and the program rather short, results were distinct: all those who studied the entire program modified it successfully and all those who limited their attention to particular locales did not. While these results argue for a thorough examination of a program before modifying it, the large size of many programs and short time available for many maintenance activities tend to discourage the practice (Soloway, 1988). This paper reviews techniques in several broad groups — those that take advantage of software tools developed
either for other texts or specifically for program source text; and offline techniques that involve analysing printed listings and output.

Using editing tools for program study
Programmers apply a great many software tools to the problem of creating and studying program source code, with varying degrees of success (for example, word-processing text editors, context-sensitive editors, syntax-directed editors and complete programming environments). Many of these are noted for their extremely powerful searching and formatting capabilities. However, multiple simultaneous presentation of source text, an area found to be of special importance in cognitive psychology studies (Norman, 1986), is often quite limited in text editors. Although a few maintain current positions in dozens of text buffers (EMACS, for example), simultaneous presentation for comparison purposes is often limited to a few tiled windows in either a horizontal or vertical split format.

Word-processing-style text editors have often been applied for studying program source code in order to take advantage of their sophisticated string search and replacement capabilities. Several classes of editors have been developed specifically for programmers that provide text entry and formatting support based on the syntax of the language in use. Context-sensitive editors such as EMACS are text-oriented in that they provide templates for control structures and declarations that are useful for avoiding logic errors, although using them is not mandatory (Bahlke and Snelting, 1986; Stomfors, 1986). Such systems permit entry and manipulation of partial logic structures and incomplete definitions while preserving the text of the program for later analysis by external compilers. Syntax-directed editors are logic-oriented in that they construct and maintain a logical parse tree during program entry, and use it together with the language’s syntax to limit text entry to verifiably correct statements. While helpful, these features do not necessarily aid productivity (Lang, 1986), they render an uneven level of support across computer languages, and they have been developed only for the most popular languages. A few editors provide in-built macro languages for customising many actions and various aspects of the user interface.

Many dynamic systems have appeared in recent times that assist the programmer to create and modify source code. Two common examples are online debuggers and CASE tools (Insup, 1983). Online debuggers permit setting breakpoints and watch lists for detecting changes in the content of important variables, and for studying program flow patterns. Such systems are often language- and machine-sensitive, which discourages general use.

CASE tools (Leblang and Chase, 1984) such as the Unix Programmer’s Workbench, the Ada Language System developed by Softech, and the Cedar system, are programming environments that assist the system designer to formulate processing concepts, manipulate them, and turn them into programs that carry out the intended work. Typically, such tools involve the construction of a model of the desired processing, and adjusting it to optimise outcomes. They are designed to be most useful during the creation of systems, before the source code of particular programs exists (see, for example, Reiss (1984)). Few

Seabrook and Shneiderman

305
such tools provide any support for retrospective study and analysis, and those that do often construct their internal models based on compiler-generated products such as module interconnection language segments and cross-reference maps, which are seldom available for older programs with a considerable history of use.

A few tools have appeared that are capable of constructing automatically a tentative flow-of-execution path (Temin and Rich, 1987; Bigelow and Riley, 1987), based on the arrangement of module names in the source code (see Figure 4). A constructed path is usually tentative at best due to conditional branches and loops depending on characteristics of the data encountered when the program is run. While this feature is extremely useful for analysing older code, its use presumes that logical interconnections between modules are explicit; that is, via calls and returns embedded in the source code. However, in a growing body of ‘object-oriented’ systems today, modules communicate by passing messages (Smalltalk-76 in Teitelman (1984), Smalltalk/V (Digitalk 1986), Goldberg (1987), Durant (1988), Petzold (1988)) between classes of objects, particular instances of which do not exist until the program executes. In most such systems, a module may receive many different messages depending on its identity and the class to which it belongs, and two modules in different classes may perform entirely different actions after receiving identical messages.

### Static analysis of programs

Static analysis includes both manual, strictly offline activities and using a few recent tools that produce helpful study aids (Reiss, 1987). While much older as a class, static techniques have scant treatment in the literature (Munro and Robson 1987). Despite this fact, program source code lends itself to very powerful techniques to support its perusal (Ryder, 1987; Frakes and Nejmeh, 1987). Static analysis of program source code comprises a body of techniques extending from desk checking to using maps produced by automatic processing of module interconnection languages. Typically, offline techniques are supported by source listings, cross-reference listings, annotations (comments-in-code), and interviews with the original developers of the program under study.

### Desk checking

Decades old in practice, this technique consists of reading source code listings and annotations, and mentally computing the results of various processes. Its chief advantage is also its greatest limitation, in that the organisation of the study materials is left to the studier, who may peruse as many source listings as desired and the mental workload involved permits. Physical limitations often contribute to limiting the size of programs that are analysable by this means.

### Cross-reference analysis

Cross-reference analysis involves tracing control flow and variable assignment by means of an alphabetised listing of all the variable names and function calls in a program, together with the source code line numbers on which they appear. Such listings are often produced routinely by compilers, and many
recent systems include capabilities for reconstructing them automatically from the source code when they are not available.

**Call analysis**
Call analysis involves the use of a tree-like indented outline structure that identifies all the call relationships between modules; that is, places in a module where (if control reaches that point) it may call another module into execution, and perhaps resume when it finishes. The successful use of this technique in a static fashion depends on the studier's abilities to predict those parts of the conditional flow of execution which depend on data characteristics that may be detected during execution, as mentioned above.

**Dataflow analysis**
This technique is based on a network structure that identifies all data constructions and the occasions where particular data items or their addresses pass between modules, together with the circumstances under which the flows take place. Once again, an understanding of such flows in the abstract depends on an ability to predict a program's actions during execution.

**Browsing source code**
Despite their dependence on the studier's ability to predict execution flow patterns, static techniques that involve source code analysis are employed widely, and can provide a great deal of information about what a program does (Weiser, 1987). A major difficulty inherent in browsing the source code is its size. Many programs are characterised by multiple source files, each of which may contain thousands of modules. While the organisation of modules into source files usually implies that they are components of some overall program function, dependent relationships between modules in different source files are common, especially when they perform utility functions, such as opening and closing files.

The complex interrelationships between modules in the composite body of source code imply a nonlinear form of text perusal and a multiple-access approach. Further, the direction of study (that is, what to look at next) is not always predictable, especially in the case of object-oriented programs, as mentioned above. In such cases, the users must have sophisticated search capabilities to discover widely separated occurrences of the same module name, variable name, or message type, and some means of directly comparing such occurrences when they are interdependent.

**Hypertext systems**
Hypertext comprises a body of techniques, which has received considerable attention in recent years, for supporting access and perusal of text and graphics (Nelson, 1980; Conklin, 1987). Techniques used in such systems are of two main kinds: organisational techniques, which effect the content and arrangement of the text in the system; and interactive techniques, which assist the ultimate user of the system to gain access. Hypertext systems developed for online reference and research are characterised by nonlinear text perusal, and therefore provide
a body of techniques to support the study of program source code. Some hypertext systems are encyclopaedic, for example Hyperties (Shneiderman, 1987), in that they are organised for browsing textual information according to predesignated conceptual links. Hyperties on the PC is primarily a single-window system, although multiple windows are used on the Sun version (Lifshitz and Shneiderman, 1987). Hyperties on the PC provides a suitable model for implementing a hypertext program browser, but annotation and multiple-window access would be useful additions.

Multiple-window source code browsers
Several systems have been created that provide a multiple-window browsing capability for program source code, for example Smalltalk (Ingalls, 1978) and Cedar (Teitelman, 1984) of the Xerox Palo Alto Research Center, and Neptune (Delisle and Schwartz, 1986) of the Tektronix Computer Research Laboratory. Both tiled and overlapped windows are used. Smalltalk maintains a hierarchical classification of all defined objects and provides a Class Hierarchical Browser utility for examining the variables and methods contained in each class, and for editing them as necessary. Cedar combines '... high-quality graphics, a sophisticated editor and document preparation facility, and a variety of tools for the programmer to use in the construction and debugging of programs' (Teitelman, 1984). Neptune is a system for CAD applications, based on the Hypertext Abstract Machine (HAM), 'a generic hypertext model which provides storage and access mechanisms for nodes and links' (Delisle and Schwartz, 1986).

Cognitive issues in program study
A hypertext multiwindow environment can provide a full range of support services for program browsing. The nonintrusive, static nature of hypertext systems coupled with a display capability that permits simultaneous presentation of several source code segments, notes and navigation aids comes closer to providing program studiers with flexibility than other existing environments. Ideally, the proof of such a claim would rest upon gaining a relatively higher degree of program understanding using this environment than using other systems. Unfortunately, what we mean by 'understanding a program' or 'sufficient level of understanding' is not clear, and may range from being able to predict the behaviour of a program in particular circumstances to constructing a complex mental model (Letovsky, 1986).

The offline study environment
Traditional program study methods included a great deal of offline work. This is not entirely due to the absence of good interactive tools — it has many advantages as well. In a desk study of a large program, the number of code segments the studier may consider simultaneously is limited only by the desk size. Placing listings side-by-side allows easy comparisons with low mental workload (Norman et al., 1986). The studier may write notes on the listings, draw diagrams and create other aids to traversing the code. The pattern of study
is completely up to reader, who may move from segment to segment at will, or return to earlier segments. The use of printed listings implies a stability in the object under study — there is no possibility of accidental modification, and all changes made to the paper copies are tentative.

The principal disadvantages of offline study are potential disorganisation, search limitations, and work space obsolescence. Experienced programmers use multiple forms of information during program study, for example listings, notes, and screens (Grantham and Shneiderman, 1984). Since the organisation of the offline materials is left completely to the studier, it is easy to get lost in stacks of listings and not recall where something was in order to get back to it.

Searching during software study typically has three phases: selection, detection and interpretation. The searcher selects a pattern for search, scans through a large amount of text looking for occurrences of the pattern, and having found it makes certain comparisons or interpretations about whether the occurrence is a desirable one. If it is, the searcher either stops or proceeds to detect other occurrences. When no occurrences of the search pattern are found, the searcher might stop or return to the first phase for selecting a new pattern.

Detecting a search pattern during offline program study is highly user-dependent, in terms of experience, skills and general well-being. While programming language syntax often provides clues to where to look for particular kinds of things (such as identifier declarations, function calls) within a program, searching through a large number of modules is tedious at best. Occasions that demand such searching are common; for example, finding out which of a large number of routines call the one under study, locating all the places where a particular global variable is used or modified, and so on. Following detection, interpretation is actually easier during offline study than in most online systems, especially those where only a single window is provided, as in this case, the searcher must remember the salient features of several code segments in order to make comparisons.

Whenever a program supported by an offline study environment is modified, new listings of all affected parts must be created and the previous versions discarded, along with any marginal notes and diagrams that may have been entered on those listings during analysis. Keeping all drawings and marginal notes in a separate place, such as a laboratory notebook or central documentation file tends to avoid eliminating notes prematurely while causing another problem — retaining obsolete notes whose code segment has been deleted or modified in some way.

The support services provided by an interactive environment for program study should retain all of the flexibility and power of offline perusal of printed listings, and solve some of its organisational, searching and note-keeping problems as well. Ideally, the services should be usable on programs in any language, during all kinds of study and maintenance tasks, and execute in most popular hardware environments.

Language implications
After almost 40 years of software development, libraries contain programs in several hundred computer languages and variants. Any one of these might be a
candidate for study. Our aim is for generality, that is to provide as few language-specific constraints on a support tool as possible. Such constraints might arise in several ways.

Given a source program, the way one proceeds to acquire an understanding of it may vary considerably across computer languages. For example, one might choose to scan the Data Division at the top of a Cobol program before looking into paragraphs in the Procedure Division, while at the other extreme, analysing a Pascal program might begin with a brief look at the text of the top-level procedure (at the end of the program). Even with the choice of language fixed, study strategies are known to vary considerably across individuals (Littman, 1986) based on their prior history of success in finding bugs or making major modifications. Thus, without a predictable starting point, a language-independent browsing tool must provide immediate access to any part of the code.

Programming languages vary widely in their use of special characters to represent variables, operations and data. Thus, the full set of printable ASCII characters must be available for search and display. A tool for program browsing must accommodate source files of many sizes. Despite the fact that most languages support today’s widely-practiced modular technique — breaking a program into many single-function modules — it has not been applied consistently over the years, making the size of source files unpredictable.

**Top-down study**

An effort to study an entire program may be conducted in a ‘top-down’ fashion, starting with the main or driver routine and proceeding from there to the routines which it calls, then the ones that they call, and so on. The sequence of calls from the top to all of the routines that do not call others is a particular kind of mapping called a ‘call tree’. At any point in the tree, the studier may choose to proceed either vertically down the chain of calls in a depth-first fashion, or to consider all of the routines that a particular one calls, breadth-first fashion, or some combination of the two. However, the process of moving from one routine to the next is usually an orderly one that follows the flow of control.

**Bottom-up study**

Correcting an error or introducing a modification of a program’s behaviour usually starts from the point where the behaviour is most likely to be manifest, and proceeds from there to higher levels in the call tree in attempts to determine what other routines may be affected. This kind of study is often carried out in some haste, due to operational pressures for improved performance. Unfortunately, the bottom-to-top progress of such a study is often impeded by the programming practice of ‘open subroutines’ where a particular routine may be called any number of times by others. Lacking a view of the entire calling tree forces searching through all the remaining routines in the program to see which ones call the one at hand. Even with a call tree map to locate possible search targets, this kind of search usually involves comparing variable names and assignments in the calling routine and the one under study to select what code segment to study next.
Searching for logical links between code segments is becoming more common with the advent of object-oriented programming techniques, as noted above, where the identity of a message sender or receiver may be ambiguous. In these systems a more useful view might be a mapping according to message types rather than explicit calls — isolating pairs of routines that send and receive certain messages.

Experience factors
The studier’s experience is also known to affect the general conduct of the study (Wiedenbeck, 1986). Novice programmers often read the entire program text where more experienced programmers appear to pay attention to frequently occurring code patterns, such as loop and counter combinations, often focusing on certain key lines or ‘beacons’. Littman (1986) observed a distinct improvement in success rates for subjects who employed a ‘systematic’ strategy over those who employed an ‘as-needed’ strategy, the former tending to have greater experience. Together, these results imply that experienced programmers are likely to study the salient parts of a larger number of program modules attempting to gain a broad view of program function, while novices are more likely to read the entire code in a smaller number of routines.

Stable source text base
Static techniques, in view of their breadth and simplicity, provide a good basis for a program study tool. The dynamic nature of an analytic tool has few advantages for the features the user will need to support software browsing, since these features primarily affect the studier’s abilities to test and change the object under study. Such an environment tends to complicate the system’s support for configuration control and other areas not directly related to software study. For example, saving multiple versions of a module under test might be desirable in order to select the best one or back out some inopportune choice of code; the nature of the programming language may imply the use of syntax-directed and other sophisticated editing capabilities; trial execution implies incorporating or communicating with compilers and linkers; and retaining study notes in the system with links to specific code segments implies some way of determining when the code has been moved about or deleted (more on this topic below). In general, features to support dynamic analysis appear to be extensions of those for static analysis.

Notes base
Most studiers of programs make notes about them as they go along — quite often on the printed listings — not trusting themselves to remember all important details. To support this activity, a system that prohibits changes to the source code must provide a notes space elsewhere, in this case a dynamic space that may be studied, edited, and changed at any time.

Ideally, a program studier will record thoughts about a particular piece of code and its relationship to other code segments, using a predictable format and keywords that enable rapid search and retrieval. The studier will use the vehicle to identify problems for future study, speculate on the effect of proposed
modifications, and record information related to the program components or the entire system (see Figure 1). The system should record the time, date, and person who created each note, and the text segment to which it pertains. Besides text, a good browsing tool might permit the user to create brief sketches with a graphics system and insert these as well. With frequent and careful use, the collection of notes on a program under study should become almost as important as the program source code itself in a study of how the program operates (Ambras and O’Day, 1987). In fact, as more studiers of the same program review earlier notes and make their own, the growing notes base should become the repository of knowledge about the program. Thus, to take advantage of existing knowledge about a program, the notes should be available for browsing as well as the code itself, ideally side-by-side.

Providing a note-taking facility in a source code browsing system is a passive technique — it does not guarantee that knowledge about programs under study will accrue, since the studier may elect not to use the facility at any time. Even with considerable use, a free-format text entry facility does not guarantee that any particular thoughts about the program will be recorded. This situation is less than the best since program studiers may not use it. Without notes, each new studier must start from scratch to acquire an understanding.

Two kinds of solutions to the passive note-taking problems were identified.

```c
short n;

n = GetWindowWord(hWnd, GWW_ID);
switch (iMessage)
{
    case WM_KEYDOWN:
        if(wParam == VK_SHIFT)
            SendMessage(GetParent(hWnd), W_DOWN);
        break;

    case WM_KILLFOCUS:
        SendMessage(GetParent(hWnd), W_DOWN);

    default:
        return CallWindowProc(EDITPROC(hWnd, iMessage, wParam, lParam));
}
```

Figure 1. Recorded notes about a program segment are on the right side of this HYBROW screen, overlaying the related program segment.
— requiring notes-entry with varying degrees of harassment, and capturing the user’s interaction. The first solution involves presenting a notes window at key points in the interaction (such as when moving between windows) and requiring an entry to complete the transition. A weaker form of the same solution is to permit cancelling or ignoring the notes window whenever it appears. Neither technique is likely to be looked upon kindly by anyone in a hurry.

A hypertext system, especially one with scrollable, variable-sized windows and attention tracking (see below) can provide an added dimension to knowledge-retention features — capturing the sequence of text segments to which the user of the system paid the greatest attention. Several systems, Neptune for example (Delisle and Schwartz, 1986; Conklin, 1987), have included capabilities for automatically recording this information. It would be relatively easy to create a session log of byte ranges, together with the elapsed time they were on the screen and a list of the other byte ranges on the screen at the same time. Such a log, built up over time, would support producing a map of the text segments that received the greatest attention, and the set of text segments most closely related to each. We introduce the terms 'significance' and 'affinity' to describe these concepts. It remains for additional research to determine whether having a significance and affinity map of an extended set of source code contributes in any way to rapid access or understanding of the program.

```
exit(0);
}
strcpy(infile.argv[1]);

if((fp = fopen(infile,"r")) == NULL)
    exit(1);

while((c=getc(fp)) != (char) EOF)
{
    chars++;
    if(c == '\n')
        lines++;
    if(c != ' ' && c != '\t' && c != '\n' && c != ',')
        inword=1;
    else
        if(inword)
            }
```

Figure 2. Notes links in the HYBROW source code window appears as <N> on the right side of the screen. Placing the cursor on the first occurrence of <N> causes it to turn inverse.
The notes retained in a browsing system may range in subject matter from broad general impressions to brief explanations of a few characters in a specific code segment. At the specificity end of this range, the system should provide an easy way of moving from the note to a window containing the segment (see Figure 2), or from a segment window to any of its corresponding notes. Storing the note with the code is one solution to this problem, although this implies modifying the source code content. All other ways involve linking notes to code segments, whether the notes are stored separately or together in a large file. Other advantages to having notes links include being able to produce annotated, printed listings of the source code, and being able to consolidate all the notes about a particular code segment quickly.

However, any system that provides a note-linking capability must come to grips with several policy decisions inherent in such a scheme. If a code segment changes or moves to another file, its note-links are likely to be incorrect. The notes associated with deleted code segments may no longer be useful, and their links will be meaningless. Although a hypertext code browsing system may not permit such code segment changes itself, it must provide for the possibility of obsolete links when the notes file is opened. Actions might include making an automatic adjustment, converting the note’s relevance to ‘general system’, or notifying the studier of the linkage difficulty. Another policy governs actions during notes editing. When a note is modified, the old version may be replaced or retained as a superseded note following the practice of many experimental laboratories.

As mentioned earlier, the notes in a browsing system should be available for browsing as well as the code segments (alongside, if possible). Access alternatives might include a full-text search for a particular string, search by date, and linkage search to identify notes that refer to the same file. If the system provides a structured note-taking capability, using for example subject keywords, studier IDs and date stamps, then more sophisticated search techniques might be provided.

Mapping program code

Any translation of the original code of a program, whether it reduces the total amount or not, is a mapping. Several mappings are common, as noted above: cross-reference listings, call trees, and data flow diagrams, for example (Munro and Robson, 1987). Such mappings are not intended to replace the original code, but merely to provide an additional means for gaining access to it (Bigelow and Riley, 1987). A browsing tool for source code ought to provide all such facilities and others as well, for optional simultaneous presentation (Goldberg, 1987). For example, several systems (e.g. Smalltalk, Cedar and Magpie) provide a hierarchical browser (Shneiderman, 1986). This feature of browsing systems is a mapping of important features in the body of text under study, similar in concept to an index or table of contents in other literatures. Hierarchical browsers are most useful when presented in close proximity to the text, serving to focus attention and guide access. For this reason, a hierarchical browser presentation often involves tiled windows.

The idea behind most mappings is to provide the user with a summary view
of some aspect of the program — the way the functions call each other perhaps, or the way they pass data. It is held widely (Shneiderman, 1986; Letovsky, 1986; Reiss, 1987; Goldberg, 1987) that having such information about module interrelationships will permit the studier to follow the flow of control more easily in any particular processing situation, ignoring routines that distract attention from the chosen track. A useful map ought to reduce much of the program detail (Shneiderman, 1986), and provide only information relevant to the reason for the study. To be helpful, the map must also be easier to read and understand than the program itself.

Two kinds of software maps are common: feature and module. A third kind, navigational, may be required to support switching among a large number of related windows. The hierarchical browser discussed above is an example of the first kind, which includes all those mappings constructed on individual program modules from actual text features, such as loops and conditional statements. Such maps support understanding the control flow and data passing in isolated modules. Such maps can be constructed automatically by the system when the program text is loaded. The second kind includes maps that show the relationship between modules as determined by their interaction and communication activities. While in traditional programming styles relationships based on explicit calls and returns may be mapped automatically, certain newer styles make this difficult. For example, modules developed in an object-oriented-programming style often communicate by exchanging messages that pass through a central dispatcher that determines routing during execution. Message destinations may be particular objects with an identity assigned at run-time, or classes of objects. For this reason it is frequently difficult or impossible to identify particular message recipients by browsing the source code.

Navigational maps can be a way to move among a large number of code segments, retrace one’s steps, and select possible routes for additional study. Such a map would relate program text segments to file names and windows, provide orientation in the current display, and assist the studier to recall what was on the screen earlier in the interaction, perhaps avoiding the need to retrace unnecessary steps. A code browser that permits many windows to be open at the same time has special responsibilities in this area, as discussed below.

Multiple-window issues

Studying a large program implies that many localities will be of interest, often several at the same time. For example, comparing the logic in several code segments, checking an assignment statement against variable declarations, reading notes alongside the code to which they refer, and comparing the syntax of a function call with the function’s definition. This accounts for much of the use of offline listings as a study technique, since the studier may view as many segments at the same time as desired and even cut them up to take greater advantage of limited space. However, as the number of listings grows, so does the need to organize and keep track of them, a factor that places an upper limit on the number of segments in view. An interactive environment may provide
similar support by permitting an unlimited number of code windows to be open at the same time, and organisational support for moving easily among them.

Using windows for code segments is common among text editors, although the number, size and positioning of the windows is often severely limited. More powerful systems (Reiss, 1987) are now appearing that provide the flexible windowing required, although they often have large screens with which to provide more organisational support. A text window is often represented as a rectangular box on the screen with a certain border style and a caption in the top border (Myers, 1988).

If a window has its own selectable actions, markers appear in the caption border area. Menus of actions are often included with windows to conserve border space. A menu is a list of actions represented by words that appears when the mouse is clicked over its distinctive menu marker in the border. Selecting an action from a menu or selecting another menu makes the menu disappear again.

When a new window is opened, anything that was on the screen within its scope is no longer visible, giving the appearance of the window overlaying older material, like sheets of paper on a desk. As soon as a window is removed, any characters that formerly occupied the screen space are restored to the screen.

One or two windows pose few organisational problems. As the number of open code windows grows, it becomes increasingly important to identity their content and provide clues to their relationships with other windows. Window labels and placement on the screen help in this regard, with more important windows appearing nearer the top of the screen.

The number of simultaneously open windows is limited, not only by the number of displayable characters on the screen, but also by the smallest horizontal and vertical size a window ought to be in an application and still present a useful amount of information. For program text, this minimum size is dictated by the format of typical program statements. In most programming languages, every character is significant—none may be ignored when studying a program, except perhaps for some words in comments. This implies that in any language where program statements are written horizontally, the windows should be as wide as possible to avoid moving the window to see more of a statement.

An isolated statement from an arbitrary program is seldom enough to determine the program’s function. It is often necessary to look at several statements together in order to grasp what is going on. Also, modern programming languages provide special marking capabilities for the beginning and ending of logical blocks of statements, for directing the logical clustering to the compiler. Baecker proposed taking advantage of block markers and other C-language features to enhance source code presentation (Baecker, 1986). Typical blocks may extend from a few program lines to dozens, depending on the action being carried out. The lower end of this range indicates a minimum vertical space for a window of about 4–5 lines.

The limitations upon the minimum size of a window and the number of
screen lines (25 in small screens) combine to limit the maximum number of windows that may be in view at the same time. Often this will be fewer than the number of windows desired to be available. Many systems solve this problem by allowing windows to overlap, partially hiding windows that are not of immediate interest. Others permit replacing an active window with a small symbolic representation which, when selected, restores the window to the screen. Many systems incorporate both strategies. Such features solve the many-window problem by extending the meaning of 'open' to include many levels of openness, rather than a simple binary relation.

Levels of openness
A fully open window is one that is in full view, occupying most of the screen space. Such a window should include material of immediate interest. Next is a window that occupies a somewhat smaller space — perhaps containing text related to other similar-sized windows on the screen, none of which is uniquely of interest. Third is a partially obscured or overlapped window. Such a window should contain material that has recently been considered or is likely to be considered again soon, but is not of immediate interest. The amount of overlap might be affected by several factors, including the number of windows on the screen, how many have been selected more recently, and so on. A completely overlapped window presents a particular problem of being open yet invisible. Fourth is a window that was used recently or may soon be of interest that has been replaced temporarily either by a very small, virtually unreadable version of itself, or by a symbol or icon. The small size usually requires placement in a fixed 'staging' area on the screen where they can be found easily when needed. The icon staging area should be as small as possible in order to preserve screen space for windows at higher levels of openness.

Openness levels are distinguished not only by the amount of on-screen material they provide, but also the amount of work, i.e. key presses and mouse movements, the user must do to make a window more or less available (Myers, 1988). Since all the windows are of some interest, this work should be as small as possible, and limited by the work involved in creating a new window from stored material.

Moving from window to window implies different actions depending on their relative degrees of openness. Moving between two windows at the same level of openness should require no work at all. All other moves imply changes in the levels of openness of the windows involved, and in the extreme case, replacing a window that is occupying more screen space than its level requires.

Window replacement policies
Transitions between levels of open windows should require as little work as possible. In particular, the studier should not have to select a window and also decide its size and position at each such transition. One solution to this problem is to let the support system modify the display automatically, within the guidelines of an overall replacement policy. A replaced window might either move to the next lower level of availability, or close entirely. While automatic replacement is likely to work well in the greatest number of cases, it
is easy to think of situations where any given policy will lead to the wrong decision. Window replacement policies are similar to those typical of other resource activating mechanisms as follows.

**Replace-least-recent**
Under this policy the support system decreases the level of openness of a window that occupies significant screen space and has been on the screen longer or has not been of immediate interest longer than any other. This situation is most likely to arise when the studier is moving through a sequence of windows, each of which makes some reference to the next. Being able to make a recency decision implies some way of knowing when the studier is looking at a particular window. Lacking eye-position sensing or other sophisticated apparatus, this means the studier must press a keyboard or mouse button at each attention shift, to avoid premature replacement.

**Replace-most-recent**
Under this policy, the new window replaces the latest one opened, or the one most recently designated of immediate interest. This situation may occur when the studier has a text segment on the screen that is related to a number of other windows and is trying to determine which one to look at next. Once again, attention-tracking is required in the system to make an automatic replacement decision.

**Replace-random**
While this policy avoids the need to track the studier’s attention patterns, it is expected to make poor replacement decisions a significant amount of the time.

**Rotation**
This policy governs relative window placement rather than replacement. For example, each newly-opened window might occupy the topmost or bottommost position on the screen, enabling the studier to anticipate where new information will appear. Rotation can contribute to the stability of the presentation during rapid window switching in a multiple-window browser. For example, bottom-most rotation coupled with a replace-most-recent strategy would have the effect of leaving the uppermost windows on the screen in place while scanning through a sequence of related windows.

**Freezing windows**
Problems caused by automatic replacement policies making the wrong decision can be avoided by allowing the studier to designate certain windows that should never be candidates for automatic replacement — in other words, to ‘freeze’ them on the screen (see Figure 3). Such windows should change levels of openness only at the user’s request, after which they are no longer frozen. In the extreme case, where all on-screen windows are frozen, the support system should advise of its inability to modify the screen automatically, and request user action.

Tracking which window is currently of interest has several advantages
Figure 3. The topmost HYBROW window is frozen (window bar indicates FROZEN) so that it will not be replaced by subsequent windows.

Beyond supporting automatic replacement decisions. For the system developer, knowing which one of several windows on the screen is under active scrutiny permits searching and positioning to be carried out immediately without the need to ask the studier to select a particular window. As an alternative, full capabilities might be included with each window, so that by choosing an action the user decides which window to use at the same time.

Attention marking
Systems that track the studier's interest in various windows should communicate this information back to the studier. This might be carried out by changing some feature of the selected window, such as its title or border (see Figure 4), or by placing a unique window identifier somewhere else on the screen (Myers, 1988). The notification problem arises when there are several important windows on the screen big enough just to look at one or another without interacting with the system. In such cases, the studier needs to know when the attention track is out of date and may lead to an unwanted replacement decision.

Placemarking
Systems that permit many windows to be open to various degrees and
Figure 4. The currently selected window in HYBROW, Text Window: 2 — CONCG.C, is indicated by a colour change, shown in inverse here

positioned individually should permit marking particular situations to enable the studier to return to them quickly without having to repeat all the interaction needed to set them up in the first place. Such ‘placemarking’ features imply keeping a constantly updated inventory of all open windows, and their associated text segment origins and current window sizes and positions. The display information might be written out to a file either automatically (say every 10 minutes), or at the studier’s request for later recall. As a refinement of the placemarking capability, a multiwindow system might keep track of an extended interaction history, i.e. all the mouse movements, typed commands and menu selections that produced a particular display, in order to repeat the sequence later for demonstration purposes.

Navigation aids
A multiwindow system must support navigation. It must provide a way to help the user retain some sense of the relationships between windows on the screen and how they came to be placed where they are (Koved, 1985). Without such means, all the windows will soon look alike, requiring some tedious backtracking. Navigation aids might include an index of the window and source code segments in the system, with special markings for segments presently in open

```c
inword = FALSE; /* check it against stop list */
if(goodword(words[nwords]-)word, stopwords))
    nwords++;

for(i=0; i<nstop; i++) /* check word against stop list */
    if(!strcwp(word,stopwords[i]))
        return FALSE;

typedef struct { /* the word structure, contains a word */
    char word[20]; /* and the line number it came from */
    int lineno;
}
```

320 Interacting with Computers vol 1 no 3 (1989)
windows on the screen, and segments which have been looked at during the current session. An index can also help make the decision of what to look at next. The placemarking capability discussed above might be classed as a navigational technique, since it provides a stationary reference point, a way to leave a particular situation and return to it at a later time.

Multiple-window searching
Perhaps no other capability is more important in a multiwindow program browser than searching. Without a preexisting map, determining which code segments are related to the current one either through a calling or a data sharing relationship requires rapid search of multiple code segments. The current window might also require searching to identify additional uses of key identifiers, or other places where the same function is called. Having found a particular identifier, it is often useful during study to check its declaration, which may be in another file. Some program organisations include a single code segment where all important data elements are defined. It may be useful under such organisations to include an ‘always search’ capability, where particular windows are automatically searched and positioned whenever any search request is made.

The great variety in program organisations implies a high degree of flexibility in search schemes. The studier ought to be able to designate any set of segments or windows for simultaneous searching, or limit it to the current window, as desired. Search schemes should provide the option of automatically increasing the openness level of a text window for any destinations that are found, and are not part of a code segment presently on the screen.

Widening the search scope to include source code files that are not yet part of any window presents both important advantages and added difficulties. Often it may not be possible to determine which of a large number of available files holds a text segment that is related to one under study. Searching unopen files implies either the additional work of identifying the files to be searched, or long waits as the system processes all available files recorded on the medium in use. The latter technique is likely to be needed only in extreme circumstances where the text in available code segments gives no clue to what material may be related.

Search directions in a program browser need include only the common ones — forward, backward, and proceed in one direction or the other from the current position (to continue a search, for example). Since an ‘always search’ segment is likely to be short, its direction might default to forward. Hypertext systems sometimes require an additional radial search capability, where the desired target is the closest one in the node-link diagram. Another useful facility is to limit search to nodes within a fixed distance of the current node.

Hypertext issues in program browsing
While hypertext systems provide a useful model for many features in a program browser, there are a number of important design differences implied by the nature of the text and requirements for a multiple-window capability.

Seabrook and Shneiderman
Keywords
Most hypertext systems in use today are designed for use with text and graphics divided into many short segments (see, for example, Cognetics Corp., 1987; Shneiderman, 1987; Kreitzberg and Shneiderman, 1988). The links between the segments are determined during ‘authoring’, when the text is loaded into the system. Most links are based on selecting certain keywords in the text of a segment, marking them for highlighting when the segment is displayed during browsing, and providing the name of another segment to be displayed whenever a highlighted keyword is selected. Choosing keywords during authoring implies making a distinction between important and unimportant words in a segment. As long as the author can guess what words the browser will consider to be important, the system will provide everything the browser needs.

Full-text search capabilities are vital in a program browser. In program text, all words are important — no string of characters in any segment may be considered potentially uninteresting to the browser. However, several common study techniques imply emphasising words that link widely separated code segments, such as function calls and declarations, variable assignments and declarations, and so on.

Window size
As long as the word order is not changed, most forms of English language text convey the same meaning regardless of the physical form. A hypertext system may provide the text in windows of any size and the browser’s understanding of the material should not vary greatly. In program text, the physical arrangement of the text lines is often important (Wiedenbeck, 1986). Even where a programming language does not require any particular format, programmers tend to use horizontal and vertical spacing to indicate groups of statements that have some relationship to each other. Such statements might refer to the same data elements or be part of the same plan (Letovsky and Soloway, 1986). Thus program text implies windows at least as wide as a typical program line and at least as tall as a typical statement group. The maximum size appropriate for a program text window is not as clear. A study of various sizes of program text windows (Reisel and Shneiderman, 1987) revealed that the largest window yielded the best comprehension. This result implies that a browsing system should provide a way to expand any single window temporarily to full-screen size during prolonged study.

Reuse
The meaning of a particular segment of English language text is relatively constant. A person who has read a typical prose passage is unlikely to read it again soon, if ever. However, program text often is characterised by a high degree of generality. What a program will do in a particular setting depends on what values it gets to work with, and what use is made of its products. For this reason, a studier may return to the same code segment many times to speculate on its actions in different settings, especially whenever a module is known to contain ‘mysteries’, sequences of statements that are not well-understood as yet.
(Ramamoorthy, 1986). The longevity of such ‘mysteries’ is also aided by the tendency of programmers to make plausible but incorrect assumptions (Letovsky and Soloway, 1986) during program study. In most cases such assumptions will remain active only until specific information proves them false, although occasions where an individual retained conflicting assumptions are known.

Returning to a previously studied segment is made even more likely in the light of the software industry’s growing concern with reuse (for example, see Frakes and Nejmeh, 1987; Fischer, 1987) — the retrieval and incorporation of existing program modules into new systems. In many systems, the more often a studier has looked at a particular code segment, the more likely it may be examined again. Navigation and placemarking capabilities that permit easy return to earlier screens are more important in program browsers than in text browsers. A note-taking facility in which the studier may record remarks and speculations is also implied, in order to take advantage of them at a later time.

Multiple windows
The need for several open windows on the screen is greater in program browsers than in text browsers. The meaning of any particular statement often depends on several other statements that may be widely separated in the code, even if they are in the same segment. For example, understanding the effect of a statement that assigns variable $A$ the value of variable $B$ may imply checking the places where both variables are declared, how they are passed among functions, and all statements that assign a value to $B$. Understanding statements more complicated than simple assignments is likely to require studying several code segments at once in close proximity. This also implies a greater need for tiled windows where the statements of interest are less likely to be obscured than in overlapped windows (Lifshitz and Shneiderman, 1987). With multiple windowing comes a need for a broad range of screen management and display features, many of which must be placed under the control of the system user.

Summary
The interface in a hypertext multiwindow program browser should therefore include the following characteristics.

- **Multiple windows**
  - tiled presentation for comparisons
  - selectable expansion for detailed study
  - selectable minimisation
  - quick-reference capability
- **Ease of use**
  - common hardware environment
  - unmodified source code
  - system-prepared access aids
  - getting in and out
  - navigating
  - recording notes

Seabrook and Shneiderman 323
• Source code sensitivity
  ○ permit studying programs of any language, size and organisational complexity
• Orientation support
  ○ support marking a place in order to come back later to the same place
  ○ manage large numbers of related windows
  ○ permit easy movement from window to window
  ○ support the studier's mental model
• Search and retrieval
  ○ maps — by module, feature, window
  ○ full-text
• Sophisticated notes support
  ○ multiple reference levels
  ○ automatic movement
  ○ search and retrieval

The HYBROW system

HYBROW is a test environment for studying how hypertext interface techniques might be applied to assist studiers of program source code files. The program was developed in the C language under the Microsoft Windows operating system, in an environment that includes PC/XT-level technology, at least a 20 Mbyte hard disk, a monochrome 13-inch screen and a two-button mouse. The MS-Windows operating system was selected to take advantage of a window-oriented operating environment with considerable potential use. The mouse-driven support features of HYBROW running under the Windows environment permit the user to select all system features and take advantage of most of them without requiring typing on the keyboard. Solutions that work in this small screen environment are likely to improve when translated to the displays that are growing more common.

Interface
The interface in the HYBROW system contains a hierarchical menu structure for selecting all system functions, an unlimited set of program text windows that may be open at the same time, a hierarchical browser for navigation support, a multiple-window search capability, and a comprehensive notes base that includes its own set of support features. The HYBROW menu appears at the top of the screen, from which individual feature menus may be pulled down using the mouse. Selecting a feature either starts its particular process immediately, as in the case of presenting the browser, or begins a dialog during which the system user supplies additional information necessary to carry out the service. An example of the latter is the search capability where the user must provide the search string and the scope — the set of text windows over which the search is to be carried out.

Screen organisation
The 25-line by 80-character screen is divided into 3 areas: the menu area in the
top 3 lines, the code window area in the next 20 lines, and the staging area in the
bottom 2 lines. All open code windows are positioned in the code window area
using a modified tiled format where each window occupies an equal portion of
the vertical space and extends for 80 characters from side to side. The maximum
number of panes, or windows that may appear simultaneously is a system
parameter called Maxpanes that is set before compilation. All open windows are
completely visible up to this maximum number, after which newly opened
windows either overlay or replace previous ones, depending on the selected
replacement policy. HYBROW users may move any window vertically to
uncover a window underneath it, but may not change the size of a window. As
each window is selected by clicking in it with the mouse, that window appears
to be in the foreground with respect to all other open windows. If only one
window is open, it automatically occupies the entire code window space. Thus,
the minimum vertical size a window can have is about 5 lines, one of which is
taken up by the top border that contains the system menu block. This size is
considered a minimum for studying a sequence of program statements. The
small size of this code space encouraged adding a ‘maximize’ feature: pressing
the shift key expands the currently selected code window to full-screen size for
scrolling through and studying larger code segments (about 20 lines). A second
press returns it to its original size and position. Mouse support for this feature
is also available via the ‘View’ menu, where if the system user clicks on
‘Expand’, the most recently selected code window is expanded to full-screen
size. Since the menu is covered up by an expanded window, double-clicking
the mouse in the staging area restores an expanded window to its former size.
HYBROW distinguishes between three levels of open windows:

- **maximised**: one window may occupy the full screen
- **open**: several windows may share the screen
- **minimised**: windows that are represented in the staging area at the bottom of
  the screen by a small icon.

The staging area is for parking windows that are not of immediate interest, but
nevertheless must be available for quick access. Any window may be mini-
nised at any time by selecting it, pulling down the ‘View’ menu and selecting
‘Minimise’. The immediate result of minimising a window is that it is removed
from the screen, its window number appears in the staging area (staging area
spaces are numbered from left to right), and the remaining windows on the
screen resize themselves to share the screen space. Of course, if more than
Maxpanes windows were open the readjustment will not result in any size
changes, merely a reordering of the windows on the screen. Any window in the
staging area may be restored to its normal size by first clicking on its number in
the staging area, at which time the file name of the text in the window appears
just above the number, and then clicking on the file name. This two-step restore
process was designed to permit reviewing which windows are in the staging
area without having to restore them.

**Usability evaluation**

HYBROW provides an environment for testing the features of a hypertext

Seabrook and Shneiderman

325
program browser. Two usability studies were conducted to demonstrate HYBROW's support for source code browsing: an evaluation of two window replacement strategies, and a comparison with a program text editor. The replacement policies implemented for this evaluation included replace-most-recent (MR) and replace-least-recent (LR). Under the MR policy, a newly opened or restored window replaces whichever window has most recently been selected by clicking the mouse on it, while under an LR policy a similar move replaces the window least recently selected. In both cases, replacement results in the new window appearing on the screen wherever the replaced window was, and the replaced window's icon appearing at the bottom of the screen in the staging area.

Trials
The study included a self-evaluation involving the first author, and several thinking aloud interviews with professional programmers. In each case, the subjects were presented typical study problems involving several source files. For example, one subject was required to select the next few steps in the browsing process and at each step provide comments about general expectations and judgments as to the usefulness of the resulting configuration. Any discrepancies between the desired and the actual configuration were noted. Two trials were carried out, the first with an MR policy selected and the second with an LR policy.

Effective use of an MR or LR replacement policy presumes that the system has some information about the user's selection history. HYBROW retains this information by storing the current window number in a 50-element circular queue whenever a window becomes current by clicking it with the mouse. During an extended session, the selection queue will contain a sequence of window numbers. Since each window automatically becomes the current window when it is opened, the sequence will be at least as long as the number of windows that have been opened. The queue will contain additional entries for each time the user has clicked the mouse in an open window that was not current at the time. Some queue entries will designate windows that are no longer on the screen.

The subject performed a call tree study of the code, starting with the header file and the main() function, and proceeding to functions which main() calls. The call tree in the concordance program used in the test is a flat structure where the main() function calls most other functions.

Ideally, changes in the screen configuration will leave both members of a caller-callee relationship on the screen in order to compare argument sequences and declarations, etc. When the call tree is flat, the MR strategy produces this effect only if the browser selects an uninteresting window after the caller function and before the first callee function, otherwise the callee's window will replace the caller's window. In general, the MR strategy produces satisfactory window replacement results only if the previously selected window is not of immediate importance. This is likely to be the case in a program that exhibits a flat call tree, as long as the calling function stays on the screen for comparison with each called function.
The LR strategy in a study involving a flat call tree will replace the calling program only if there are just two windows on the screen, otherwise it will replace a previously selected one. In general, if the screen contains \( n \) windows, the calling function's window will be replaced after \( n \) selections, as long as only minimised windows are selected. At this point the calling function and the last \( n-1 \) called functions will be on the screen. In general, the LR strategy produces satisfactory replacement results only if the last \( n-1 \) selected windows continue to be of some interest. Such a policy is more likely to produce satisfactory results when the call tree of the program under study is not flat, since in this case selecting functions down the call tree will leave the last \( n-1 \) caller–callee related functions on the screen.

In a second evaluation using a professional programmer, the subject stated a preference for following the call tree and examining the code in called functions prior to viewing header files and other noncontrol source code. The subject suggested that looking at a header file would be a temporary reference need and that ideally, such a file should not replace any screen windows, but instead appear in a temporary window that overlays or overlaps others on the screen and then disappears on request. The subject claimed little need for an MR policy except as an option to provide greater flexibility in managing what happens on the screen.

The subject stated a clear preference for the LR policy in browsing source code, coupled with a rotation policy that places each new window at the bottom of the screen. The current LR policy, in which a new window brought to the screen occupies the same position as the least-recently-selected window it replaced was judged helpful to the extent that it allowed viewing several functions in the call tree on the screen at the same time. However, the subject claimed a need for the vertical positioning of the functions to reflect the sequence in which they call each other, the topmost calls the function in the next window, which calls the function in the next, and so on.

The subject acknowledged a need for the call-tree diagram and cross-reference listing included in the HYBROW browser dialogue window, but stated that the studier should be able to modify them to reflect increased understanding of the data and control relationships among the functions in the program. On each occasion of use, the system should aid the studier to recall to mind the previously derived conceptual model without having to rediscover it from basic diagrammatic tools. One way to implement such a capability would be to permit copying all or part of the browser displays into a notes window after which they might be edited and augmented as ordinary text.

The subject acknowledged a need for a placemarking capability in the work environment due to constant interruptions during an extended study session, but stated a preference for placemarks to retain conceptual-model information like that described above, as well as information about which files were open, their relative scroll position and what place they occupied on the screen.

**Tests**

A study was conducted to compare HYBROW's browsing and searching features with those of a commercially available program text editor, the M editor.
distributed with Microsoft C. HYBROW was designed with the mouse as the principal interactive tool while the M editor offers no mouse support. For this reason, several test results were adjusted to remove any bias towards HYBROW. A common set of browsing tasks was devised and carried out in each environment, keeping a careful count of the keystrokes required. The tasks selected were as follows:

- open three source files
- switch attention among three open files
- locate a specific function
- locate a system function call

**Task 1 — Open three source files**
To open a file in HYBROW, the user must first select the File menu and then the Index dialogue, which provides a directory listing. As long as the files have a `.c` extension, they can be opened and automatically positioned on the screen with four keystrokes each. Opening three files required a total of twelve keystrokes. Opening three files under the editor required typing their names, since no directory was provided for selecting available files. The total number of keystrokes required was 29, although eliminating the filenames and assessing one keystroke for each file to put the comparison on a more equitable footing brought this total to twelve keystrokes again.

**Task 2 — Switch attention among three open files**
Both packages maintained current buffer positions in all open files. Since HYBROW has a multiple-window presentation, selecting the current window amounts to moving the mouse and clicking its button once. Thus, only one keystroke was required to switch among open windows and report this change of attention to the system. The total for three files was three keystrokes. The editor required switching to its index of open text buffers, a cursor movement to the correct file name, and then a keystroke combination to select the file name and present the buffer on the screen. Replacing multiple cursor-key presses with a single keystroke resulted in six keystrokes for each change. The total for three files was eighteen keystrokes.

**Task 3 — Locate a specific function**
This task involved positioning a text window to a particular function's source code, without prior knowledge of which file it was in. For the HYBROW test, the function name browser was used to select a function name, and then a multiple-window search was initiated, which positioned all open windows simultaneously (see Figures 5 and 6). The total for three files was eight keystrokes. The editor test involved typing the name of the function, after which three window switches and searches were required. The result was 26 keystrokes.

**Task 4 — Locate a system function call**
This test was run twice to take advantage of special features in HYBROW. Since
system function names are not included in HYBROW's module browser, a window search required typing the function name and selecting 'All active' windows as the search scope. The total was eleven keystrokes. HYBROW's feature browser provides a list of all program words followed by a left parenthesis. Switching to the feature browser in order to select the function name rather than type it resulted in a total of six keystrokes for this test. Locating a system function call with the editor required typing the function name and then searching each text buffer individually. The total was 23 keystrokes. A summary of the keystroke comparisons is shown in Table 1.

Analysis
While this usability evaluation is not conclusive, it does provide subjective and objective evidence of the benefit of HYBROW strategies. The informal nature of the tests and smallness of the sample size limit interpretation of the test results to a generally favourable impression with a strong recommendation for additional study. Such a study should include repeated presentation in both academic and professional environments, using subjects with many different levels of experience, and several programs that differ in their relative degree of modularity. Also, test programs in several languages should be employed to understand and correct for the language bias included in HYBROW features.
One further positive anecdote about HYBROW is that as portions were completed the system was immediately put to work to support further development.

Conclusions
The need to study existing programs has grown from a once-in-a-while bothersome task to a frequent endeavour, crucial to successful system development and maintenance. Many software tools and environments have been

Table 1. Keystroke comparisons on several browsing tasks

<table>
<thead>
<tr>
<th>Task</th>
<th>HYBROW keystrokes</th>
<th>editor keystrokes</th>
</tr>
</thead>
<tbody>
<tr>
<td>open three source files</td>
<td>12</td>
<td>12*</td>
</tr>
<tr>
<td>switch attention</td>
<td>3</td>
<td>18*</td>
</tr>
<tr>
<td>locate a specific function</td>
<td>8</td>
<td>26</td>
</tr>
<tr>
<td>locate a system function call</td>
<td>11</td>
<td>23</td>
</tr>
<tr>
<td>locate using browser</td>
<td>6</td>
<td>no browser</td>
</tr>
</tbody>
</table>

*adjusted downward to remove bias
developed to assist programmers in the initial design and development of systems. Few, however, provide support for software study after implementation. As a result, many practitioners use program text editors for studying programs, in order to take advantage of powerful string-search capabilities. While many of these retain current positions in internal lists of open buffers, often screen management and support are rudimentary.

Hypertext systems have arisen in the past 10 years as a way to provide rapid access to interrelated text segments by means of internal links, indexes, and such high-level abstraction tools as hierarchical browsers. Some hypertext systems provide multiple-window support, although most window size and positioning decisions and the effort required to carry them out are left to the user. The potential benefits to be derived from understanding program browsing and from building a superior support tool are enormous. By reducing the cognitive load in traversing a complex program, the user’s mental resources can be devoted to understanding the program, finding bugs and planning improvements. In addition to maintenance considerations, the software industry’s interest in reusing existing modules implies a strong need to study previously developed programs.

The need for programmer training and computer science education has not abated, despite periodic predictions of the advent of ‘programmerless’ systems, languages and work environments. The wide use of program examples as teaching aids implies that a carefully designed support system will be invaluable for helping the student understand the nature of control, communication and data relationships in a set of modules. As the complexity of programs increases through the adoption of such advanced techniques as object-oriented programming and parallel processing, aids to understanding software become critical.

This paper clarifies how study aids may be included in a support system for program browsing. The varied techniques employed in software design and the rapid pace of new developments indicate an urgent need for continued study of the browser interface, both as means of increasing the productivity of existing programmers and the learning rate of new ones.

References

Cognetics Corp. (1987) Hyperties user’s manual Princeton, NJ, USA

Seabrook and Shneiderman
Digitalk Corp. (1986) Smalltalk/V: tutorial and programming handbook Digitalk. Los Angeles, USA
Frakes, W.B. and Nejmeh, B.A. (1987) ‘Software reuse through information retrieval’ Proc. 20th Annual Hawaii Conf. on System Sciences 530

332 Interacting with Computers vol 1 no 3 (1989)


Ryder, B.G. (1987) ‘An application of static program analysis to software maintenance’ Proc. 20th Annual Hawaii Conf. on System Sciences 82

Shneiderman, B. (1980) Software psychology: human factors in computer and information systems Winthrop, Cambridge, Massachusetts, USA


Shneiderman, B. (1987) Designing the user interface: strategies for effective human–computer interaction Addison-Wesley, Reading, Massachusetts, USA


Seabrook and Shneiderman
Appendix A: HYBROW documentation

The HYBROW menu provides the following alternatives: 'File', 'View', 'Placemark', 'Debug', 'Search', 'Freeze' and 'Notes'. These menus provide general file and window operations, window sizing, saving and returning to a particular set of windows and positions within the windows, a debug facility, searching and automatically positioning open windows, freezing windows on the screen so that they will not automatically be removed, and notes services, respectively.

File menu
The File menu provides the alternatives 'Index . . .', 'Browser', 'Policy . . .', 'Close All', and 'Quit'. The services provided by these alternatives are described in the following sections.

Index
Clicking on 'File' and then on 'Index' commences the index dialogue. A dialogue box appears on the screen containing a list of the files in the current directory with the default specification, '*.c'; that is, all source code files in the C language. Typing a new specification, possibly with a different directory named, produces a different list. The user may scroll the list up and down to see the rest of the names if there are more than will fit in the box. Any file on the list may be opened in one of three ways: by double-clicking on its name; by clicking on its name and then on the 'Open' button; or by clicking on its name and then on the 'MinOpen' button. Clicking on the 'Cancel' button terminates the dialogue. As soon as a window has been designated for opening by either of the first two ways, the dialogue terminates, the screen is readjusted to include the new window, and the user may proceed to browse. Selecting a file and then clicking on 'MinOpen', prepares the window for opening but places its number in the staging area, where it may be restored at any time by the process outlined above. In this case the index dialogue does not terminate, but continues to present the file list in case the user wants to open a number of files for later use. After staging several windows in this way, the user may click on 'Cancel' to terminate the dialogue.

Browser
Clicking on the 'Browser' alternative of the 'File' main menu selection results in replacing the entire screen presentation with a browser window, and initiating a dialogue. The browser dialogue includes a list box in which the user may see a mapping of a segment of the source code in the system. Three mappings are available at present: a function listing, a word listing, and a hierarchical browser. The function listing includes the names of all functions in the system, grouped according to the source file they are in. The word listing includes all words in the system in alphabetical order together with the source file and line number on which they appear. The hierarchical listing includes the function names in the source code organised according to the way they call each other. Clicking on any element of the three mappings and then on the 'Search' button
results in terminating the dialogue restoring the former screen presentation, and positioning the window that includes the selected item. At present, the mappings are constructed outside the system, although they might easily be created and maintained whenever a new window is opened.

Policy
Clicking on the 'Policy' alternative of the 'File' main menu selection initiates a brief dialogue where the user may establish the window replacement policy to be used whenever new windows are opened or staged windows are restored to their normal size. Current policies are 'Add/Only', 'Replace oldest' and 'Replace most recent'. The first policy results in adding a new window or restoring a minimised one to the screen, overlaying an existing window if the screen already contains Maxpanes windows. Under the second policy, HYBROW searches a list of previously selected window numbers backwards until the earliest entry that is still on the screen is found. It minimises this window and replaces it with the newly opened or restored window. The 'Replace most recent' policy causes the newly opened or restored window to replace whichever one was selected when the 'Browser' alternative was clicked.

Close All
The 'Close All' alternative of the 'File' main menu selection checks to see if the 'Place' or window configuration has changed since the last time it was saved (see the 'Placemark' alternative below), asks about saving the configuration to a placemark file and then proceeds to close all open or minimised windows, returning the user to the initial presentation before any windows were opened. The text of all the windows remains in memory to avoid reading it in again in case new windows are opened on the same source file.

Quit
The 'Quit' alternative of the 'File' main menu selection carries out the same actions as the 'Close All' alternative discussed above, and then terminates HYBROW, returning control to the Windows operating system.

View menu
The 'View' menu provides the alternatives 'Normal', 'Expand', and 'Minimise'. The 'Expand' alternative expands the most recently selected window to full-screen size, while the 'Minimise' alternative removes it from the screen and places its number in the staging area, as discussed above. The 'Normal' alternative restores either an expanded or minimised window to its normal size and screen position.

Placemark
The 'Placemark' menu provides two alternatives: 'Mark a current place . . . ' and 'Open a place . . . '. These services allow the user to save any window configuration in order to start up the system later with the same windows open, positioned and staged.
Mark a place . . .
The ‘Mark a place . . .’ alternative of the ‘Placemark’ menu initiates a dialogue wherein the user is asked to provide the name of a file in which to record all relevant information about the current HYBROW configuration: which windows are open, their scrolled positions relative to the beginning of each file, and whether they were minimised or not. The user may type a full file name specification, or just a short name in which case the default extension ‘MRK’ is used. If the file name already exists, the user must either give permission to write over it or must provide another file name. After marking the place the dialogue terminates and the user may proceed as before. Successive configurations in a session may be saved in this way for later resumption from any point.

Open a place . . .
The ‘Open a place . . .’ alternative of the ‘Placemark’ main menu selection initiates a dialogue wherein the list of all files in the current directory with the ‘MRK’ extension is presented. The user may type a different file specification to see a different list. Double clicking on any placemark file replaces the current configuration with the configuration that was in existence at the time that placemark file was created. If the current configuration has changed since it was opened or restored from another placemark file, the ‘Mark a place . . .’ dialogue is initiated automatically, during which HYBROW asks users if they want to save the current configuration before replacing it with the new one.

Debug alternative
The ‘Debug’ alternative of the main menu replaces the current screen display with a new window full of selected variable names and their contents. The complicated nature of windows programming encourages frequent use of the debug window during development. No changes to the debug window are permitted at present. As soon as ‘Debug’ is clicked again, the previous display is restored to the screen.

Search alternative
The ‘Search . . .’ alternative of the main menu initiates a dialogue during which the user specifies a character string to search for and the scope of the search to be carried out. The user may either type a search string or proceed to search for a string selected from a recent program text window. The same string is provided by the notes-searching capability to enable easy searching of the notes as well. Specifying the scope of the search involves selecting any combination of three alternatives: ‘Current’ which specifies searching the currently selected window; ‘Open’ which specifies searching all windows on the screen except the current one, and ‘Active’ which specifies searching the staged windows. In each case searching begins at the current text position in the window and proceeds forward. Finding the search string results in repositioning the text in the window so that the first occurrence of the search string is at the top of the screen. Searching staged windows by selecting ‘Active’ does not restore such windows to the screen. Instead HYBROW indicates that the search string has been found in a staged window by highlighting its window number in the
staging area. The window must be restored in order to see the text beginning at
the located string. The scope of a search may be changed at any time — even
when continuing the current search.

Freeze alternative
Clicking on the ‘Freeze’ alternative results in freezing a window. When a
window is frozen, it remains on the screen near the top as new windows are
opened or restored no matter what replacement policy has been selected.
HYBROW indicates that a window is frozen by placing the word ‘FROZEN’ in
its caption border. Selecting a frozen window and then clicking on the ‘Freeze’
alternative results in unfreezing, or ‘thawing’ the window.

Notes . . . menu
The ‘Notes . . .’ menu provides the alternatives ‘Notes’, ‘Open’, ‘Search’, and
‘Close’. The services provided by these alternatives enable access to a text
window where the user may type, edit or search for current program notes. The
alternatives on this menu are described in the following sections.

Notes alternative
The ‘Notes’ alternative of the ‘Notes . . .’ menu displays the current notes
window if it has been opened. Selecting this alternative hides the notes
window if it is visible at present. Whenever a notes file is open, a ‘NOTES’
button is displayed at the bottom of the screen which performs the same action
as the ‘Notes’ alternative of the ‘Notes . . .’ menu.

Open alternative
The ‘Open’ alternative of the ‘Notes . . .’ menu initiates a dialogue in which the
user may name a text file of program notes to be opened. If a file name is
provided and the file does not exist, a blank notes window is opened. When the
file exists, it is opened and the text made available in a minimised notes
window. To see the window with the notes text the user must click the ‘NOTES’
button at the bottom of the screen, or select the ‘Notes’ alternative from the
‘Notes . . .’ menu. Whenever a notes file is open, all the windows shorten their
width by one character to provide the impression that the notes window
underlies the right end of the source code windows. Full editing capabilities are
provided in the notes window, i.e. search, select, cut, paste, and copy.

Close alternative
The ‘Close’ alternative of the ‘Notes . . .’ menu updates the notes file on the disk
with any changes made during the session. A new notes file will be created if it
does not already exist.