Chapter 4

The Usability Engineering Lifecycle

Usability engineering is not a one-shot affair where the user interface is fixed up before the release of a product. Rather, usability engineering is a set of activities that ideally take place throughout the lifecycle of the product, with significant activities happening at the early stages before the user interface has even been designed. The need to have multiple usability engineering stages supplement each other was recognized early in the field, though not always followed on development projects [Gould and Lewis 1985].

Usability cannot be seen in isolation from the broader corporate product development context where one-shot projects are fairly rare. Indeed, usability applies to the development of entire product families and extended projects where products are released in several versions over time. In fact, this broader context only strengthens the arguments for allocating substantial usability engineering resources as early as possible, since design decisions made for any given product have ripple effects due to the need for subsequent products and versions to be backward compatible. Consequently, some usability engineering specialists [Grudin *et al.* 1987] believe that "human factors involvement with a particular product may ultimately have its greatest impact on future product releases." Planning for future versions is also a prime reason to follow up the release of a product with field studies of its actual use.

1. Know the user
    a. Individual user characteristics
    b. The user's current and desired tasks
    c. Functional analysis
    d. The evolution of the user and the job
2. Competitive analysis
3. Setting usability goals
    a. Financial impact analysis
4. Parallel design
5. Participatory design
6. Coordinated design of the total interface
7. Apply guidelines and heuristic analysis
8. Prototyping
9. Empirical testing
10. Iterative design
    a. Capture design rationale
11. Collect feedback from field use

Table 7 *The stages of the usability engineering lifecycle model.*

For a company that sells software or other products on the open market, the usability of each product will contribute to the company's general reputation as a quality supplier, and just a single product with poor usability can cause severe damage to the sales of the entire product family.

Table 7 shows a summary of the lifecycle stages discussed in this chapter. It is important to note that a usability engineering effort can still be successful even if it does not include every possible refinement at all of the stages. Section 4.13, *Prioritizing Usability Activities*, on page 112 contains a discussion of how to choose usability methods under varying levels of resource constraints.

The lifecycle model emphasizes that one should not rush straight into design. The least expensive way for usability activities to influence a product is to do as much as possible before design is started, since it will then not be necessary to change the design to comply with the usability recommendations. Also, usability work done before the system is designed may make it possible to avoid developing unnecessary features. Several of the pre-design usability activities might be considered part of a market research or product planning process as well, and may sometimes be performed by

marketing groups. However, traditional market research does not usually employ all the methods needed to properly inform usability design, and the results are often poorly communicated to developers. But there should be no need for duplicate efforts if management successfully integrates usability and marketing activities [Wichansky *et al.* 1988]. One outcome of such integration could be the consideration of product usability attributes as features to be used by marketing to differentiate the product. Also, marketing efforts based on usability studies can sell the product on the basis of its benefits as perceived by users (*what* it can do that they want) rather than its features as perceived by developers (*how* does it do it).

4.1 Know the User

The first step in the usability process is to study the intended users and use of the product. At a minimum, developers should visit a customer site so that they have a feel for how the product will be used. Individual user characteristics and variability in tasks are the two factors with the largest impact on usability, so they need to be studied carefully. When considering users, one should keep in mind that they often include installers, maintainers, system administrators, and other support staff in addition to the people who sit at the keyboard. The concept of "user" should be defined to include everybody whose work is affected by the product in some way, including the users of the system's end product or output even if they never see a single screen.

Even though "know the user" is the most basic of all usability guidelines, it is often difficult for developers to get access to users. Grudin [1990b, 1991a and b] analyzes the obstacles to such access, including

- The need for the development company to protect its developers from being known to customers, since customers may bypass established technical support organizations and call developers directly, sidetracking them from their main job.
- The reluctance of sales representatives to let anybody else from the company talk to "their" customers, fearing that the devel-

opers or usability people may offend the customer or create dissatisfaction with the current generation of products.

- User organizations only making users available for a short time, either because they are highly paid executives or because they are unionized and dislike being studied.

All these issues are real and need to be addressed when trying to get to "know the user." No universal solutions are available, except to recommend an explicit effort to get direct access to representative users and not be satisfied with indirect access and hearsay. It is amazing how much time is wasted on certain development projects by arguing over what users *might* be like or what they *may* want to do. Instead of discussing such issues in a vacuum, it is much better (and actually less time-consuming) to get hard facts from the users themselves.

Individual User Characteristics

It is necessary to know the class of people who will be using the system. In some situations this is easy since it is possible to identify these users as concrete individuals. This is the case when the product is going to be used in a specific department in a particular company. For other products, users may be more widely scattered such that it is possible to visit only a few, representative customers. Alternatively, the products might be aimed toward the entire population or a very large subset.

By knowing the users' work experience, educational level, age, previous computer experience, and so on, it is possible to anticipate their learning difficulties to some extent and to better set appropriate limits for the complexity of the user interface. Certainly one also needs to know the reading and language skills of the users. For example, very young children have no reading ability, so an entirely nontextual interface is required. Also, one needs to know the amount of time users will have available for learning and whether they will have the opportunity for attending training courses: The interface must be made much simpler if users are expected to use it with minimum training.

The users' work environment and social context also need to be known. As a simple example, the use of audible alarms, "beeps," or more elaborate sound effects may not be appropriate for users in open office environments. In a field interview I once did, a secretary complained strongly that she wanted the ability to shut off the beep because she did not want others to think that she was stupid because her computer beeped at her all the time.

A great deal of the information needed to characterize individual users may come from market analysis or from the observational studies one may conduct as part of the task analysis. One may also collect such information directly through questionnaires or interviews. In any case, it is best not to rely totally on written information since new insights are almost always achieved by observing and talking to actual users in their own working environment.

Task Analysis

A task analysis [Diaper 1989a; Fath and Bias 1992; Johnson 1992] is essential as early input to system design. The users' overall goals should be studied as well as how they currently approach the task, their information needs, and how they deal with exceptional circumstances or emergencies. For example, systematic observation of users talking to their clients may reveal input and output needs for a transactions-processing system. Sometimes, interviewing or observing the users' clients or others who interact with them can provide additional task analysis insights [Garber and Grunes 1992].

The users' model of the task should also be identified, since it can be used as a source for metaphors for the user interface (see page 126). Also, seek out and observe especially effective users and user strategies and "workarounds" as hints of what a new system could support. Such "lead users" are often a major source of innovations [von Hippel 1988]. Finally, one should identify the weaknesses of the current situation: points where users fail to achieve goals, spend excessive time, or are made uncomfortable. These weaknesses present opportunities for improvements in the new product.

A typical outcome of a task analysis is a list of all the things users want to accomplish with the system (the goals), all the information

they will need to achieve these goals (the preconditions), the steps that need to be performed and the interdependencies between these steps, all the various outcomes and reports that need to be produced, the criteria used to determine the quality and acceptability of these results, and finally the communication needs of the users as they exchange information with others while performing the task or preparing to do so.

When interviewing users for the purpose of collecting task information, it is always a good idea to ask them to show concrete examples of their work products rather than keeping the discussion on an abstract level. Also, it is preferable to supplement such interviews with observations of some users working on real problems, since users will often rationalize their actions or forget about important details or exceptions when they are interviewed.

Often, a task analysis can be decomposed in a hierarchical fashion [Greif 1991], starting with the larger tasks and goals of the organization and breaking each of them down into smaller subtasks, that can again be further subdivided. Typically, each time a user says, "then I do *this*," an interviewer could ask two questions: "*Why* do you do it?" (to relate the activity to larger goals) and "*How* do you do it?" (to decompose the activity into subtasks that can be further studied). Other good questions to ask include, "why do you *not* do this in such and such a manner?" (mentioning some alternative approach), "Do errors ever occur when doing this?," and "How do you discover and correct these errors?" [Nielsen *et al.* 1986].

Finally, users should be asked to describe exceptions from their normal work flow. Even though users cannot be expected to remember *all* the exceptions that have ever occurred, and even though it will be impossible to predict all the future exceptions, there is considerable value to having a list indicating the *range* of exceptions that must be accommodated. Users should also be asked for remarkable instances of notable successes and failures, problems, what they liked best and least, what changes they would like, what ideas they have for improvements, and what currently annoys them. Even though not all such suggestions may be followed in the final design, they are a rich source of inspiration.

Functional Analysis

A new computer system should not be designed simply to propagate suboptimal ways of doing things that may have been instituted because of limitations in previous technologies. Therefore, one should not analyze just the way users currently do the task, but also the underlying functional reason for the task: What is it that really needs to be done, and what are merely surface procedures which can, and perhaps should, be changed [Schmidt 1988].

For example, many projects in the *computer-supported cooperative work* (CSCW) field assume that face-to-face interaction is the ultimate in communication and that computers should emulate *physically proximate reality* (PPR) as closely as possible. In contrast, the "beyond being there" approach [Hollan and Stornetta 1992; Brothers *et al.* 1992] separates the needs of human communication from the media through which communication has been achieved so far. Computerized communication tools might be built to take advantage of the strengths of the computer medium, such as asynchronism, anonymity, searchable archives, and automated replies and filters, even if the resulting communication mechanisms do not resemble the way people talk when they are in the same room.

As a more mundane example, initial observations of people reading printed manuals could show them frequently turning pages to move through the document. A naive design of online documentation might take this observation to imply really good and fast paging or scrolling mechanisms. A functional analysis would show that manual users really turn pages this much to find specific information, but they have a hard time locating the correct page. Based on this analysis, one could design an online documentation interface that first allowed users to specify their search needs, then used an outline of the document to show locations with high search scores, and finally allowed users to jump directly to these locations, highlighting their search terms to make it easier to judge the relevance of the information [Egan *et al.* 1989].

Of course, there is a limit to how drastically one can change the way users currently approach their task, so the functional analysis should be coordinated with a task analysis.

The Evolution of the User

Users will not stay the same. Using the system changes the users, and as they change they will use the system in new ways. Carroll and Rosson [1991] refer to this dialectic phenomenon as the "coevolution of tasks and artifacts." For example, spreadsheets were initially invented as aids for calculation, but having such a malleable computerized medium available encouraged users to integrate noncalculation data in a spreadsheet. Users have often been known to use spreadsheets for databases [Nielsen *et al.* 1986], and these and other uses have led spreadsheet vendors to include noncalculation features in later versions.

It is impossible to forecast these changes completely as users will always discover new uses for computer systems after some period of use, but a flexible design will stand a better chance of supporting these new uses. Try to make an educated guess based on your knowledge about how other users have changed in the past. One way of getting such knowledge is through the post-deployment field studies discussed on page 109.

A typical change is that users become experts after some time and want interaction shortcuts (sometimes called accelerators). For example, a business graphics package might lead novice users through a series of question–answer screens to specify the main characteristics of the main types of charts, but expert users will probably want to be able to change the charts by direct manipulation and maybe even to be given access to a kind of specialized programming language for the construction of graphics. It is important not to design just for the way users will use the system in the first short period after its release.

4.2 *Competitive Analysis*

As discussed in Section 4.8, prototyping is an important part of the usability process, and existing, perhaps competing, products are often the best prototypes we can get of our own product [Byrne 1989]. It is desirable to analyze existing products heuristically

according to established usability guidelines and to perform empirical user tests with these products. A competing product is already fully implemented and can therefore be tested very easily [Bachman 1989]. Also, the developers of the existing systems often have put a reasonable amount of effort into their development process so that the competing products may work fairly well. This again means that user testing with existing products can be more realistic than a test of other prototypes. Users can perform real tasks on the competing system, making it possible to learn how well its functionality and interaction techniques support the kinds of tasks the planned new product is expected to support based on the initial analysis of the intended users.

If several competing products are available for analysis, one can furthermore perform a comparative analysis of their differing approaches to the various user interface design issues for the kind of product being studied. This will provide ideas for the new design and will give a list of ad hoc guidelines for approaches that seem to work and those that should be avoided. Also, reading trade press reviews can provide some insights into the usability characteristics and different approaches of a large number of competing products. Such reviews should be complemented with more thorough and principled analysis and testing of a smaller number of important products. Sometimes, competitive analysis will involve the study of non-computer interfaces. For example, an electronic reference book project should first studying how people use traditional printed encyclopedia [Marchionini 1989].

Note that a competitive analysis does not imply stealing other people's copyrighted user interface designs. One would hope to be able to do better than the previous designs as a result of the analyses of their strengths and weaknesses.

4.3 *Goal Setting*

As discussed in Chapter 2, usability is not a one-dimensional attribute of a system. Usability comprises several components that can sometimes conflict. Normally, not all usability aspects can be

given equal weight in a given design project, so you will have to make your priorities clear on the basis of your analysis of the users and their tasks. For example, learnability would be especially important if new employees were constantly being brought in on a temporary basis, and the ability of infrequent users to return to the system would be especially important for a reconfiguration utility that was used once every three or four months.

As also discussed in Chapter 2, the different usability parameters can be operationalized and expressed in measurable ways. Before starting the design of a new interface, it is important to discuss the usability metrics of interest to the project and to specify the goals of the user interface in terms of measured usability [Chapanis and Budurka 1990]. One may not always have the resources available to collect statistically reliable measures of the usability metrics specified as goals, but it is still better to have some idea of the level of usability to be strived for.

For each usability attribute of interest, several different levels of performance can be specified as part of a goal-setting process [Whiteside *et al.* 1988]. One would at least specify the minimum level which would be acceptable for release of the product, but a more detailed goal specification can also include the planned level one is aiming for as well as the current level of performance. Additionally, it can help to list the current value of the usability attribute as measured for existing or competing interfaces, and one can also list the theoretically best possible value, even though this value will typically not be attained. Figure 7 shows one possible notation, called a *usability goal line*, for representing the range of specification levels for one usability goal.

In the example in Figure 7, the number of user errors per hour is counted. When using the current system, users make an average of 4.5 errors per hour, and the planned number of user errors is 2.0 per hour. Furthermore, the theoretical optimum is obviously to have no errors at all. If the new interface is measured at anything between 1.0 and 3.0 user errors per hour, it will be considered on target with respect to this usability goal. A performance in the interval of 3–5 would be a danger signal that the usability goal was

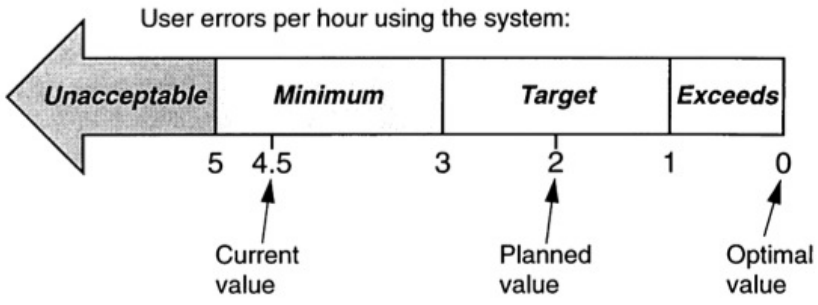

Figure 7 *An example of a usability goal line in a notation similar to that used by Rideout [1991].*

not met, even though the new interface could still be released on a temporary basis since a minimal level of usability had been achieved. It would then be necessary to develop a plan to reduce user errors in future releases. Finally, more than 5.0 user errors per hour would make this particular product sufficiently unusable to make a release unacceptable.

Usability goals are reasonably easy to set for new versions of existing systems or for systems that have a clearly defined competitor on the market. The minimum acceptable usability would normally be equal to the current usability level, and the target usability could be derived as an improvement that was sufficiently large to induce users to change systems. For completely new systems without any competition, usability goals are much harder to set. One approach is to define a set of sample tasks and ask several usability specialists how long it "ought" to take users to perform them. One can also get an idea of the minimum acceptable level by asking the users, but unfortunately users are notoriously fickle in this respect; countless projects have failed because developers believed users' claims about what they wanted, only to find that the resulting product was not satisfactory in real use.

Financial Impact Analysis

At about the same time as usability goals are being specified, it is a good idea to make an analysis of the financial impact of the

usability of the system. Such an analysis involves estimating the number of users who will be using the system, their loaded salaries or other costs, and the approximate time they will be using the system. The cost of the users' time is not just their salary but also other costs, such as various pensions and benefits, employment taxes or fees charged by the government, and general overhead costs like the rent of office space. This total cost will be referred to as the *loaded salary* or *loaded cost* of a user.

Financial impact analyses are easiest to make for software that is being developed in-house or under contract directly from the user organization, as the savings are readily available as true bottom-line benefits. For example, consider the development of a software system for a group of 3,000 specialized staff processing some kind of service orders. If the loaded cost of the technicians is assumed to be $25 per hour, and the technicians can be assumed to be using the system about a third of their working day, we immediately[1] find that the annual financial impact of the user interface is approximately $47,000,000. Furthermore, let us assume that the system is planned for introduction in two years and that it will then be used for four years until it is replaced by a new system or a major redesign. Under these assumptions, the annual impact translates into a total financial impact of $129,000,000, when considering the time value of money and deflating the impact of money spent in future years by 10% per year.[2]

Usability activities that might improve the learnability of the user interface sufficiently to cut down learning-time by one day would be worth $600,000, corresponding to a present value of $500,000. Similarly, usability improvements leading to a 10% increase in user

1. By multiplying by 8 hours per day and 236 working days per year.

2. The deflator should be derived from the expected real return from alternative investments. A 10% deflator for the real value of money corresponds to an investment return of 13% per year (the stock market average) minus an inflation rate of 3%. A more elaborate financial model might use a deflator that has not been adjusted for inflation and then increase the annual estimates of various cost categories by their expected rate of increase. This latter approach is more accurate if the increase in salaries and overhead is expected to differ significantly from the inflation rate.

productivity would be worth $4,700,000 per year, or $12,900,000 over four years (again deflating the value of money saved in later years).[3] We would normally find it worthwhile to invest a reasonable amount in usability work in this kind of development project. Also, this calculation makes it clear that productivity improvements would be worth more than learning time savings for this product, assuming that the one day and 10%, respectively, are approximately the magnitude of improvement that can be expected.

In the case of software being developed for sale on the open market, user savings are not directly available as profits for the development organization. Therefore, the financial impact analysis should have two components: an estimate of the impact on the development organization (to help determine the magnitude of the usability budget) and an estimate of the impact on the user organizations (to help prioritize the focus of the available usability resources). Analyses of the financial impact of usability on the development organization should include estimates of revenue loss or enhancements as well as cost estimates like the expense of servicing calls to customer support lines. Unfortunately, specific data about these two aspects is usually considered highly secret proprietary information, since user interfaces now constitute a major aspect of a company's competitive advantage.

Anecdotal evidence indicates that some vendors have found that a product with a usability level below a certain point is simply not worth selling, since one can predict that it will fail in the market. Alternatively, customers may buy the product, but they will then make such excessive demands on the vendor's technical support staff that each sale ends up losing money. One example was an upgrade to a spreadsheet, where the installed base of customers

3. Calculations of present value for this example assume that the savings in training costs are realized on the first day after the introduction of the system (that is, 2 years from the present). Productivity savings are realized throughout the year but for simplicity's sake, they are calculated as occurring on a single day half-way through the year. Thus, for example, savings in the system's first year of use are deflated by the compound value of 10% over 2.5 years = 27%.

guaranteed the "success" (in terms of sheer number of sales) of the upgrade. The installation program supplied with the upgrade had such a horrible user interface that the customers needed on average two 20-minute calls to the vendor's toll-free support line before they had succeeded in installing their upgrade. Given that it costs about $20 to service a more typical, 5-minute support call, the installation user interface (which could probably have been fixed with a minimum of usability engineering effort) ended up costing the vendor more than the $70 per user they made selling the upgrade.[4] In general, the need to save on customer support is a driving force for usability engineering in many companies. The median loaded cost of servicing a customer support call was $23.33 according to a 1993 survey of 148 software vendors in the industry newsletter *Soft Letter.*

As an example analysis of the financial impact of a user interface on the customers, assume that you were developing a word processor that is expected to sell one million copies. About half of the users are expected to be secretaries who will be using the word processor about half of their working day, and the other half of the users are expected to be business professionals who will be using the word processor about 10% of their working day. Furthermore, assume that the loaded cost of a secretary is $20 per hour and that the loaded cost of a business professional is $100 per hour. This means that the amount of money spent by users while using the word processor is about $19,000,000,000 annually (calculated at 8 hours per day and 236 working days per year). This amount is an indication of the potential value being influenced by the usability engineer in charge of the word processor's user interface, even though it will never show up on the development organization's budget.

Assume that we are considering the potential benefits from improving the efficiency of the word processor's editing features

4. This example also indicates the necessity of paying attention to the total user interface in the usability engineering lifecycle. The spreadsheet itself might have had perfect usability, but the install utility ended up destroying the product.

by 5%. To calculate the savings from such an improvement, we furthermore need to estimate the proportion of the users' time spent editing as opposed to just entering text. Such data should preferably be gathered from field studies or by logging data from instrumented copies of installed versions of previous systems. For the sake of argument, we will assume that 10% of the secretaries' word processor use is editing and that the corresponding proportion for the business professional users is 25%. This means that total annual value of the time spent editing by the users of the word processor is $3,300,000,000, and that the value[5] of a 5% savings would be $165,000,000. Of course, the vendor of the word processor package will not get this money, but there is still *some* value to having the users save $165,000,000, and the usability work that could bring about such savings would be worthy of a larger part of the budget than work on some other feature that might save users no more than a few million dollars.

Much of the information needed for the financial impact analysis should be available from the marketing department. Specifically, they should have data about the current or projected number of users in different markets and perhaps be able to provide estimates of the users' salary levels.

4.4 *Parallel Design*

It is often a good idea to start the design with a parallel design process, in which several different designers work out preliminary designs [Nielsen *et al.* 1993, 1994]. The goal of parallel design is to explore different design alternatives before one settles on a single approach that can then be developed in further detail and subjected to more detailed usability activities. Figure 8 is a conceptual chart of the relation between parallel and iterative design.

5. In principle, the value of saved time is not the average cost of the employees' time, but the *marginal* value of their time, necessitating the use of a so-called hedonic wage model [Sassone 1987], but for practical purposes one can use average values for the type of rough estimate we are making here.

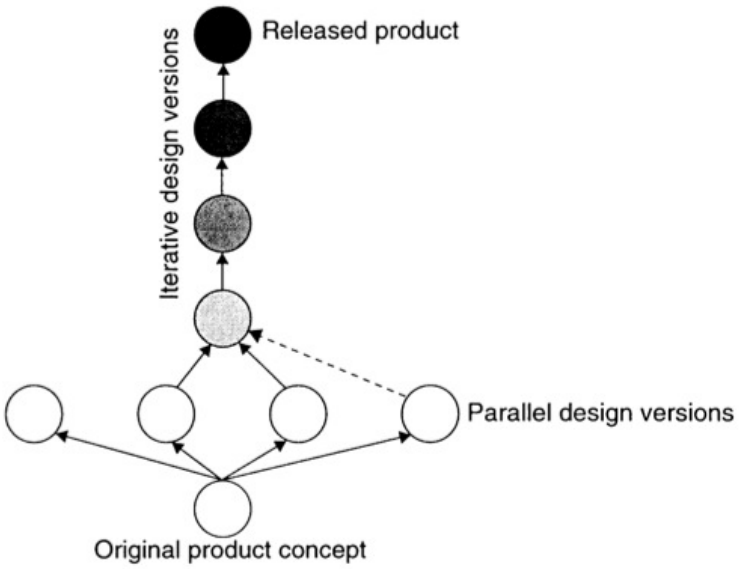

Figure 8 *Conceptual illustration of the relation between parallel and iterative design. Normally, the first prototype would be based on ideas from several of the parallel design sketches.*

Typically, one can have three or four designers involved in parallel design. For critical products, some large computer companies have been known to devote entire teams to developing multiple alternative designs almost to the final product stage, before upper management decided on which version to release. In general, though, it may not be necessary for the designers to spend more than a few hours or at the most one or two days on developing their initial designs. Also, it is normally better to have designers work individually rather than in larger teams, since parallel design only aims at generating rough drafts of the basic design ideas.

In parallel design, it is important to have the designers (or the design teams) work independently, since the goal is to generate as much diversity as possible. Therefore, the designers should not discuss their designs with each other until after they have produced their draft interface designs.

When the designers have completed the draft designs, one will often find that they have approached the problem in at least two drastically different ways that would give rise to fundamentally different user interface models. Even those designers who are basing their designs on the same basic approach almost always have different details in their designs. Usually, it is possible to generate new combined designs after having compared the set of initial designs, taking advantage of the best ideas from each design. If several fundamentally different designs are available, it is best to pursue each of the main lines of design a little further in order to arrive at a small number of prototypes that can be subjected to usability evaluation before the final approach is chosen.

A variant of parallel design is called *diversified parallel design* and is based on asking the different designers to concentrate on different aspects of the design problem. For example, one designer could design an interface that was optimized for novice users, at the same time as another designer designed an interface optimized for expert users and a third designer explored the possibilities of producing en entirely nonverbal interface. By explicitly directing the design approach of each designer, diversified parallel design drives each of these approaches to their limit, leading to design ideas that might never have emerged in a unified design. Of course, some of these diversified design ideas may have to be modified to work in a single, integrated design.

It is especially important to employ parallel design for novel systems where little guidance is available for what interface approaches work the best. For more traditional systems, where competitive products are available, the competitive analysis discussed in Section 4.2 can serve as initial parallel designs, but it might still be advantageous to have a few designers create additional parallel designs to explore further possibilities.

The parallel design method might at first seem to run counter to the principle of cost-effective usability engineering, since most of the design ideas will have to be thrown away without even being implemented. In reality, though, parallel design is a very cheap way of exploring the design space, exactly *because* most of the ideas

will not need to be implemented, the way they might be if some of them were not tried until later as part of iterative design. The main financial benefit of parallel design is its parallel nature, which allows several design approaches to be explored at the same time, thus compressing the development schedule for the product and bringing it to market more rapidly. Studies have shown that about a third of the profits are lost when products ship as little as half a year late [House and Price 1991], so anything that can speed up the development process should be worth the small additional cost of designing in parallel rather than in sequence.

4.5 *Participatory Design*

Even though the advice to "know the user" may have been followed before the start of the design phase, one still cannot know the user sufficiently well to answer all issues that come up in doing the design [Kensing and Munk-Madsen 1993]. Instead of guessing, designers should have access to a pool of representative users after the start of the design phase. It is important to have access to the people who will actually be using the system, and not just to their managers or union representatives. Even well-intentioned managers will often not know the exact issues facing users in their everyday work, and they will normally have different characteristics from the real users in many ways. Elected union leaders may not be typical workers either, and they may also have spent too much time in administration.

Users often raise questions that the development team has not even dreamed of asking. This is especially true with respect to potential mismatches between the users' actual task and the developers' model of the task. Therefore, users should be involved in the design process through regular meetings between designers and users. Users participating in a system design process are sometimes referred to as *subject matter experts*, or SMEs.

Users are not designers, so it is not reasonable to expect them to come up with design ideas from scratch. However, they are very good at reacting to concrete designs they do not like or that will not

work in practice. To get full benefits from user involvement, it is necessary to present these suggested system designs in a form the users can understand. Instead of voluminous system specifications, concrete and visible designs, preferably in the form of prototypes, should be employed for this purpose. In early stages of the design where functional prototypes are not yet available, paper mock-ups or simply a few screen designs can be used to prompt user discussion.[6] Even simple, guided discussion can elicit ideas from users.

It is important to realize that participatory design should not just consist of asking users what they want, since users often do not know what they want or what they need, or even what the possibilities are. For example, in one study users were first asked to rate the usefulness of some new features of an editor on the basis of a description of the features and then asked the same question after they had tried out the actual features [Root and Draper 1983]. It turned out that the correlation between the users' ratings before and after actual experience with the features was as low as 0.28, indicating essentially no relation between the two sets of ratings. See also the discussion of the usability slogan, *The User Is Not Always Right*, on page 11.

For larger development projects, thought should be given to periodically refreshing the pool of users who participate in the project since they risk becoming less representative of the average user population as their involvement with system development grows. A user representative who has been to too many design meetings will be steeped in the developers' way of thought and will understand the proposed system structure and possibly have a tendency to accept the rationale for awkward design elements. Fresh users who are brought in later in the project are more likely to question such potential problems since they will not know the history of the design. Furthermore, of course, users are different, so it is dangerous to rely too much on information from a small set of users

6. I once gave a presentation to a group of users about a proposed user interface using (then) new terminal technology. They listened politely and did not say anything, until the time when I put a screen dump on the overhead—after which the audience erupted with questions and comments [Nielsen 1987a].

that never changes. On the other hand, there are trade-offs involved in changing user representatives, since one also does not want to spend time explaining the project to new people, so such changes should not be made more than a few times during a project.

4.6 *Coordinating the Total Interface*

Consistency is one of the most important usability characteristics (see Section 5.4 on page 132). Consistency should apply across the different media which form the total user interface, including not just the application screens but also the documentation, the online help system, and any online or videotaped tutorials [Perlman 1989] as well as traditional training classes. For example, in one case studied by Poltrock [1994], training materials described an obsolete way of using an interface because the training department had not been informed about the introduction of a redesigned, and presumably better, interface.

Consistency is not just measured at a single point in time but should apply over successive releases of a product so that new releases are consistent with their predecessors. Also, since very few companies produce only a single product, efforts should be made to promote consistency across entire product families. Corporate user interface standards are one common way of promoting that goal. In spite of the general desirability of consistency, it is obviously not the only desirable usability characteristic, and consistency may sometimes conflict with other interface desiderata [Grudin 1989]. It is necessary to maintain some flexibility so that bad design is not forced upon users for the sake of consistency alone.

To achieve consistency of the total interface it is necessary to have some centralized authority for each development project to coordinate the various aspects of the interface. Typically this coordination can be done by a single person, but on very large projects or to achieve corporate-wide consistency, a committee structure may be more appropriate. Also, interface standards (discussed further in Chapter 8) are an important approach to achieving consistency. In addition to such general standards, a project can develop its own

ad hoc standard with elements like a dictionary of the appropriate terminology to be used in all screen designs as well as in the other parts of the total interface.

In addition to formal coordination activities, it is helpful to have a shared culture in the development groups with common understanding of what the user interface should be like. Many aspects of user interface design (especially the dynamics) are hard to specify in written documents but can be fairly easily understood from looking at existing products following a given interface style. Actually, prototyping also helps achieve consistency, since the prototype is an early statement of the kind of interface toward which the project is aiming. Having an explicit instance of parts of the design makes the details of the design more salient for developers and encourages them to follow similar principles in subsequent design activities [Bellantone and Lanzetta 1991].

Furthermore, consistency can be increased through technological means such as code sharing or a constraining development environment. When several products use the same code for parts of their user interface, then those parts of the interfaces automatically will be consistent. Even if identical code cannot be used, it is possible to constrain developers by providing development tools and libraries that encourage user interface consistency by making it easiest to implement interfaces that follow given guidelines [Tognazzini 1989; Wiecha *et al.* 1989].

4.7 *Guidelines and Heuristic Evaluation*

Guidelines list well-known principles for user interface design which should be followed in the development project. In any given project, several different levels of guidelines should be used: *general guidelines* applicable to all user interfaces, *category-specific guidelines* for the kind of system being developed (e.g., guidelines for window-based administrative data processing or for voice interfaces accessed through telephone keypads), and *product-*

specific guidelines for the individual product. All these guidelines can be used as background for heuristic evaluation as discussed in Section 5.11 on page 155.

For example, a general guideline could be to "provide feedback" to the user about the system's state and actions. This general advice could be made more specific in a category-specific guideline for graphical user interfaces: Ensure that the main objects of interest to the user are visible on the screen and that their most important attributes are shown. Finally, this guideline could be further developed into a product-specific guideline for the design of a graphical file system: Have each file and subdirectory represented by an icon and use different icon shapes to represent different classes of objects (data files, executable files, and subdirectories). It would then be possible to check that each aspect of the file system complied with this latter rule.

As another example, the same general guideline, "provide feedback," could be applied to hypermedia navigation to recommend that users be informed about the transition that takes place when they move from one node to another. Experience with existing products or reading of the research literature [Merwin *et al.* 1990] could then lead to a further ad hoc guideline for a particular hypertext document stating that an animated visual effect should be used to signify navigational transitions rather than having an instantaneous change to the destination screen.

The difference between standards and guidelines is that a standard specifies how the interface should appear to the user, whereas a set of guidelines provides advice about the usability characteristics of the interface. Standards are discussed further in Chapter 8 and have interface consistency as one of their major objectives. Hopefully a given standard will follow most of the traditional usability guidelines so that the interfaces designed according to the standard will also be as usable as possible. For example, a guideline may state that users should always be able to back out from any undesired system state. One standard might instantiate that general guideline by specifying that an undo command should always be available and that it should be shown as an icon at the top right of

the screen. Another standard might follow the same guideline by returning to the previous system state whenever the user hits the escape key.

Several very extensive collections of general user interface guidelines exist, including

- [Brown 1988] with 302 guidelines
- [Marshall *et al.* 1987] with 162 guidelines
- [Mayhew 1992] with 288 guidelines
- [Smith and Mosier 1986] with 944 guidelines

It is thus normally possible to rely on the international user interface community for general guidelines, whether expressed individually in research papers or collected in larger guidelines reports. Chapter 5 in this book provides a short list of the most important general guidelines. Some category-specific guidelines can also be found in the research literature, but they are also often a product of corporate memory, to the extent that lessons from previous projects are generalized and made available to future projects. Finally, product-specific guidelines are often developed as part of individual projects as project members gain a better understanding of the special usability aspects of their system. Such understanding can be gathered early on through competitive analysis as discussed on page 78, and additional insights typically come from user testing of prototypes of the new system.

4.8 *Prototyping*

One should not start full-scale implementation efforts based on early user interface designs. Instead, early usability evaluation can be based on prototypes of the final systems that can be developed much faster and much more cheaply, and which can thus be changed many times until a better understanding of the user interface design has been achieved.

In traditional models of software engineering most of the development time is devoted to the refinement of various intermediate

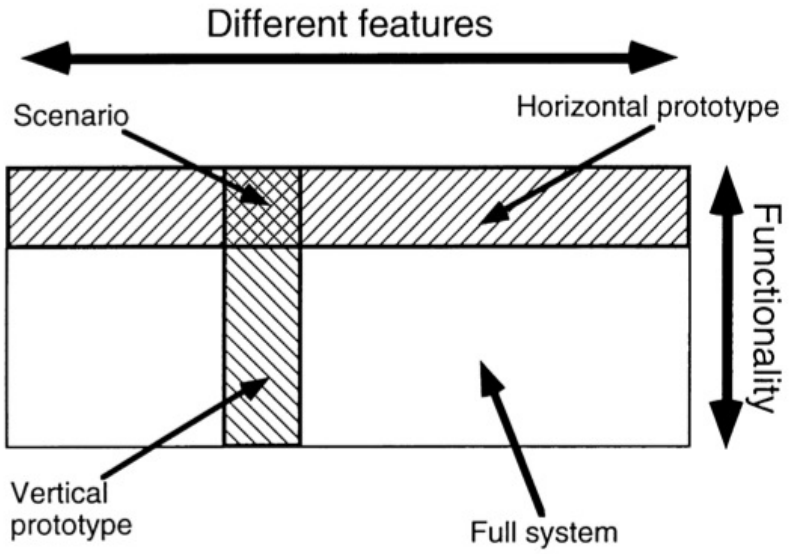

Figure 9 *The two dimensions of prototyping: Horizontal prototyping keeps the features but eliminates depth of functionality, and vertical prototyping gives full functionality for a few features.*

work products, and executable programs are produced at the last possible moment. A problem with this "waterfall" approach is that there will then be no user interface to test with real users until this last possible moment, since the "intermediate work products" do not explicitly separate out the user interface in a prototype with which users can interact. Experience also shows that it is not possible to involve the users in the design process by showing them abstract specifications documents, since they will not understand them nearly as well as concrete prototypes.

The entire idea behind prototyping is to save on the time and cost to develop something that can be tested with real users. These savings can only be achieved by somehow reducing the prototype compared with the full system: either cutting down on the number of features in the prototype or reducing the level of functionality of the features such that they *seem* to work but do not actually *do* anything. These two dimensions are illustrated in Figure 9.

Cutting down on the number of features is called *vertical prototyping* since the result is a narrow system that does include in-depth functionality, but only for a few selected features. A vertical prototype can thus only test a limited part of the full system, but it will be tested in depth under realistic circumstances with real user tasks. For example, for a test of a videotex system, in-depth functionality would mean that a user would actually access a database with some real data from the information providers.

Reducing the level of functionality is called *horizontal prototyping* since the result is a surface layer that includes the entire user interface to a full-featured system but with no underlying functionality. A horizontal prototype is a simulation [Life *et al.* 1990] of the interface where no real work can be performed. In the videotex example, this would mean that users should be able to execute all navigation and search commands but without retrieving any real information as a result of these commands [Nielsen 1987a]. Horizontal prototyping makes it possible to test the entire user interface, even though the test is of course somewhat less realistic, since users cannot perform any real tasks on a system with no functionality. The main advantages of horizontal prototypes are that they can often be implemented fast with the use of various prototyping and screen design tools and that they can be used to assess how well the entire interface "hangs together" and feels as a whole.

Finally, one can reduce both the number of features and the level of functionality to arrive at a scenario that is only able to simulate the user interface as long as the test user follows a previously planned path. Scenarios are extremely easy and cheap to build, while at the same time not being particularly realistic. Scenarios are discussed further on page 99.

In addition to reducing the proportion of the system that is implemented, prototypes can be produced faster by:

- Placing less emphasis on the efficiency of the implementation. For example, it will not matter how much disk space the prototype uses since it will only be used for a short time. Similarly, test users may be able to cope with slow response times that would never be acceptable in the final product. Note, however, that

response times are an important aspect of usability and that test users may get very frustrated and make errors if the prototype is *too* slow. Of course, efficiency measures of the users' performance will be invalid if the prototype slows them down too much, so inefficient prototypes are better suited for early evaluation of interface concepts than for measurement studies.

- Accepting less reliable or poorer quality code. Even though bugs and crashes do distract users during testing, they can often be compensated for by the experimenter.
- Using simplified algorithms that cannot handle all the special cases (such as leap years) that normally require a disproportionately large programming effort to get right.
- Using a human expert operating behind the scenes to take over certain computer operations that would be too difficult to program. This approach is often referred to as the *Wizard of Oz technique* after the "pay no attention to that man behind the curtain" scene in this story. Basically, the user interacts normally with the computer, but the user's input is not relayed directly to the program. Instead, the input is transmitted to the "wizard" who, using another computer, transforms the user's input into an appropriate format. A famous early Wizard of Oz study was the "listening typewriter" [Gould *et al.* 1983] simulation of a speech recognition interface where the user's spoken input was typed into a word processor by a human typist located in another room.[7] When setting up a Wizard of Oz simulation, experience with previously implemented systems is helpful in order to place realistic bounds on the Wizard's "abilities" [Maulsby *et al.* 1993].
- Using a different computer system than the eventual target platform. Often, one will have a computer available that is faster or otherwise more advanced than the final system and which can therefore support more flexible prototyping tools and require less programming tricks to achieve the necessary response times.
- Using low-fidelity media [Virzi 1989] that are not as elaborate as the final interface but still represent the essential nature of the

7. Dye *et al.* [1990] survey additional simulations of listening typewriters.

interaction. For example, a prototype hypermedia system could use scanned still images instead of live video for illustrations.

- Using fake data and other content. For example, a prototype of a hypermedia system that will include heavy use of video could use existing video material, even though it did not exactly match the topic of the text, in order to get a feel for the interaction techniques needed to deal with live images. A similar technique is used in the advertising industry, where so-called ripomatics are used as rudimentary television commercials with existing shots from earlier commercials to demonstrate concepts to clients before they commit to pay for the shooting of new footage.
- Using paper mock-ups instead of a running computer system. Such mock-ups are usually based on printouts of screen designs, dialog boxes, pop-up menus, etc., that have been drawn up in some standard graphics or desktop publishing package. They are made into functioning prototypes by having a human "play computer" and find the next screen or dialog element from a big pile of paper whenever the user indicates some action. This human needs to be an expert in the way the program is intended to work since it is otherwise difficult to keep track of the state of the simulated computer system and find the appropriate piece of paper to respond to the user's stated input.

  Paper mock-ups have the further advantage that they can be shown to larger groups on overhead projectors [Rowley and Rhoades 1992] and used in conditions where computers may not be available, such as customer conference rooms. Portable computers with screen projection attachments confer some of the same advantages to computerized prototypes, but also increase the risk of something going wrong.
- Relying on a completely imaginary prototype where the experimenter describes a possible interface to the user orally, posing a series of "what if (the interface did this or that) . . ." questions as the user steps though an example task. This verbal prototyping technique has been called "forward scenario simulation" [Cordingley 1989] and is more akin to interviews or brainstorming than a true prototyping technique.

Obviously, several prototyping techniques can be combined either in one, especially cheap prototype, or as alternative prototypes,

each exploring one aspect of the usability of the total system. For example, one could create one prototype hypermedia interface with scanned still images of the actual topic material, and another prototype interface with "ripomatic" live video from an existing system on another topic. The still-image prototype could then be used to test the integration of text and images to support learning the domain of the hyperdocument, and the live-video prototype could be used to test interaction mechanisms for controlling the time-variant media, such as super–fast-forwarding as a way to scan a long video clip in a short amount of time. Of course, one would ultimately have to produce a single, integrated prototype with domain-specific live video to test the integration of text and video, but that more expensive version could be put off while the cheaper prototypes were used to clean up the interface and help decide the types of new video material one would need to film.

Special prototyping tools [Hix and Schulman 1991] and languages are a major means of fast implementation of user interface prototypes. In addition to specialized prototyping tools, fast prototyping is often achieved by the use of hypertext systems [Hartson and Smith 1991; Nielsen 1989a; Young *et al.* 1990], courseware authoring tools, database systems [Lee *et al.* 1990], so-called fourth-generation application generators, specialized screen generator tools, and the features some spreadsheets have for constructing general user interfaces as a front-end to an underlying spreadsheet.

Prototypes may sometimes be used for a special form of participatory design called *interactive prototyping*, where the prototype is developed and modified on the fly as a test user comments on its weak spots. If a crack programmer is available and a flexible interface construction system is used, such interactive prototyping can be a powerful experience for the users who get the immediate gratification of seeing their design suggestions implemented. Also, the design may proceed rapidly as multiple variations are tested and modified in a single test session.

Unfortunately, reality is often less ideal, since even a true wizard programmer will often make mistakes when hacking code in real time. Programming errors and system difficulties will sidetrack the

test session from the focus on the user's task and the interface, and the user may feel severely alienated by the many extra windows popping up for split-second editing by the programmer. These problems may be avoided by using paper mock-ups for interactive prototyping sessions and allow the users to modify the paper designs. One such technique is PICTIVE (Plastic Interface for Collaborative Technology Initiatives through Video Exploration) [Muller 1991, 1992] where designs are put together as multiple layers of sticky notes and overlays that can be changed by simple colored pens. A final PICTIVE design may be somewhat of a mess of loose paper and plastic, which is why the two last characters of the acronym imply using a videotape of the design session to convey the result to the implementers. PICTIVE is especially suited for prototyping activities carried out as part of a participatory design process since the low-tech nature of the materials make them equally accessible to users and to developers.

A prototype is a form of design specification and is often used as a major way of communicating the final design to developers. Unfortunately, the prototype can be *over*specified in some aspects that are not really intended to be part of the design. Whenever something is made concrete, there is a need to instantiate a multitude of representational details that might not have been explicitly designed by anybody. For example, a screen design will have to use certain colors and fonts, even though the designer's focus may have been on the wording and positioning of the dialogue elements. Basically, one needs to be aware that not every aspect of the prototype should be replicated in the final system, and the designers should inform developers about which aspects of the prototype are intentional and which are arbitrary.

Scenarios

Scenarios are the ultimate minimalist prototype in that they describe a single interaction session without any flexibility for the user. As such, they combine the limitations of both horizontal prototypes (users cannot interact with real data) and vertical prototypes (users cannot move freely through the system).

The term "scenario" has seen widespread use in the user interface community with slightly different meanings [Campbell 1992; Karat and Karat 1992]. Carroll and Rosson [1990] give examples of the term in at least seven different meanings. Therefore, I will try to clarify the terminology by the following definition [Nielsen 1990d]:

A scenario is an encapsulated description of

- an individual *user*
- using a specific set of computer *facilities*
- to achieve a specific *outcome*
- under specified *circumstances*
- over a certain *time interval* (this in contrast to simple static collections of screens and menus: The scenario explicitly includes a time dimension of what happens when).

As such, scenarios have two main uses: First, scenarios can be used during the design of a user interface as a way of expressing and understanding the way users eventually will interact with the future system. Second, scenarios can be used during early evaluation of a user interface design to get user feedback without the expense of constructing a running prototype.

For example, a scenario for the use of an automated teller machine (ATM) might read as follows for used during the design phase:

1. The user approaches the machine and inserts a bank card. No matter what side is up, the machine reads the card correctly.
2. The machine asks the user to input a four-digit personal identification number, and the user does so using the numeric keypad.
3. The machine presents the user with a menu of four options, "`withdraw $100`," "`withdraw other amounts`," "`make a deposit`," and "`other transactions`." There is a button next to each of the menu options.
4. The user presses the button for "`withdraw $100`," and the machine pays out that amount, deducting it from the user's account. If the user has more than one account tied into the bank card, the amount is deducted from the account with the largest balance.

5. The machine returns the bank card to the user.

This scenario immediately raises some questions for the design of the user interface to this machine. For example, is $100 the best amount to have available as a single-button choice?[8] Is it even a good idea to have this accelerated option for a pay-out at a single push of a button, or should the user always be asked to specify the account in case there are several possibilities? And so on. In general, scenario descriptions are good tools in early design stages because they can be generated and edited before the user interface has been fully designed [Carroll and Rosson 1992]. Scenarios describing possible uses of envisioned future systems are also helpful for early participatory design exercises, since users will find it easier to relate to the task-oriented nature of the scenarios than to the abstract, and often function-oriented, nature of systems specifications.

Scenarios can also be used for user testing if they are developed with slightly more detail than a pure narrative. In the previous example, it would be possible to make mock-up drawings of the ATM screens with the buttons and menus, and present them to users, asking them to "use" the screens to withdraw money, and asking them what they would think should happen in each step.

Elaborate scenarios are sometimes produced in the form of "day-in-the-life" videotapes [Vertelney 1989]. These videos show enactments of "users" (actors) interacting with a simulated system in the course of their daily activities. Because the interactions are shown on video, the simulated system can be produced using all kinds of special effects and can be made to look quite sophisticated [Dubberly and Mitsch 1987]. These videos can then be shown to users; for example, to prompt discussions in focus groups.

8. One way of empirically answering this question would be to analyze the bank's database of previous ATM withdrawals. If it turned out that most withdrawals were for the amount of $50, then the $100 should be changed to $50.

4.9 *Interface Evaluation*

The most basic advice with respect to interface evaluation is simply to *do it*, and especially to conduct some user testing. The benefits of employing some reasonable usability engineering methods to evaluate a user interface rather than releasing it without evaluation are much larger than the incremental benefits of using exactly the right methods for a given project.

Whitefield *et al.* [1991] provide a classification of evaluation methods on two dimensions: whether or not real users are involved and whether or not the interface has actually been implemented. One would certainly expect the best results from testing real users and real systems, but doing so may not always be feasible. The prototyping methods described above provide a means of performing evaluations early enough to influence a project while it can still change direction, and the heuristic evaluation method discussed in Chapter 5 allows you to assess usability without the expense of a user test.

User testing is covered in more detail in Chapter 6.

Severity Ratings

From whatever evaluation methods are used, a major result will be a list of the usability problems in the interface as well as hints for features to support successful user strategies. It is normally not feasible to solve all the problems, so one will need to prioritize them. Priorities are best based on experimental data about the impact of the problems on user performance (e.g., how many people will experience the problem and how much time each of them will waste because of it), but sometimes it is necessary to rely on intuitions only.

Severity ratings are usually gathered by sending a group of usability specialists a list of the usability problems discovered in the interface and asking them to rate the severity of each problem. Sometimes, the severity raters are given access to use the system while making their estimates, and sometimes they are asked to judge the problems based only on written description. Note that

the latter approach is possible because the severity raters are supposed to be usability specialists. They should therefore be able to visualize the interface based on the written description (and possibly some screen dumps) in a way that regular users would normally not be able to do. Typically, evaluators need only spend about 30 minutes to provide their severity ratings, though more time may of course be needed if the list of usability problems is extremely long. It is important to note that each usability specialist should provide his or her individual severity ratings independently of the other evaluators.

Unfortunately, severity ratings derived purely by subjective judgment from usability specialists are not very reliable. People have too different opinions about usability. I therefore recommend that you never rely on severity ratings from any single usability specialist (especially not yourself!). Instead, collect ratings from several independent evaluators. Even with just three to four evaluators, the mean of their ratings is much better than the ratings from any single one of them. In one case study, the probability for getting within ±0.5 rating unit from the true severity of a problem on a 5-point rating scale was only 55% with ratings from a single usability specialist, but 95% with the mean of ratings from 4 independent specialists [Nielsen 1994b].

Two common approaches to severity ratings are either to have a single scale or to use a combination of several orthogonal scales. A single rating scale for the severity of usability problems might be

0 = this is not a usability problem at all

1 = cosmetic problem only—need not be fixed unless extra time is available on project

2 = minor usability problem—fixing this should be given low priority

3 = major usability problem—important to fix, so should be given high priority

4 = usability catastrophe—imperative to fix this before product can be released

		Proportion of users experiencing the problem	
		Few	*Many*
Impact of problem on the users who experience it	*Small*	Low severity	Medium severity
	Large	Medium severity	High severity

Table 8 *Table to estimate the severity of usability problems based on the frequency with which the problem is encountered by users and the impact of the problems on those users who do encounter it.*

Alternatively, severity can be judged as a combination of the two most important dimensions of a usability problem: how many users can be expected to have the problem, and the extent to which those users who do have the problem are hurt by it. A simple example of such a rating scheme is given in Table 8. Of course, both dimensions in the table can be estimated at a finer resolution, using more categories than the two shown here for each dimension. Both the proportion of users experiencing a problem and the impact of the problem can be measured directly in user testing. A fairly large number of test users would be needed to measure reliably the frequency and impact of rare usability problems, but from a practical perspective, these problems are less important than more commonly occurring usability problems, so it is normally acceptable to have lower measurement quality for rare problems.

If no user test data is available, the frequency and impact of each problem can be estimated heuristically by usability specialists, but such estimates are probably best when made on the basis of at least a small number of user observations.

One can add a further severity dimension by judging whether a given usability problem will be a problem only the first time it is encountered or whether it will persistently bother users. For example, consider a set of pull-down menus where all the menus are indicated by single words in the menubar except for a single

menu that is indicated by a small icon (as, for example, the Apple menu on the Macintosh). Novice users of such systems can often be observed not even trying to pull down this last menu, simply because they do not realize that the icon is a menu heading. As soon as somebody shows the users that there is a menu under the icon (or if they read the manual), they immediately learn to overcome this small inconsistency and have no problems finding the last menu in future use of the system. This problem is thus not a persistent usability problem and would normally be considered less severe than a problem that also reduced the usability of the system for experienced users.

4.10 *Iterative Design*

Based on the usability problems and opportunities disclosed by the empirical testing, one can produce a new version of the interface. Some testing methods such as thinking aloud provide sufficient insight into the nature of the problems to suggest specific changes to the interface in many cases. Log files of user interaction sequences often help by showing where the user paused or otherwise wasted time, and what errors were encountered most frequently. It often also helps if one is able to understand the underlying cause of the usability problem by relating it to established usability principles such as those discussed in Chapter 5, or by using a formal classification scheme for different categories of problems [Booth 1990]. In other cases alternative potential solutions need to be designed based solely on knowledge of usability guidelines, and it may be necessary to test several possible solutions before making a decision. Familiarity with the design options, insight gained from watching users, creativity, and luck are all needed at this point.

Houde [1992] presents an interesting case study of iterative design of a graphical user interface for the manipulation of three-dimensional objects on a two-dimensional computer screen. One of the issues that was addressed in the iterative design was the design of cursors and handles for movement and rotation. The initial design

used a single picture of a grasping hand, but users were soon seen to be disturbed by having the cursor seem to grasp empty air next to the object they wanted to move rather than the object itself. The second iteration replaced the static image of the cursor with an active area on each movable object, such that a customized picture of a hand grasping the object in an appropriate manner for the object would appear when the user clicked in the active area. Unfortunately, the concept of an active area frustrated users who had to click all over the objects to find the spot where the picture of the grabbing hand would appear. The third iteration therefore introduced multiple customized hands that would appear on an object when it was selected. Users could then move these hands as handles to manipulate the object. Again, user testing indicated problems, this time because the way people would want to use hands to move objects was very individual and would depend on the shape of the object (for example, lifting a lamp would be done differently than lifting a chair). Finally, the fourth, successful, solution was to surround each selected object by a wire-frame bounding box and attach the hands to the box. Because of the regular shape of the box, users were less confused about how to use the hands to move it.

As shown by this example, some of the changes made to solve certain usability problems may fail to solve the problems. A revised design may even introduce new usability problems [Bailey 1993]. This is yet another reason for combining iterative design and evaluation. In fact, it is quite common for a redesign to focus on improving one of the usability parameters (for example, reducing the user's error rate), only to find that some of the changes have adversely impacted other usability parameters (for example, transaction speed).

In some cases, solving a problem may make the interface worse for those users who do not experience the problem. Then a trade-off analysis is necessary as to whether to keep or change the interface, based on a frequency analysis of how many users will have the problem compared to how many will suffer because of the proposed solution. The time and expense needed to fix a particular problem is obviously also a factor in determining priorities. Often,

usability problems can be fixed by changing the wording of a menu item or an error message. Other design fixes may involve fundamental changes to the software (which is why they should be discovered as early as possible) and will only be implemented if they are judged to impact usability significantly.

Furthermore, it is likely that additional usability problems appear in repeated tests after the most blatant problems have been corrected. There is no need to test initial designs comprehensively since they will be changed anyway. The user interface should be changed and retested as soon as a usability problem has been detected and understood, so that those remaining problems that have been masked by the initial glaring problems can be found.

I surveyed four projects that had used iterative design and had tested at least three user interface versions [Nielsen 1993b]. The median improvement in usability per iteration was 38%, though with extremely high variability. In fact, in 5 of the 12 iterations studied, there was at least one usability metric that had gotten *worse* rather than better. This result certainly indicates the need to keep iterating past such negative results and to plan for at least three versions, since version two may not be any good. Also, the study showed that considerable additional improvements could be achieved after the first iteration, again indicating the benefits of planning for multiple iterations.

During the iterative design process it may not be feasible to test each successive version with actual users. The iterations can be considered a good way to evaluate design ideas simply by trying them out in a concrete design. The design can then be subjected to heuristic analysis and shown to usability experts and consultants or discussed with expert users (or teachers in the case of learning systems). One should not "waste users" by performing elaborate tests of every single design idea, since test subjects are normally hard to come by and should therefore be conserved for the testing of major iterations. Also, users get "worn out" as appropriate test subjects as they get more experience with the system and stop being representative of novice users seeing the design for the first

time. Users who have been involved in participatory design are especially inappropriate as test subjects, since they will be biased.

Capture the Design Rationale

The rationale for the various user interface design decisions can be made explicit and recorded for later reference [Moran and Carroll 1994]. Having access to an audit trail through the design rationale is important during iterative development and during development of any future releases of the product. Since changes to the interface will often have to be made, it is helpful to know the reasons underlying the original design so that important usability principles are not sacrificed to attain a minor objective. Similarly, the design rationale can help technical writers develop documentation and translators develop foreign versions. Furthermore, the design rationale can help in maintaining user interface consistency across successive product versions.

Design rationales can be captured either in traditional written form or in a hypertext [Nielsen 1990a] structure such as the gIBIS system [Conklin and Begeman 1988; Conklin and Yakemovic 1991] with links between alternative design options and the supporting evidence or arguments leading to the choice of one of them. Figure 10 shows an example of a design rationale in hypertext, using a QOC-notation (questions, options, and criteria) similar to that suggested by MacLean *et al.* [1989, 1991a and b]. Future repositories for design rationales may even include video records of design meetings and selected user tests [Hodges *et al.* 1989].

During the development process, a design rationale can also be captured by a low-tech solution on the walls of the design team's meeting room. Karat and Bennett [1990, 1991b] used such a technique by taping notepaper on the walls, using different walls for different perspectives on the design. One wall was used for design sketches, one for design constraints, one for scenarios (cf. page 99), and one for open questions. The scenarios are interaction examples illustrating the flow of specific user actions needed for some result, concentrating on what the user will see, what the user must know, and what the user can do [Karat and Bennett 1991a]. Since design

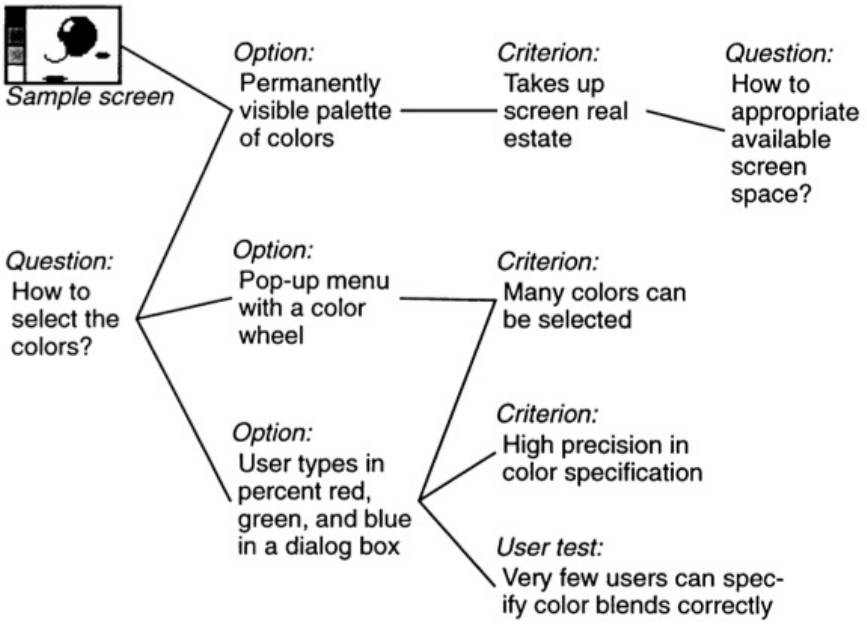

Figure 10 *A partial example of a design rationale for a small part of an interface design for a hypothetical color paint program. The full design rationale might include more sample screens and links to additional design questions like the* "How to appropriate..." *question hinted at here. The lines might denote hypertext links in an online representation, or they could be supported by simple proximity in a paper document.*

issues are often difficult to understand fully in the abstract, the concreteness of the scenarios adds value to a design rationale.

4.11 Follow-Up Studies of Installed Systems

The main objective of usability work after the release of a product is to gather usability data for the next version and for new, future products: In the same way that existing and competing products were the best prototypes for the product in the initial competitive

analysis phase, a newly released product can be viewed as a prototype of future products. Studies of the use of the product in the field assess how real users use the interface for naturally occurring tasks in their real-world working environment and can therefore provide much insight that would not be easily available from laboratory studies.

Sometimes, field feedback can be gathered as part of standard marketing studies on an ongoing basis. As an example, an Australian telephone company collected customer satisfaction data on a routine basis and found that overall satisfaction with the billing service had gone up from 67% to 84% after the introduction of a redesigned bill printout format developed according to usability engineering principles [Sless 1991]. If the trend in customer satisfaction had been the opposite, there would have been reason to doubt the true usability of the new bill outside the laboratory, but the customer satisfaction survey confirmed the laboratory results.

Alternatively, one may have to conduct specific studies to gather follow-up information about the use of released products. Basically, the same methods can be used for this kind of field study as for other field studies and task analysis, especially including interviews, questionnaires, and observational studies. Furthermore, since follow-up studies are addressing the usability of an existing system, logging data from instrumented versions of the software becomes especially valuable for its ability to indicate how the software is being used across a variety of tasks.

In addition to field studies where the development organization actively seeks out the users, information can also be gained from the more passive technique of analyzing user complaints, modification requests, and calls to help lines (see Section 7.5, *User Feedback*, on page 221). Even when a user complaint at first sight might seem to indicate a programming error (for example, "data lost"), it can sometimes have its real roots in a usability problem, causing users to operate the system in dangerous or erroneous ways. Defect-tracking procedures are already in place in many software organizations and may only need small changes to be useful for usability engineering purposes [Rideout 1991]. Furthermore, infor-

mation about common learnability problems can be gathered from instructors who teach courses in the use of the system.

Finally, economic data on the impact of the system on the quality and cost of the users' work product and work life are very important and can be gathered through surveys, supervisors' opinions, and statistics for absenteeism, etc. These data should be compared with similar data collected before the introduction of the system.

4.12 Meta-Methods

To ensure the successful application of the usability engineering methods discussed here, it is important to supplement each of them with the following *meta-methods* (methods that apply to methods):

- Write down an explicit plan for what to do when using the method. For example, a plan for empirical user testing would include information about how many users to test, what kind of users to test (and how to get hold of them), what test tasks these users would be asked to perform (which itself should be based on task analysis and user observation), and a time schedule for the studies.
- Subject this plan to an independent review by a person who is not otherwise on your team and who can critique it from a fresh perspective. This person should preferably be experienced with respect to usability engineering.
- Perform a pilot activity by investing about 10–15% of the total resources budgeted for the use of the method. Then revise your plan for the remaining 85–90% to fix the difficulties that invariably will be found during the pilot activity. For example, with empirical user testing, the original test instructions are often misinterpreted by the users; you want the main test to focus on the usability of your system and not on your ability to write readable test instructions. See page 174 for more information about pilot tests.

Furthermore, as early as possible in the project, an overall usability plan should be established listing the usability activities to be

performed throughout the lifecycle. Not all projects can afford to use all the methods, and the exact methods to use will depend on the characteristics of the project.

These meta-methods may involve a little extra work up front, but they save work in the long term and ensure that your efforts are on the right track to increase usability, thereby reducing the risk of truly wasting the main effort.

4.13 *Prioritizing Usability Activities*

It is not always possible to perform all the recommended usability activities in any given project. My own approach to budget constraints or time pressures is outlined in Section 1.4, *Discount Usability Engineering* (page 16), and stresses

- visit to user sites (see page 73)
- prototyping through scenarios (see page 99)
- simplified thinking aloud (see page 195)
- heuristic evaluation (see page 155)

To get additional prioritizing advice, I surveyed 13 usability engineering specialists and asked them to rate 33 different usability methods for their importance to the usability of the final interface [Nielsen 1992b]. The ratings were on a scale from one (no impact on usability) to five (absolutely essential for usability).

The top six methods according to rated impact on usability were

1–2 Iterative design and task analysis of the user's current task. Both rated 4.7.

3 Empirical tests with real users. Rated 4.5.

4 Participatory design. Rated 4.4.

5–6 Visit to customer location before start of design and field study to find out how system is actually used after installation. Both rated 4.3.

The usability specialists were also asked to what extent they had actually used the 33 methods on their most recent project. There