12

User-Centered Design Methods

One usability engineer watched while a new user tried to insert a floppy disk into a crack in the front of the computer housing instead of the proper disk slot. Then there was the user who when told to press any key, pressed the break [halt everything] key and brought the system down. Said the user, "It said press any key."

Traditionally, a lead sentence right about here says that of course a single chapter like this won't make you competent; you'll need to read many other books, go back to school, hire the author as a consultant. In this case, I have my doubts. Experienced experts will do better evaluation, but they won't know the application as well as those who are responsible, and they may not be listened to as respectfully. This chapter says most of what needs reading before getting started. The best next step is doing. Nobody has a magic formula beyond the general kinds of activities in which to engage. Little extra knowledge is essential. Certainly once a particular aspect is in focus—say, screen design for a form filling interface—reading up on what others have done and said will provide useful guidance. Practice, however, appears to be critical; every one of the cited successes came from an experienced team. Here is a list of techniques that have been involved in significant achievements.

Task Analysis

Chapters and books have been devoted to task analysis. They set forth pseudo-formal methodologies and structured sets of activities with

specific reports and analyses, and they make very boring reading.[1] In practice, task analysis is a loose collection of formal and informal techniques for finding out what the job is to which the system is going to be applied, how it is done now, what role the current and planned technology might play. Probably the most important aspect is identifying the goals of the work. This means not just the goals of the computer system but the goals of the business or entertainment activity for which it is contemplated. The most effective attitude is that a noncomputer solution is equally welcome.

The first step is always to go the place of work and see what is being done now. Analysts watch people and ask questions; they talk to executives, managers, supervisors, and especially to the people doing the work. The way work is actually done is seldom the way it is officially prescribed; real workers find better procedures. Analysts hanging about the workplace at Xerox discovered a rich underground culture of job knowledge passed by example and word of mouth that was nowhere in the supervisors' descriptions, the official job specifications, or the training materials. Such observations directly suggested solutions ranging from furniture and office architecture to electronic message facilities to facilitate informal communication rather than suppress it (Brown 1991).

Many analysts try to get workers to talk aloud as they work. The context of actively performing the job brings things to mind that aren't thought of later or mentioned in an interview. The opportunity to delve deeper and ask what might help is greater. Some analysts make videotapes and find footage of struggling workers valuable for converting unbelieving executives and designers. (Whether extensive taping is worthwhile as a supplement to live observation is debatable. It is very labor expensive.) The analyst looks for tasks that might be done better or not at all and tries to determine if a reorganization of the work, or even a change of goals, is called for. Thinking about computer function and design comes very late in the game.

Task analysts always take careful notes about how much work of what kind is being done. They measure how long people spend doing what. They stick around long enough to find out what the major problems are, not just by asking supervisors but by watching and asking many workers. They notice what kind of errors workers are urged to avoid, the "stupidi-

ties" on the part of workers about which managers complain. (Worker errors and stupidities are usually due to bad procedures and systems.) How much time does each part of the work or process take? How much variation is there in that time? On what does the variation depend? Are some workers much faster than others? Are some kinds of jobs finished much faster than others? How could slow jobs or slow people be converted into fast jobs and fast people? All of this investigation takes time, typically several full days.

No one set of methods or activities characterizes all successful task analyses. Almost every situation in which a new computer system is contemplated is different in important respects from every other. What appears to be required is a questioning approach that puts its emphasis on finding the relevant facts and keeping an open mind about how improvements are to be realized.

All consultants know "how much of their sustained employment they owe to the fact that few managers actually know what goes on in their workplaces" (Zuboff 1991, 164).

Here are some more specific techniques that have often been successful:

Learn the Job. On learning to do a job, experienced computer designers see all kinds of ways in which it can be improved. Effective task analysts never stop there. They ask users whether the improvement would be worthwhile; as often as not, the users will have thought of the same ideas and will have rejected them for good reason. The analyst is not the only smart cookie on the block.

Consult the Users. When it comes to jobs, it's hard to be wiser than the people doing them. Opinions and suggestions are collected both informally and formally. One formal method is the questionnaire, but most questionnaires are misunderstood by the people who answer them and provide uninterpretable or misleading data. In this aspect of task analysis, expertise is important. An expert analyst never writes a questionnaire before doing observations and never administers one without testing. A questionnaire is a user interface and is never gotten right the first time. Involving users in the design process increases the chances that they will

like what they get. The so called Scandinavian school of system design prescribes going out into the work environment in which the system is to be used and bringing users into the process at every step. The emphasis is on personal, social, and organizational factors in the introduction and use of new technology. Spending time with workers in the workplace and involving them in design helps ensure that the system will promote rather than interfere with important aspects of work and work life, essential work habits of individuals, and communication patterns of groups. For example, if the normal communications among the nursing and medical staff of a hospital are maintained through the bedside chart, replacing the chart with a computer could have disastrous consequences for the exchange of informal news about patients and the maintenance of good working relationships.

Use Subject Matter Experts. An indirect sort of observation and questioning is often achieved by using a subject matter expert, someone with extensive experience of the work in question. A talkative low-level supervisor is the usual source. The wise analyst never takes the information extracted from a subject matter expert as gospel. Experts have a tendency to think things fine and easy that aren't. They have a fish-in-water view. Information from subject matter experts is best used to guide and interpret field observations.

Conduct Time and Motion Studies. The old technique of time and motion studies is still used to good effect. The analyst first categorizes the various subparts and activities of a business process, then applies an inconspicuous stopwatch. These measurements call attention to time-consuming activities that can be reduced or eliminated. Whiteside, Bennett, and Holtzblatt (1988) report a case in which people spent significant time accessing an online help system when they didn't need any help.

Consult Normal Business Records. Measurements that most businesses keep as routine data (even though they may never use them) are often helpful. The amounts of money spent on consultants, training, overtime for error correction, and the like can be revealing. The budget tells where important gains are to be made. The analyst measures the number of whatever per day and asks what makes there be so much or little. Xerox homed in on its usability problems by counting service calls.

Formative Design Evaluation

"Engineering design shares certain characteristics with the positing of scientific theories. But instead of hypothesizing about behavior of a given universe, . . . engineers hypothesize about assemblages of concrete and steel that they arrange into a world of their own making" (Petroski 1982, 43).

The terminology *formative evaluation* is borrowed from the development of instructional methods, which often contrasts it with *summative evaluation.* Formative evaluation is used to guide changes; summative evaluation is testing to determine how good something is. Unfortunately, most usability testing for computer systems is summative: somebody wants to check that the finished system works or to produce data for marketing purposes. Such evaluation rarely produces useful design guidance.[2] In formative evaluation, by contrast, the notion is not just to decide which is better, system A or system B, but to produce detailed information about why system A is better or what is good and bad about both systems—what needs fixing, amplification, or replacement. The Olympic Message System, DEC, and SuperBook iterative development stories were about formative design evaluation. Formative evaluation can run a gamut of techniques from simple observations and questions to elaborate laboratory experiments. Some of the deeper methods are described in a later section on performance analysis. Here we focus on quick and practical approaches that have been successfully applied in ordinary software development settings.

The Gold Standard: User Testing

We want information that will help us make what people do with a system more productive. Only by studying real workers doing real jobs in real environments can we be sure that what we find out is truly relevant. User testing tries to get as close to this ideal as possible. The Olympic Message System trials and the Xerox video vignettes were the nearest we've seen. Field trials with finished systems can get even closer.

Usually practicalities get in the way. Real workers and real designers are busy in different cities. Measurement and observation is too

cumbersome, intrusive, and slow in the real workplace. The real system can't be tested until there is a real system, but much earlier guidance is much better. Compromises and approximations are necessary. The most common compromise is to test people like those for whom the system is intended with tasks like those they will most often do, using a prototype or early version in a laboratory setting that is vaguely like the intended office or other place of use. Experience suggests that such compromises are usually good enough. Information gained from user tests has been the most frequent source of major usability improvements.

User testing is straightforward. Someone—it has been most successful when the person was a specialist but also a full-fledged member of the design team—thinks up a set of tasks for users to try with the system, and gets test users to come in or goes to them. While users try, the tester watches, notes errors, measures times, and later asks questions. Many practitioners urge the test users to talk aloud as they work so that their conceptions, misunderstandings, and suggestions can be gathered and discussed. Others prefer the greater realism and more accurate measures of work efficiency afforded if users don't try to divide their time this way.

Mike Grisham at Bell Labs worked the bugs out of instructions to operate a voice messaging system by having volunteers come into the lab and try to operate a crude mockup of the system. With the initial, professionally written instructions, the majority of users made fatal errors of one kind or another. For a few weeks, Grisham tried one wording and procedure sequence after another, modifying each on the basis of what he learned from the previous ones. In later field tests, fatal error rates were less than 1 percent.

It is a good idea to start formative evaluation before building the system. Rapid prototypes that can be quickly built, then thrown away or changed, are extremely useful. Unfortunately, it is often either impossible or too expensive to build them. Moreover, it is best to get an even earlier head start, to evaluate how to build the prototype. A variety of techniques have been devised for testing user-oriented systems before they exist.

Wizard of Oz Experiments. The DEC email system, where Dennis Wixon played the part of the computer, was an example of this technique. Another is provided by John Gould (Gould, Conti, and Hovany-

ecz 1983), who wanted to see how well businesspeople could use a typewriter that took dictation using automatic speech recognition. Since no such system existed, he faked it by having a human sit behind a wall and translate. The idea is to test function and interface without having to build it first.

A Good Second Best: Heuristic Evaluation

In this technique real users, and sometimes real systems, are replaced by expert judgment. This distinctly doesn't mean just letting the boss or a consultant take a look. Rather, it means a systematic, disciplined inspection by several specially trained evaluators working independently. Jakob Nielsen, working with Rolf Molich at the Technological University of Denmark, evolved a set of ten heuristics for judging the quality of a computer's user interface. (A heuristic is a general principle or rule of thumb that is usually but not always effective.) His heuristics represent a consensus of usability guideline wisdom (Nielsen 1993c).

People already knowledgeable about computers spend a half-day to a few weeks learning the meaning of the heuristic rules and their basis and practice applying them. Then they spend an hour to a half-day examining the system, either real, in prototype, or as a set of written descriptions and drawings. They search for usability problems, aspects of the system's user interface that violate the heuristic rules.

Here are Nielsen's ten heuristics, briefly paraphrased:

1. Use simple and natural dialogue. Tell only what is necessary, and tell it in a natural and logical order. Ask only what users can answer.
2. Speak the users' language. Use words and concepts familiar to them in their work, not jargon about the computer's innards.
3. Minimize the users' memory load by providing needed information when it's needed.
4. Be consistent in terminology and required actions.
5. Keep the user informed about what the computer is doing.
6. Provide clearly marked exits so users can escape from unintended situations.
7. Provide shortcuts for frequent actions and advanced users.
8. Give good, clear, specific, and constructive error messages in plain language, not beeps or codes.

9. Wherever possible, prevent errors from occurring by keeping choices and actions simple and easy.
10. Provide clear, concise, complete online help, instructions, and documentation. Orient them to user tasks.

Evaluators need to know quite a lot more to cash these principles out in practice. Catching wordings that are computer gibberish is relatively easy, but knowing what wordings are truly communicative for intended users takes experience (indeed, may not be possible without repeated user tests). In one case, Nielsen (1992) found that usability experts, people with training and professional experience in usability engineering, noticed almost twice as many problems, on average, as programmers who had had a half-day introduction to the method. The more sensitive detectives weren't just being pickier; once identified, the problems they found were judged to be just as real and bad by evaluators who had missed them.

Nielsen has found that a single evaluator typically finds only about a third of the lurking problems—sometimes more, sometimes less, depending on system, experience, and luck. But different evaluators find different problems. In fact there's very little correlation between the 20 percent that one expert finds and the 40 percent that the next one detects. That means that more and more of the bugs can be detected by sending out more and more experts to look.[3]

Nielsen also found enormous variation between one system and another; you never know how many usability bugs to expect, although there are usually a lot of them. Over one set of eleven user interfaces subjected to exhaustive evaluation, the number of problems ranged from 9 to 145, with an average of 42. Of usability problems found this way, perhaps half are both easy to fix and well worth fixing (Nielsen 1993a). No matter how many problems there are, a single evaluator finds about the same proportion of them, usually around a third.

The number of usability problems found by heuristic evaluation is roughly the same as the number found by user testing (Virzi 1992). That is, one expert examination will find about the same number as one test user will tumble over. (And different user tests expose different problems, just as different experts do.) However, the kind of problems found by the two methods may not be the same. Those found by testing appear to be

somewhat more severe, and more frequently occurring problems tend to be experienced first. More important, user testing seems more likely to reveal why users are doing well or poorly and to offer insights into how to improve usefulness and polish usability. Heuristic evaluation is aimed primarily at catching common interface design errors rather than analysis of the adequacy of a system's functionality for getting a job done. User tests have more often led to significant innovations. One of Nielsen's studies suggests that the best approach may be a combination of two to four expert examinations and a like number of user tests at each iteration.

Paper, Pencil, Plastic, and Palaver

Another shortcut method is a sort of brainstorming session held over a cheap mock-up of the interface. People try to simulate a new interface and mimic the operations they would do with it. This method is particularly useful for initial designs of screens for information input and display. Recently it has been expanded into a cute technique for designing graphical user interfaces. In the PICTIVE technique a design team sits around a table with a drawing of a computer screen. Using a number of common interface tools—icons, menus, and cursors—in the form of plastic overlays or glued note sheets, they arrange and rearrange interface components and act out work activities done with the interface, attempting to design layout, dialogue, and functionality for a system they have in mind. If the design team includes actual users as well as usability engineers and programmers, the first try will be closer to the best (Muller 1992).

A related technique is called a cognitive walkthrough. A group of experts reads the specs, then gathers round to imagine going through the same mental steps a user would, discussing and commenting on the good and bad. In all, these group design methods appear to find fewer problems than Nielsen's heuristic evaluation but perhaps different ones (Jeffries et al. 1991; Muller, Dayton, and Root 1993).[4]

Engineering Models

More formal methods rely on explicit models of human task performance; the best is the so-called GOMS model (Card, Moran, and Newell,

1983). Skilled tasks are broken down into atomic components consisting of goals (what the user is trying to do—say, change *this* to *thus*), operators (the actual action needed—a thought or a keystroke), and methods (the strategy chosen—for example, deleting and replacing the whole offending word or line or just editing one letter). Careful observations are made of the use of a system, and the time for each kind of operation is measured. To estimate how well people would perform with a variation of the design, the engineer specifies the new sequence of operations that expert users would execute. The time required to achieve goals with the new system is estimated by adding up the times for all the component actions. The predicted times for different strategies are compared, and the expert user is assumed to choose the best one. For the kinds of systems and tasks to which it is applicable, GOMS analysis is sometimes sufficiently accurate to substitute for heuristic evaluation or user tests (Gray, John, and Atwood 1992; Nielsen and Phillips 1993).

GOMS works best when the operators are simple perception-motion sequences like keystrokes. Its major deficiencies are the incorrect assumption that experts always choose the best of the available ways to do something and the lack of a good way to predict errors, which often eat a large share of a user's time and patience.

Another form of analysis has been tried for cases in which the action is mostly mental. In cognitive complexity analysis, the thought processes are simulated by an artificial intelligence program. To measure the difference in difficulty between two designs, the number of steps in the program is counted. Similarly, the proportion of common steps in the two predicts transfer of training between one and the other. The authors, Kieras and Polson (1985), have had some luck predicting which systems are easier to learn for people who already know which other systems. The method, however, is labor intensive and difficult to apply successfully.

Performance Analysis

Performance analysis refers to the kind of systematic experiments and observations that were illustrated in the SuperBook project; in contrast

to the methods reviewed so far, their main goal is deeper understanding on which to base innovation and initial design rather than assessment and improvement of existing designs. Performance analysis is more likely to be found in an industrial or academic applied research organization than in the usability assurance group of a software house. Broadly, performance analysis studies people doing information processing tasks in an attempt to understand what they do well and poorly, where help is needed, and, if possible, what might help. Performance analysis is done either in the laboratory with somewhat abstracted tasks, such as suggesting titles or key words for information objects, or explicitly with an existing technology for the performance of some job. Performance analysis, when successful, leads to the identification of a human work problem that a computer can ameliorate. Computers are so powerful and flexible and there are so many things that they can be made to do that finding the right problem is often harder than finding its solution. An example is the unlimited aliasing technique already described. Once it had been discovered that many more names were needed, providing them with a computer was easy. There are a variety of aspects of user task performance that can be observed and used in performance analysis.

Time Is the Essence

In striving for work productivity, we fundamentally want to reduce the amount of time that a user needs to spend to accomplish a given amount of work. Eric Nilsen, in his 1991 University of Michigan thesis, measured the time users took to make selections from menus of different design. He discovered that the popular walking menu results in excessive selection times because it requires carefully aimed curved movements with the mouse. These movements are difficult for people. By substituting two short, straight movements for one long, curved movement, Nilsen found an alternative that saved substantial time.

Errors Are the Villains

Errors are the main thieves of time (and satisfaction). In an early and influential book on the psychology of human-computer interaction, Card, Moran, and Newell (1983) presented detailed analyses of the actions involved in using text editors. The data showed that the largest

source of wasted time was errors. Finding what causes errors and getting rid of the cause (or at least providing more than a beep or secret code number to help the user recover) has big benefits.

"Beep," System has quit due to error 593.

Learning from Learning

Learning time is both a diagnostic and a design object in its own right. In some software development environments, standard operating procedure is to assume that usability is the exclusive province of training; just design a powerful feature, then let someone figure out how to teach its use. The bigger the training manual, the longer the time required to learn a particular function, the more redesign is needed. (Beware a 400-page manual beginning, "This application is very easy to learn and very easy to use.") Analysis of what aspects users find hard to learn illuminates both how to change the system and how to write the instructions if all else fails.

Variability Is a Source of Progress

In Darwinian evolution, natural genetic variation provides the opportunity for change. Something similar applies in usability design. Looking where there is the most variability between one task and another, or between one user and another, can reveal paths to progress. If some people do things well that others do poorly, the more efficient may have found a strategy that everyone could use. The power of the structured search feedback technique of SuperBook was discovered this way.

The flip side is that especially slow or error-prone users may have discovered an especially poor way to perform the task that deserves extinction. These kinds of situations occur most frequently in powerful, feature-rich systems that offer users many ways to do the same job. Some users are sure to compound enormously complex, and sometimes enormously poor, procedures. The same comments hold for variations across task problems. If some problems are dealt with quickly and others slowly, an opportunity may be hiding or a soft spot in system design may be at fault. SuperBook again offers an example—this time a flaw. With

SuperBook, chemists did very well on most information search problems, much better than with paper technology. However, when the problems required information from pictures of chemical structures, they did quite poorly. Analysis disclosed that students using the online version often failed to bring up the pictures.

The Talent Search

Productive technology for the service sector multitudes should not require rare talent. Yet as figures 10.2 to 10.4 showed, many, and perhaps most, do. Tracking down the special abilities needed can unearth hints about what needs to be done. In the case of each of the problems illustrated in these figures, ways to reduce the dependence on special abilities were found. In text editing, when so-called full-screen editors were introduced, older people found them much easier. Young people got better too, but the older people improved more. Designers of full-screen editors had believed that it was the WYSIWYG (what you see is what you get) visual nature of full-screen displays that accounted for their superiority. Gomez and Egan showed that, instead, it was the way the user specifies a place in the text that mattered. The hunt that found the reason was guided by looking for parts of the task that required special abilities. Gomez and Egan discovered that people with poor spatial memory couldn't think abstractly about positions in the text so they had to point. Older people had trouble formulating complex statements. Replacing complex commands with simple syntax or arrow keys reduced the need for youth and talent.

In the case of the database query languages studied by Greene and associates, the analysis suggested that some method of querying that did not require logical reasoning was needed. Other experiments showed that people, regardless of logical ability, can identify the cells in a table that contain the data they want. (They may not know much about data, but they know what they like.) Greene, and coworkers devised an interface in which the computer provided the tables and the users had only to mark the cases they wanted. As figure 12.1 shows, this interface made everyone almost equal, again by bringing up the laggards without hurting the champs.

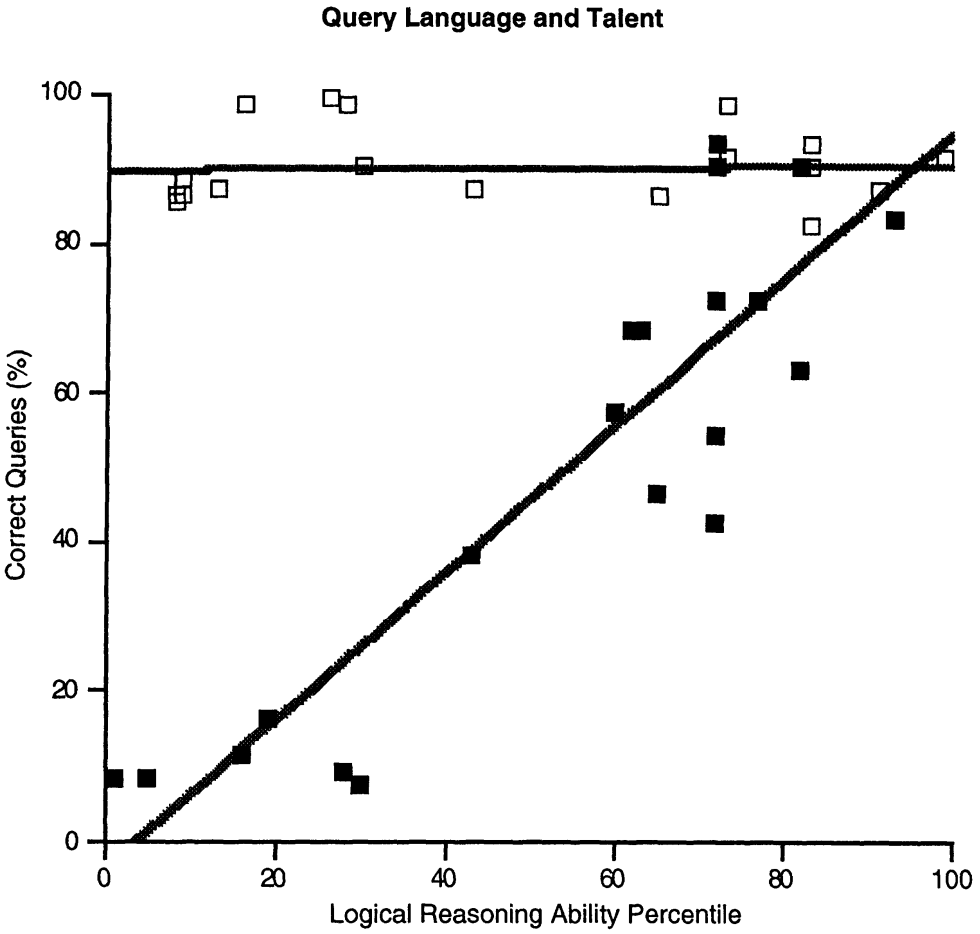

Figure 12.1.
After discovering that standard query languages require special logic expressing abilities but picking the right cell in a data table doesn't, a new query language that everyone can use was invented. Data from Greene, Gomez, and Devlin 1986.

The final example, shown in figure 12.2, comes from Susan Dumais's work on text querying and makes a similar point. Even when people could say what they wanted in their native language, their verbal fluency—the ability to remember words of a particular meaning—had a large impact on success. In an alternative search technique, once users find examples of desired documents, they can ask the system to find similar ones. When they are using this technique, the need for high verbal fluency virtually disappears.

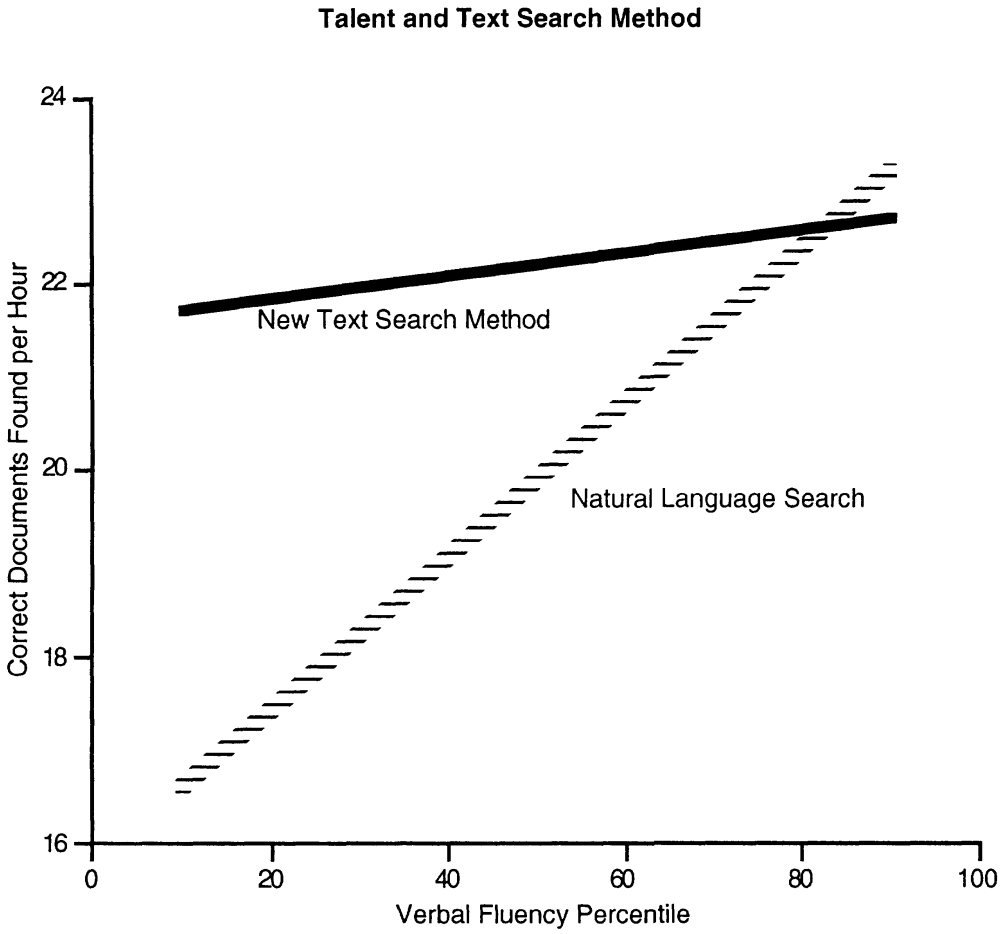

Figure 12.2.
Telling the computer what you want, even in your own words, demands high verbal fluency. When people instead can provide examples of the documents they want, everyone does well. Data from Dumais and Schmitt 1991.

An Aside: Different Strokes for Different Folks?

The fact that some interaction styles can be easy for one person and difficult for another has not been lost on software designers; as the two previous figures show, it's hard to miss. However, some have drawn from this a dangerously incorrect conclusion: systems should provide a wide variety of ways to do the same thing, so as to fit all comers, and to make interfaces reconfigurable so users can more or less design for themselves.

This conclusion has several problems. First, while it is often true that a particular method is much easier for Jill than for Jack, it is rarely true

that some other method will be much easier for Jack than for Jill. In virtually every case I know of, the real situation is like the one depicted in figures 12.1 and 12.2: those who do well with method A will do well with method B too, while those who do poorly with A may sometimes do better with B. If a method is found that helps Jack, it will not hurt Jill—it just equalizes the two of them somewhat. There are techniques that few can master and techniques that all can master—not different easy strokes for different folks.

The only exception, and even it is often overestimated, is the difference between novices and experts. What the user can be expected to know or needs to be told obviously depends on experience. It's not a good idea to make beginners memorize eighty-eight arcane key chords before they can get started, even if chords may be helpful after mastery. Advanced, powerful techniques are best reserved for experienced users. However, the line can be fuzzy. Many menu-and-mouse-driven interfaces provide alternative keyboard accelerators—keystroke chords that have the same effect as a menu selection. These are intended, and believed by most expert users, to make those who are willing to learn them faster. Tognazzini (1992) claims the advantage is illusory. He says timing studies of experts invoking commands by key chords and mouse with menus find them equally fast. He claims the keystroke method just *seems* faster because of a perceptual phenomenon by which time appears to run slower when it is occupied by things you consciously see and think about, such as menu alternatives and cursor movements on the screen.

The absence of cases where different methods are optimal for different people would not surprise educational researchers. Despite a widely believed myth, decades and hundreds of attempts to show these kinds of effects in teaching methods—that one method is best for one kind of learner and another for others—have generally met with failure (Cronbach and Snow 1977). Most graphs of learning speed for different teaching methods used for students with differing aptitudes, say verbal or spatial abilities, look like figures 12.1 and 12.2. One method stresses the ability more heavily than the other; the other makes all students more nearly equal, but there is no flip-over. In the rare cases when there is a flip—student type V does better with method 1, student type M with

method 2—the effect is always small; a tiny percentage of all students change places in the ranking because the teaching method changes. Again, the major exception to this generalization is experience. An advanced text is more helpful for advanced students than for beginners, and an introductory text the opposite. Interestingly, when students are offered a cafeteria of learning methods, having choice sometimes helps, but only for the top-ranking students. Only the most able are able to take advantage of using different methods at different times (Cronbach and Snow 1977). Perhaps we have a parallel in computer systems. Perhaps the most talented—the programmers and power users—do profit from a wide variety of alternative methods and features. This could explain why designers and data processing department gurus are attracted to variety and self-tailored interfaces. On the other hand, as we will see later, given a choice between two interfaces, one of which is objectively quite superior, most users, even programmers, may have little better than a fifty-fifty chance of choosing the more effective. Thus, providing both would make half the users less efficient than necessary.

The second problem with variety is that it is confuses and delays. The more techniques there are to learn, the longer it takes and the more likely the learner will mix up the options and actions of one with another. The more options available at one time, the longer it will take to choose—it takes twice as long to decide among eight alternatives as between two—and the greater will be the chance of choosing wrong. One popular spreadsheet program offers more than eight different ways to move from one cell to another. Even highly expert users of the system often select the less efficient method for particular tasks (Nilsen et al., 1992)

An additional problem with user tailorability (letting users design their own functions and interfaces) is that usability engineering is not an amateur sport. It's easy enough to be done by any software development team but takes much more than tinkering by average users, who will not know what's best or even how to tell. Making users design their own interfaces is not much more sensible than making drivers design cars or highway bridges. In summary, Jack Spratt and wife are not a good model for productivity software design.[5]

Guidelines, Standards, and Examples

When iron was first used in bridges, there were very frequent collapses—one in four bridges by certain accounts. In 1847 Queen Victoria appointed a commission to find out what was going on and charged it to look into the situation and propose rules. The commission came up with some useful strictures, though the theoretical reasoning behind them was entirely mistaken (Petroski 1982).

A final way to go about user-centered design is to heed good advice. Advice based on wisdom gained from experience, the best guesses of experts, and the results of research, both basic and applied, have been codified in compendia of design guidelines. These how-to books range from the minutia of one company's suggestions for one product type to Noah's arks covering all species. Collections of general guidelines range from 162 to 944 entries; a catalog of all the separate admonishments plausibly relevant to a single system can easily exceed 1,000 (Brown 1988; Marshall, Nelson, and Gardiner 1987; Mayhew 1992; Smith and Mosier 1984).

Undoubtedly, careful attention to applicable guidelines would improve usability over current norms. Clear violations of clearly established principles are rampant in commercial products: commands like **A3492-Q6,** yellow letters on gray backgrounds, missing help information for incomprehensible menu choices. There are, however, severe limitations to the value of guidelines. First, in the current state of the art and science, their validity is often questionable. Pick any specific guideline from any collection and ask three experts; at least one is likely to disagree or qualify its application. Like commonsense aphorisms, usability guidelines are sometimes contradictory: "provide alternate accelerators for experts" but "always keep it simple." The guidelines all say to make instructions specific and clear, but clarity can be assured only by user testing. Second, guidelines are hard to follow. There are so many of them that finding all the relevant ones is difficult. Many are vague: for example, "provide feedback"—but about what, when, how much? Third, guidelines provide little support for the critical analytic and creative parts of design. In practice, guidelines are infrequently consulted and inconsistently obeyed

(Bellotti 1990). Most designers appear to copy their designs from predecessors and competitors, not a bad idea. Nielsen's ten general heuristics are easier to apply than 1,000 specific guidelines but are successful for evaluation of designs, not for their creation. He has found that the same training in heuristics that makes programmers into reasonably good interface critics has virtually no effect on the quality of interfaces they produce before inspection begins. A principle that helps you recognize problems does not necessarily proffer the skill to avoid them.

Standards differ from guidelines in being more specific and in being enforced, or at least agreed in some official manner. Their main goal is consistency. Provided they do not cast in concrete bad elements of design—a real concern at our current stage of knowledge—standards can make it easier for users to go from one application or system to another. As I write, companywide and international committees are laboriously negotiating standards for human interfaces for computers. Luckily for all of us, many are concentrating most heavily on urging user-oriented methods of design rather than specifying detailed solutions.

Science

It is tempting to think that cognitive psychology, human factors and human engineering principles, and the newly emerging field of cognitive science might provide theory and fact to steer this effort. They can, but only to a modest extent. Psychological science has provided some real advances in understanding human behavior but only in limited domains, usually attached to narrow problems that have been brought to the laboratory. A few "laws" and principles are available that speak to system design. For example, the Hick-Hyman law says that decision time is proportional to the log of the number of equal alternatives. Useful in designing menus, it implies that a single screen with many choices (well organized and laid out) is better than a series of screens with a few choices each (Landauer and Nachbar 1985). Fitts' (1954) law tells how long it takes to point to an object depending on how large and how far away the object is. Fitts' law helps in designing pointing devices and laying out the icons and buttons on a screen. Yet another law, the power law of practice, tells how response speed increases over time. Given

appropriate user tests, it could predict by how much and when experts with a new system would outperform those using an old system. There is knowledge about how much people can remember from one screen to the next and about how long it takes them to do most of the simple stimulus–response things they do when interacting with a computer. This knowledge forms the foundation of the GOMS method.

Knowing all this, plus immersion in all the lab lore and rules of thumb enshrined in guidelines, combined with experience, is unquestionably valuable. In the Bailey experiment cited earlier, half the designers had training and experience in computer system human factors. Their designs for the recipe file system tested out significantly superior to those of programmers lacking such backgrounds (although it took them longer to write the programs). In fact, their initial designs were better, in terms of flaws and user efficiency, than the final versions arrived at by unwashed programmers.

Nevertheless, as a foundation for design, cognitive science is up against a tough, possibly impossible problem. The human mind is an extremely complex information processing device. Physically it is based on a brain that has hundreds of billions of mysterious parts that interact in extraordinarily complex and almost completely unknown ways. It uses enormous amounts of information from enormous-squared amounts of experience; we ingest billions of bits of perceptual information every second. Its complexity is similar to what one bumps up against trying to predict weather or model the turbulence of air flow around an airplane wing. Like other such dynamic physical systems, the brain is vulnerable to chaotic disorganization. The mind—the brain's function—may be at least as complex. There have been repeated expressions of hope that mental activity and behavior will somehow be subject to simpler organizing principles than the physical machinery on which it rests. Such occurrences are not without precedent in nature; the heart beats in finely conducted rhythms despite the fact that its cellular systems for neural control are of vast complexity. However, the hope of discovering simplicity in mental function has so far gone mostly unfulfilled. (I hear psychologists and AI proponents screaming. They want to voice confidence in one or another theory. But read their literature. Every theory has serious holes

and ever-changing countertheories; almost none has the validity, generality, or precision needed as a base for technology.)

We cannot rely on scientific theory for answers; however, all is not lost. The same situation applies in most areas of engineering, and where it does not apply today, it certainly did in times not long past. There's a great deal of debate over the relation between science and technology. Some knowledgeable commentators, such as the philosopher of science Kuhn (1977), go so far as to argue that technology progresses faster when science is in abeyance. Technological advances most often come from accumulated practical wisdom. People try things, see what's wrong with them, try to fix, and so forth. Only at occasional critical points does even a modern field of engineering such as electronics need to renew its scientific principles to solve a problem or to suggest new directions. It is said that the interplay between science and technology is, despite common belief, much more often in the opposite direction. Problems raised by technology raise curiosity that drives scientific research.

"We have not had a thousand failures. We have discovered a thousand things that don't work." Attributed to Thomas Edison.

Here is an illustrative example of both the success and limitations of science as so far applied to computer usability. Early research efforts focused on the design of command languages. The question was what words to use and what syntax when stringing them together. Some authorities assumed that "all natural" words would be a big advantage, as would consistent, natural syntax. Experiments comparing ways to choose words made several discoveries. First, for small systems, say a basic text editor, and frequent users, the choice of words didn't make much difference. Total nonsense strings were a problem, but using the words *allege, cipher,* and *deliberate* instead of *omit, add,* and *change* was inconsequential (Landauer Galotti, and Hartwell, 1983). Why? In the first place, new users were so busy learning about the system—the very concept of a typewriter that did mysterious things behind the scenes—that the added difficulty of learning a new meaning for a few words in special context didn't slow them down. Second, humans have many

names for the same thing, and computer actions don't fit known word meanings perfectly, so there isn't an enormous difference between one word and another as a choice for a computer command. Investigations of syntax had a similar upshot. Using "normal" English word order sometimes made a difference but not a great deal. Using a consistent word order—verb before object, for example—sometimes was helpful. However, sometimes, as in natural languages like English, it is more useful to have different orders for different kinds of functions or activities (Barnard and Grudin 1988; Barnard et al. 1981). The language research with the best payoff was on how to construct abbreviations. It turned out that using a consistent rule was more important than what rule was used (Streeter, Acroff, and Taylor 1983).

Although I am pessimistic about basic science as the main basis of usability engineering, nevertheless it can do good. We are surrounded by bad design decisions that could have been better guided by existing knowledge about perception and cognition; for example, screen color is often used in ways that science says confuses and slows. Perhaps some such knowledge helped Bailey's sophisticates, and perhaps all designers would profit from better training in usability-relevant science. Moreover, the examples we have seen show that research aimed at better understanding of the cognitive underpinnings of information system use can pay off. However, it will take time, and much greater volume than the current trickle, to make a major contribution.

Now, the most-needed aspect of science may be its skepticism. Current design of information tools is largely based on intuition and art. It is popular among practitioners to compare computer interface design to architecture. The problem with intuition as a basis for design in this realm is that intuition about human thought is unreliable. People do not understand their own minds, nor can they predict their own behavior or that of their best friends. There is endless evidence for this assertion. Here are some examples. Remember the numbers 2, 4, 3, 7, 1. Got them?

Do this in your head only, and don't look back at the list.

Quick, was the number four part of that set? How did you determine that it was? Most people will say that they compared four with 2, then with 4, decided it matched, and answered yes. The evidence is over-

whelming, however, that they ordinarily do nothing of the sort. Instead, they unconsciously compare the target with all the numbers in the set before deciding. The evidence is that the time to make such a decision increases by 35 milliseconds for each additional number in the set; it doesn't matter if the added numbers come before or after the matching one (4 here) or even if none match.

You meet with a group of people for a day, or live with them in a dorm for a semester. Then you try to judge which ones think that you like them and which don't. If you are an average person, you will be quite confident in your predictions, but you would do almost as well by flipping a coin.

You talk to somebody for an hour, then try to rate his or her intelligence or honesty. You would have done almost as well without ever having met the person. You probably don't believe that. Your intuition tells you otherwise, largely because your intuition has never had a chance to test itself against truth in such situations and therefore goes on supporting your self-confidence.

Here is one of my favorites: almost everyone's intuition tells them that if they want to remember a telephone number permanently, they should repeat it over and over to themselves right after they have heard it. This method will keep the memory fresh as long as they keep repeating it, but it has virtually no benefit for remembering it half an hour or more later (Landauer and Bjork 1978).

The point is that the same complexities and uncertainties that plague the science of human interaction with computers plague our intuitions about such matters. The analogy to architectural design is both apt and inappropriate—apt because architects rarely do the kind of upfront task analysis that their products deserve. (Thus the almost universal provision of the same number of toilet facilities for men and women in public places, despite the painful differences in queue length at every public event.) Like current software designers, architects have very poorly developed mechanisms for feedback of usability results into their designs. The sense in which the analogy is inappropriate is that architects are largely concerned with aesthetics. People, but especially architects, care a great deal about the beauty of their surroundings and are manifestly willing to trade some comfort and convenience for eye appeal. Although eye appeal

in computer screens is certainly something users like, it seems unlikely that they would knowingly trade much usability or usefulness for better looks in their work tools.

This overview of the available technology for doing user-centered design has demonstrated that the quiver is far from empty and provides almost enough instruction to get started. For readers who are interested in more how-to, the chapter by John Gould in the *Handbook of Human Computer Interaction,* edited by Martin Helander (1988), followed by the recent book *Usability Engineering,* by Jakob Nielsen (Nielsen 1993b), will take you a long way down the path to being as much of an expert as book learning alone can promise.

13

User-Centered Development

"Who the hell thought of this? It doesn't make any sense. Nobody has talked to any users." Supervisor of an order department after installation of a new system.

Usability and Development

After design comes production. Right? Wrong. Neither ideally nor in practice does it work that way for software. With automobiles and television sets, a good design is just the starting place. The critical factor is devising an economical manufacturing process to spew out millions of near-identical copies. With software, the copying part is trivial; the critical step is completing the first model. Doing that corresponds roughly to the combined design and development stages in automobile manufacturing. Ideally, since it is impossible to get usability right without iterative prototyping and testing, software design must evolve throughout development. In practice too, design evolves throughout development but for different reasons. As a system is implemented, the original plans frequently turn out to be too hard to execute; its parts get in each other's way. Often usability-related aspects—what screens will look like, how error messages will be worded, what action options will be in which menus—are not specified in advance. Usually programmers get new ideas as they work. This means that opportunities for UCE continue to abound. It quickly gets harder to change what the system does, its basic functionality, as implementation progresses. However, if the architecture—the overall organization of the software's components and how