



# First Principles

---

Introduction 10

- 1.1 Principle 1: Focus on the users and their tasks, not the technology 10
  - 1.2 Principle 2: Consider function first, presentation later 22
  - 1.3 Principle 3: Conform to the users' view of the task 30
  - 1.4 Principle 4: Don't complicate the user's task 36
  - 1.5 Principle 5: Promote learning 39
  - 1.6 Principle 6: Deliver information, not just data 44
  - 1.7 Principle 7: Design for responsiveness 49
  - 1.8 Principle 8: Try it out on users, then fix it! 51
- Further reading 56

## Introduction

The main purpose of this book is to describe user interface bloopers that are often made by developers of computer-based products and services, and to provide design rules and guidelines for avoiding each blooper. First, however, it is useful to ground the discussion of the bloopers by laying out the principles that underlie the design of effective, user-friendly user interfaces. That is the purpose of this chapter.

The principles given in this chapter are *not* specific design rules for graphical user interfaces (GUIs). There is nothing in this chapter about the right or wrong way to design dialog boxes, menus, toolbars, Web links, and so on. Specific design rules about such issues are provided in later chapters of the book, in the design rules for avoiding each blooper.

Specific user interface design rules are also given in “official” style guides for specific industry-standard GUI platforms, such as the *Java Look and Feel Design Guidelines* [Sun, 1999], the *Windows Interface Guidelines for Software Design* [Microsoft, 1995], the *OSF/Motif Style Guide: Rev 1.2* [OSF, 1993], and the *Macintosh Human Interface Guidelines* [Apple, 1993]. Specific GUI guidelines are also provided by many “unofficial” but nonetheless good books, such as Bickford [1997], Fowler [1998], Mandel [1997], McFarland and Dayton [1995], Mullet and Sano [1995], Nielsen [1999d], Shneiderman [1987], Tufte [1983], Weinshenk et al. [1997].

Rather than providing specific GUI design rules, the principles given in this chapter provide the basis for GUI design rules. They are based on the cumulative wisdom of many people, compiled over several decades of experience in designing interactive systems for people. They are also based on a century of research on human learning, cognition, reading, and perception. Later chapters of this book refer to these principles as rationale for why certain designs or development practices are bloopers, and why the recommended remedies are improvements.

### 1.1 Principle 1: Focus on the users and their tasks, not the technology

This is Principle Numero Uno, the Main Principle, the mother of all principles, the principle from which all other user interface design principles are derived:

*Focus on the users and their tasks, not the technology.*

Now that I’ve stated it, and you’ve read it, we’re done, right? You now know how to design all your future products and services, and I needn’t say anything more.

I wish! Alas, many others have stated this principle before me, and it doesn't seem to have done much good. And no wonder: it is too vague, too open to interpretation, too difficult to follow, and too easily ignored when schedules and resources become tight. Therefore, more detailed principles, design rules, and examples of bloopers are required, as well as suggestions for *how* to focus on users, their tasks, and their data. Not to mention that this would be a very thin book if I stopped here.

Before proceeding to those more detailed principles, design rules, and bloopers, I'll devote a few pages to explaining what I mean by "focus on the users and their tasks."

Focusing on users and their tasks means starting a development project by answering the following questions:

- For whom is this product or service being designed? Who are the intended customers? Who are the intended users? Are the customers and the users the same people?<sup>1</sup>
- What is the product or service for? What activity is it intended to support? What problems will it help users solve? What value will it provide?
- What problems do the intended users have now? What do they like and dislike about the way they work now?
- What are the skills and knowledge of the intended users? Are they motivated to learn? How?
- How do users conceptualize and work with the data in their task domain?
- What are the intended users' preferred ways of working? How will the product or service fit into those ways? How will it change them?

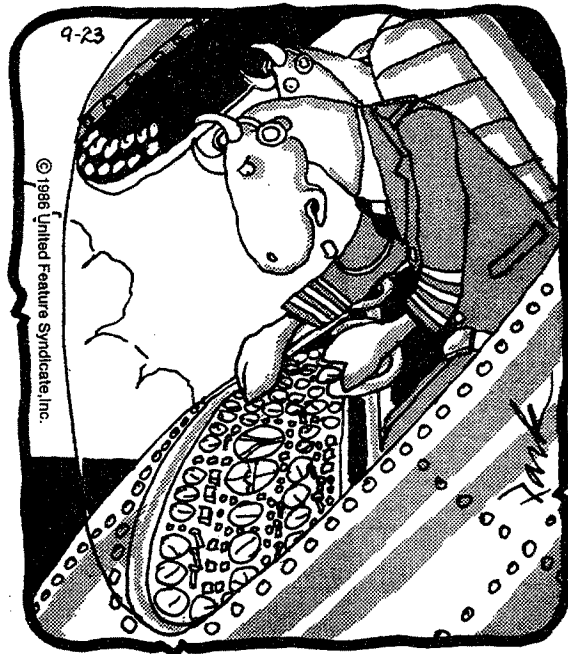
It would be nice if the answers to these questions would fall out of the sky into developers' laps at the beginning of each project. But, of course, they won't. The only way to answer these questions is for the development organization to make an explicit, serious effort to do so. Making a serious effort to answer these questions takes time and costs money. But it is crucial, because the cost of not answering these questions before beginning to design is much, much higher.

### 1.1.1 Understand the users

Several of the questions listed above that developers should answer before starting to design are questions about the intended users of the product: Who are they? What do they like and dislike? What are their skills, knowledge, vocabulary, and motivation? Are they going to be the ones who make the decision to buy the software, or will someone else do that? These questions are best

---

1. Hint: The customer is the person who makes the buying decisions, while the user is the person who . . . well . . . uses the software. These different roles may imply different criteria and needs.



"Darn these hooves! I hit the wrong switch again! Who designs these instrument panels, raccoons?"

answered using a process that is part business decision, part empirical investigation, and part collaboration.

### *Decide who the intended users are*

To a certain extent, the development organization needs to decide who it is developing the product or service for. It is tempting to decide that the intended user base is "everyone"; most development organizations want the broadest possible market. However, that temptation must be strongly resisted: a product or service designed for everyone is likely to satisfy no one. Developers should choose a specific primary target population as the intended user base in order to focus their design and development efforts, even if they suspect that the software might also have other types of users.

In reaching this important decision, the designers should seek input from other parts of the organization in order to

assure that the target user base of the product or service is in line with strategic goals. In particular, developers should seek input from the Marketing and Sales departments because it is they who are usually responsible for identifying and categorizing customers. However, it is important to keep in mind that Marketing and Sales focus—as their job responsibilities dictate—on future *customers* of the product or service, whereas the designers need to understand the future *users*. A product's customers and its users are not necessarily the same people, or even the same type of people, so Marketing and Sales' ideas about who the product is aimed at may have to be filtered or augmented in order to be useful to designers.

### *Investigate characteristics of the intended users*

As described above, understanding the users also requires empirical investigation. Empirical investigation means making an effort to learn the relevant characteristics of potential users. This investigation, first of all, provides information to guide the above-described decision: going out and surveying potential users helps developers discover specific populations of users whose requirements and demographics make them attractive as a target user base. Second, once the above-described decision has been made and the primary target user

population has been identified, developers should use empirical methods to learn as much as possible about that target population.

How do developers gather information about the intended users? By going and talking with them. By talking to their management. By inviting groups of them—and possibly their management—to participate in focus groups. By reading market analyses and magazines about their business. Perhaps even by socializing with them.

### ***Collaborate with the intended users to learn about them***

Finally, understanding the users is best accomplished by working with them as collaborators. Don't treat users merely as objects to be studied. Bring some of them onto the development team. Treat them as experts, albeit a different kind of expert than the developers. They understand their job roles, experience, management structure, likes and dislikes, and motivation. They probably don't understand programming and user interface design, but that's fine—that's what the rest of the team is there for. A useful slogan to keep in mind when designing software is

*Software should be designed neither **for** users  
nor **by** them, but rather **with** them.*

### ***Bringing it all together***

The goal of this three-part process—part decision, part investigation, part collaboration—is to produce a profile that describes the typical intended user of the planned product or service, or perhaps a range of intended users. The profile should include information such as job description, job seniority, education, salary, hourly versus salaried, how their performance is rated, age, computer skill level, and relevant physical or social characteristics, if any. With such a profile in hand, developers know what they are aiming at. Without it, they are, as Bickford [1997] says, “target shooting in a darkened room.”

Some designers even go beyond constructing profiles for the intended users of a software product or service. Cooper [1999] advocates making user profiles concrete by instantiating them as fully elaborated *personas*: characters with names, vocations, backgrounds, families, hobbies, skills, and lifestyles. This is similar to the practice among novelists and script writers of writing “backstories” for every significant character in a book or movie. A character's back-story helps ground discussions about the sorts of things that character would and would not do. Similarly, the *personas* in Cooper's design methodology help ground discussions of the sorts of designs a particular target user type would find easy, difficult, annoying, fun, useful, useless, and so on.

The trick, of course, is to build profiles and *personas* from real data obtained from prospective users. If software developers just make up target user

profiles and personas based on armchair speculation, they might as well not have them.

### 1.1.2 Understand the tasks

Some of the questions listed above pertain to the intended function of the product or service, that is, the “task domain” it is intended to support. As with understanding the users, understanding the task domain is best accomplished by a three-part process: part business decision, part empirical investigation, and part collaboration.

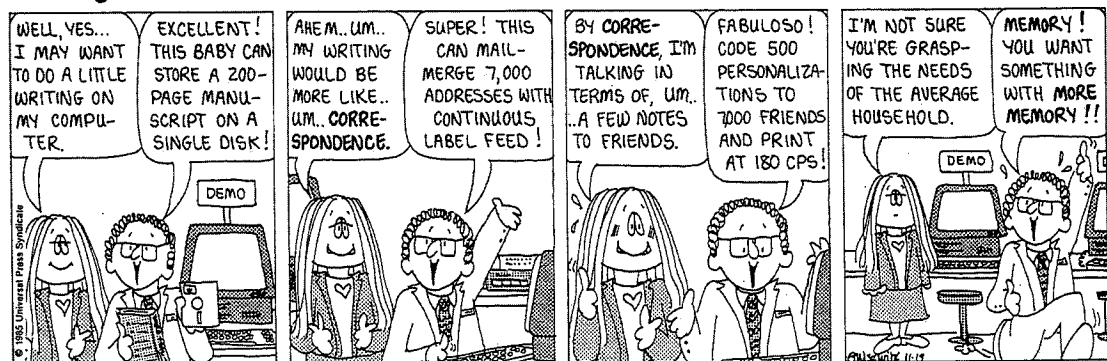
#### *Decide what the intended task domain is*

Understanding the intended task domain is partly a business decision because a software development organization is never completely open-minded about what products or services to develop. No software development organization is going to pick a group of potential customers purely at random, figure out what they need, and design a product to fill that need. Instead, decisions about what products and services to offer are strongly influenced—one might even say “predetermined”—by

- the organization’s strategic goals, reflecting the interests of its founders, top management, and shareholders
- the expertise of its employees
- its past history
- its assets, processes, and infrastructure
- its perception of market opportunities and niches
- new technologies that have been developed by its researchers

**cathy®**

**by Cathy Guisewite**



Cathy © 1985 Cathy Guisewite. Reprinted by permission of Universal Press Syndicate. All Rights Reserved.

As a concrete example, a company that had been founded to develop music instruction software would be unlikely to decide to develop air traffic control systems unless it underwent such a radical change of management and employees that it was really a different company that happened to have the same name.

The last item in the previous list, "new technologies that have been developed by its researchers," is an especially important one. In the computer and software industries, decisions about what products or services to bring to market are often more strongly influenced by technological "push" than by other factors [Johnson, 1996b]. Whether this is good or bad is a subject of frequent debate. Norman [1999] argues that it is often good for emerging markets, but usually bad for mature ones. In my opinion, it is more often bad than good.

Regardless of which of the above factors are responsible, most development organizations decide in advance what general product or service area to target, such as document creation and management, information retrieval, banking, music, home finance, presentation slide-making. This decision combines with the decision about the primary target user base to yield a fairly specific product category, such as document-editing software for technical writers, or banking software for bank tellers.

As with identifying the target users, designers should seek input from other parts of their organization in order to assure that the target task domain of the product or service is in line with strategic goals. Again, developers should seek input from the Marketing and Sales departments because it is they who are usually responsible for identifying market opportunities, and because they often have at least a secondhand understanding of the work the target users do.

### *Investigate the intended task domain*

Once a product category has been decided, the empirical investigation part of "understanding the tasks" comes into play. Before starting to design or implement anything, developers need to learn as much as they can about exactly how the intended users do the tasks that the software is supposed to support. This is called conducting a "task analysis." The goal of a task analysis is to develop a thorough understanding of the activities the product or service is intended to support.

The best way to conduct a task analysis is for members of the development team to arrange to talk with and observe people who either will eventually be users or whose demographics match the profiles of the intended users. Since the product or service will not have been developed yet, these interviews and observation sessions are concerned with understanding the work as it is done *before* the new product or service is introduced. Users can be interviewed or observed individually and in groups, face-to-face or by telephone or email, while they work or apart from their work.

It is important to both interview users and observe them working; the two techniques are complementary. Interviews and focus groups provide explana-

tions, rationales, goals, and other information that cannot be directly observed. However, interviews may also provide *mis*-information, such as how a process is *supposed* to (but doesn't) work, or what the user thinks the interviewer wants to hear. Observation, on the other hand, lets developers see what actually happens, but leaves it to the observers to interpret what they see. Since the observers are, by definition, naive about the task domain—otherwise they wouldn't need to learn about it—their ability to interpret correctly what they observe is severely limited.

It is possible to combine interviewing with observing users: developers can interview prospective users at their workplaces, encouraging them to answer questions not only verbally, but also by demonstrating what they normally do. When gathering task information in this way, developers should ask users to talk while they work in order to explain what they are demonstrating; otherwise many users fall into their common habit of working silently or mumbling under their breath.

Developers can supplement interviews and observation of users by interviewing the users' managers. This provides another perspective on the same tasks, which is always useful. However, even more than interviews of users, interviews of users' managers must be interpreted carefully; managers are highly likely to talk in terms of how the work is *supposed* to be rather than how it really is.

### ***Collaborate with users to learn about the task domain***

Collaborating with users is even more important for understanding the task domain than it is for understanding the users themselves. The limitations of both straight interviews and straight observation are too great for developers to rely upon conclusions obtained purely by those methods. These limitations can be overcome by introducing two-way feedback into the task discovery and analysis process. Designers should not simply collect data from users; they should provide preliminary analyses and conclusions to users and seek their reactions. In such a process, the designers are not the only ones who learn; the users also gain a greater awareness of how they themselves work and about what sorts of technology might help them work better. In return for the effort required to establish a collaborative working relationship, the designers get more reliable data from the users.

Further information about involving users in analyzing a target task domain is provided in an article by Dayton et al. [1998] and a book by Greenbaum and Kyng [1991].

### ***Bringing it all together***

Fortunately, analyzing the task domain involves much the same activity as investigating the users. Although the two investigations were discussed separately here in order to better explain each one, developers usually conduct them



at the same time, in the same interview and collaboration sessions. This synergy is helpful because access to prospective users is likely to be quite limited.

This principle began with a list of questions developers should answer before designing a product or service. Those questions were fairly broad. A well-done task analysis answers some fairly detailed questions:

- What tasks does the person do that are relevant to the application's target task area?
- Which tasks are common, and which ones are rare?
- Which tasks are most important, and which ones are least important?
- What are the steps of each task?
- What is the product of each task?
- Where does the information for each task come from, and how is the information that results from each task used?
- Which people do which tasks?
- What tools are used to do each task?
- What problems, if any, do people have performing each task? What sorts of mistakes are common? What causes these problems and mistakes?
- What terminology do people who do these tasks use?
- How are different tasks related?
- What communication with other people is required to do the tasks?



### Example task analysis questions

In 1994, as part of a research project on the benefits of task-specific versus generic application software [Johnson and Nardi, 1996], a colleague and I investigated how people prepare presentation slides. Our investigation amounted to a task analysis. We interviewed people in their offices, encouraging them to both talk about and demonstrate how they work. The questions covered in the interviews are listed below. The interviewer allowed the conversation to flow naturally rather than strictly following the list of questions, but made sure answers to all of the questions had been captured on tape before ending the interview.

1. What is your role in producing presentation slides?
  - 1.1 Do you produce slides yourself or do you supervise others who do it?
    - 1.1.1 What sort of training or experience is required to do the job you do?
    - 1.1.2 [If supervises others] What is the skill level of your employees?
  - 1.2 How much of your total job involves producing presentation slides?
  - 1.3 For whom do you produce these presentation slides?
    - 1.3.1 Who is the customer (i.e., who approves the slides)?
    - 1.3.2 Who is the audience for the presentations?

- 1.4 What sort of quality level is required for the slides?
  - 1.4.1 How important are elaborate special effects (e.g., animation, dissolve)?
  - 1.4.2 Who decides on appearance and quality, you or the customer?
  - 1.4.3 Are there different kinds of presentations with different quality requirements?
- 1.5 Do you (your department) follow slide formatting standards?
  - 1.5.1 How do you assure that slides adhere to those standards?
  - 1.5.2 Does your slide-making software help with standardization of presentations?
2. What software do you use to create presentation slides?
  - 2.1 Who decides what software you use for this?
  - 2.2 Do you use one program or a collection of them?
    - 2.2.1 [If many] What are the different programs used for?
  - 2.3 Do you use general-purpose drawing software or slide-making software?
    - 2.3.1 Why?
  - 2.4 What do you like about each of the programs you use?
  - 2.5 What do you dislike about each one? What would you like to see changed?
    - 2.5.1 Describe some of the things you do to "work around" limitations of the software.
  - 2.6 How easy is the software for new users to learn?
    - 2.6.1 How do they learn the software (classes, manuals, using, asking)?
    - 2.6.2 How did you learn it?
  - 2.7 What other software have you used, tried, or considered for making slides, either here or in previous jobs?
    - 2.7.1 Why don't you use it now?
3. What is involved in making slides?
  - 3.1 Describe the complete process of producing a presentation, from when you take the assignment to when you deliver it to the customer.
    - 3.1.1 How much revision is usually required before a presentation is considered done?
  - 3.2 Do you usually create new presentation slides?
    - 3.2.1 What is hard and what is easy about creating new material (i.e., what goes quickly and what takes time and work)?
  - 3.3 Do you reuse old slides in new presentations?
    - 3.3.1 What is hard and what is easy about reusing old material (i.e., what goes quickly and what takes time and work)?
  - 3.4 How do you (your department) organize and keep track of slides and presentations?
    - 3.4.1 Is each slide a separate file, or are all the slides in a presentation together in one file?
    - 3.4.2 Do you use directories (folders) and subdirectories (subfolders) to organize your material?

- 3.4.3 How do you name your slide (or presentation) files?
- 3.4.4 Do you ever fail to find a slide you know you have?
- 3.4.5 How does your software hinder you in reusing material?
- 3.4.6 How easily can you include a single slide in several different presentations?
- 3.5 What kinds of revisions are often required in the process of preparing a presentation?
  - 3.5.1 Which are easy and which are hard?
  - 3.5.2 Are the same revisions easy and hard for each of the slide-making programs you use?
  - 3.5.3 Some specific cases we'd like to know about:
    - 3.5.3.1 A slide used in multiple presentations is changed.
    - 3.5.3.2 A company logo or standard border must be added to every slide in a presentation.
    - 3.5.3.3 The order of slides in a presentation must be changed.
    - 3.5.3.4 The round bullets throughout a presentation must be changed to square bullets.
    - 3.5.3.5 The font used throughout a presentation must be changed.
    - 3.5.3.6 Each of the points on a particular slide must be expanded into a separate slide.

### 1.1.3 Interactive products and services function in a broad context

When engineers design computer hardware, software, online services, and electronic appliances, they often regard what they are designing as if it were the center of the universe, or even the only thing in the universe. They fail to consider the broad context in which the technology will be used, and what the users' total experience will be in using the technology in that context.

Sometimes even people who purchase technology fall prey to technocentric tunnel vision. They have a problem they want to solve and are hoping that, by simply acquiring and using some technology, they can fix it. Wishful thinking often predisposes technology buyers to perceive the problem as being simpler than it really is, that is, easily correctable by available technology. It also often predisposes them to believe the often inflated claims of technology manufacturers and vendors.

The fact that engineers and technology buyers often manifest technocentric tunnel vision does not mean that they are stupid or overly gullible, just that they are human. People focus on their own goals and desires or those of their organization, and often fail to take into account that there are other things in the environment that influence the outcome of applying technology to a problem.

Consider a person who designs a car alarm or the person who buys one for his or her automobile. The designer is focused on the goal of designing an appliance that can signal when a car is being burglarized or vandalized. The




### Wishful thinking and military computer applications

The following are cases in which both developers and potential buyers of computer-based military technology were seduced by technocentric wishful thinking into overlooking important contextual factors.

*SDI:* In the 1980s, then-President Ronald Reagan proposed that the United States build a defensive "shield" against nuclear attack. Out of that proposal grew a research program called the Strategic Defense Initiative (SDI), often referred to in the popular press as "Star Wars." An important part of the SDI "shield" would be satellites for detecting and shooting at enemy missiles. A major problem with SDI was that these orbiting detectors and battlestations could easily be tracked and shot down by an enemy. Thus, crucial components of SDI would only remain operational if the enemy were nice enough not to shoot at them. SDI proponents were seduced by the appeal of a highly desirable goal—a shield against nuclear attack—to overlook critical facts about the world in which the system would have to operate.

*Remote-controlled vehicle:* An article in the *Wall Street Journal* on November 28, 1990, described a prototype of an autonomous vehicle that U.S. soldiers might someday use to explore enemy territory without risking their own lives. The vehicle was being designed so that it could be driven "like an ordinary car" to the edge of an enemy-occupied area, then driven into the area using remote controls and a "virtual reality" helmet that allowed a soldier to see the vehicle's surroundings as if he were still physically in the vehicle. The designers of the vehicle overlooked two important contextual factors: (1) the possibility that people and dogs might notice the vehicle as it clanked around, and (2) that soldiers in or near a combat zone might hesitate to put on a helmet that obscured their view of their immediate surroundings.

*Range-finding guns:* An article in the *Christian Science Monitor* on April 6, 1998, described a prototype of a new gun that would allow soldiers to "shoot around corners." The gun would shoot shells that would explode at a specified distance and rain shrapnel onto enemy soldiers hiding behind walls, rocks, vehicles, and so on. To gauge the distance to the target, the gun had a laser-based range finder. However, like an autofocus camera, the gun's range finder would have to be pointed at some object. The gun's designer failed to consider that when an enemy is hiding around a corner, there is no real target at which to aim the range finder. Thus, soldiers trying to use the gun in combat would probably have to aim at objects *near* the corner, shoot, and manually adjust the distance estimate—perhaps several times—before finding the correct distance. 

buyer is thinking solely of how to protect the car. Neither one stops to consider that the alarm must function in an environment in which many other people have similar car alarms, and in which many events besides break-ins can trigger the alarm. When an alarm goes off in the neighborhood, it will be difficult

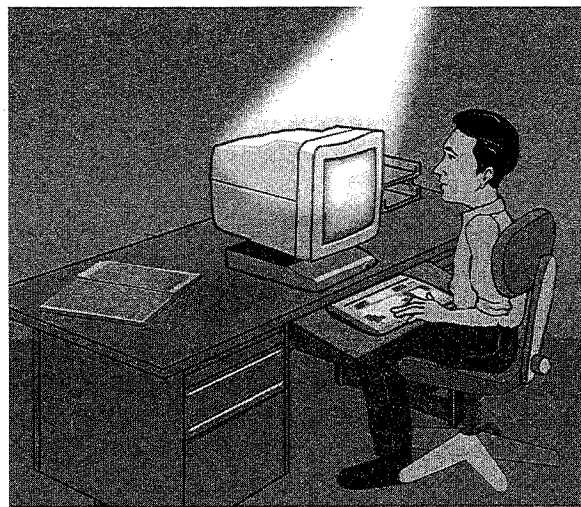
to tell whose car it is and whether it signals an actual attempt at theft or vandalism. It usually will not. An otherwise sound idea and resulting product fails to provide value, not because of a flaw in the idea itself, but because the designer and buyer didn't consider the larger picture.

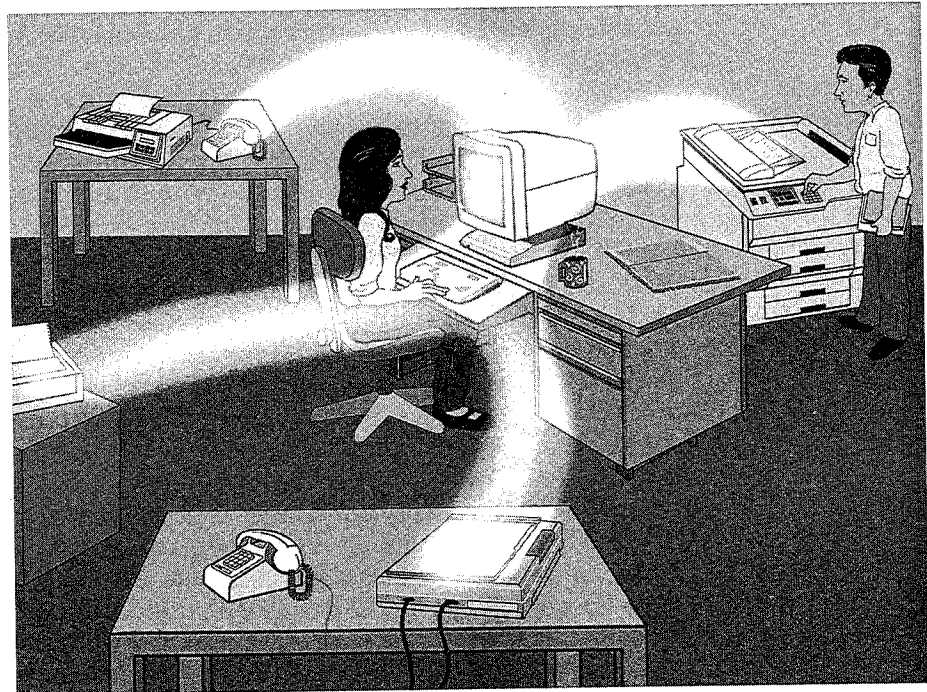
Applying technological tunnel vision to an office containing a desktop computer, one would see the office as a darkened room with a spotlight on the computer. One would see people (users) come into the spotlight, use the computer, and leave, disappearing back into the darkness (see Figure 1.1). The things the people do with the computer would be seen as disconnected, and those they do without it would be seen as unimportant. Using the computer, the people word process, they create graphics, they perform calculations, they enter and retrieve data. Where does the input come from? Unimportant. What is the output used for? Irrelevant. This perspective induces engineers to ask questions like "What should this product do?"

In fact, the office computer would be embedded in a work context. The proper way to view the office is as a large collection of paths of light flowing into, through, and out of the office. In each light path is a person. Some of the light paths intersect the computer; others don't (see Figure 1.2). The paths represent the flow of information, communication, and work; they show where it comes from and where it goes. This contextual view of the office induces designers to ask rather different questions. Instead of asking, "What should this product do?", designers would ask, "What does this *office* do?", "How does it do it?", and "What kind of computer software would support doing that (better)?"

To produce an effective product or service, developers must understand the context in which it will be used. When designers of a software application don't consider the context in which the application will be used, a common result is

**Figure 1.1**



**Figure 1.2**

that the application's users find themselves typing data into it that just came out of another computer program they use. The application's designers didn't think about where the data would come from, and so didn't design their application to take input directly from the other program.

To understand the context in which a planned product or service will be used, developers must study that context. Studying the context means talking to prospective or representative users, observing them, and analyzing the data thus collected. Representative users can even be recruited as fellow analyzers. Good advice on how to do these things is provided by two books: Greenbaum and Kyng [1991] and Beyer and Holtzblatt [1998].

## 1.2 Principle 2: Consider function first, presentation later

When given an assignment to develop a computer-based product or service, many GUI developers—even many user interface designers—immediately begin trying to decide how it will look. Some people sketch designs using paper and pencil or computer drawing tools. Some use interactive GUI or Web site construction tools to lay out the product's displays and controls. Some begin hacking actual implementation code.

Starting by worrying about appearances is putting the cart before the horse. It is tempting, but it is almost always a mistake. Yielding to this temptation results in products and services that lack important functionality, that contain many components and functions that are irrelevant to the work users have to do, and that seem ad hoc and arbitrary to users and so are difficult to learn and to use.

### 1.2.1 What “consider function first” does not mean

There is a danger that some readers will misinterpret Principle 2, so I will take a moment to explain what it does not mean. It does not mean “Get the functionality designed and implemented first, and worry about the user interface later.” This misinterpretation matches the approach many software developers and development managers use to develop software. I want to state clearly that that approach will not produce successful software. The user interface is not something that can successfully be “worried about” or tacked on at the end of a development project.

The assumption underlying Principle 2 is that the user interface of a software product or service is not only—not even mainly—about the software’s presentation. User interfaces are like human skin. Human skin isn’t just the surface layer that we can see. It has aspects besides the superficial ones such as color, smoothness, and amount of hair. It has depth. It has structure: the epidermis, the endodermis, capillaries, nerve endings, hair follicles, pores, and so on. Most importantly, it has a function, or rather many functions, such as protection, transfer of moisture, and tactile sensation. Furthermore, we have different kinds of skin, each serving different functions in addition to the basic ones. The skin on the back of our hand looks and feels different from that on our palm, or from that on our lips, or on our knees, or on our scalp because it serves different purposes. Finally, the structure and function of our skin is related to our overall structure and function.

Similarly, there is more to user interfaces than meets the eye (or the mouse). They have depth, structure, and function. They also have variety: they vary according to their function, within an application as well as between applications. And, like human skin, user interfaces are related to the overall structure and function of the software they serve.

### 1.2.2 What “consider function first” does mean

Principle 2 states that software developers should consider the purpose, structure, and function of the user interface—and of the software as a whole—before considering the presentation—the surface appearance—of the user interface. The word “function” here does not mean “implementation”—how does it work? It means “role”—what does it do?

Before sketching displays, laying out controls, cutting foam prototypes, or hacking code, developers should focus their efforts on answering first the questions given under Principle 1, and then the following questions:

- *What concepts will the product or service expose to users?* Are they concepts that users will recognize from the task domain, or are they new concepts? If new, can they be presented as extensions of familiar concepts, or are they completely foreign concepts from the domain of computer science or electrical engineering?
- *What data will users create, view, or manipulate with the software?* What information will users extract from the data? How? What steps will they use? Where will data that users bring into the product or service come from, and where will data produced in the product or service be used?
- *What options, choices, settings, and controls will the application provide?* This is not a question about how to *represent* controls (e.g., as radiobuttons, type-in fields, menus, sliders); it is about their *function, purpose, and role* in the product or service (e.g., day of the week, dollar amount, email address, volume level). It is about what options the software provides and what the possible values of those options are.

### 1.2.3 Develop a conceptual model

Determining the answers to the above questions is often referred to as developing a “conceptual model” for the product or service.

#### *What is a conceptual model and what is it not?*

A conceptual model is, in a nutshell, the model of a product or service that the designers want users to understand. By using the software and reading its documentation, users build a model in their minds of how it works. Hopefully, the model that users build in their minds is close to the one the designers intended. This hope has a better chance of being realized if the designers have explicitly designed a clear conceptual model beforehand.

Developing a conceptual model before designing a user interface is often difficult; it is tempting for developers to jump right into discussing user interface concepts, such as control panels, menus, and data displays. The temptation is exacerbated by the tendency of Sales and Marketing personnel to state functional requirements in terms of window layout and mouse clicks. When marketing requirements are stated in such terms, developers should gracefully decline them, and ask instead for requirements stated in terms of the task domain: the problems users face and the goals they wish to achieve.

A conceptual model is not a user interface. It is not expressed in terms of keystrokes, mouse actions, dialog boxes, controls, or screen graphics. It is



expressed in terms of the concepts of the intended users' task domain: the data that users manipulate, the manner in which that data is divided into chunks, and the nature of the manipulations that users perform on the data. It explains, abstractly, the function of the software and what concepts people need to be aware of in order to use it. The idea is that by carefully crafting an explicit conceptual model, then designing a user interface from that, the resulting software will be cleaner, simpler, and easier to understand.

### ***Keep it as simple as possible, but no simpler***

One goal, when developing a conceptual model for a planned software product or service, is to keep it as simple as possible, that is, with as few concepts as are needed to provide the required functionality. In the design of computer-based products and services, as in many things, the following slogan is a helpful design guide:

*Less is more.*

A related goal is to keep the conceptual model as focused on the task domain as possible, that is, with few or no concepts for users to master that are not found in the task domain.

However, the overriding goal of a conceptual model is to capture the way the software's intended users think about the target task domain. If, in trying to minimize complexity, developers omit important concepts from the conceptual model, or merge concepts that users regard as separate, the usability and usefulness of the resulting software will suffer just as it will if extraneous concepts are included.




### **A case of oversimplification**

Consider the Unix operating system. In its raw forms, Unix does not provide a function for renaming files. The designers of Unix did not consider an explicit *rename* command necessary because files can be renamed using the *move* (*mv*) command. To rename a file, users just move it to a file that has a different name.

For example, the command "*mv MyInfo JeffsInfo*" moves the content of the file *MyInfo* into a new file named *JeffsInfo* and then deletes *MyInfo*. Effectively, *MyInfo* has been renamed "*JeffsInfo*".

The problem is that when most Unix users need to rename a file, they will look for some kind of *rename* command, not a *move* command. Thus, many Unix users had trouble figuring out or remembering how to rename files. Some such users created command aliases on their computers so that they could have an explicit *rename* command.

The moral of this story is that Unix's conceptual model was oversimplified: it merged two concepts that users regarded as distinct. 

### *Perform an objects/actions analysis*

The most important component of a conceptual model is an objects/actions analysis. An objects/actions analysis is a listing of all the objects that the planned product or service will expose to users, and the actions that users can perform on each of those objects.

The software's implementation may include objects and associated operations other than those listed in the conceptual model, but those are not supposed to be visible to users. In particular, purely implementation objects and their associated actions—such as a text buffer, a hash table, or a database record—have no place in a conceptual model. Furthermore, the conceptual model and the implementation model may differ radically from each other. The implementation may not even be object oriented. For example, the application might be written in Visual Basic on top of an old database.

For the benefit of the developers of a planned software application, the objects in the conceptual model can be organized into a type hierarchy, with subtypes inheriting actions and attributes from their parent types. This makes commonalities and relationships between objects clearer to the developers.

It should be kept in mind, however, that organizing the objects in a class hierarchy is purely a way of facilitating the implementation of the software. Subclasses and inheritance are not concepts that will make sense to users (unless the software's target task domain is programming). Therefore, if the conceptual model is to be presented to users (e.g., in participatory design sessions or in software documentation), great care should be taken to express computer science concepts such as class hierarchies in terms that make sense to users. For example, most people can understand the relationship "is one type of," as in "A checking account is one type of bank account."

Depending on the application, objects may also be organized into a containment hierarchy, in which some objects contain other objects. For example, an email folder may contain email messages and other email folders.

Laying out a conceptual model's objects and actions according to these two hierarchies—type and containment—greatly facilitates the design and development of a coherent, clear user interface.

### *An example of a conceptual model*

If the software to be developed was an application to help people manage their checking accounts, the conceptual model should include objects like checks, accounts, and amounts of money, and actions like depositing, withdrawing, voiding, and balancing. The conceptual model should exclude non-task-related objects like buffers, dialog boxes, modes, databases, tables, and strings, and non-task-related actions like clicking on buttons, backing up databases, editing table rows, and flushing buffers.

Because computer-based checkbook software sometimes includes capabilities not found in paper checking account registers, some additional concepts

not found in the conventional (i.e., noncomputer) task domain may creep into the conceptual model, for example, objects like transaction templates and actions like defining templates. But it is important to realize that each such additional concept comes at a high cost, for two reasons:


- It adds a concept that users who know the task domain will not recognize and therefore must learn.
- It increases the complexity of the application exponentially because each added concept interacts with many of the other concepts in the application.

Therefore, additional concepts should be strongly resisted, and admitted into the design only when they provide high benefit and their cost can be minimized through good user interface design.



### **An extraneous concept: Save**

One concept that developers often add to software conceptual models despite the fact that it doesn't exist in most actual task domains is an explicit action for saving the results of the user's work. When a person writes or draws on a physical piece of paper, there is no need for the person to do anything to save his or her work. Computer software has been adding Save actions to software conceptual models for so long that frequent computer users now consider it to be natural to most task domains. It isn't.

Software developers add Save actions to conceptual models partly because doing so gives users a way to back out of changes they have made since the last Save. However, the ability to back out of changes could also be provided in other ways, such as (1) making all changes reversible, or (2) automatically creating back-up versions of users' data. This is in fact how most computer file managers operate: when users move data files and folders from place to place, they do not have to save their changes. Users back out of changes by simply reversing the operations. Why are file managers designed differently than, say, most document editors? Tradition and habit, nothing more. 

### **Develop a lexicon**

A second component of a conceptual model is a *lexicon* of terminology to be used in the product or service and its documentation. Once the team agrees what each user-visible concept in the software is, the team should also agree on what to call that concept. The lexicon is best managed by the team's head technical writer. As the software is developed and the corresponding documentation written, it is the lexicon manager's role to make sure the terminology that appears in the documentation and in the software is consistent. For example:

"Hey, Bill. We called this thing a 'widget' in this dialog box, but we call it a 'gadget' in this other dialog box. Our official name for them is 'widgets,' so we need to correct that inconsistency."<sup>2</sup>

The goal is to avoid referring to objects or actions with different names in different places in the software or documentation, and also to avoid having distinct objects or actions being referred to with the same name. It is also the lexicon manager's role to be on the lookout for user-visible concepts in the software or documentation that aren't in the lexicon, and to resist them. For example:

"Hey, Sue, I see that this window refers to a 'hyper-connector.' That isn't in our conceptual model or lexicon. Is it just the wrong name for something we already have in our conceptual model, or is it something new? And if it's something new, can we get rid of it, or do we really, *really* need it?"

### **Write task scenarios**

Once a conceptual model has been crafted, it should be possible to write scenarios depicting people using the application, using only terminology from the task domain. In the case of the checkbook application, for example, it should be possible to write scenarios such as "John uses the program to check his checking account balance. He then deposits a check into the account and transfers funds into the checking account from his savings account." Note that this scenario refers to task domain objects and actions only, not to specifics of any user interface.<sup>3</sup>

The *user interface design* translates the abstract concepts of the conceptual model into concrete presentations and user actions. Scenarios can then be rewritten at the level of the user interface design. For example: "John double-clicks on the icon for his checking account to open it. The account is displayed, showing the current balance. He then clicks in the blank entry field below the last recorded entry and enters the name and amount of a check he recently received...."

### **Extra benefit: implementation guide**

A side benefit of developing a conceptual model is that the model can help guide the implementation because it indicates the most natural hierarchy of implementation objects and the methods each must provide. A conceptual

- 
2. In an ideal development organization, Bill the programmer would have little to do with reconciling labels. All labels would be in a separate file, which the writer would translate to proper terminology. However, in many development organizations, messages are often still embedded in source code.
  3. It is also possible to begin writing task scenarios as soon as developers have performed a task analysis (i.e., before devising a conceptual model), then translate them to the conceptual model lexicon when it is ready.

model can also simplify the application's command structure by allowing designers to see what actions are common to multiple objects and so can be designed as generic actions (see Section 1.3.4).

For example, imagine a software application in which users can create both Thingamajigs and Doohickeys. If Thingamajigs and Doohickeys are related, similar if not identical user interfaces can be used to create them. Thus, after learning how to create one of them, users will already know how to create the other. Similarly, copying, moving, deleting, editing, printing, and other functions might have similar UIs for both Thingamajigs and Doohickeys.

This, in turn, makes the command structure easier for users to learn because instead of a large number of object-specific commands, a smaller number of generic commands apply to many kinds of objects.

### ***Summary: Benefits of developing a conceptual model***

Starting a design by devising a conceptual model has several benefits:

- By laying out the objects and actions of the task domain, it allows designers to notice actions that are shared by many objects. Designers can then use the same user interface for operations across a variety of objects. This, in turn, makes for a user interface that is simpler and more coherent, and thus more easily mastered.
- Even ignoring the simplification that can result from noticing shared actions, devising a conceptual model forces designers to consider the relative importance of concepts, the relevance of concepts to the task domain (as opposed to the computer domain), the type hierarchy of objects, and the containment hierarchy of objects. When these issues have been thought through before the user interface is designed, the user interface will be "cleaner"—simpler and more coherent.
- A conceptual model provides a product lexicon—a dictionary of terms that will be used to identify each of the objects and actions embodied in the software. This fosters consistency of terminology, not only in the software, but also in the accompanying documentation. Software developed without such a lexicon often suffers from (1) a variety of terms for a given concept, and (2) using the same term for distinct concepts. For examples, see Blooper 33: Inconsistent terminology (Section 4.1.1).
- A conceptual model allows the development team to write, at a level of description that matches the target task domain, imaginary scenarios of the product in use. Those scenarios are useful in checking the soundness of the design. They can be used in product documentation, in product functional reviews, and as scripts for usability tests. They also provide the basis for more detailed scenarios written at the level of detail of the eventual user interface design.

- Finally, developing a conceptual model provides a first cut at the object model (at least for the objects that users will be aware of), which developers can then use in implementing the software. This is especially true if the developers are coding the software in an object-oriented language.

For more detail on how to construct a conceptual model, see Dayton et al. [1998]. Most of the industry standard style guides for specific GUI platforms also provide advice on devising conceptual models before designing a GUI for a planned product or service.

## 1.3 Principle 3: Conform to the users' view of the task

Computer software and appliances should be designed from the users' point of view. Obviously, developers cannot do that if they don't know what the users' point of view is. The best way to discover the users' point of view is to talk with representative users, observe their work, and collaborate with them to perform a task analysis and develop a conceptual model, as described in Principles 1 and 2.

Conforming to the users' point of view has several subprinciples, which I'll discuss in turn.

### 1.3.1 Strive for naturalness

One important benefit of performing a task analysis before beginning to design is that it provides an understanding of what activities belong "naturally" to the target task domain and what activities are extraneous, artificial, "unnatural." Stated simply, the goal is to avoid forcing users to commit "unnatural acts."

#### *Don't make users commit unnatural acts*

"Unnatural acts" is my term for steps users have to perform to get what they want that have no obvious connection to their goal. Software that requires users to commit unnatural acts seems arbitrary, nonintuitive, and amateurish to users because unnatural acts are difficult to learn, easy to forget, time-consuming, and annoying. Unfortunately, many computer-based products, services, and appliances force users to commit unnatural acts.

#### *An example: Playing chess*

As an example of how performing a task analysis can help clarify what actions are natural, consider the game of chess. An important action in playing chess is



making a move, that is, moving a chess piece to a new board position. In order to move a piece, what must be specified? Think about this for a few seconds. Answer the question in your own mind first, then read on.

Moving a piece in a chess game requires indicating (1) which piece is to be moved, and (2) where it is to be moved. I'm not talking about the user interface of a chess program. I'm talking about the task of playing chess, whether it is played on a computer, across a table on a wooden chess board, by mail, or by fax. Wherever, whenever, and however chess is played, moving a piece requires specifying the piece to be moved and where it is to be moved.

Now, let's consider a computer chess program. If a chess program requires users to specify *anything* other than the piece to be moved and the destination square, it is requiring unnatural acts. What sorts of unnatural acts might a computer chess program require? Here are some:

- *Switching to command mode in order to be able to give the Move command.* The software might have a mode for specifying moves and a mode for typing messages to the other player. If the software is always in one or other of these modes, it is a safe bet that users will often forget to switch modes, and either type Move commands when in message mode, or type messages when in command mode. For more on modes, see Blooper 51: Unnecessary or poorly marked modes (Section 5.3.2).
- *Stating the reason for the move.* Perhaps the software requires users to record their reasoning for each move, to provide a record that can be used in later postmortem analyses of the outcome.

- *Assigning a name to this move.* Perhaps the software requires users to give each move a name so that there is a way to refer back to that move later, or to allow the move to be saved for reuse.
- *Specifying which of several ongoing chess games this move is for.* Perhaps the software allows users to play several different games with different opponents at once, but provides only one place on the screen for specifying moves. The user must therefore identify the game in addition to the piece and the destination.

The bottom line is that moving a piece in a chess game should be a simple operation, but software can easily make it more complicated by adding extra, “unnatural” steps. Similar analyses can be performed for the other operations provided by a chess program. The ideal is for *all* operations in the program to be as free as possible of actions that are foreign to playing chess.

### ***Other examples***

Software developers should perform this sort of analysis on every operation of every product or service they develop. Here, for example, are some operations for widely used products and services that might benefit from an analysis of what actions the user must specify to complete each listed action:

- *Automatic teller machine:* withdraw money, deposit money, transfer money, check balance
- *Checkbook accounting software:* record check, record deposit, record transfer, reconcile with bank statement
- *Document editor:* create new document, open existing document, find text in document, print document
- *Email program:* retrieve new messages, view received message, save message, print message, compose message, add attachment to message
- *Web shopping service:* open account, close account, find product, order product, download product, submit comment or complaint
- *Airline reservation system:* find flights on required dates, check availability of seats on flights, check ticket price, check restrictions, make reservation

### ***Imposing arbitrary restrictions***

Another way in which computer-based products and services can violate users’ sense of naturalness and intuitiveness is by imposing arbitrary or seemingly arbitrary restrictions on users. Examples of such restrictions include

- limiting person names to 16 characters
- allowing table rows to be sorted by at most three columns



- providing Undo for only the last three actions
- forcing all address book entries to specify a fax number even though some people don't have fax machines

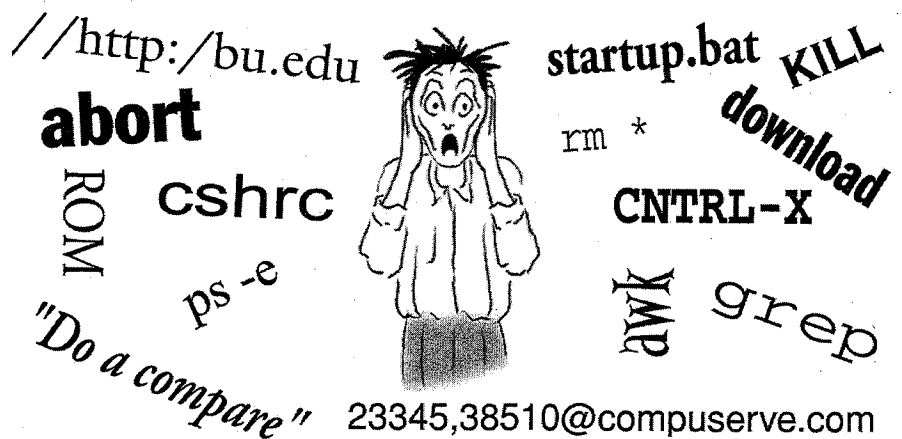
Arbitrary restrictions, like unnatural acts, are hard for users to learn, easy for them to forget, and annoying. A product with many arbitrary restrictions won't have many satisfied users. Obviously, size and length restrictions are more bothersome the more users bump into them. If a limit is so large that users never encounter it, it isn't really a problem. For examples of bloopers involving arbitrary restrictions, see Blooper 43: Exposing the implementation to users, Variation B (Section 5.1.1).

### 1.3.2 Use the users' vocabulary, not your own

Computer-based products and services are infamous for being full of technobabble—jargon that computer engineers understand but most users do not. Even when words in computer software come from the users' standard vocabulary, they are often redefined to have specific technical meanings; users don't understand this either. An overabundance of technobabble is one of the enduring shames of the industry. Figure 1.3 shows some examples of the technobabble that computer users are exposed to. For more examples, see Blooper 35: Speaking Geek (Section 4.1.3).

When writing text for the software or its documentation, avoid computer jargon. Developing a conceptual model for the product or service produces, among other things, a project lexicon. The lexicon should name each and every concept (object or action) that will be seen by users. The terms in the lexicon should match those used in the task domain. Once the lexicon has been developed, text in the software or in the documentation should adhere strictly to it.

Figure 1.3



### 1.3.3 Keep program internals inside the program

Software users are not interested in how the software works. They just want to get their work done. Details of the software's internal workings should therefore remain internal—out of sight and out of mind of the users.

The user interface of a computer-based product or service should represent the concepts (objects and actions) of the users' target task domain, and *only* those concepts. All other concepts, especially concepts from the implementation that are not in the target task domain, are extraneous. General computer technology concepts are also extraneous, since they are almost certainly not part of the target task domain (unless the software is a programming tool).

Principle 2: Consider function first, presentation later (Section 1.2), recommends developing a conceptual model before designing a user interface. Part of developing a conceptual model is analyzing the objects and actions of the target task domain. The result of such an analysis is an outline listing all objects in the task domain, the relationships between objects, the attributes of objects, and all actions on those objects. This outline provides an excellent filter for deciding which concepts to expose to users; if a concept isn't listed in the conceptual model, it shouldn't be in the user interface.

### 1.3.4 Find the correct point on the power/complexity trade-off

There is usually a trade-off between power and usability. For every feature, function, or capability in a computer-based product or service, there must be a way for users to invoke or control it. Unfortunately, in the computer industry, there is too much of an emphasis on lengthy feature lists, and not enough recognition of the price that one pays for power.

Computer programmers tend to believe "the more options, the more controls, the more power, the better." In contrast, most people who use computer products and services want just enough power or functionality to do their work—no more, no less. Most users learn to use only a small portion of the features of any given software product or service, and ignore many—if not most—of the software's other features. Sometimes users ignore a software feature because they regard it as too complicated to be worth learning. Sometimes the problem isn't the complexity of the feature itself, but rather a matter of sheer quantity of stuff to learn. The user sees the feature as "just one more thing to learn" and is not sufficiently motivated to do so. The mere presence of less important features can make more important ones harder to find and use, causing common and simpler tasks to become harder.

The problem for software designers is finding the optimal point on the trade-off between power and complexity, or devising the software so that users can set that point themselves. Clearly, in order to do that, developers must talk

Name	Phone	FAX
Ramon	321-4465	320-446
Granger	321-6745	320-674
Petty	322-0474	332-047
Granson	322-4839	320-483
Peculia	321-4738	320-473
Washoe	334-5748	333-574
Morgana	321-4783	320-247
Applegate	322-4739	332-473
Magpie	321-4837	320-483

Font Family:	Helvetica
Boldface:	<input type="checkbox"/> Yes <input checked="" type="checkbox"/> No
Size:	<input checked="" type="checkbox"/> 10 <input type="checkbox"/> 12 <input type="checkbox"/> 18 <input type="checkbox"/> 14 <input type="checkbox"/> 36 <input type="checkbox"/> 48
Slant:	<input checked="" type="checkbox"/> Italic
Color:	Black

Blood Pressure:  Heart Rate:

CO<sub>2</sub>:  H<sub>2</sub>O:

8.8 8.9 9.2 9.4 9.6 9.8 10.0 10.2 10.4 10.6 10.8

Hightower NPR Jazz Rush Buchanan

FM 12:00 AM

Low High

Volume

<input checked="" type="checkbox"/> Cheese	<input checked="" type="checkbox"/> Mushrooms
<input type="checkbox"/> Onions	<input checked="" type="checkbox"/> Garlic
<input type="checkbox"/> Sausage	<input type="checkbox"/> Pepperoni
<input checked="" type="checkbox"/> Pesto	<input type="checkbox"/> Anchovies
<input type="checkbox"/> Olives	

Order

ITEM	JAN	FEB	MAR	A
Car	202	202	202	
Rent	1003	1003	1003	1
Groc.	763	236	305	
Fun	5262	3582	4532	2
Cats	1349	0	0	
Util.	156	122	99	

with and observe representative users, maybe even bring some onto the design team. Otherwise, developers are just guessing.

Once developers have learned how much functionality users need, they can also use one or more of these important design techniques for reducing complexity.

- **Sensible defaults:** Make sure that every setting in an application has a default value. Users should be able to leave all or most of the settings at their default values and still get a reasonable result. This allows users to ignore most of the settings for most of what they do with the application.
- **Templates or canned solutions:** Instead of making users start every task from scratch, provide partially or fully completed solutions for users to choose from and then modify to satisfy their specific goals. This approach, like sensible defaults, allows users to simply bypass most of the software's functionality.

Users can get useful results without even knowing how to produce results from scratch.

- **Progressive disclosure:** Hide detail and complexity until the user needs it. One way to do this is to deactivate controls and not display menubar menus until they are relevant. A second way is to hide seldom-used settings, or controls requiring advanced knowledge of the software, under auxiliary panels labeled "Details" or "Advanced." A third way to hide detail—and hence to provide progressive disclosure—is to assign names to *combinations* of settings, allowing users to work with the named combinations instead of all the individual settings. For example, Macintosh users would face an overwhelming selection of operating system "extensions," except that the MacOS Extension Manager lets users name different combinations of extensions (e.g., "Basic Mac Extensions," "My Normal Extensions") and turn entire named combinations on or off at once. Progressive disclosure is discussed more fully under Blooper 46: Overwhelming users with decisions and detail (Section 5.2.1).
- **Generic commands:** Use a small set of commands to manipulate all types of data objects. A carefully chosen set of generic commands, mapped onto all of the types of data objects in an application, can provide most of the required functionality while giving users the perception that there isn't much to learn. Generic commands that have been used successfully are Create, Open, Move, Copy, Save, Delete, Print, Show Properties, and Follow Link.

Almost anything people do with computers can be expressed in terms of these nine commands. Some products have even simplified the set further. For example, the Xerox Star didn't have a Print command; files were printed by Copying them to a printer icon. The alternative to generic commands is a separate set of commands for each type of data object that users can manipulate using the software, greatly increasing the amount that users have to learn in order to use the software productively. As described under Principle 2 (Section 1.2), starting the design process by developing a conceptual model of the application can show designers which actions are common to multiple objects and so are candidates for generic commands.

- *Task-specific design*: Support a very limited range of tasks very well. Instead of offering users big feature-rich programs that attempt to support a wide range of tasks, offer them a collection of small specialized programs, each of which supports one task extremely well. For example, instead of developing a single document editor that can be used for creating everything from song lyrics and to-do lists to corporate financial reports and presentation slides, develop many small, simple editors, such as a song lyric editor, a to-do list editor, a corporate financial reports editor, and a presentation slide editor. This approach has been used successfully for household tools and appliances, including information appliances. It is even more successful when task-specific appliances and software can transfer information to and from each other. For a more complete discussion of the advantages and disadvantages of task-specific software applications, see the articles by Nardi and Johnson [1994] and Johnson and Nardi [1996].

## 1.4 Principle 4: Don't complicate the users' task

An important design principle is that the users' task—whatever it is that the user is trying to do—is complicated enough; the computer shouldn't complicate matters further. This principle is best explained as two separate subprinciples.

### 1.4.1 Common tasks should be easy

In any task domain, users will have goals ranging from common to rare. Computer software should be designed to recognize this range. If a user's goal is predictable and common, the user shouldn't have to do or specify much in order to get it. On the other hand, it is OK if unusual goals require more effort and specification to achieve.

Here is a more formal way of stating this design principle: The amount users should have to specify in order to get a desired result should not be proportional to the absolute complexity of the desired result. It should be proportional to how much the desired result *deviates* from a standard, predefined result.

To understand this principle, consider the following: If a man goes into a restaurant every day for dinner and always orders the same thing, he can just say, "I'll have the usual." Perhaps he usually has a hamburger. Perhaps he usually has a wilted spinach and belgian endive salad topped with organic walnuts and curried free-range chicken and a basil-artichoke mustard dressing. Whether his usual dinner is simple or complicated, he can just order "the usual." Furthermore, if on a particular day he wants a little change from his usual meal, he can specify it as a change, for example, "I'll have the usual with the mustard on the side." However, if he ever decides to eat something out of the ordinary, he's got to tell the waiter what menu item he wants and specify all of that menu choice's options.

Returning now to computer software, if someone is preparing a presentation and wants simple, mostly word slides with a few business graphics, all conforming to the company's standard format, the slide preparation program should make that a snap—just plug in the content and the slides are done. However, it may take significant work—including help from graphics experts—to prepare a presentation that includes fancy dissolves, animations, or other visual effects, or that doesn't conform to the standard format.

There are several ways to make common tasks easy. Two of them were already mentioned under Principle 3 (Section 1.3) as ways of reducing user interface complexity: providing sensible defaults, and providing catalogues of templates or "canned" solutions. Both approaches allow users to get a lot by specifying only a little. Two additional important techniques for making common tasks easy are

- *Support customization:* Allow users—or developers local to the users' organization acting on behalf of users—to set their own defaults, define macros, create templates, and otherwise customize the software for their specific requirements.
- *Provide wizards:* Give users the option of doing tasks using multipage dialog boxes that guide users step by step through otherwise complicated processes, with copious instructions and choices specified mainly through menus instead of type-in fields.

The bottom line is that users should be able to get a lot without specifying much. Do a little; get a lot. That's what users want.

### 1.4.2 Don't give users extra problems to solve

The human mind is amazingly good at multitasking. It allows us to handle many different tasks at once, for example, carrying on a phone conversation while beating an egg while keeping watch on our child while planning a pending vacation, all while tapping a foot to a song we heard on the radio this morning.

However, our ability to multitask is pretty much limited to activities that are well learned or based on perceptual and motor skills. One way to categorize activities that we can multitask is “stuff we already know how to do.” In contrast, working out solutions to novel problems is one activity that human minds cannot multitask effectively. Problem solving—which can be regarded as “stuff we don’t already know how to do”—requires concentration and focused attention. We’re pretty much limited to solving one novel problem at a time.

People have plenty of their own problems to solve and goals to achieve in the domain of their work, their hobbies, and their personal lives. That is why they use computer products and services: to solve those problems and achieve those goals. They don’t need or want to be distracted from those problems and goals by extra ones imposed by computer products and services. Unfortunately, computers often require users to stop thinking about their real problems and goals, and think instead about some computer-related problem. For example:

- A student wants to put a picture on his Web site, but the picture is in TIFF format rather than the required GIF or JPEG graphics format. His graphics software won’t convert images from TIFF to GIF or JPEG; it will only convert from BMP format to GIF. He checks around the dorm to see if anyone else has a graphics program that will perform the desired conversion, or at least convert from TIFF to BMP.
- A manager has installed a new program on her computer, but it won’t work. The installation instructions say that the program may be incompatible with software accessories already on the computer. She begins turning off the accessories one by one in an attempt to figure out which one is conflicting with her new software.
- An author is using his computer to type a chapter of a book. While he is typing, a message suddenly appears on the screen saying, “Out of memory. Try closing some other applications before continuing.” No other applications are open. He stares at the computer screen, trying to figure out what to do.
- A stockbroker is using her computer to check the status of a client’s stock order. To search for the order, the investment-tracking software wants her to specify an “Instrument ID.” She thinks this is probably the same thing as the “Stock Symbol” she specified when she first recorded the order, but she isn’t sure. She asks a coworker.

Computer-based products and services should be designed to let users focus their attention on their own problems and goals, whatever they may be: analyzing financial data, looking up job prospects on the Web, keeping track of relatives’ birthdays, and so on. Software should be designed so as to support activity—including problem solving—in the software’s target task domain, but it should minimize or eliminate the need for users to spend time problem solving in the domain of computer technology.

Minimizing the need for problem solving in the domain of computer technology includes not requiring users to figure out how software works by a process of elimination. Users should not have to go through thought processes such as the following:

- "I want page numbers in this document to start at 23 instead of 1, but I don't see a command to do that. I've tried the 'Page Setup' settings, the 'Document Layout' settings and the 'View Header and Footer' commands, but it isn't there. All that's left is this 'Insert Page Numbers' command. But I don't want to *insert* page numbers; the document already has page numbers. I just want to change the number they start at. Oh well, I'll try 'Insert Page Numbers' because that's the only one I haven't tried."
- "Hmmm. This checkbox is labeled 'Align icons horizontally.' I wonder what happens if I uncheck it. Will my icons be aligned vertically, or will they simply not be aligned?"
- "This online banking service is asking me for a 'PIN number.' But the card they sent me has a 'password.' I wonder if that's it? It must be, because they haven't sent me anything called a 'PIN number.'"

The function of controls, commands, and settings in a user interface should be clear and obvious. Operating computer software should not require deductive reasoning. To the extent that it does, it distracts users from their own tasks and goals.

## 1.5 Principle 5: Promote learning

A frequent complaint about computer-based products and services is that they are too difficult to learn. Learning takes time; the more a user has to learn in order to use a product or service, the longer it will be before that user can be productive. Time is money. Furthermore, if a user is not *required* to learn how to use a particular product or service (e.g., because of his or her job), the user may simply decide not to bother; there are plenty of other products (or services) to choose from.

The user interface of a software product or service can promote learning in several different ways, described in each of the following subprinciples.

### 1.5.1 Think "outside-in," not "inside-out"

Developers of computer-based products often design as if they assume that the users will automatically know what the developers intended. I call this thinking "inside-out" instead of "outside-in." When a software developer has designed

software, he or she knows how it works, what information is displayed in what place and at what time, what everything on the screen means, and how information displayed by the software is related. Most designers think inside-out: they use their own knowledge of the software to judge whether the displays and controls make sense. They assume that users perceive and understand everything the way the designer intended it to be perceived and understood.

The problem is, users *don't* know what the designer knows about the software. When people first start using a software product, they know very little about how it works or what all that stuff on the screen is supposed to mean. They are not privy to the designer's intentions. All they have to base their understanding on is what they see on the screen and, in some cases, what they read in the software's documentation.

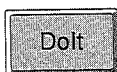
### **Examples: Textual ambiguity**

Thinking inside-out is a problem that is not limited to software developers. Newspaper headline writers sometimes commit the same error when they fail to notice that a headline they've written has interpretations other than the one they intended. Some examples:

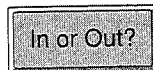
Crowds Rushing to See Pope Trample 6 to Death  
 Drunk Gets Nine Months in Violin Case  
 Vineland Couple to Take on Missionary Position  
 Teacher Strikes Idle Kids  
 British Left Waffles on Falkland Islands  
 Iraqi Head Seeks Arms  
 New Vaccine May Contain Rabies  
 Two Soviet Ships Collide, One Dies  
 Farmer Bill Dies in House

### **Examples: Typographical ambiguity**

A famous example of textual ambiguity in a computer system—and thus inside-out thinking by the designers—is a LISP workstation, circa 1985, that had a key on its keyboard labeled “DoIt”, as in “do it”. The key was labeled in a sans serif font, in which lowercase *L* and uppercase *I* characters looked alike. The key is shown in Figure 1.4. The designers of this keyboard apparently assumed that users would read the label as intended, but predictably, some users read it as “Dolt”—D-O-L-T—and wondered why anyone would press that key.



**Figure 1.4**



**Figure 1.5**

### **Examples: Graphical ambiguity**

Ambiguity need not be textual. It can also be graphical. For example, is the button shown in Figure 1.5 poked out or pressed in?



It turns out that whether you see it as poked out or pressed in depends on your assumption about where the light source is. If you assume that the light is coming from the upper left, the button looks poked out. If you assume that the light is coming from the lower right, the button looks pressed in. In fact, you can make the button flip back and forth between poked out and pressed in by mentally switching the light source from upper left to lower right.

Window-based operating systems often use shading effects to create an illusion of three-dimensional controls on the screen of a computer. However, for users to perceive such images in the proper orientation—or even in 3D at all—they must imagine that the light is shining from a particular direction. In GUIs, the standard location of the imaginary light source is somewhere to the upper left of the screen.

People don't automatically perceive these simulated 3D displays as the designers intended; they have to *learn* to do that. People who have never encountered such a display before may either perceive no 3D effects or may perceive them in the “wrong” orientation. However, many designers of simulated 3D displays have known for so long that the imaginary light in GUIs comes from the upper left that they have forgotten that they had to learn that. They believe everyone else will just automatically know where the light is coming from and see the 3D controls as the designers intended. They are thinking inside-out, and they are wrong.

Similarly, graphic designers are thinking inside-out if they assume that an icon they've drawn will “naturally” convey the intended meaning to users. Designers at one of my client companies were surprised when usability testing showed that most test participants thought an antenna symbol for a Transmit function was a martini glass with a swizzle stick in it (see Figure 1.6).

Similarly, a colleague told me: “There is an icon in Lotus Notes that everyone at my company refers to as ‘the lollipop.’” Further examples of graphic designers thinking inside-out are given in Section 10.1.

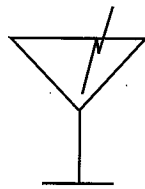


Figure 1.6

### *The right way to design: Think outside-in*

Thinking outside-in requires assessing the meaning of the displays and controls based on what a user can be assumed to know at that point. It is a part of what it means to design from the user's point of view. It does not mean assuming that users are stupid. The users, in fact, probably know much more about the tasks that the software is intended to support than the developers do. What users don't know is the software. They don't know the meaning of the various sections of the display. They don't know what is dependent on what.

The bottom line is that if the user interface of the software that you develop is ambiguous—textually or graphically—and users misinterpret it, they aren't the real losers; you are. If your intended users misperceive or misunderstand your design, they may have an immediate problem, but you will have the more serious and longer-term problem: unsatisfied users, resulting in diminished sales.

### 1.5.2 Consistency, consistency, consistency, but don't be naive about it

Computer-based software should foster the development of habits. When using interactive software and electronic appliances, users want to fall into unconscious habits as quickly as possible. They want to be able to ignore the software or device and focus on their work. The more consistent the software is, the easier it is for users to do that.

#### *Consistency avoids "gotchas"*

Software that is full of inconsistencies, even minor ones, forces users to keep thinking about it, thereby detracting from the attention they can devote to the task at hand. I characterize this sort of software as being full of "gotchas": it's constantly saying to the user "Aha! Gotcha! Stay on your toes, buster, or I'll getcha again." Designers should try to minimize the number of "gotchas" in their software by striving for a high degree of user interface consistency.

When beginning a design, develop a conceptual model and perform an object/action analysis to expose operations that are common to many different types of objects. This allows the design to make use of generic commands (see Principle 3, Section 1.3.4), or at least to use highly similar user interfaces for similar but distinct functions. When adding new features to a product or service, reuse user interfaces from other parts of the software instead of inventing new user interfaces for the specific new feature.

The alternative is to provide a bewildering array of different ways of doing more or less the same thing in different contexts. For example, Figure 1.7 illustrates the large number of different ways found in computer-based products for deleting items; the method of deletion depends on the type of item to be deleted.

#### *Dangers of trying to be consistent*

Exhorting software developers to make their user interfaces "consistent" is somewhat risky; it could do more harm than good. Why? Because consistency is a more complex concept than many people think it is. In particular, it is

- *Difficult to define:* Many experts have tried without success.
- *Multidimensional:* Items that are consistent on one dimension (e.g., function) may be inconsistent on another (e.g., location).
- *Subject to interpretation:* What seems consistent to one person may seem inconsistent to another.

Many designers have naive notions of consistency, and either are unaware that users might see things differently or arrogantly believe that they can define con-

Figure 1.7



sistency for users. Some astonishingly bad designs have been produced in the name of consistency. Examples include software applications in which everything is controlled by forms or by hierarchies of menus even though forms or menus may not be appropriate for controlling certain functions. Grudin [1989] even goes as far as to suggest that consistency is such an ill-defined concept that it should be abandoned as a user interface design principle.

### *Why I advocate consistency anyway*

While there are indeed dangers in advocating consistency in user interfaces, I don't advocate abandoning the concept. Just because theorists haven't yet found a formal definition of consistency doesn't mean that the concept is useless. It clearly has value to users, even though their ideas of what is and is not consistent may not match those of user interface designers. Users search for consistency along the dimensions that are relevant to them. They are so anxious to disengage their conscious mind from the task of controlling the computer—so they can apply it to their own problems—that they make up consistency even when it is absent.

For example, programmers argue: "Yes, most items on this computer are opened by double-click, but this application is different because items can only be opened, not selected, so a single-click should open them." So they design it that way, and users of the new application double-click to open items anyway and are annoyed when items often open accidentally. Similarly, the designers of the Apple Macintosh decided that it made sense for dragging a file within a disk to be a Move operation, whereas dragging it between disks is a Copy operation, and—surprise—Mac users complain about not knowing when their files will be moved versus copied.

Users gladly expend physical effort in controlling the computer to reserve mental effort for working on their own tasks. For example, while being observed using a software application developed at a company where I once worked, a user said:

*I was in a hurry so I did it the long way.*

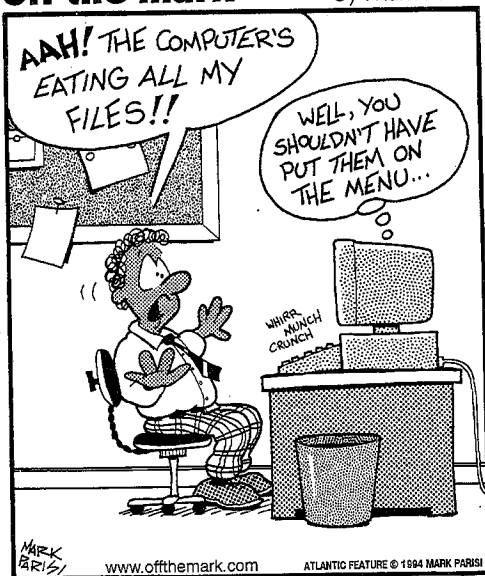
### Making consistency user-centered

Instead of abandoning consistency as a user interface design goal, we need to refine it to make it more user-centered. Therefore, when interviewing or observing users to understand them, the target tasks, and the work environment, software designers should try to determine how the users perceive consistency. What aspects of the users' current tools seem consistent and inconsistent to them?

When sketches or other prototypes of the new software are available, they should be tested on representative users, and developers should be on the lookout for aspects of the user interface that the users perceive as inconsistent. The bottom line is that the consistency of a user interface should be evaluated based not on how "logical" it seems to designers and developers, but rather on how predictable it is to users.

### off the mark

by Mark Parisi



### 1.5.3 Provide a low-risk environment

An important fact to consider when designing an interactive product or service is that people make mistakes. A product or service that is risky to use is one that makes it too easy for users to make mistakes, does not allow users to correct their mistakes, or makes it costly or time-consuming to correct mistakes. People won't be very productive in using it; they will be wasting too much time correcting mistakes. Such a product or service will not be popular.

Even more important than the impact on time is the impact on users' learning. A high-risk situation discourages exploration; people will tend to stick to familiar, safe paths. When exploration is discouraged, learning is severely hampered. A low-risk situation, in which people don't have to worry about mistakes—either

because they are hard to make or are easy to correct—encourages exploration and hence greatly fosters learning. In such situations, users aren't hesitant to try unfamiliar paths: "Hmmm, I wonder what *that* does."

## 1.6 Principle 6: Deliver information, not just data

A common saying about computers is that they promise a fountain of information, but mainly deliver a glut of data ... most of it useless. Data is not necessar-

ily information. In particular, the information in data is what the recipient learns from examining it.

If I want to know whether my colleague Bill is in his office or not, all I want is one bit of information: yes or no, true or false, 1 or 0. But I can get the answer to my question by a variety of methods, involving the transfer of widely different amounts of data. For example, I can

- knock on the wall separating our offices and listen to hear whether Bill knocks back (one bit)
- send him email and wait to see if he sends a reply (several hundred bytes, including the mail headers)
- call him on the intranet phone and listen to see if he answers (several kilobytes)
- visit his Web site and download an image of him that is updated once a minute from the video camera in his office (several megabytes)

Regardless of the method I use and the amount of *data* transferred, I receive only one bit of *information*, assuming that all I care about is the answer to my question.

Historically, computer-based products and services have had an unfortunate tendency to treat data as if it were the same thing as information: they put it all in your face and make you find the information in it.

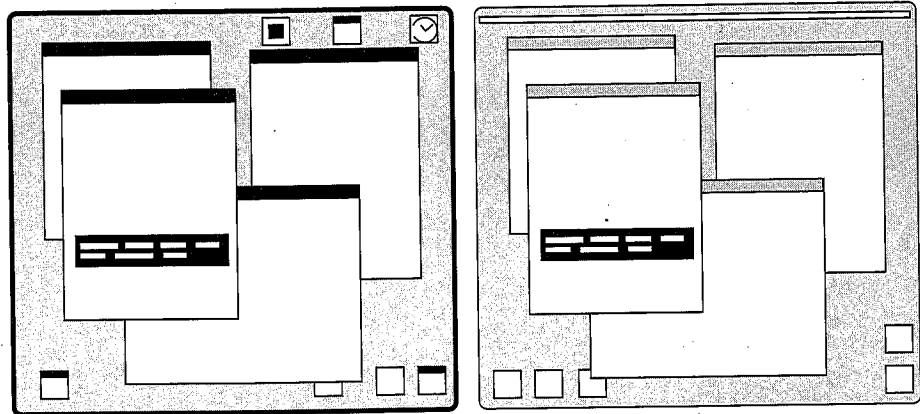
Software designers need to counter this tendency by designing interactive software so that it focuses users' attention on the important information and does not distract them from it. That is what this principle and its various sub-principles are about.

### 1.6.1 Design displays carefully; get professional help

Principle 2 (Section 1.2) says, "Consider function first, presentation later." There comes a time, however, in every software development effort when it is necessary to consider presentation. I am talking about the presentation of the software's data, of the data's properties and relationships, and of the software's controls and status. When the time comes to consider presentation, designers should consider it very seriously and carefully. That means trying to achieve the following:

- *Visual order and user focus*: One important feature of a successful presentation is that it doesn't simply present. It *directs* users' attention toward the important information. For example, find the selected text in each of the computer screens shown Figure 1.8. The large amount of contrast present on the screens of many window-based computer systems (shown in the left screen) makes it difficult for users to focus on the relevant information. On the computer screen, the current selection should be the users' main focus;

Figure 1.8



it is the object of the next operation. The screen on the right illustrates how the displays of the Xerox Star and the Apple Macintosh minimize the presence of contrast on the screen in order to focus attention on the current selection.

- *Match the medium:* One mark of a poorly designed user interface is a failure to match the design to the limitations of the medium in which it is presented. The software marketplace has plenty of examples of mismatches: window systems presented on character displays or on two-inch screens, visual metaphors presented aurally through a telephone, point-and-click GUIs on computers that have no pointing device other than arrow keys, using subtle colors or gray shades on a display that cannot render them well. Well-designed user interfaces match the medium in which they will be presented.
- *Attention to detail:* It is often said that success is in the details. Nowhere is this more true than in the design of effective information displays. Hiring user interface and graphic designers for a software development project may seem expensive, but they bring an eye for detail that few other members of the team can provide, and thus easily repay their cost, especially considering the alternative. A lack of attention to detail yields incoherent displays, design inconsistencies, horrendous installation experiences, indecipherable symbols, and a generally cheesy appearance. In short, it yields many of the bloopers described elsewhere in this book. These bloopers can hamper the success of the product or service by giving prospective customers an impression of a shoddy product and also by decreasing the product's usability.

In order to assure that all of the important issues are considered in designing presentations, software developers should make sure they have the right people for the job. You would not hire a plumber—even a skilled one—to repair

your car. Yet many software development teams assign GUI programming to student interns and new hires, or expect GUI programmers to design as well as implement the GUIs, or use icons created by programmers or managers. This sort of corner-cutting is misguided and counterproductive.

Don't assume that just anyone can design information displays well. The people with the necessary skills are user interface designers (sometimes called "interaction designers") and graphic designers: user interface designers for gross characteristics of the display, and graphic designers for the exact details.

Programmers, even GUI programmers, lack the required experience and skills. Though they are professional programmers, they are amateurs in the area of presentation and graphic design. Products and services that rely on graphics drawn or designed by programmers will be perceived by users (and customers) as amateurish, and rightly so. For further discussion of the difference between user interface designers, graphic designers, and GUI programmers, see Blooper 76: Misunderstanding what user interface professionals do (Section 8.1.1) and Blooper 77: Treating user interface as low priority (Section 8.1.2).

### 1.6.2 The screen belongs to the user

This design principle was recognized as far back as the mid-1970s by researchers at Xerox Palo Alto Research Center (PARC). At the time, personal computers with bit-mapped displays, mouse pointers, and windows were still new, and the GUI as we know it today hadn't yet settled down; researchers at PARC were still trying all sorts of different design ideas. Some worked better than others.

Fairly quickly, PARC researchers realized that one thing that did *not* work very well was for the computer—or more accurately, its software—to unilaterally move objects around on the screen. Some researchers had initially assumed that it would be helpful to users and more efficient for software to sometimes move or "warp" the mouse pointer automatically to new positions. Similarly, some researchers tried having their software automatically reposition, stretch, and shrink windows "when appropriate." Although well-meaning, these attempts to be helpful and efficient disoriented and frustrated users more than they helped them. In a well-meaning attempt to help users, the designers had interfered with users' perception of the screen as being under their control.

This is especially true for the screen pointer. Moving the screen pointer is a hand-eye coordination task. After a user has gotten used to using a pointing device such as a mouse, moving the pointer becomes a reflex—an action controlled more by "muscle memory" than by conscious awareness. The users' conscious mind is freed to think about other things besides moving the pointer, such as the task he or she is trying to accomplish. Automatic, unilateral movement of the pointer by the software disrupts the coordination, causing disorien-

tation and yanking the user's conscious mind back to the task of moving the pointer. Users aren't sure which pointer movements are the result of their actions versus those of the computer.

At Xerox PARC, years of experience and observation of users eventually gave rise to the important GUI principle "The screen belongs to the user." Graphical user interfaces are supposed to be based on direct manipulation of data by users, and that is what users expect. When software changes too much on its own initiative, users become disoriented and annoyed.

The principle can be generalized beyond screen pointers and windows to include desktop icons, lists of email messages, and many other sorts of data objects that people manipulate using software. It suggests that, in general, software should avoid trying to "help" users by rearranging their data displays for them. It should let users arrange and manage their data themselves, as they wish.

To assist users in arranging and formatting their data, software can provide "rearrange" or "reformat" commands that users have to invoke explicitly. Examples of such commands are the MacOS Clean-up Desktop command, the Balance or Align functions of some graphics editors, and Microsoft Word's AutoFormat command.

### 1.6.3 Preserve display inertia

Closely related to the principle that "the screen belongs to the user" is the principle of "display inertia."

When software changes a graphical display to show the effect of a user's actions, it should try to minimize what it changes. Small, localized changes to the data should produce only small, localized changes on the display. Another way of stating the principle is that when a user changes something on the screen, as much of the display as possible should remain unchanged.

Failing to localize changes in the display to what has actually changed can be quite disorienting to users. For example, if someone were editing the name of a file displayed in a folder, it would be very disorienting and annoying if the file constantly jumped around to different alphabetical positions as the name was being edited. Therefore, most file managers—wisely—leave the file temporarily where it is until users indicate that they are done by pressing RETURN or moving the selection elsewhere.

When large or nonlocal changes in the display are necessary (such as repaginating a document, or swapping the positions of branches of a family tree diagram), they should not be instantaneous. Rather, they should be announced clearly, and they should be carried out in ways that

- foster users' recognition and comprehension of the changes that are occurring
- minimize disruption to users' ability to continue working



## 1.7 Principle 7: Design for responsiveness

To design software that satisfies its users, designers must of course ask, What do the users want?

Considerable evidence has been amassed over the past four decades of computer use that responsiveness—the software’s ability to keep up with users and not make them wait—is the most important factor in determining user satisfaction with computer-based products and services. Study after study has found this [Miller, 1968; Thadhani, 1981; Barber and Lucas, 1983; Lambert, 1984; Shneiderman, 1984; Carroll and Rosson, 1984; Rushinek and Rushinek, 1986]. The findings of all these studies are well summarized by Peter Bickford in his book *Interface Design* [1997]:

Many surveys have tried to determine what it is about a computer that makes its users happy. Time and time again, it turns out that the biggest factor in user satisfaction is not the computer’s reliability, its compatibility with other platforms, the type of user interface it has, or even its price. What customers seem to want most is speed. When it comes to computers, users hate waiting more than they like anything else.

Bickford goes on to explain that by “speed,” he means *perceived* speed, not actual speed:

Computers actually have two kinds of speed: ... real (machine) speed and ... perceived speed. Of these two, the one that really matters is perceived speed. For instance, a 3-D rendering program that saves a few moments by not displaying the results until the image is complete will inevitably be seen as slower than a program that lets the user watch the image as it develops. The reason is that while the latter program’s users are watching an image form, the first program’s users are staring impatiently at the clock, noticing every long second tick by. Users will say that the first program ran slow simply because the wait was more painful.

Research also suggests that improving the responsiveness of software, in addition to increasing user satisfaction, actually improves the productivity of its users [Brady, 1986].

### 1.7.1 Defining responsiveness and the lack thereof

Responsive software keeps up with users even if it can’t fulfill every request immediately. It prioritizes feedback based on *human* perceptual, motor, and cognitive requirements. It provides enough feedback for users to see what they are doing. It lets users know when it is busy and when it isn’t. It gives users ways to judge how much time operations will require. Finally, it does its best to let users set their own desired work pace.

In contrast, when software is said to exhibit poor responsiveness, it means that the software does not keep up with users. It doesn't provide timely feedback about user actions, so users are often unsure of what they have done or are doing. It makes users wait at unpredictable times or for unpredictable periods. It limits—moderately or severely—users' work pace.

Examples of poor responsiveness include

- delayed or nonexistent feedback for user actions
- time-consuming operations that block other activity and cannot be aborted
- providing no clue how long lengthy operations will take
- periodically ignoring user input while performing internal "housekeeping" tasks

When software manifests these problems, it not only impedes users' productivity, it frustrates and annoys them as well. Unfortunately, a lot of software manifests these problems. Despite all the research showing that responsiveness is very important to user satisfaction and productivity, much of the software in today's marketplace exhibits poor responsiveness.

### 1.7.2 Responsiveness on the Web: A big deal

Responsiveness is an especially important issue on the Web. One reason is that the Web is completely user-driven. A user of a traditional desktop computer application is somewhat "captive" because of the work required to install the application and the additional work that would be required to replace it. On the Web, users are in no way captive. If they don't like a Web site for any reason, they can go elsewhere with a few simple clicks. There is no shortage of sites out there to choose from. If a user gets frustrated with how long it is taking for a Web site to display the desired information, he or she simply hits the Back key and moves on to the next site.

Responsiveness is also important on the Web for a technical reason: the huge difference in the time it takes to execute operations that can be performed entirely on the client side (e.g., displaying pop-up menus in forms or executing Java/Javascript functions) versus operations that require communicating with a Web server (e.g., loading a page). This difference places strong constraints on the design of Web user interfaces. For example, forms on the Web can't check the validity of users' input as frequently as do forms in conventional desktop applications, thereby limiting the immediacy of error feedback.

### 1.7.3 Summary of responsiveness design principles

In addition to describing the many ways software can be designed so as to be unresponsive to its users, Chapter 7, *Responsiveness Bloopers*, provides design

principles and techniques for designing software that is highly responsive. For present purposes, it will suffice to summarize the seven responsiveness design principles that are described fully in Chapter 7. They are:

- *Responsiveness is not the same thing as performance:* Software can be responsive even if its performance—that is, speed of execution—is low; conversely, improving performance does not necessarily improve responsiveness
- *Processing resources are always limited:* Computer users probably have older computers than developers have, and when users do get faster computers, they just load them down with more software
- *The user interface is a real-time interface:* It has time constraints and deadlines that are just as important as the time constraints on software that interfaces with real-time devices
- *All delays are not equal:* Software can and should prioritize its responses to users
- *The software need not do tasks in the order in which they were requested:* Tasks can be reordered based on priority or to improve the efficiency with which the entire set of tasks can be completed
- *The software need not do everything it was asked to do:* Some requested tasks are unnecessary and can simply be ignored, freeing time for the software to perform necessary tasks
- *Human users are not computer programs:* People do not interact with computer software in the same way as other software does, and should not be forced to do so

For examples of design bloopers developers make that contribute to poor responsiveness, and techniques that can help designers avoid responsiveness bloopers, see Chapter 7, Responsiveness Bloopers.

## 1.8 Principle 8: Try it out on users, then fix it!

Most people in the computer industry have heard the saying “Test early and often.” Although there are many different kinds of testing to which computer software and hardware can be subjected, the kind that is relevant to this book is “usability testing”—trying a product or service out on prospective users to see what problems they have in learning and using it. Such testing is extremely important for determining whether a design is successful, that is, whether it helps users more than it hinders them.

### 1.8.1 Test results can surprise even experienced designers

Developers can learn surprising things from usability tests. Sometimes the results can surprise even user interface experts.

I usually review the user interface of software products or services before testing them on representative users. Reviewing the user interface beforehand gives me an idea of how to design the test, what sorts of problems to look for in it, and how to interpret the problems I observe users having. However, conducting the test almost always exposes usability problems I hadn't anticipated.

For example, one of my client companies was developing software for analyzing and predicting the performance of database servers. One feature of the software was the ability to plot the expected performance of one or more servers for a varying number of simultaneous users. To do this, users had to specify not only the characteristics of the database servers they wanted analyzed, but also the type of plot they wanted. The software provided radiobuttons for choosing a plot type. The radiobutton choices were labeled textually, but next to the radiobutton setting was an image panel showing a small thumbnail example of the currently chosen type of plot. What surprised both the developers and me was that the usability tests showed that many users thought the small thumbnail images were the actual data plots for the specified servers! The moral of this story is that it is always useful to test; you never know what you will learn, but you *will* learn something that will help you improve your software.

### 1.8.2 Schedule time to correct problems found by tests

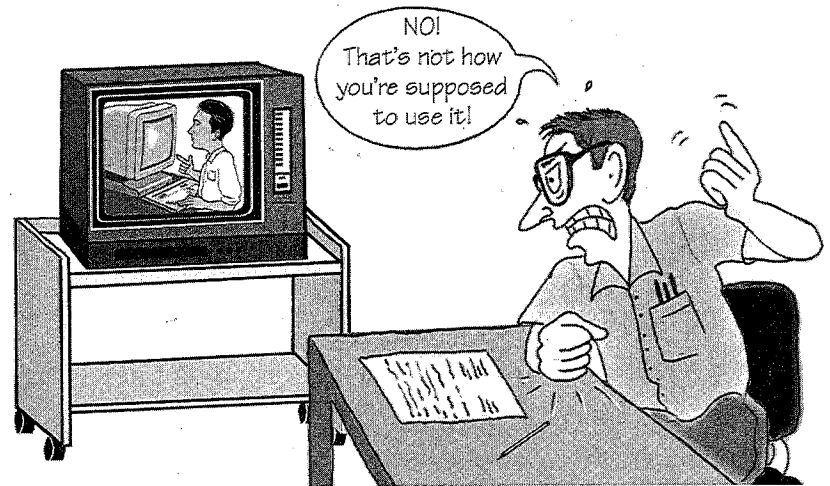
Of course, it isn't enough just to test the usability of a product or service. Developers must also provide enough time in the development schedule to correct problems uncovered by testing. Otherwise, what is the point of testing?

### 1.8.3 Testing has two goals: Informational and social

Usability testing has two important but different goals: one informational, the other social.

#### *Informational goal*

The informational goal of usability testing is the one most people are familiar with: find aspects of the user interface that cause users difficulty, and use the



exact nature of the problems to suggest improvements. This goal can be accomplished by a wide variety of testing and data collection methods, some expensive and time-consuming, some cheap and quick (see Section 1.8.4).

### Social goal

The social goal of usability testing is at least as important as the informational goal. It is to convince developers that there are problems in the design that need to be corrected. Most developers naturally are resistant to suggestions for change, partly because of the time and effort required and partly because of the perception that the need to change a design reflects poorly on the original design and hence on the designer. To achieve the social goal of usability testing, I have found it most effective to either have developers present as passive observers during testing, or to videotape the test sessions and show the tapes—or excerpts from them—to developers. When developers observe tests in person, it is important to stress that they remain *passive* observers because, in my experience, programmers can become quite agitated while watching a user who is having trouble with the programmers' software.

I have found that emphasizing the importance of the social goal of usability testing has benefits beyond convincing developers to make the indicated changes. It also makes them much more accepting of the notion that usability testing is an essential development tool rather than a way to evaluate developers. Some programmers who initially resisted usability testing became "converts" after witnessing a few tests; in later development efforts, they actively sought opportunities to test their designs in order to get feedback.

### 1.8.4 There are tests for every time and purpose

Many people in the computer industry have the mistaken impression that usability testing is always conducted when a software product or appliance is nearly ready to ship, using elaborate testing facilities and equipment. In fact, a wide variety of different sorts of tests can be considered “usability testing,” each having its benefits and drawbacks. I find it useful to categorize usability testing along two independent dimensions: (1) the point in the development process when testing occurs, and (2) the formality of the testing method.

#### *Implementation stage of testing*

First let’s consider the point in development when testing is done. I find it helpful to divide this dimension into three categories:

- *Before development.* Before anything has been implemented, developers can test a design using mock-ups. Mock-ups can be storyboards—hand-drawn sketches of the design on paper that are shown to users in approximately the order they would see them in the running product. They can be screen images created using interactive GUI building tools, printed out onto paper, and placed in front of users with a person acting as “the computer” as the user “clicks” the controls shown on the screens. They can be “cardboard computers” with rear projection screens that show slides simulating the display (see the chapter “Cardboard Computers” in the book by Greenbaum and Kyng [1991]). Finally, they can be “Wizard of Oz” systems in which the display on the computer screen is controlled by a person in another room who watches what the user does and changes the user’s screen accordingly.
- *During development.* A development team can use prototyping software (e.g., Macromedia Director, Hypercard, Toolbook) to create working software mock-ups of an application. GUIs prototyped with such tools would be only prototypes; their underlying code would not be used in the eventual implementation. Alternatively, some GUI building tools allow user interfaces to be partially wired together and “run.” This allows the early versions of the user interface to be tested even before any back-end code has been written. Another way to test during development is to test parts of the planned application in isolation, to resolve design issues. For example, you could test a table widget used in an application long before any of the rest of the application was ready.
- *After development.* Many usability tests are conducted when most or all of the product has been implemented and it is being prepared for release. At this late stage, it is rare for the results of usability tests to have much of an impact on the pending release because the development organization is usually anxious to ship it. The main reason for conducting usability tests at this stage is to provide guidance for a preshipping cleanup of minor usability problems. Nonetheless, such testing can also uncover “showstopper” problems that are deemed important enough to delay the release. Another



DILBERT reprinted by permission of United Feature Syndicate, Inc.

reason for testing “completed” software is to get feedback on how to improve the user interface for *future* releases. This sort of testing can be conducted after a release, when the schedule is not so frantic and the team is beginning to think about the next release.

### Formality of testing

The second dimension along which usability testing can be categorized is the formality of the testing method. Usability test formality has to do with the degree of control the tester exerts over what people do in the test session, and whether the measurements are qualitative or quantitative. Again, I find it helpful to divide this dimension into three categories:

- *Informal testing.* This type of testing includes situations where users are interviewed about the software, for example, how they use it, what they use it for, how they like it. Sometimes this is done in front of a computer with the software running so the user can show what he or she is talking about, and sometimes it isn't. Informal testing also includes situations in which users are observed while they do their real work or while they explore the software in an unguided fashion. The observers note the users' expressed likes and dislikes, and record any problems they notice users having. The session may be videotaped or audiotaped. The observations are qualitative.
- *Quasi-formal testing.* In this type of testing, users are asked to do tasks that have been predetermined by the testers, who also prepared the necessary materials and data files and set up the software as required. In other words, users aren't just exploring the software or doing their own work; they are doing what the tester asks them to do. However, as in informal testing, the measurements are mainly observational. The testers record (and count) any errors users make, as well as situations in which users need help (either from online help documents or from the testers). The testers also record the time required to complete each of the tasks. The session may be videotaped.

It is best to start each test session with simple tasks and give the test participant progressively harder tasks over the course of the session.

- *Formal testing.* This is the type of usability testing that is most similar to the psychology experiments many people remember from their college days. They are true “controlled experiments,” conducted to compare alternative designs (e.g., control layouts A versus B) or to determine an optimal value of a design parameter (e.g., number of cascading menu levels). The tasks that test participants are asked to perform are highly prescribed, and their relation to real-world tasks often seems tenuous (e.g., use the mouse to hit a sequence of randomly positioned targets on the screen; find a file using this file browser). In fact, the materials or software used for the test often are devised purely for the test itself, rather than being part of a pending product. The data collected are mainly quantitative (e.g., user reaction or completion time, number of errors) and are analyzed statistically. Often, the data are collected automatically, by the same computer that is running the test software. In addition, such sessions are usually videotaped.

Each level of formality has its place in product development. The important thing to realize, however, is that the formality level of the test is independent of the point in development when the test is conducted. One can find or devise examples of usability tests at any level of formality at any stage of development.

For examples from my consulting files of usability tests conducted at a variety of development stages and formality levels, see the design rule for Blooper 78: Discounting the value of testing and iterative design (Section 8.1.3).

## Further reading

### *Methodologies for understanding users and tasks*

- Beyer, H., and Holtzblatt, K. 1998. *Contextual Design: Defining Customer-Centered Systems*. San Francisco: Morgan Kaufmann Publishers.
- Dayton, T., McFarland, A., and Kramer, J. 1998. “Bridging User Needs to Object Oriented GUI Prototypes via Task Object Design.” In L. Wood, ed., *User Interface Design: Bridging the Gap from Requirements to Design*, pp. 15–56. Boca Raton, FL: CRC Press.
- Greenbaum, J., and Kyng, M. 1991. *Design at Work: Cooperative Design of Computer Systems*. Hillsdale, NJ: Lawrence Erlbaum Associates.

### *Official platform-specific GUI style guides*

- Apple Computer, Inc. 1993. *Macintosh Human Interface Guidelines*. Reading, MA: Addison-Wesley.
- Microsoft. 1995. *The Windows Interface Guidelines for Software Design: An Application Design Guide*. Redmond, WA: Microsoft Press.



- Microsoft. 1999. *Microsoft Windows User Experience*. Redmond, WA: Microsoft Press.
- Open Software Foundation. 1993. *OSF/Motif Style Guide: Rev 1.2*. Englewood Cliffs, NJ: Prentice Hall.
- Sun Microsystems. 1999. *Java Look and Feel Design Guidelines*. Reading, MA: Addison-Wesley.

### **Platform-independent design guidelines**

- Bickford, P. 1997. *Interface Design: The Art of Developing Easy-to-Use Software*. Chestnut Hill, MA: Academic Press.
- Fowler, S. 1998. *GUI Design Handbook*. New York: McGraw-Hill.
- Mandel, T. 1997. *The Elements of User Interface Design*. New York: John Wiley and Sons.
- McFarland, A., and Dayton, T. 1995. *Design Guide for Multiplatform Graphical User Interfaces*. Issue 3, LPR13. Piscataway, NJ: Bellcore.
- Nielsen, J., 1999d. *Designing Web Usability: The Practice of Simplicity*. Indianapolis, IN: New Riders Publishing.
- Shneiderman, B. 1987. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Reading, MA: Addison-Wesley.
- Weinshenk, S., Jamar, P., and Yeo, S. 1997. *GUI Design Essentials*. New York: John Wiley and Sons.

### **Graphic design guidelines**

- Mullet, K., and Sano, D. 1995. *Designing Visual Interfaces*. Mountain View, CA: SunSoft Press.
- Tufte, E. R. 1983. *The Visual Display of Quantitative Information*. Cheshire, MA: Graphics Press.