CHISIG

ERGONOMICS SOCIETY OF AUSTRALIA

HCI Australia '89

Proceedings of the Annual CHISIG Conference

Monash University / Telecom Research Laboratories
Melbourne

23-24 November 1989

Sponsored by:
Telecom Australia
Research Laboratories

Monash University,
Psychology Department

Inpro Services Pty Ltd

A CASE STUDY IN USER INTERFACE DESIGN

Jim Welsh
Key Centre for Software Technology
University of Queensland

Mark Toleman
Darling Downs Institute of Advanced Education

ABSTRACT

User interface choices are vital to the success of software use by skilled users, such as software engineers themselves. This paper presents a case-study of some significant user-interface choices made in the design of language-based editors for software development at the University of Queensland, and discusses the difficulties perceived in their validation.

INTRODUCTION

In design and production areas where trained human intellect is the primary resource, computer aids play an ever increasing role. Software development itself is a typical example, as is computer-aided design in a variety of engineering disciplines. In these areas the objective is to maximise the product quality and productivity achieved, by enabling the users to perform their creative intellectual activity under optimal conditions, by preventing or detecting human errors as they occur, and by relieving the users of routine mental and physical activity associated with the productive process. In designing user interfaces for such systems the emphasis is on optimising the physical and intellectual ergonomics involved, rather than on ensuring comprehensibility for users unfamiliar with the system.

For the developers producing such tools, the problem is that rapid advances in interface hardware technology imply correspondingly rapid change in the factors that determine user interface success. Ideally, the interface choices concerned should be determined, or at least validated, by application of a well-defined set of interface evaluation criteria or procedures. In practice, such criteria or procedures are not used by many tool developers, either through ignorance, or through the lack of a workable set of procedures for doing so.

In this paper we review the problems encountered, and strategies adopted, in a project to develop an effective software preparation tool for use on high-powered personal workstations. In practice the tool is intended to support the preparation of a range of software-related documents, specifications, designs and programs, but for the purpose of discussing the user interface issues involved we will describe it as a *program editor* which assists in the preparation, inspection and modification of programs written in languages such as Pascal, Modula-2 or Ada. At the time of writing, a first-generation generic editor UQ1 exists [Welsh, Rose and Lloyd (1986), Hayes, Neucom and Welsh (1989)], while development of a second-generation editor UQ2 is well advanced [Broom, Welsh and Wildman (1990)].

The features and associated interface issues discussed in this brief paper are a judicious abstraction of the actual features of either system.

The issues are discussed under two headings: physical interface representation, and conceptual interface issues. The order of discussion is arguably the opposite to that in which the issues should be addressed in practice, but reflects a progression of increasing difficulty for the designer concerned.

PHYSICAL INTERFACE REPRESENTATION

At the lower level the software designer determines the physical representation of the user interface — how input commands and output responses are represented. The designer must choose between many input devices such as keyboard, mouse, etc. for user (-initiated) inputs, between output devices and on-screen styles for system outputs, and between paradigms such as menu display, form-filling, etc. for system-elicited user responses.

In general the relative merits of the options available at the physical level are amenable to empirical evaluation, and a wide literature is available to designers who care to look — Eberts (1987) has reviewed over 100 articles in the area. The help available ranges from simple formulae like Fitt's Law for sizing and placement of on-screen controls, to more comprehensive models such as the GOMS model of Card et al (1983) which integrate issues related to the physical level.

Many handbooks and guidelines focus on this physical level. Smith and Mosier (1986) have published a set of about 1000 guidelines covering the areas of data entry, data display, sequence control, user guidance, data transmission and data protection. Other examples cover specific application areas or user communities.

From the software designer's viewpoint, work-stations with bit-mapped displays and mice enable a bewildering plethora of physical interface options. Recently, however, the user need for uniformity across a range of tools and systems has led to the formulation of "look and feel" guidelines, such as the OPEN LOOK proposal from AT&T and Sun (1988) . The extent to which such guidelines become "standards" remains to be seen, but provided they are consistent with results arising from human factors research, they provide an effective channel for communicating these results to the software developer.

What impact have these aids had on our development of the UQ editors? UQ1 is a prototype system primarily intended to demonstrate the conceptual choices involved (as discussed in the next section). As such, it has an idiosyncratic physical interface which might be seen as typical of a purely intuitive approach. In retrospect, this intuitive interface has a number of major deficiencies for professional use:

- user commands are activated via a control panel which is permanently displayed and consumes too much screen "real estate",
- button sizes and placement are not tailored to efficient normal use,
- insufficient keyboard "accelerators" are provided to avoid slow interleaving of mouse and keyboard activity, and
- multi-window display management is inconsistent with the host windowing system's conventions.

In contrast, the UQ2 interface design takes due account of current thinking and guidelines on interfaces for professional use:

- permanently displayed control features are limited to a narrow band of buttons through which menus or command windows are displayed when required,
- frequently used edit commands are available from a dynamically changing "on-mouse" menu within the edit area,
- default keyboard accelerators are defined for all appropriate commands, with additional or alternative accelerators definable via a user-customising option, and
- multi-window display management conforms to the host windowing conventions.

In its detailed representation, the interface conforms to the OPEN LOOK guidelines, and its implementation has been considerably assisted by Sun Microsystems' XView toolkit.

UQ2's functional design is still subject to experimental adjustment. For this reason, its physical interface choices have not yet been validated by application of any empirical or model-based evaluation. We are reasonably confident, however, that such evaluation should lead only to fine-tuning of the current physical interface choices, rather than their wholesale redesign.

In summary, therefore, a reasonable degree of assistance appears to be available to the software designer in determining the physical representation of a user interface, in the form of guidelines, models for comparative evaluation of alternatives, and implementation support packages. While the human factors inherent in these aids may be the subject of ongoing research, their existence provides a communication channel by which new findings become available to the software designer. This is in direct contrast to the situation with respect to conceptual problems, which we now consider.

CONCEPTUAL INTERFACE ISSUES

More critical problems for the software designer arise at the conceptual level of user interface design, where the user's cognitive processes, as well as their perceptual and motor skills, have to be considered. As Eberts and Eberts (1989) indicate, validation of an interface design at this level requires application of cognitive psychology and cognitive science theory, to show that the information processing involved is optimal for the human/computer partnership. Applicable theories that have been advocated include those on problem solving, (Newell and Simon 1972, Card et al 1983), analogical reasoning, (Mayer 1975, Teitelman 1979, Bewley et al 1983), spatial reasoning, (Gomez et al 1983, Molzberger 1983), attentional models and goals, plans and scripts, (Ehrlich and Soloway 1982, Soloway et al 1982, Soloway et al 1983).

From the software designer's point of view, the problem is not that potentially relevant theories have not been developed, but that an effective procedure for the identification and application of an appropriate theory is not yet available. In these situations, the designer falls back on intuition to determine the choices involved.

To illustrate the problem we present three examples of conceptual choices inherent in the design of the UQ editors, and the intuitive reasoning which led to them. First, however, we review the conceptual models of programs that users may employ, as these underly the reasoning involved.

User models of programs

When a text-editor is used for program construction, the conceptual structure of the program is purely the user's concern, and all interaction with the editor is expressed in terms of character manipulation. The final program, of course, must have a well-defined structure, as a tree of well-formed constructs, which the user must appreciate and work towards during the construction process.

When designing a program-specific editor, it is tempting to express all program manipulation in abstract structural terms — in effect to give the user a tree editor, within which character sequences are significant only as lexical symbols at the leaves of the tree. However, limitations of this approach arise from two major sources.

Firstly, the assumption that all users are willing and able to think exclusively in tree terms is clearly false. In practice, many users have a pluralistic view of the programs they manipulate, seeing them sometimes as tree structures, sometimes as lines of text, sometimes as symbol sequences, and sometimes as character sequences, according to their purpose at that time. Consider the hierarchy of constructs that appear in programs written in languages such as Pascal, Modula-2 or Ada:

compilable units, programs, etc.	*tree-like*
nested modules, packages, procedures and functions	*tree-like*
declaration parts, structured statements and structured types	*trees, lines, symbols or text?*
simple statements and declarations	*trees, lines, symbols or text?*
expressions, lists, etc.	*trees, lines, symbols or text?*
identifiers, literals etc.	*text-like*
characters	*text-like*

We have little difficulty in putting a lower bound on the user's perception of tree structure. No user thinks of identifiers or literals as trees of characters, and editors must provide straightforward text editing operations at this level. Immediately above this level, the most common perception is probably of a sequence of lexical symbols, but as in text editors users switch between perceptions of symbol sequences and character sequences with little conscious effort.

At the other end of the hierarchy, we have little difficulty putting an acceptable upper bound on textual or symbol perception. Few users usefully perceive a hierarchy of nested modules and blocks as anything other than a hierarchic tree structure, and having to manipulate it as a flattened textual space is one of the basic disadvantages of text editors.

Between these obvious bounds, it is difficult to justify an exclusively tree-like or an exclusively textual perception of the constructs at any level. Depending on the user's purpose at the time either perception may be the more effective. An editor that recognises only tree structure at such levels is no better, and may well be worse, than one that recognises only character texts.

The second limitation of the tree based approach arises from the nature of the feedback which users receive via the program display. Even if the user is willing to think exclusively in tree terms, the feedback provided does not normally reinforce this view. While workstations with bit-mapped displays are capable of generating graphical representations for program constructs, the display area consumed by such representations is too high to allow even modest program fragments to be displayed effectively. This limitation forces the program display that the user sees to be in textual form, albeit with structural overtones achieved by indentation, highlighting, etc. This textual feedback not only influences the user's

perception, it also acts as the primary frame of reference within which the user must express the program manipulations required. It is commonly recognised that users prefer commands which represent *direct manipulation* of the displayed representation to those whose effect is defined in terms of some abstraction of the display. The textual display enforced by display limitations therefore biases the user towards operations defined in textual terms.

For both these reasons, the need to support a pluralistic view of program structure is a critical factor in the design of program editors, and will remain so until radical changes occur both in user training and in input/output technology.

Example 1: Block-oriented display

From the user model of program structure suggested in the preceding section, we may assume that the user perceives the structure of a Pascal program, say, as a tree of nested blocks. In practice, however, the user is most commonly interested in looking at one block at a time, less frequently in comparing the content of two or more blocks, and occasionally in reviewing the overall hierarchic structure itself.

To meet the user's predominant need (to look at one block at a time) the solution adopted in the UQ editors is that each view presented is a single block and its associated heading, with any nested blocks abbreviated by elision of their bodies. In the UQ1 screen shown as Figure 1 the topmost sub-window shows the display of the outermost block of a trivial Pascal program *Example*. The ellipsis points (...) after each procedure heading indicate that the corresponding procedure bodies have been suppressed by the display system.

To view any of these abbreviated procedures, the user invokes **zoom in**. This replaces the current view by that of the procedure concerned, in which the bodies of its nested procedures or functions are again abbreviated. From this point, the user may again **zoom in**, **zoom out**, or **pan forward** and **pan back** between sibling blocks.

Alternatively, absolute selection of a new context can be made at any time from a menu of all viewable blocks in the program. In principle, our user model suggests that this menu should be displayed in a tree-like form, but in practice its display as an indented list of block names, as shown on the right in Figure 1, is as effective and more easily managed. This menu of viewable blocks also fulfills the user's occasional need to review the overall hierarchic structure.

To meet the user's remaining need, that of comparing the content of two or more blocks, simultaneous display of views in two or more sub-windows is enabled. Figure 1 shows the use of a second sub-window to display the procedure *FixValue*, in parallel with the main program itself.

No formal validation of these block-oriented display choices in the UQ editors has been attempted, but informal experience with UQ1 to date suggests that all users find them preferable to a flat scrollable representation. This is consistent with the unequivocal user model at block level, and in direct contrast with the problem presented as example 3.

Example 2: Detail suppression

Automatic formatting is an essential feature of any program editor. To avoid prodigal consumption of display capacity, the UQ editors employ *adaptive formatting* as defined by Rose and Welsh (1981). This allows trivial occurrences of potentially large constructs

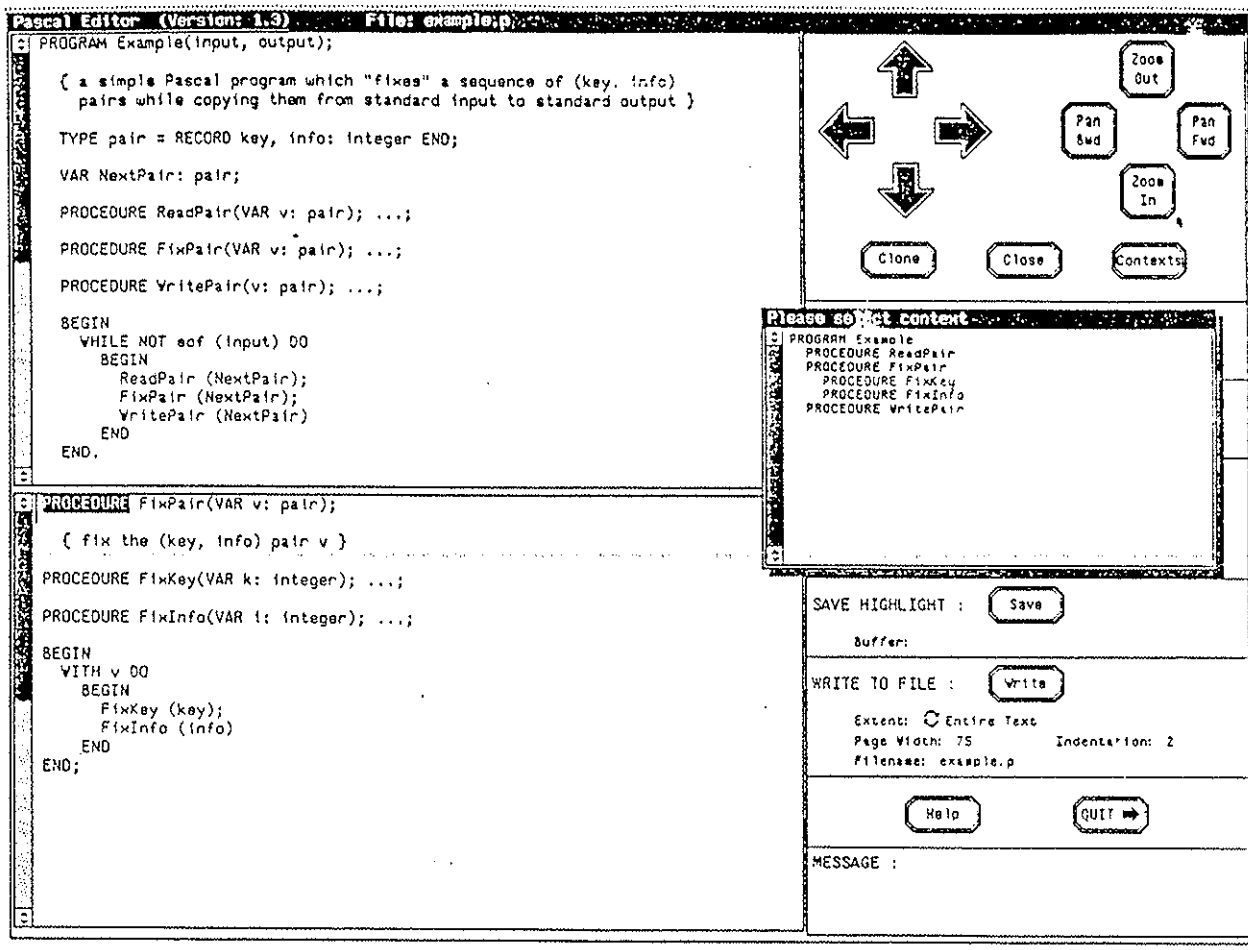

Figure 1: Block-oriented program display by UQ1

to assume one-line formats, while larger occurrences adopt the minimum multi-line form consistent with their structure, extent and readability.

The combination of nested block abbreviation and adaptive formatting minimises the display requirements of a program block while maintaining its readability, but it cannot guarantee that the resultant view fits in a window of given size.

In UQ1, this problem was solved by resorting to the text-editor solution of *scrolling* the viewable text within the display window. While such scrolling is consistent with a purely textual view of the block concerned, it is unsatisfactory when structural issues are relevant, since structurally related symbols which are textually separated by more than the window size cannot be viewed simultaneously.

For this reason, UQ2 offers *suppression by structural distance* [Broom and Welsh (1986)] as a selectable alternative to scrolling when a block view is too large for the available sub-window. With this option selected, text furthest from the user's focus of attention (in some structural sense) is suppressed until the view fits.

Figure 2 shows the effect of distance suppression when the top sub-window in Figure 1 is reduced to half its height, and the user's focus or highlight is on the conditional expression controlling the while-loop in the program body.

By providing scrolling and distance suppression as user-selectable alternatives, variations in user preference are catered for, but a significant interface choice remains to be resolved within the distance suppression option. In a system that combines adaptive formatting and

```
PROGRAM Example(input, output);
  ......
  BEGIN
    WHILE [NOT eof(input)] DO
    BEGIN
      ...
    END
  END.
```

Figure 2: Detail suppression by structural distance

text compression an important issue is whether the formatting decisions take account of text suppression in determining the layouts required. If they do, then a statement whose unsuppressed layout is

```
IF condition THEN
  BEGIN
  statements
  END
ELSE
  BEGIN
  statements
  END;
```

may under partial suppression take the one-line form

```
IF condition THEN ... ELSE ...;
```

While this approach makes optimum use of the available display area, it means that the "shape" of a program fragment alters as the user's focus of attention changes. Such changes of shape create recognition problems for the user, and for this reason we prefer a suppression strategy which suppresses only whole lines of the formatted unsuppressed view.

At this stage, this decision is a strictly intuitive one on the designers' part, which has not been validated either by user experiment or by application of known human factor results. However, its underlying principle, which preserves an object's shape while varying the relative levels of projected detail of its components, must be one which has been investigated in other contexts.

Example 3: Program editing

In language-based editors to date, two basic paradigms for the input and editing of program fragments have been recognised.

One is the *tree-building paradigm*, in which the user is only allowed operations which ensure the structural correctness of the program tree at all times. The user extends an existing program by expanding a previously unexpanded node of the program tree, usually by selecting an appropriate node template from the menu of allowable templates at that point.

Likewise the user may only delete existing constructs that are optional, or replace an existing construct with another of the same syntactic class.

Tree-building by template selection has the obvious advantages that construction of syntactically incorrect programs is precluded, and the user is freed from all need to type syntactic sugar in the program — the only textual input required is the user-determined lexicals such as identifiers and literals which appear as leaves of the tree. However tree-building is an unnatural and very tedious way of constructing low-level constructs such as expressions and identifier lists, so most editors adopting the paradigm allow users some form of text input and editing at these levels.

The conceptual model of program structure underlying this approach is simple, consisting of a strict tree model at those levels where manipulation by template selection is required, with textual words or phrases as the leaves of the tree concerned. From the user's viewpoint, however, the tree model is compromised by the fact that a textual display is normally used at all levels. To formulate the next operation required, the user is therefore forced to abstract from this display to the tree structure involved.

The alternative to tree-building is the *(text) recognition paradigm*, in which the user manipulates the displayed representation in textual terms, and the editor parses or "recognises" the textual changes to deduce the program tree required.

The delete, insert and replace operations permitted by this approach are clearly a superset of those allowed by a tree-editor, giving the user greater flexibility as to how a given change is effected. Depending on the nature of the text inserted or deleted, the user may conceive the change involved as a tree manipulation, a symbol sequence alteration, or a character-based textual alteration, but need not distinguish which it is. The editor is thus tolerant of a pluralistic model of program structure.

Text recognition is more natural for input of low-level constructs, and in general has the advantage of "direct manipulation", i.e., the user perceives and effects the change involved directly in terms of the displayed representation. It does not, however, preclude user error, and in general the editor must tolerate incorrect intermediate program states as changes are made. Its effectiveness in achieving error-free programs (and in reducing the keystroke effort required of the user) depends critically on the nature of the parsing process applied, the degree of its synchronisation with the actual keying of the text involved, and its capacity to propagate the consequences of changes in an efficient manner.

The choice between tree-building and text-recognition paradigms has been an issue in language-based editor design over the past eight years, with much intuitive comment appearing in the literature. To the best of our knowledge, however, no systematic attempt to demonstrate the advantage of one paradigm or the other, either by application of relevant theories or by controlled experimental evaluation, has been attempted. In this case, the problem is complicated by the interaction of several factors - in addition to the user's conceptual model of program structure, factors such as the error discipline to be applied, the model bias produced by textual display, and the adequacy of implementation of the text-recognition approach, have all to be taken into account.

The UQ editors both use the text recognition paradigm, but the recognition parser developed for UQ2 [Kiong and Welsh (1986)] has significant advantages in its efficiency and capacity for user feedback compared to that used in UQ1. The initial choice was based on an intuitive preference for the direct manipulation style, and the evolutionary advantages for professional users who are already accustomed to the text-editing paradigm. Experience with UQ1 has shown that implementation performance is critical to the success of the recognition approach, but in UQ2 we believe we have developed algorithms which meet

these critical requirements. The only reliable test of this belief currently envisaged is the ultimate reaction of users.

CONCLUSION

Interface issues are critical to the provision of computer-aided tools for professional users. At the physical representation level, the maturing knowledge of the human factors relating to current interface technology, and its flow-through to software developers via guidelines and support software, can ensure a reasonable standard of interface design.

At the conceptual level, however, continuing problems are more likely. Of the three program editor examples cited, the first posed few problems, mainly because the user model of program block structure is relatively clear. The second illustrates a detailed issue where a more general display principle seems to pertain, but was adopted by intuitive reasoning on the designers' part. The third is by far the most complex, and involves reconciliation of several interacting factors. That the problem remains a matter of subjective debate in the language-based editor community, without any documented attempts to resolve it, may be an indication either of its complexity or of a failure of some kind.

The Apple ideal, of every design team consisting of a software engineer, a cognitive psychologist and a graphic designer, is still a long way off for most software development organisations, simply through lack of the trained personnel required. To remedy this situation, by changes in education and training, takes time. In the meantime, the software and CHI communities must strive for a greater awareness of each others needs and abilities, to achieve the most effective use of the limited resources available.

References

AT&T, and Sun Microsystems Inc., 1988, OPEN LOOK Graphical Interface Specification .

Bewley, W.L., Roberts, T.L., Schroit, D. and Verplank, W.L., 1983, Human factors testing in the design of Xerox's "Star" office workstation. In A. Janda (Ed.) Human Factors in Computing Systems. New York, ACM: 72-77.

Broom B. and Welsh J., 1986, Detail suppression systems for interactive program display, *Proc. 9th Australian Computer Science Conference*, 83-93.

Broom B., Welsh J. and Wildman L., 1990, A multilingual document editor, submitted to 13th Australian Computer Science Conference, Monash, February, 1990.

Card, S.K., Moran, T.P. and Newell, A., 1983, The Psychology of Human Computer Interaction. Hillsdale, NJ, Lawrence Erlbaum Associates.

Eberts, R.E., 1987, Human-computer interaction. In P.A. Hancock (Ed.) Human Factors Psychology. Amsterdam, North Holland: 249- 304.

Eberts, R.E. and Eberts, C.G., 1989, Four approaches to human computer interaction. In P.A. Hancock and M.H. Chignell (Eds.) Intelligent Interfaces: Theory, Research and Design. Amsterdam, North Holland: 69-127.

Ehrlich, K. and Soloway, E., 1982, An Empirical Investigation of the Tacit Plan Knowledge in Programming (technical Report). Yale Univ., Dept. of Comp. Sci. no. 236.

Gomez, A.D., Egan, D.E. Wheeler, E.A., Sharma, D.K. and Gruchacz, A.M., 1983, How interface design determines who has difficulty learning to use a text editor. In A. Janda (Ed.) Human Factors in Computing Systems. New York, ACM: 176-181.

Hayes I. and Neucom R. and Welsh J., 1989, An editor for Z specifications, *Proc. CASE '89*, Imperial College, London.

Kiong D. and Welsh J., 1986, An Incremental Parser for Language-Based Editors, *Proc. 9th Australian Computer Science Conference*, 107-118.

Mayer, R.E., 1975, Different problem solving competencies established in learning computer programming with and without meaningful models. Journal of Educational Psychology, 67: 725-734.

Molzberger, P., 1983, Aesthetics and programming. In A. Janda (Ed.) Human Factors in Computing Systems. New York, ACM: 247-250.

Newell, A. and Simon, H.S., 1972, Human Problem Solving. Englewood Cliff, NJ, Prentice Hall.

Rose G. A. and Welsh J., 1981, Formatted Programming Languages, *Software - Practice and Experience*,

Smith, S.L. and Mosier, J.N., 1986, Guidelines for Designing User Interface Software. Technical Report ESD-TR-86-278, MTR 10090. Mitre Corp., Bedford, Mass.

Soloway, E., Ehrlich, K., Bonar, J. and Greenspan, J., 1982, What do novices know about programming? In A. Badre and B. Shneiderman (Eds.) Directions in Human/Computer Interaction. Ablex: 27-54.

Soloway, E., Ehrlich, K. and Black, J.B., 1983, Beyond numbers: Don't ask "How many"...ask "why". In A. Janda (Ed.) Human Factors in Computing Systems. New York, ACM: 240-246.

Teitelman, W., 1979, A display oriented programmer's assistant. International Journal of Man-Machine Studies, 11: 157-187.

Welsh J. and Rose G. A. and Lloyd M., 1986, An Adaptive Program Editor, *Australian Computer Journal*, 18, 67-74.