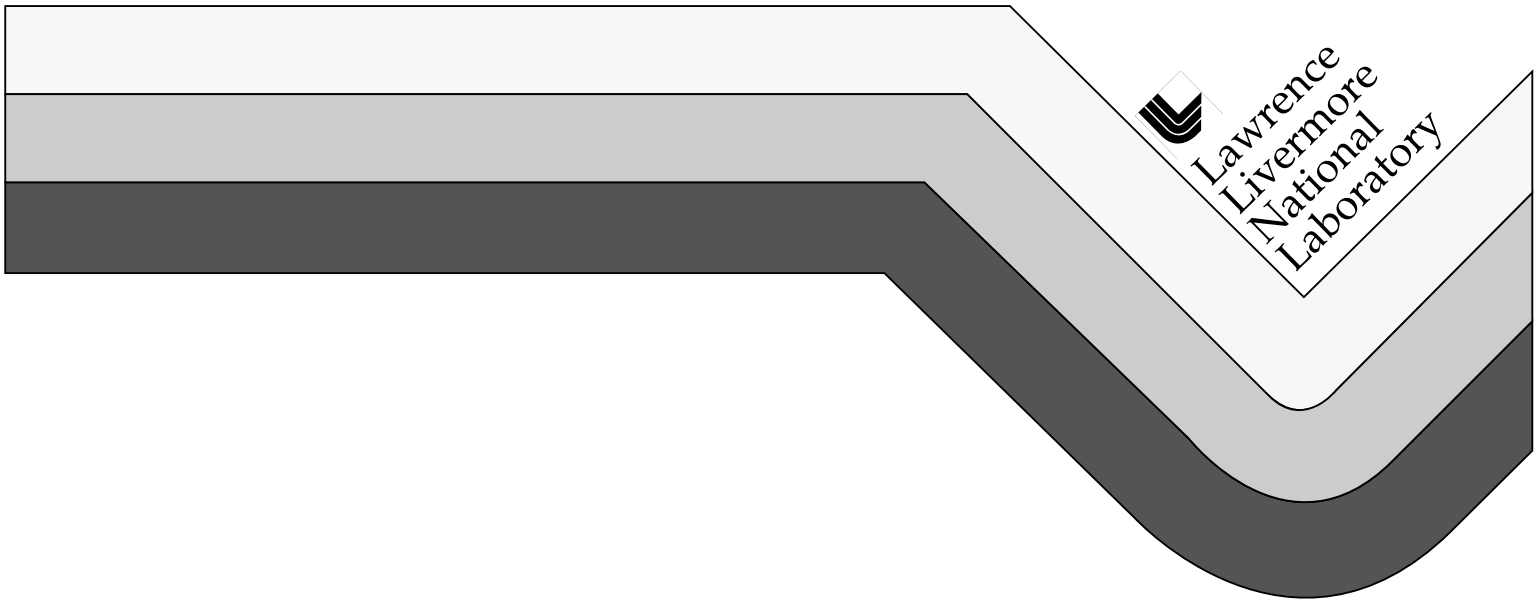
UCRL-CR-131046
B 330442

Improving the Quality of Numerical Software Through User-Centered Design

Cherri M. Pancake

June 1998



Work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-ENG-48.

Improving the Quality of Numerical Software through User-centered Design

Cherri M. Pancake

Department of Computer Science

Oregon State University

Corvallis, OR 97331

Final Report for Contract B330442

Submitted to Lawrence Livermore National Laboratory

Improving the Usability of Numerical Software through User-Centered Design

Cherri M. Pancake
Department of Computer Science
Oregon State University
pancake@cs.orst.edu

The software interface — whether graphical, command-oriented, menu-driven, or in the form of subroutine calls — shapes the user's perception of what software can do. It also establishes upper bounds on software usability. Numerical software interfaces typically are based on the designer's understanding of how the software should be used. That is a poor foundation for usability, since the features that are "instinctively right" from the developer's perspective are often the very ones that technical programmers find most objectionable or most difficult to learn. This paper discusses how numerical software interfaces can be improved by involving users more actively in design, a process known as user-centered design (UCD). While UCD requires extra organization and effort, it results in much higher levels of usability and can actually reduce software costs. This is true not just for graphical user interfaces, but for all software interfaces. Examples show how UCD improved the usability of a subroutine library, a command language, and an invocation interface.

Introduction

A "build it and they will come" mentality has dominated the design of scientific software for some time. It is increasingly clear, however, that this attitude is responsible for the failure of many software systems. Software users are no longer willing to put up with products that are difficult to learn or use [8]. This is actually a positive impetus for change. As one author writes, "Much of the improvement in software is attributable to research and knowledge supplied by research in human cognition and behavior, expertise that the computer scientists who designed the earlier systems never realized they needed until consumer discontent became impossible to ignore" [9].

Unfortunately, very few computer scientists have any formal training or expertise in cognitive psychology, ethnology, or even the subdiscipline of computer science known as HCI (human-computer interaction). One consequence is that only software targeted at mass markets like home computing shows

real evidence of being designed to please the consumer. Consider, for example, the use of metaphors and symbols to convey relationships and actions. In an environment based on the desktop metaphor, file folders and page icons can be used effectively to identify the origin or type of directories and files. The successes have been scored in metaphors for home and business computing environments, however, and not for scientific or numerical applications. In fact, there is remarkably little understanding of what human factors are important for software targeted at scientists and engineers [4, 7, 25].

Meanwhile, technical users are becoming increasingly vocal about how difficult it is to learn software tools and libraries, particularly in the area of high-performance computing (HPC). It is also becoming more difficult to woo these users to new products, even though libraries and software tools have great potential for facilitating HPC application development [13, 26, 24]. Technological sophistication is no longer enough of a drawing-card; the technical community expects software to be able to address their requirements, and to do it in ways that map well to their established patterns for developing applications. In addition to more traditional software characteristics, like robustness and accuracy, scientists and engineers now expect usability.

Usability encompasses a variety of factors, including how easy the software is to learn, how easily it can be remembered by infrequent users, its efficiency in the hands of advanced users, its forgiveness of user errors, and how well it can be applied to new tasks or needs that evolve. These are human factors, requiring that the software interface designer know and understand the target users. Fifteen years ago, Moran observed that software developers think:

> the best way to deal with the user is simply to take more care in considering the user -- all the system designer needs is to be given the time to do so. The designer is, after all, human and has the intuitions to predict what will be easy for the user. It is mostly common sense, anyway, isn't it? The limitation of this approach is obvious: The designer's intuitions do not necessarily match the user's. [18:2]

Historically, HPC software has been based on computer scientists' understanding of how the software will be used. This is a poor foundation for usability. The HPC user community is composed of scientists and engineers, who approach programming very differently from their CS counterparts. Consequently, the features that are "instinctively right" from the software developer's perspective are often the very ones that technical programmers find most objectionable or most difficult to learn [25, 24, 26].

This paper discusses software usability within the framework of numerical software for HPC. Attention is drawn to the software interface itself, which shapes the user's perception of what can be done with the software, in what ways, and with what degree of effort. The first section outlines the factors that contribute to usability. This is followed by a description of why and how users should be involved in software design, a process known as user-centered design (UCD). Case studies from the HPC industry illustrate what kinds of information UCD can yield and how that feedback can be applied to improve usability. A final section draws conclusions about where numerical software developers should concentrate their UCD efforts.

Factors Influencing Software Usability

Usability has several dimensions, of which four are particularly important for numerical software. Table 1 summarizes the key design objectives associated with each dimension.

The first factor, *ease-of-learning*, is particularly important for attracting new users. The interface presents the user with an implicit model of the underlying software. This shapes the user's understanding of what can be done with the software, and how. Learning a new piece of software requires that the user

discover, or invent, a mapping function from his/her logical understanding of the software's domain, to the implicit model established by the interface [12].

Designers often fail to take into account the fact that the interface is really the only view of the software that a user ever sees. Each inconsistency, ambiguity, and omission in the interface model can lead to confusion and misunderstanding during the learning process. For example, providing default settings for some objects, but not for all, hinders learning because it forces users to recognize subtle distinctions when they are still having to make assumptions about the larger patterns; a common result is the misinterpretation of what object categories mean or what defaults are for. In fact, any place the interface deviates from what users already know — about the domain this software supports, or about any other software with which they are familiar — is a likely source of error [16, 27]. Consequently, it is a mistake to rely on computer science concepts or terminology in designing an interface that will be used by non-computer scientists.

It is also important to recognize that the time a user invests to learn a library or tool will not be warranted unless it can be amortized across many applications of the interface. If the interface is a poor match to users' logical models, lack of regular use forces them to re-learn the interface many times over ([13] gives a good example). The short lifespan of HPC machines exacerbate the problem. Like it or not, HPC programmers will end up migrating their applications across several machine platforms over the course of time. The investment in learning a software package may not be warranted unless it is supported, and behaves consistently, across multiple platforms.

Once an interface is familiar to the user, other usability factors begin to dominate. *Ease-of-use* refers to the amount of attention and effort required to accomplish a specific task using the software. In general, the more a user has to memorize about using the interface, the more effort will be required to apply that remembered knowledge [12, 2]. This is why mnemonic names, the availability of menus listing operations, and other mechanisms aimed at prodding the user's memory serve to improve usability. Interface simplicity is equally important, since it allows users to organize their actions in small, meaningful units; complexity forces users to pause and re-consider their logic at frequent intervals. Ease-of-use also suffers dramatically when features and operations are indirect, or "hidden" at other levels of the interface. For example, the need to precede a desired action by some apparently unrelated action forces the user to expend extra effort, both to recognize the dependency, and to memorize the sequencing requirement.

Table 1. Usability objectives for software interfaces

Dimension	*Objectives*
Ease of learning	• provide an intuitive conceptual framework • map terms/actions to user's (not developer's) frame of reference • make terminology and operations consistent
Ease of use	• base the interface on recognition rather than recall • minimize interface complexity
Usefulness	• help user understand how to apply software to new situations • provide recoverability from potential errors
Throughput	• streamline common sequences of operations • reduce likely frequency of errors • must be efficient enough to increase user productivity

Where ease-of-use refers to how easy it is to figure out what actions are needed to accomplish some task, *usefulness* focusses on how directly the software supports the user's own task model. That is, as the user formulates goals and executes a series of actions to reach each goal, how direct is the mapping between what the user wants to do and what he/she must do within the constraints imposed by the interface? If a lengthy sequence of steps must be carried out to accomplish even very common goals, usefulness is low. On the other hand, if the most common user tasks are met through simple, direct operations, usefulness will be high (in spite of the fact that long sequences may be required for tasks that occur only rarely). Another aspect of usefulness is how easily users can apply the software to new task situations. If the implicit model presented by the interface is clear, it should be possible to infer new uses with a low incidence of error. For example, if the user knows how to generate a 3-way stencil for data access, it should be straightforward to extend that knowledge to 4-way stencils.

Since the inherent goal of software is to increase user productivity, *throughput* is also important. This measure reflects the degree to which the tool or library contributes to user productivity in general. It includes the efficiency with which the software can be applied to accomplish the user's goals, as well as the negative influences exerted by frequent errors and situations where corrections are difficult or time-consuming. For graphical interfaces or other software with start-up costs, throughput also measures the amount of time required to invoke the software and begin applying it to tasks.

It should be clear that all four dimensions contribute to how quickly and generally a software package will be adopted by the target user audience. It should be equally clear that users are the only ones who will have the insight needed to accurately identify which interface features contribute to usability, and which represent potential sources of confusion or error. The basis for usability lies in how responsive the software interface is to user needs and preferences — something that can only be determined with the help of actual users.

Involving Users in Design

User-centered design is based on the premise that usability will be achieved only if the software design process is customer-driven. The designer must make a conscious effort to understand the target users, the set of tasks they will want to perform, and the logical models they will use in applying the software to those tasks [21, 20]. The concept that usability should be the driving factor in software design and implementation is not particularly new; it has appeared in the literature under the guises of usability engineering, participatory design, and iterative design, as well as user-centered design [21, 23, 30, 28, 24]. There is not yet a firm consensus on what methodology is most appropriate, nor on the frequency with which users should be involved in design decisions (cf. [10, 29]).

What is clear is that the tradition of soliciting user feedback only during the very early and very late stages of development is not adequate for assessing and improving usability. During early stages, the design is too amorphous for a user to really assess how the interface structure might enhance or constrain task performance. During late stages such as alpha testing, the software structure has already solidified so much that user impact will be largely cosmetic. User involvement and feedback is really needed throughout the design process, since different types of usability problems will be caught and corrected at different points. Moreover, it is important to work with at least a few individual users on a sustained basis. The introduction of any computerized tool does more than replace a sequence of manual operations by automated ones; only by observing how a user interacts with the interface as he/she becomes familiar with it can designers understand the real issues affecting usability [14, 19, 6].

Our research work as a mediator between HPC vendors and their user communities, has provided a number of opportunities to observe how user involvement can improve interface usability dramatically. The most immediate benefit of UCD is that it allows developers to concentrate their attentions on those

aspects of the software that reflect users' highest priorities. Our experiences suggest a four-step model for incorporating UCD in the development of numerical software:

(1) Ensure that initial software requirements are based on demonstrable user needs. Realistic requirements can be identified only by soliciting input directly from the user community. Specifically, a library or tool will not be useful unless it facilitates tasks that the user already does and that are time-consuming, tedious, or error-prone when performed manually. If, instead, design is driven by the kinds of support that the software's developers are ready or able to provide, it will miss the mark.

(2) Analyze user tasks within the context of actual task environments. The first step in design is to study the intended audience in their world, where the software ultimately will be used. The point is to observe how users organize their efforts to accomplish tasks, what tasks need to be easiest or fastest, and what tasks are most subject to variation or indecision. For example, the observation that users write down or sketch out certain information for themselves provides important clues about how visual representations should be structured and how the interface should be documented, as well as indicating the need for associated utilities or toolkits.

(3) Design incrementally, with many cycles of design—user-test—redesign-and-expand. Based on the task analysis, the developer should begin to organize the proposed interface so that the most common user tasks are the best supported. "Paper prototypes" or mockups can be constructed to show basic interface concepts. This allows user evaluation to begin long before implementation efforts have been invested in features that will have low usability payoffs. It also allows the developer to observe users' instinctive reactions, one piece at a time. For example, the user might be presented with a few subroutine names and asked to guess what each does or what arguments are required for each. Early reactions might suggest major changes in thrust that will ultimately have repercussions throughout the interface.

(4) Have users evaluate every aspect of interface structure and behavior. As the paper designs are converted into prototype, then full implementation form, user tests should be performed at many points along the way. This permits feature-by-feature refinement in response to specific sources of user confusion or frustration. It also provides the developer with valuable insight into sources of user error — and what might be done to minimize the opportunity for errors or ameliorate their effects.

The idea of repeatedly asking users for input is intimidating to many software developers, whose experience has been that users are remarkably diverse and inconsistent. Interestingly, people who have actually applied this strategy note that it is not necessary to show a given version to more than a handful of representative users, since the iterative process means than premature or ineffective changes will be caught in later cycles. Moreover, a remarkable degree of consensus is reached by the later stages of iteration [15, 5]. The fact that users have seen, questioned, and commented on the entire interface structure from the ground up, so to speak, means that the structure actually does reflect user concepts and user preferences.

What User Involvement Reveals

This section describes the problems that users identified in the design of three software interfaces. In each case, we were involved in assessing the product's usability through a series of user-centered

activities, organized by us on behalf of the vendor companies. Although the deficiencies may seem obvious here, they were not at all obvious to software developers prior to the user reviews. Each of the three designs was considered adequate in terms of usability, and the reviews were not expected to reveal significant problems or insights. (In actuality, they led to further development and further cycles of user involvement.) The examples have been modified slightly to protect the identity of the companies involved.

The first example is the *programmatic interface* (also known as an API, or application program interface) for a message-passing library. Like many numerical and scientific subroutine libraries, this defines alternate syntax/semantics for invocation from Fortran and C programs. It includes four major categories of routines: point-to-point communications (e.g., send), group communications (broadcast), global operations (global maximum), and informational (buffer status). The developers were professional library writers, who already had several years of experience in developing math or operating-system libraries.

The *command language interface* from a parallel debugger serves as a second example. The language designers were influenced strongly by existing serial debuggers. Most of the commands simply augmented serial commands with a processor specification, indicating to which of the total processor set a command should apply. The developers assumed that by basing their work on existing products, they could leverage user familiarity to arrive at a language that was easy to learn and apply. The development team included a person with compiler and language design experience, as well as a person specializing in debuggers.

The third example is the *invocation interface* from a run-time environment, where each "command" issued by the user serves to invoke some component of the environment (e.g. loader, memory management utility, file migration utility, event tracing monitor). The components had been developed by multiple teams, so it was anticipated that their invocation syntax and error message structure would reveal some inconsistencies. These differences were exacerbated by the fact that each team's composition (in terms of member experience and interests) and software design objectives were somewhat different. Although the collection of teams as a whole included interface and language designers, individual teams were highly skewed toward specific system software components.

Users were involved in several different evaluation activities, which varied somewhat for the three examples. We extracted information from the specification documents for the products, to create "paper prototypes" outlining specific subsets of functionality and the language constructs associated with them. In several cases, users were presented with alternative designs and asked to choose among them or revise them to improve their intuitiveness. Later evaluations were conducted using alpha and beta versions of the software.

For the purposes of this discussion, we have organize those aspects of the three interfaces that were consistently criticized by users into several categories. Each has been assigned a general name and definition, but there is some overlap among categories. Examples are given with each definition. Table 2 provides additional examples, together with the solution suggested in user/developer interactions.

Inconsistency: lack of symmetry or consistency among elements within a given interface. The most blatant inconsistencies (e.g., spelling, naming of elements, or punctuation) can be caught through a careful checking by the software developers themselves. Nevertheless, users always find additional inconsistencies that are likely to result in problems. In the case of the programmatic interface, for example, users noted that in most routine calls, the source (of a communication) appeared first in the argument list, followed by the destination and the message itself, but in the global operations the message preceded the source/destination info. For the command language, users were quick to point out that some command modifiers were preceded by hyphens but others weren't. In the third case, they noted that while some options are controllable via both command-line flags and environment variables, just one of the two

was available for controlling others. In all these cases, the developers cited practical justifications for the inconsistencies — but users insisted that they would cause errors and confusion.

Incongruency: the interface's operation and object don't relate to each other in the same way that logical action and object do. An example from the programmatic interface was that the result was undefined if a non-current message identifier was specified as an argument to the status-checking routine; the users' comment was: "but how do you know it has become stale in the first place? The status code should just say 'I don't know what this identifier refers to'." Note that this category of error occurs because the interface fails to match the logical model of users. Another example, from the users who evaluated the command language interface, was that it was necessary to issue a whole series of commands in order to set breakpoint at multiple locations. Although a list of program identifiers could be specified on the command controlling the display of data values, only one location could be specified on the breakpoint command.

Incompatibility: the interface specification contradicts or overrides accepted patterns of usage. Where consistency compares elements within an interface, compatibility assesses how well the interface conforms to "normal practice." Users were quick to complain that the programmatic interface did not order in-arguments (i.e., those furnished as input to the routine) ahead of out-arguments (generated as output), but mixed them in seemingly arbitrary ways. For the second case study, users noted that one command had a triadic state (**more on** and **more off** controlled the status of the option, while **more** displayed its status), in contradiction to other examples, where informatory commands (e.g., **show** xxx) were distinct from those provoking actions (**add**/**remove** xxx). With the invocation interface, users complained that although some toggle-like options were controlled with binary flags (e.g., **+rv**/**-rv** or **-on**/**-off**), others had only unary flags (**-s**, interpreted as "on", with "off" applying where the flag was omitted).

Ambiguity: the choice of interface names leads to user misinterpretation. In the programmatic interface, users were confused by the fact that both "reduce" and "combine" were routines, where one referred to the operation the users traditionally call reduction or combination (i.e., acquiring values from each of multiple nodes and calculating the sum, minimum, etc.) and the other was a shortcut referring to that same operation, followed immediately by a broadcasting of the result to all nodes. The users complained that it would be very hard to remember which meaning went with which name. For the command language interface, similar criticisms were levied at the use of **list** (to display a source code listing) versus **source** (to establish the path for locating source code files).

Indirection: one operation must be performed as a preliminary to another. In the cases examined, indirection most often involved some sort of table lookup operation, so that the index (rather than the name assigned by the user) could be supplied as an argument to some other operation. In the first case study, users noted that they could not simply invoke the routine to set the so-called options mask, but rather had to retrieve the current mask first. Similarly, the invocation interface made freeing a node subset a two-step operation: the user had to obtain the subset's identifier through a status command, then issue a remove command specifying the identifier (although subset allocation could be accomplished in a single step). For the command-language interface, users complained that they couldn't begin executing the program in single-step mode; instead, they had to insert a breakpoint at the first executable line, then issue the "run" command, wait until the debugger stopped at the breakpoint, and only then start issuing single-step commands.

Obfuscation: operations or key information are hidden from the user's view. These problems are often due to poorly conceived default values. Users pointed out that the debugger command language,

for example, truncated the display of queue entries at 42 elements for one queue, but 51 for another (and neither of those quantities was under the user's control). In the invocation interface, environment variables were the only mechanisms for controlling some settings (others are set by command-line flags), which meant that the settings were effectively hidden from the user, who had to remember which setting fit into which mechanism category.

Fragility: subtleties in syntax or semantics that are likely to result in errors. In the programmatic interface, for example, all blocking operations involve routines whose names begin with "b" (e.g., bsend, brecv), but one routine beginning with that letter (bcast) is non-blocking. Fragility increases when the errors are essentially undetectable (that is, the software will still work, but results will be incorrect or unexpected). Users pointed out in the third case study that environment variables often cause users to think that they have specified an option, when in fact the software is not even aware of the user action, because the variable was spelled incorrectly or with lower-case letters.

Ergonomic problems: too many (or too clumsy) physical movements must be performed by the user. In most cases, problems occur because users are forced to do unnecessary or redundant typing. In the first case study, for example, users complained that they must specify the number of processes to be involved in a barrier or global operation, although that number almost always is the total count of all active processes (and so could be specified with a wildcard argument, such as "*" or -1). Similarly, users evaluating the command language interface pointed out that the way sequences of nodes are specified (using counts within parentheses, such as "(1:9)") involves many shift/unshift actions, making it both awkward and slow, even though the punctuation is really just "syntactic sugar".

The Cost of Responding to User Input

As part of analysis of the three case studies cited in the last section, we considered what kinds of changes had to be made to remedy the problems identified by user review. There were essentially six levels of improvement:

superficial change: modification limited to the documentation and/or procedure prototypes (e.g., to change the names of arguments)
trivial syntactic change: modification limited to the name of the operation
syntactic change: modification of the order of arguments or operands
trivial semantic change: modification of the number of arguments or operands
semantic change: relatively minor modification of the operation's meaning
fundamental change: addition of a new feature to the interface and/or major changes in operational semantics

Although collectively the users identified a large number of interface problems (almost two hundred), an overwhelming majority fell into the categories of superficial or trivial syntactic changes. That is, simple changes in the names used to refer to operations or operands were sufficient to eliminate the problem, from the users' perspective. Only six problems fell into the category of fundamental changes, requiring significant implementation work. Thus, the actual cost of responding to user comments was extremely low.

In fact, the widespread notion that UCD prolongs the software development process in terms of costs or time-to-market. Experience has shown that this is not really true [1, 11, 17, 22]. Usability improvements reduce ultimate costs, particularly maintenance, training, and technical support activities. In some cases, development time is actually shortened because features that would have required major implementation effort turn out to be of no real interest to users. Essentially, more developer time is spent earlier in the product design cycle (i.e., at the requirements and prototyping stages), rather than making

adjustments once the product has jelled. Generally speaking, the earlier in the design cycle that input is solicited from users, the easier and less expensive it is to make changes — particularly semantic or fundamental changes.

Finally, UCD engenders real interest and commitment on the part of users [15, 14, 5]. From their perspective, the developers are making a serious attempt to be responsive to their needs, rather than "making half-hearted, cosmetic changes when it's too late to do any good anyway" [31]. In our experiences, users have spontaneously helped in ways that go well beyond interface evaluation, such as developing example applications or publicizing the software among their colleagues.

Table 2. Examples of potential solutions for problems identified by users.

Category	*Example (from interface 1, 2, or 3*	*User-responsive solution*
inconsistency	(I2) "loadpath" sets the path for executables, while "source" sets the path for locating source code files	change command name from "source" to "srcpath"
incongruency	(I3) default value for environment variable only applies if user has never set its value in this session	setting a variable to the null string or unsetting it should result in use of default value
incompatibility	(I1) in the argument list, an array or list item is specified ahead of the integer indicating the number of elements	change the order of arguments so that the count occurs before the array/list to which it refers
ambiguity	(I3) the verb "monitor" enables program event tracing, but is easily confused with the concept of a debugging monitor	change the command to "trace" or "log_events"
indirection	(I2) to remove a breakpoint, user must first obtain a list of current breakpoints and their internal identifiers, then specify the identifier on the "remove" command	support the same specifiers (e.g., line number, function name) for both setting and removing breakpoints
obfuscation	(I1) some routines use "mode arguments" (predefined constants or strings) to distinguish mode, but only some of them have fallback defaults	every mode argument should have a fallback value, and it should be the mode that is most commonly needed
fragility	(I2) a breakpoint is set at a line number by prefixing the number with "#"; an unmodified number refers to an instruction address	unmodified numbers should refer to the most common usage; have addresses require the prefix "@"
ergonomics	(I3) many commands require that the user specify "all" if the command is to apply to all nodes	make "all" the default (no need to specify), since that's what's wanted 95% of the time

Conclusions

There can be little dispute that how users perceive and respond to software is critical to its success. Creating an elegant, powerful piece of software does not guarantee that it will be accepted by users. Ease-of-learning, ease-of-use, usefulness, and throughput are all important indicators of software usability, and they depend more on the software interface than on the underlying software structure.

User-centered design, as a methodology, attempts to capitalize on the observation that users are the ones who are best qualified to determine how software should support their work patterns. Making sure that software requirements reflect user needs, that interface organization correlates well with established ways of carrying out tasks, and that interface features and terminology are clear and efficient for users — all of these are relatively obvious, once the decision has been made to focus on potential users.

In the case studies reported here, a number of the problems identified by users could also have been found by usability or HCI specialists, or even interface developers who knew what to look for. More importantly, had user task structure been studied in the first place, many of the problems would never have occurred. Traditional approaches to software design rely on the insight of developers to identify where efforts should be concentrated. In contrast, UCD centers on repeated, frequent interactions with users to validate or re-focus every design decision. It is the user audience who establish what the key focal areas should be.

While this sounds onerous, experience has shown that it can be dynamic and positive for developers and users alike. Although the cases reported in the literature are not completely analogous, since they refer to business/commercial or custom-designed software interfaces, their results have been quite encouraging. Our experiences have borne this out. The problems found by technical users were all ranked as "important" when independent groups were asked to review the results of earlier sessions. In an overwhelming majority of cases, the improvements that were recommended proved to be both fast and relatively easy. They also had considerable impact on the clarity and intuitiveness of the interfaces. Usability is, in fact, within the reach of numerical software developers, if they can learn to center design more on potential users.

References

[1] R. G. Bias and D. J. Mayhew, eds., *Cost-Justifying Usability*, Academic Press, 1994.

[2] J. M. Carroll, J. C. Thomas and A. Mahotra, ‘‘Presentation and Representation in Design Problem-Solving,’’ *British Journal of Psychology*, Vol. 71, 1980, pp. 143-153.

[3] K. A. Ericsson and H. A. Simon, *Protocol Analysis: Verbal Reports as Data*, MIT Press, 1984.

[4] J. C. French, A. K. Jones and J. I. Pfaltz, eds., ‘‘Multidisciplinary interfaces (Panel Report),’’ *Proceedings Workshop on Scientific Database Management: Panel Reports and Supporting Material*, NSF sponsored workshop held at School of Engineering and Applied Science, University of Virginia, 1990, pp. 2-12.

[5] J. Grudin, ‘‘Interactive Systems: Bridging the Gaps between Developers and Users,’’ *IEEE Computer*, April 1991, pp. 59-69.

[6] D. L. Heppe, W. H. Edmondson and R. Spence, ‘‘Helping Both the Novice and Advanced User in Menu-Driven Information Retrieval Systems,’’ *Proceedings Conference of the British Computer Society*, Sept. 1985, pp. 95-100.

[7] B. Hesse, L. Sproull, S. Kiesler, and J. Walsh, ‘‘Returns to Science: Computer Networks in Oceanography,’’ *Communications of the ACM*, Vol. 36, No. 8, 1993, pp. 90-101.

[8] K. Holtzblatt and H. Beyer, ‘‘Making Customer Centered Design Work for Teams,’’ *Communications of the ACM*, Vol. 36, No. 10, Oct. 1993, pp. 93-103.

[9] W. Howell, ‘‘How Social Scientists Can Contribute to the Information Revolution, *Chronicle of Higher Education*, June 8, p. A40.

[10] R. Jeffries, J. R. Miller, C. Wharton and K. M. Uyeda, ‘‘User Interface Evaluation in the Real World: A Comparison of Four Techniques,’’ *Proc. CHI’91*, 1991, pp. 119-124.

[11] C.-M. Karat, ‘‘Usability Engineering in Dollars and Cents,’’ *IEEE Software*, May 1993, pp. 88-89.

[12] D. E. Kieras and S. Bovair, ‘‘The Role of a Mental Model in Learning to Operate a Device,’’ *Cognitive Science*, Vol. 8, 1984, pp. 255-273.

[13] J. Kuehn, ‘‘NCAR User Perspective,’’ *Proc. 1992 Supercomputing Debugger Workshop*, Jan 1993.

[14] M. Kyng, ‘‘Designing for Cooperation: Cooperating in Design,’’ *Communications of the ACM*, Vol. 34, No. 12, December 1991, pp. 64-73.

[15] T. K. Landauer, *The Trouble with Computers: Usefulness, Usability, and Productivity*, MIT Press, 1995.

[16] C. Lewis and D. A. Norman, ‘‘Designing for Error,’’ in *Readings in Human-Computer Interaction*, ed. R. M. Baecker and W. A. S. Buxton, Morgan Kaufman, 1983, pp. 627-638.

[17] M. M. Mantei and T. J. Teorey, ‘‘Cost/Benefit Analysis for Incorporating Human Factors in the Software LifeCycle,’’ *Communications of the ACM*, Vol. 31, No. 4, 1988, pp. 428-439.

[18] Moran, T. P., ‘‘An Applied Psychology of the User,’’ *ACM Computing Surveys*, Vol. 13, No. 1 (1981), pp. 1-12.

[19] National Research Council, *Information Technology and the Conduct of Research: The User’s View*, National Academy Press, 1992.

[20] J. Neilsen, ‘‘Non-Command User Interfaces,’’ *Communications of the ACM*, Apr. 1994, pp. 83-98.

[21] J. Neilsen, ‘‘The Usability Engineering Life Cycle,’’ *Computer*, March 1992, pp. 12-22.

[22] J. Nielsen, ‘‘Usability Engineering at a Discount,’’ in *Designing and Using Human-Computer Interfaces and Knowledge-Based Systems*, ed. G. Salvendy and M. J. Smith, Elsevier Science, 1989, pp. 394-401.

[23] D. A. Norman, ‘‘Cognitive Engineering,’’ in *User Centered System System Design: New Perspectives in Human-Computer Interaction*, eds. D. A. Norman and S. W. Draper, Erlbaum Associates, 1986, pp. 31-62.

[24] C. M. Pancake and C. Cook, ‘‘What Users Need in Parallel Tool Support: Survey Results and Analysis,’’ *Proc. Scalable High Performance Computing Conference*, 1994, pp. 40-47.

[25] C. M. Pancake and D. Bergmark, ‘‘Do Parallel Languages Respond to the Needs of Scientific Researchers?’’ *IEEE Computer*, Vol. 23, No. 12, Dec. 1990, pp. 13-23.

[26] C. M. Pancake, *et al*., unpublished results of user surveys conducted on behalf of Intel Corporation, IBM Corporation, Hewlett-Packard Corporation, Convex Computer Corporation, and the Parallel Tools Consortium, 1989-1995.

[27] A. Rizzo, S. Bagnara and M. Visciola, ‘‘Human Error Detection Processes,’’ *International Journal of Man-Machine Studies*, Vol. 27, 1987, pp. 555-570.

[28] A. Rose, B. Shneiderman and C. Plaisant, ‘‘An Applied Ethnographic Method for Redesigning User Interfaces,’’ *Proceedings of the Symposium on Designing Interactive Systems: Processes, Practices, Methods, & Techniques*, Ann Arbor, 1995, pp. 25-31.

[29] J. Whiteside, J. Bennett and K. Holtzblatt, ‘‘Usability Engineering: Our Experience and Evoluation," in *Handbook of Human-Computer Interaction*, ed. M. Helander, North-Holland, 1988.

[30] S. Wilson and P. Johnson, ‘‘Empowering Users in a Task-Based Approach to Design,’’ *Proceedings of the Symposium on Designing Interactive Systems: Processes, Practices, Methods, & Techniques*, Ann Arbor, 1995, pp. 25-31.

Technical Information Department • **Lawrence Livermore National Laboratory**
University of California • Livermore, California 94551