1987

THE USER INTERFACE IN INFORMATION PROCESSING: AN EMPIRICAL STUDY OF STUDENT PROGRAMMERS

Donna M. Kaminski
Western Michigan University

THE USER INTERFACE IN INFORMATION PROCESSING: AN EMPIRICAL STUDY OF STUDENT PROGRAMMERS

Donna M. Kaminski
Computer Science Department
Western Michigan University

ABSTRACT

The importance of human factors considerations is clearly recognized in the computing field today. However, there has been relatively little attention given to such issues in information systems and computer science curricula. This study examined the user interfaces designed by student COBOL programmers for an online, information processing application. A variety of problems were found in the visual displays and interaction designs suggesting a need for greater explicit consideration of this area in programming and software design training. A variety of suggestions are made as to ways of incorporating this issue into the curriculum.

The notion of "user friendliness" in computing has become quite popular during the last decade, particularly with the advent of widespread microcomputer usage, the influx of vast numbers of casual and non-technical users, the increasing number of online processing applications, and the rapidly expanding volume of information being stored on computers today. Hardware design, the software's user interface and documentation have all increasingly begun to reflect this greater awareness of the importance of considering human factors in computing, i.e., concern for users' needs, desires, abilities, tendencies and shortcomings, tailoring the computer system to the user rather than vice versa.

Indeed, research has shown that there is a payoff for this effort; "the human-computer interface can make a substantial difference in learning time, performance speed, error rates and user satisfaction" (Shneiderman 1987, p. v). Problems which users encounter on the interface, such as "cumbersome data entry procedures, obscure messages, intolerant error handling, inconsistent procedures, confusing sequences of cluttered screens" can all significantly reduce productivity (Morland 1983, p. 484). But direct costs such as wasted human and computer time are not the only costs of poor human engineering. Other more hidden costs can be even more expensive. There are extensive indirect costs as, for example, the time spent in trying to understand manuals, errors, system vocabulary, command abbreviations and system formats or the costs of printing excessive documentation. There is also the cost of human "suffering" -- the frustration of repeated mistakes, the anxiety from seemingly erratic behavior of the system, the pressure and fatigue from trying to understand system responses and keeping track of mentally difficult processes. And probably even worse is the cost from relatively limited use of computers because of previous negative experiences (Ledgard, Singer and Whiteside 1981).

However, in spite of the growing awareness of the importance of the user interface, relatively little direct attention has been given to this topic in most undergraduate computer science and information systems curricula. For example, the recommended standard computer science curriculum does not specifically include this topic (ACM 1979). Judging from its absence in most programming and software engineering textbooks, one would assume that user interface design issues do not receive much direct attention in such courses. (Exceptions do exist, e.g., Kacmar 1984; Marca 1984. Likewise, the relatively new approach to software development of prototyping [e.g., Boar 1984] gives much greater prominence to the user interface than the traditional approach did.) Computer science educational programs perhaps see this topic as relatively less important and beyond the scope and time of their coursework. Or perhaps it is assumed that such abilities come "naturally" with experience. What kind of user interfaces would experienced programming students develop if asked to design user friendly, interactive programs?

THE STUDY

This study looked at a sample of programs from intermediate-level computer science students' programs and examined the user interfaces they designed. They were asked to write user friendly, interactive programs allowing users to easily access and update specified information on a random access file. The interest here was to investigate and identify what visual display and communications design decisions student programmers at this level would spontaneously use without explicit instructions on interface ideals and guidelines. Were they able to take the role of user and imagine what a user might need, want and like? By analyzing particular types of choices, problems and/or less desireable features which they chose to use, specific user interface design issues could be identified to serve as a basis for further discussion and study. The specific aspects of their designs examined here include the reasonableness of command option indicators, the clarity and consistency of error messages, the completeness, intelligibility and conciseness of instructions, the efficiency of error recovery, the extent of shortcuts allowed to users and the clarity and easy readability of the visual display formats.

Alternatively, students could have been given prior instruction or readings on human-computer interaction guidelines as a method of learning about user interface issues. However, the interest in this study was in examining the design decisions these programmers used *without* specific human factors training, which no doubt reflects the more typical case in programmer development.

The analysis was descriptive rather than inferential based on a field study model rather than an experimental design. The interest here was on identifying and describing the types of problems, assumptions, and practices used in order to serve as a basis for further discussion and work. This then served as a pilot study for identifying particular factors to be used experimentally in subsequent tests with these programmers serving as users of the interfaces.

RELATED LITERATURE

Several recent books provide extensive guidelines for software designers in developing the user interface (Shneiderman 1980, 1987; Rubinstein and Hersh 1984; Bolt 1984). Shneiderman's (1987) text in particular does an excellent survey of research on user interface. Coombs and Alty (1981), Monk (1985) and Moran (1981) all provide collections of articles on user psychology and the user interface.

Some of the common complaints users have with regard to the interface include: incomprehensible terminology, no warning by the system of potentially dangerous actions, hard to remember language forms, non-uniform abbreviations, cryptic messages, redundant forms of the same operation, and many useful tasks still manual rather than automated (Ledgard, Singer and Whiteside 1981).

Morland (1983) provides guidelines for terminal interface design which address some of these issues. The objective is to reduce the frequency and consequences of user errors by keeping things simple. For example, for data input, use screens correlated with input forms, common cultural conventions (e.g., for dates), chunking (e.g., social security number), and mnemonic structures. For screen design, simplify the screen, eliminate unnecessary social amenities (e.g., "please"), use native language, and standardize and structure things. For error handling, prevent them if possible, correct or tolerate where possible, and report them immediately where users must fix them.

Dehning, Essig and Maass (1981) review research in this area and summarize some basic interface design principles: simplify, standardize, be consistent, use common human communication patterns, keep system behavior transparent and protect against human nature (e.g., forgetfulness and common mistakes). Designers should also keep in mind that users are likely to be goal directed (as a coping strategy against the barrage of new information) and may tend to skip reading all but "essential" material (Hammond and Barnard 1985).

Ledgard, Singer and Whiteside (1981) highlight aspects of their Assistant package which suggest attributes of a good user interface: an organized, visually appealing physical display that is concise and easily referenced (e.g., keywords in the margin); consistent behavioral rules with deductive capabilities and natural defaults; status information and positive reinforcement with no ambiguities; security checking (e.g., for defaults); exploitation of natural language using correct grammar, short keywords, single letter abbreviations; and several possible skill levels of interaction.

Thimbleby (1985) suggests a more general set of guidelines -- generative user-engineering principles -- rather than a specific list of rules (e.g., Smith 1982; Paxton 1984; and those above) since many rules are derived from very small experiments and cannot necessarily be generalized. For example, some generative principles include: conceal extraneous information from the user, "what you see is what you get," and be able to "run it with your eyes shut." In further support of the need for application-based decisions for interface design, Hauptmann and Green (1983) found no significant differences among menu, command and natural language interfaces in terms of user satisfaction or task performance time.

User psychology is also important to consider. Hammond and Barnard (1985) have done a variety of empirical field studies of dialogue design from which they have developed a theoretical model of user knowledge, including knowledge of the domain, system operations, interface dialogue, the physical interface and the problem itself. The designer should then specifically use the user's conceptual model in developing the interaction dialogue (Jagodzinski 1983; Gaines 1981). The system should be thought of as a communication network with information flowing among users, programmer, designer and machine (Oberquelle, Kupta and Maass 1983).

When users encounter problems in dealing with the computer, they are likely to attribute it to something they did wrong -- a threat to their self-esteem -- unlike designers (computer specialists), who probably view such things as external problems, as fixable bugs (Thimbleby 1985). Commercial users also are likely to want to view the computer as a tool to help them accomplish their task: as a means not an end (Eason and Damodaran 1981). The interface should thus accommodate the user's task and expectations as nearly as possible and keep any overhead time and effort for learning to a minimum.

THE SAMPLE

The programs examined in this study were written near the end of the course by all students enrolled in the Computer Science Department's COBOL course during one semester. This was a required course for all computer science majors, so this sample should be fairly representative. All had previously completed at least two prerequisite programming courses plus an average of three and a half additional computer classes as well as an average of one concurrent computer class (all requiring considerable programming). A typical subject had probably written a minimum of forty programs prior to this assigned task.

There were 64 usable programs, i.e., those submitted which ran roughly according to specifications. The sample was skewed towards the "better" students since programs for those not completing the course, missing this final assignment, or not following directions were not included in the study. Two-thirds reported grade point averages of 3.0 or better, both overall and in their computer courses.

These intermediate-level computer students were selected because they were advanced enough to have had a fair amount and variety of programming experience by this time. This course also provides an appropriate place in which to incorporate discussion of user interface issues (in response to this assignment) since programming per se need not be taught but, rather, a new third language.

THE TASK

The assigned task was to write an interactive, user friendly program which would allow a user to query, insert, delete, correct, update and print out specified information from a random access VSAM inventory file. General instructions were provided as to the types of functions needed and the necessity of handling possibly invalid requests. The only guidance given on the actual interaction style was the suggestion that a menu be used and the need to provide reassurance to the user that their requested function had been carried out. There was no specific discussion as to what "user friendly" meant. Rather, an attempt was made to elicit students' "natural," unguided design of the visual display and dialogue.

The compiler/system used did not easily allow students full-screen addressing capabilities as such; rather, a linear format was used. The actual processing logic of the problem was kept quite straightforward in order to encourage greater effort on the visual display and computer-user communication design aspects. The interface requirements were also kept rather straightforward in order to observe how students would handled these simple design issues.

Students ran their programs themselves using instructor test cases. Being "perfect users," there were thus no data entry errors, no misunderstandings of the appropriate responses, etc. -- certainly an unrealistic situation. The test data provided a check of both valid and invalid request routines for each function. However, no additional "bad data" was included in the test cases, thus relieving the necessity of extensive editing in the program.

THE RESULTS

In evaluating the user friendliness of a software system, some characteristics are not necessarily innately good or bad. Rather, the particular user and his or her level of expertise, frequency of use, and so on, must be taken into account. For example, should a menu repeatedly be provided every time the user is required to enter a function, or should it be provided only when the user requests it? Other measures, however, can be ranked on a more/less easy-to-use continuum. For example, when requested, is the user's range of function choices clearly specified with easy to remember, easy to enter option indicators? Both types of issues were examined here.

As suggested, all programs in the sample provided the user with an initial menu of function choices. Most also provided initial instructions as well. Most dialogues (92%) automatically repeated the entire menu each time a new function was being requested. The others allowed the user the choice of either entering a function selection indicator immediately (from memory) or else requesting to view the menu again. It was surprising that more students did not choose the more streamlined "expert" approach (i.e., providing help initially and then only again when requested), particularly given the simplicity of the program and their own experience with software. (This ties in with the suggestion by Ledgard, Singer and Whiteside (1981) and Morland (1985) of allowing a terse and a long mode of interaction). However, they perhaps interpreted "user friendly" as meaning geared to an audience of very naive beginning users. Or instead, two more likely explanations: it was easier to program with duplicate processing each time through, or else they hadn't thought about treating the function menu as optional.

Over one-half of the menus and introductory instructions, though usable, were not highly readable, nor easy to quickly follow. In an attempt to be friendly, the user instructions often tended to be quite long and tedious rather than short and to the point. This would be especially annoying to a user in conjunction with a menu (and introductory instructions) automatically repeated every time they needed to select a function. Surprisingly, some of the menus (13%) even had obvious spelling and/or grammatical errors. On more than one-quarter of the menus the actual options and their indicators were not easy to pick out. Indentation, column lists, spacing, alignment, extra blank lines and/or special visual markers would have helped considerably in clearly presenting the options to the user.

Regarding option indicators, one-third of the menus required the user to type in either the function name itself or its first letter(s); two-thirds used indicators unrelated to the function names. One might imagine that programmers would not require users to repeatedly type in entire function names, since the interface was menu-based rather than command-driven, but a number did. (And even in their own command-driven environment, most of these students themselves use abbreviations when running system software). Over one-half used numbers for indicators, though it is difficult to say whether this was specifically a design decision or merely for programming ease (e.g., a "go to/depending on" statement). Eight percent actually provided letters completely unrelated to function names.

The programs generally printed out fairly understandable error messages on invalid conditions. However, one-half of the programs used no consistent pattern for error messages; for example, sometimes using the word "invalid," sometimes "error," sometimes "not." Less than one-half used some type of "attention-getter" on error messages such as a series of *'s or beginning the message with a consistent key word such as "Invalid." The rest embedded the word "not" or "no" within the message making it appear, on a glance, to be very similar to their reassurances on valid requests. Users would have to read the entire message to determine whether their insertion or deletion had been done or whether an error had occurred.

A surprising two-thirds of the programs did not initially check the key for validity before continuing to ask the user to supply the rest of the information on that request. For example, on an invalid insertion, the user had to type in all the necessary information to build a new record, only finding afterwards that the insertion could not be

done because of an initial invalid ID key. Obviously little thought had been given to saving the user time and frustration on this point. As is often the case at this level, invalid/error situation handling receives little thought and testing.

Most programs gave reasonable responses to the user after carrying out a function. A number of minor improvements could have been made, however. One-sixth did not label the fields printed out in response to the user's request for information. When items were added or subtracted from the number in stock in the file, only one-half the programs informed the user of the original and/or new totals in their reassurance message. While not necessarily required, this feature would not have cost much in terms of either programmer or screen display time, and yet it may have been of value to users. Less than one-half the programs consistently worded reassurance messages reflecting the menu function terms (e.g., "removed" versus "deleted" versus "discontinued"). These are minor problems, but do suggest that more thought could have been devoted to considering the user's point of view during design.

Regarding overall screen layout, even though a linear rather than screen format was necessary, further consideration could have been given to the general visual appearance of the screen for both requesting and presenting information to the user. One-quarter used no blank lines at all for visual separation and one-half used no indentation to highlight things. Only 15% made good use of indentation and special characters as visual markers to indicate error conditions, set off the menu or point out that information was needed from the user.

DISCUSSION

It might be assumed that designing a computer-user interface for an interactive system is a skill that should come naturally to people. After all, it is quite similar to what people practice everyday: interacting with others, explaining things, requesting information and following directions. Programmers in particular are accustomed to explaining in detail, using very specific rules, the step-by-step instructions telling a computer what to do. Also, programmers themselves are heavy software users who have no doubt developed ideas about what they like and don't like about using certain packages and system programs. However, consider how well other types of human-computer interfaces have fared in their "friendliness": e.g., documentation, system software and actual application program structures themselves. How understandable, concise, thorough, organized, easy to use and visually appealing are these? Traditionally they are no better than user interfaces.

The programs evaluated in this study produced usable dialogues, particularly given the inherent simple nature of the task. Most flaws were not major. But taken all together they suggest a general failure to step back and actually consider the user's point of view. Much of the human factors literature concerns issues which help make software easier and faster to learn and to use, making for a smoother, less frustrating user experience. Many of these are seemingly small factors, yet together they contribute to the overall friendliness of a system.

It appeared that many of the programmers in the study spent minimal time considering the user view. One might surmise from this that they were focusing on the program's functioning and saw the interface task as more peripheral. Their actual use of the interface was perhaps more as a mechanism for testing their program rather than vice versa with the program as a mechanism for providing for human-computer interaction.

It is difficult to say whether this is reflective of the general population of programmers or merely of intermediate-level student programmers. As students, they know their programs will not be run by real users. There is thus less motivation to spend time working on the interface aspects. However, past observations of student-written systems from the graduate-level software systems development course show that their user interfaces are often little better; many projects from that course are actually intended for use by the general university computing community.

Can computer and information science education programs do anything to address this problem? There is one group of users that computer education has shown a good deal of concern for and whose needs and concerns are discussed and taught -- i.e., programmers themselves, especially maintenance programmers. Students in programming and software development courses receive specific instruction on issues such as programming style, structured programs, top-down design, modularity,

good variable naming, and internal documentation. They are expected to incorporate these concepts into all of their programs and projects. All of these make for more user friendly programs -- more readable, understandable, and modifiable programs which are easier to write, test, debug and change.

Human factors issues should be incorporated into programmer education. They need not be a central concern in a programming class, where the language itself, programming skills and ideas of maintainability deserve major attention. However, it can be a thread which runs through many courses. When interactive program assignments are given, some discussion of various alternative approaches might be discussed. It is very useful to have students run, demonstrate and comment on other students' programs from the class. The experience has been very educational for both the author and the "critic." Outside readings might be given which discuss human factors issues relevant to a particular assignment. Students could be asked to critique and suggest improvements for existent systems software and utilities which they regularly use. They might select the best and the worst they have used and elaborate on why they like/dislike it. Students might be asked to explicitly compare several different computers they've worked on noting differences with respect to ease of use, documentation readability, helpfulness, clarity of error messages, forgiveness of compilers, etc. These kinds of activities would go a long way towards making students more conscious and reflective on the user point of view; after all, they themselves are users. Yet considerable amounts of class time need not be spent on any of these topics. Ideas related to human factors should be specifically incorporated into software development courses, particularly by introducing such issues into more systems design and software engineering texts.

Whether one is a systems programmer, applications programmer or systems designer, there are end users who need to be seriously considered. User friendliness is clearly not something that comes "naturally" to all without explicit consideration and reflection. Just as concern for programmers as users has been incorporated into the computer science and information systems curricula, so should a greater concern for end users. Today's computer students are tomorrow's programmers. Online, interactive software can only become more widespread, with greater need for easy to use, less frustrating user interfaces.

REFERENCES

ACM Curriculum Committee on Computer Science. "Curriculum '78: Recommendations for the Undergraduate Program in Computer Science." *Communications of the ACM*, Vol. 22, No. 3, March 1979, pp. 147-166.

Boar, B. H. *Application Prototyping: a Requirement Definition Strategy for the 80's.* Wiley, New York, 1984.

Bolt, R. A. *The Human Interface: Where People and Computers Meet.* Lifetime Learning Publications, New York, 1984.

Coombs, M. J., and Alty, J. L. *Computing Skills and the User Interface.* Academic Press, London, 1981.

Dehning, W.; Essig, H.; and Maass, S. *The Adaptation of Virtual Man-Computer Interfaces to User Requirements in Dialogs.* Springer-Verlag, New York, 1981.

Eason, K. D., and Damodaran, L. "The Needs of the Commercial User." In M. J. Coombs and J. L. Alty (eds.), *Computing Skills and the User Interface*, Academic Press, London, 1981, pp. 115-139.

Gaines, B. R. "The Technology of Interaction--Dialogue Programming Rules." *International Journal of Man-Machine Studies*, Vol. 14, 1981, pp. 133-150.

Hammond, N., and Barnard, P. "Dialogue Design: Characteristics of User Knowledge." In A. Monk (ed.), *Fundamentals of Human-Computer Interaction*, Academic Press, London, 1985, pp. 127-163.

Hauptmann, A. G., and Green, B. Bert F. "A Comparison of Command, Menu-Selection, and Natural Language Computer Programs." *Behavioral Information Technology*, Vol. 2, No. 2, April-June 1983, pp. 163-178.

Jagodzinski, A. P. "A Theoretical Basis for the Representation of OnLine Computer Systems to Naive Users." *International Journal of Man-Machine Studies*, Vol. 18, No. 3, March 1983, pp. 215-252.

Kacmar, C. J. *On-Line: Systems Design & Implementation.* Reston, Reston, VA, 1984.

Ledgard, H.; Singer, A.; and Whiteside, J. *Directions in Human Factors for Interactive Systems.* Springer-Verlag, New York, 1981.

Marca, D. *Applying Software Engineering Principles.* Little, Brown, & Co., Boston, MA, 1984.

Monk, A. *Fundamentals of Human-Computer Interaction.* Academic Press, London, 1985.

Moran, T. "An Applied Psychology of the User." *Computing Surveys,* Vol. 13, No. 1, March 1981.

Morland, D. V. "Human Factors Guidelines for Terminal Interface Design." *Communications of the ACM,* Vol. 26, No. 7, July 1983, pp. 484-494.

Oberquelle, H.; Kupta, I.; and Maass, S. "A view of Human-Machine Communication and Cooperation." *International Journal of Man-Machine Studies,* Vol. 19, No. 4, October 1983, pp. 309-333.

Paxton, A. L. "The Application of Human Factors to the Needs of the Novice Computer User." *International Journal of Man-Machine Studies,* Vol. 20, No. 2, February 1984, pp. 137-156.

Rubinstein, R., and Hersh, H. M. *The Human Factor: Designing Computer Systems for People.* Digital Press, Billerica, MA, 1984.

Shneiderman, B. *Software Psychology: Human Factors in Computer and Information Systems.* Winthrop, Cambridge, MA, 1980.

Shneiderman, B. *Designing the User Interface.* Addison-Wesley, Reading, MA, 1987.

Smith, S. L. "User-System Interface Design for Computer-Based Information Systems." ESD-TR-82-132, Mitre Corp., Bedford, MA, 1982.

Thimbleby, H. "User Interface Design: Generative User Engineering Principles." In A. Monk (ed.), *Fundamentals of Human-Computer Interaction,* Academic Press, London, 1985, pp. 165-180.