TASK-CENTERED USER INTERFACE DESIGN

A Practical Introduction

Clayton Lewis
John Rieman

SHAREWARE NOTICE:

The suggested shareware fee for this book is $5.00, payable to Clayton Lewis and John Rieman. Send it to:

Clayton Lewis and John Rieman
P.O.Box 1543
Boulder, CO 80306 USA.

If sending U.S. dollars is difficult, for example if you aren't in the U.S., send us something else you think we'd like to have. Or send us two somethings, one for each of us.

Original files for the book are available via anonymous ftp from ftp.cs.colorado.edu.

We thank you for your support!

* v.1
*
* Table of Contents
*
*

Contents

List of HyperTopics

List of Examples

List of Tables

Foreword

In this introductory material we explain the book's goals and introduce some basic terminology. In particular, this book is about the design of user interfaces, and it's useful to discuss what we mean by "user interfaces" and why we have decided to focus on the process of their "design."

We also tell you a little about the authors, we acknowledge some people and organizations that have helped with the book, and we tell you about the shareware concept under which the book is distributed.

0.1 What's This Book All About?

The central goal of this book is to teach the reader how to design user interfaces that will enable people to learn computer systems quickly and use them effectively, efficiently, and comfortably. The interface issues addressed are primarily cognitive, that is, having to do with mental activities such as perception, memory, learning, and problem solving. Physical ergonomic issues such as keyboard height or display contrast are covered only briefly.

0.1.1 Who Should Be Reading the Book?

We've designed this book to be most useful for people who are actually developing user interfaces. That's in contrast to the full-time interface professionals who do research and evaluation in large corporations. We strongly believe that effective interactive systems require a commitment and an understanding throughout the entire development process. It just won't work to build a complete system and then, in the final stages of development, spread the interface over it *peanut butter theory of usability* like peanut butter.

With that in mind, some of the people who should be interested in this book are programmers, systems analysts, users and user-group representatives, technical writers, training coordinators, customer representatives, and managers at several levels. All of these positions have input into how the final system will look and act.

0.1.2 What Is the User Interface?

The basic user interface is usually understood to include things like menus, windows, the keyboard, the mouse, the "beeps" and other sounds the computer makes, and in general, all the information channels that allow the user and the computer to communicate.

Of course, using a modern computer system also involves reading manuals, calling help lines, attending training classes, asking questions of colleagues, and trying to puzzle out how software operates. A successful computer system or software package supports those activities, so that support is part of the user interface too.

But in a sense, all of these parts are the "peanut butter" we mentioned in the previous section. No matter how well they are crafted, the interface will be a failure if the underlying system doesn't do what the user needs, in a way that the user finds appropriate. In other words, the system has to match the users' tasks. That's why the book's central message is the need for "task-centered" design, and that's why the design of the user interface can't be separated from the design of the rest of the system.

Related exercise 0.1 ▷ see page 152

0.1.3 What Kind of User Interfaces Does This Book Cover?

The principles presented in this book were developed primarily in the context of the interfaces to computer software and hardware, but they are also applicable to a wide variety of other machines, from complex equipment such as phone systems and video cameras to simple appliances like refrigerators and power tools. Simpler machines are sometimes informative examples of problems or solutions in interface design.

0.1.4 Why Focus on Design?

This book describes design processes that help to produce good interfaces. The focus on process instead of end result deserves some explanation. Why don't we simply describe what a good interface is and leave the reader to create interfaces that fit that description?

There are several reasons. An interface has to be matched to the task it will support, as well as to the users who will work with it. There is an infinite variety of tasks and users, so there's no simple definition of a "good" interface. There have been many attempts to give broad, general guidelines for good interfaces, but those guidelines are usually too vague to be of much use. For example, a general guideline might say, "Give adequate feedback." But how will the designer determine what's "adequate"?

More specific guidelines for elements of the final interface have also been developed, describing such elements as how menus should be designed, how to label icons, and so forth. These guidelines also have problems. It's

impossible to cover every possible combination of task, user, and interface technology, so no set of specific guidelines can be complete. Even so, lists of specific guidelines are often so large and cumbersome that practicing designers find them very difficult to use. Further, in a given situation there are often several contradictory guidelines, and the designer has to rely on intuition to decide which are most important.

We might make an analogy between a designing a successful interface and a cutting a piece of string to the "right" length. General guidelines for the length of a piece of string, such as "long enough to do the job," aren't very helpful; and a list of specific definitions of the correct length for every purpose would be endless: 6 inches to tie up sagging flowers, 42 inches for a small package, 78 inches to tie down the trunk on an old VW, etc. But it's easy to describe a process that produces the right length: start with a very long piece of string, tie up your plant, package, car, or whatever, and then cut off the string that's not being used. Similarly, instead of specifying all the characteristics of the finished interface, this book present a design process that can produce good interfaces.

This is not to say that simply following the design process will magically produce a successful interface every time. The designer using the process must make many decisions along the way, relying on knowledge of users, their cognitive skills and limitations, and their tasks. In addition, the interface design process will only be successful if it is integrated into the software production process as a whole. This book presents basic information about all of these issues, and it contains pointers to other books and articles containing further useful information. All this material is organized in the context of the design process.

0.2 How to Use This Book

The main body of this book is a series of chapters describing in rough chronological order the steps needed to design a good user interface. Chapter 1 provides an overview of this process, and Chapters 2 through 7 fill in the details. Two appendices provide additional information on topics that don't fit into this chronological structure and that may not be of interest to every reader. Appendix L provides guidance on legal issues in interface design, while Appendix M gives an overview of management concerns.

0.2.1 HyperTopics and Examples

This book has been designed to achieve some of the advantages while avoiding some of the problems of computer-based hypertext. Hypertext has the advantage of providing pointers within the text that lead readers to additional material on topics that interest them. For example, a paragraph about typewriters might contain the word "keyboard," and clicking that word with a mouse could cause the computer to display a paragraph about different keyboard layouts. We've incorporated a similar technique by placing examples and supplemental material called "HyperTopics" near the text they're related to.

Hypertext has the disadvantage that readers often become confused as they jump from the middle of one concept to another, and to another, and to another, loosing track of any central theme. This book provides a mainline, the plain text you are reading now, that ties together all the details under the common theme of the design process. Chapters in the book are ordered to reflect that process, although materials within each chapter are often organized according to more abstract principles. For a quick overview or review of a chapter, you may want to read just the chapter's mainline.

0.2.2 Exercises

Readers who are seriously interested in becoming effective interface designers should work through the exercises for each chapter. A central goal of task-centered design is to reveal interface problems to the designers who create them, before the interface hits the street. But even with task-centered design, the ability to identify interface problems is a skill that can be improved with practice. The exercises are intended, in part, to give that practice.

Example: It's Obvious. . . Isn't It?

> You may read our examples of problem interfaces and say to yourself, "Well, that's an obvious problem. No one would really design something like that. Certainly I wouldn't."
>
> Here's a counter-example from the author's personal experience. John has a stereo system with a matched set of components made by the same manufacturer: a receiver, a CD player, and a cassette deck, stacked in that order. They all have the on/off button on the left side. Every time John goes to turn off all three components, he presses the top left button on the receiver, which turns it off; then he presses the top left button on the

CD player, which turns it off; then, naturally, he presses the top left button on the cassette deck — which pops open the cassette door. In retrospect, it seems "obvious" that the manufacturer could have improved the interface by putting all three buttons in the same location. But it clearly wasn't obvious to the system's designers, who (the advertisements tell us) were especially interested in building a system with a good user interface. We can guess that it wasn't obvious because the designers never considered the right task: the task of turning off all three components in sequence.

The flip side of the "it's obvious" coin is that most actions used to accomplish tasks with an interface are quite obvious to people who know them, including, of course, the software designer. But the actions are often not obvious to the first-time user. For example, imagine a first-time user of a multiuser computer who has been shown how to login to the system, has done some work, and is now finished with the computer for the day. Experienced computer users will find it obvious that a logout command is needed. But it may not occur to first-time users that a special action is required to end the session. People don't "log out" of typewriters or televisions or video games, so why should they log out of computers? Even users who suspect that something is required won't be likely to know that typing the word "logout" or "exit" might do the job.

Learning to predict problems like these by taking the user's point of view is a skill that requires practice, and that practice is a fundamental goal of the exercises.

0.3 About Shareware: How to Get and Pay for This Book

We've decided to make this book available as "shareware." That means we, the authors, have retained the copyright to the book, but we allow you to copy it and to make your own decision about how much it is worth. The details on copying restrictions and payment are included in a box at the end of every chapter, including this one.

0.3.1 Why Shareware?

We've chosen shareware as a distribution method for several reasons. For one thing, we hope it will make the book available to a wider audience, both because the cost is less ($5 + your printing/copying costs, as compared to probably $20 for a traditional book) and because anyone who can't afford the full cost is encouraged to pay just what they can afford — or what they think the book is worth. We've chosen to make the book shareware rather than freeware because we we would like some reimbursement for our development efforts.

We also hope that this distribution method will save a few trees. We've intentionally removed all sophisticated formatting so the text can be used on-line as a reference with virtually any computer system. You also have the option of printing just the chapters you need.

Finally, we like the idea of distributing our ideas directly to the "end-user" without the filter of a publisher. It's not that we think commercially published books are bad; but there's clearly room in the world for books that are published by individuals, just as there's room for handmade pottery, independent computer consultants, roving jugglers, and freelance carpenters. We count ourselves fortunate to have caught the leading edge of a technology that makes this kind of independent publishing possible.

0.3.2 Special Note to Instructors and Students

Instructors who want to use this book for class have our permission to make and sell copies to students for the price of copying, or to have the copies made and sold through a commercial copy service, or to make an original available to students so they can make their own copies. Please be sure to include the shareware notice from the front of the book in every copy (including copies of individual chapters). Students are asked to send in the shareware fee if they decide the book is useful to them.

0.3.3 Where to Get Up-To-Date Copies

To get a current copy of the electronic version of this book, use ftp to connect to "`ftp.cs.colorado.edu`" (yes, "`ftp`" is part of the address). For login name, type "`anonymous`"; for password, give your full login name. Look for the files in the directory `/pub/cs/distribs/clewis/HCI-Design-Book`.

0.3.4 Corrections and Additions

You can help us, and your fellow readers, by letting us know when our presentation is wrong or incomplete. We'll do our best to incorporate improvements into future versions.

0.3.5 Let Us Know What You Think

If you send in a shareware payment (or even if you don't!) we'd like to have your comments and suggestions. What parts of the book are especially valuable to you? What else would you like to see included? We probably won't be able to answer specific questions or reply personally to your letters, but we'll consider your comments if the book is a success and we decide to do a major revision.

0.4 About the Authors

Clayton Lewis is Professor of Computer Science and a member of the Institute of Cognitive Science at the University of Colorado. Before coming to Colorado he worked as a programmer, researcher, and manager of user interface development in corporate settings in the United States and England. He has continued to maintain close contacts with industry. Clayton holds a Ph.D. in psychology from the University of Michigan. His current research involves theoretical analysis of learning processes, assessment and design of programming languages, and development of prototyping tools.

John Rieman is finishing a Ph.D. in computer science at the University of Colorado, where he is investigating users' techniques for learning new interfaces in the absence of formal training, His interest in user interfaces developed during 10 years "in the trenches" as a user and manager of computerized editorial and typesetting systems. John's varied background also includes studies in art, mathematics, and law, as well as work experience as an auto mechanic and truck driver.

Both authors have taught courses in user interface design using draft versions of portions of this book.

0.5 Acknowledgements

This book is a practically oriented distillation of ideas that have grown out of many years of productive collaboration, formal and informal, with individuals in both the academic and the industrial human-computer interaction

(HCI) community. We have attributed ideas to individuals wherever possible, but we acknowledge a debt to many unnamed persons whose efforts have combined to provide a deeper understanding of the problems and solutions in the field.

We especially acknowledge the contribution of two of our colleagues at the University of Colorado, Peter Polson and Gerhard Fischer. Their research, as well as their insightful counter-arguments to points we might otherwise have accepted as obvious, make CU an exciting and productive environment in which to do HCI research.

Clayton also wants to acknowledge his debt to John Gould, recently retired from IBM Research. John has given Clayton help, guidance, and friendship, as well as his keen insights into all kinds of issues, technical and nontechnical, since 1970.

Many people, including our students, contributed suggestions that have helped to make this revised edition of the book a better publication. In particular, we acknowledge the detailed comments of Dieter Boecker of the GMD and John Patterson of SunSoft.

Much of the research described here has been supported by the National Science Foundation (grants IRI 8722792 and 9116640), by US West, and by CU's Center for Advanced Decision Support in Water and Environmental Systems (CADSWES).

0.6 Disclaimers

The opinions, findings, conclusions, and recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the agencies named in the acknowledgments section.

Comments about interfaces used as examples should not be taken as an evaluation of the interfaces as a whole. Every interface has its good and its bad points, and we have simply chosen problems that illustrate our topics.

Wherever trademarks have been used in this book they have been capitalized, to the best of our knowledge.

1 The Task-Centered Design Process

This chapter gives an overview of the task-centered design process that the book recommends. The process is structured around specific tasks that the user will want to accomplish with the system being developed. These tasks are chosen early in the design effort, then used to raise issues about the design, to aid in making design decisions, and to evaluate the design as it is developed. The steps in the task-centered design process are as follows:

- figure out who's going to use the system to do what
- choose representative tasks for task-centered design
- plagiarize
- rough out a design
- think about it
- create a mock-up or prototype
- test it with users
- iterate
- build it
- track it
- change it

1.1 Figure Out Who's Going to Use the System to Do What

The industry terminology for this step is "task and user analysis." The need for the task analysis should be obvious: if you build an otherwise great system that doesn't do what's needed, it will probably be a failure. But beyond simply "doing what's needed," a successful system has to merge smoothly into the user's existing world and work. It should request information in the order that the user is likely to receive it; it should make it easy to correct data that's often entered incorrectly; its hardware should fit in the space that users have available and look like it belongs there. These and a multitude of other interface considerations are often lost in traditional

requirements analysis, but they can be uncovered when the designer takes time to look into the details of tasks that users actually perform.

[Getting to know users and their tasks, Ch. 2]

Understanding of the users themselves is equally important. An awareness of the users' background knowledge will help the designer answer questions such as what names to use for menu items, what to include in training packages and help files, and even what features the system should provide. A system designed for Macintosh users, for example, should provide the generic Mac features that the users have come to expect. This might mean including a feature like cut and paste even though cut and paste plays no important part in the system's main functionality. Less quantifiable differences in users, such as their confidence, their interest in learning new systems, or their commitment to the design's success, can affect decisions such as how much feedback to provide or when to use keyboard commands instead of on-screen menus.

Effective task and user analysis requires close personal contact between members of the design team and the people who will actually be using the system. Both ends of this link can be difficult to achieve. Designers may have to make a strong case to their managers before they are allowed to do on-site analysis, and managers of users may want to be the sole specifiers of the systems they are funding. It's certain, however, that early and continued contact between designers and users is essential for a good design.

1.2 Choose Representative Tasks for Task-Centered Design

After establishing a good understanding of the users and their tasks, a more traditional design process might abstract away from these facts and produce a general specification of the system and its user interface. The task-centered design process takes a more concrete approach. The designer should identify several representative tasks that the system will be used to accomplish. These should be tasks that users have actually described to the designers. The tasks can initially be referenced in a few words, but because they are real tasks, they can later be expanded to any level of detail needed to answer design questions or analyze a proposed interface. Here are a few examples:

- for a word processor: "transcribe a memo and send it to a mailing list"
- for a spreadsheet: "produce a salary budget for next year"
- for a communications program: "login to the office via modem"
- for an industrial control system: "hand over control to next shift"

Again, these should be real tasks that users have faced, and the design team should collect the materials needed to do them: a copy of the tape on which the memo is dictated, a list of salaries for the current year and factors to be considered in their revision, etc.

The tasks selected should provide reasonably complete coverage of the functionality of the system, and the designer may want to make a checklist of functions and compare those to the tasks to ensure that coverage has been achieved. There should also be a mixture of simple and more complex tasks. Simple tasks, such as "check the spelling of 'ocassional'," will be useful for early design considerations, but many interface problems will only be revealed through complex tasks that represent extended real-world interactions. Producing an effective set of tasks will be a real test of the designer's understanding of the users and their work.

1.3 Plagiarize

We don't mean plagiarize in the legal sense, of course. But you should find existing interfaces that work for users and then build ideas from those interfaces into your systems as much as practically and legally possible. This kind of copying can be effective both for high-level interaction paradigms and for low-level control/display decisions.

At the higher levels, think about representative tasks and the users who are doing them. What programs are those users, or people in similar situations, using now? If they're using a spreadsheet, then maybe your design should look like a spreadsheet. If they're using an object-oriented graphics package, maybe your application should look like that. You might be able to create a novel interaction paradigm that's better suited to your application, but the risk of failure is high. An existing paradigm will be quicker and easier to implement because many of the design decisions (i.e., how cut and paste will work) have already been made. More important, it will be easy and comfortable for users to learn and use because they will already know how much of the interface operates.

Copying existing paradigms is also effective for the low-level details of an interface, such as button placement or menu names. Here's an example. You're writing a special-purpose forms management package and the specifications call for a spelling checker. You should look at the controls for spelling checkers in the word processing packages used by people who will use your system. That's almost certainly how the controls for your spelling checker interface should work as well.

This is an area where it's really common for designers to make the wrong

decision because they don't look far enough beyond the requirements of their own system. Let's dig a little further into the example of a spelling checker for the forms package. Maybe your analysis has shown that the spelling checker will most often pick up misspelled names, and you can automatically correct those names using a customer database. So you decide the most efficient interaction would be to display the corrected name and let the user accept the correction by pressing the Return key. But the word processor your users use most frequently has a different convention: pressing Return retains the "wrong" spelling of a word. Do you follow the lead of the existing system ("plagiarize"), or do you create your own, more efficient convention? To an extent the answer depends on how often users will be running your system compared to how often they will be running systems they already know. But more often than not, the best answer is to stick with what the users know, even if it does require an extra keystroke or two.

1.4 Rough Out the Design

The rough description of the design should be put on paper, which forces you to think about things. But it shouldn't be programmed into a computer (yet), because the effort of programming, even with the simplest prototyping systems, commits the designer to too many decisions too early in the process.

At this stage, a design team will be having a lot of discussion about what features the system should include and how they should be presented to the user. This discussion should be guided by the task-centered design approach. If someone on the team proposes a new feature, another team member should ask which of the representative tasks it supports. Features that don't support any of the tasks should generally be discarded, or the list of tasks should be modified to include a real task that exercises that feature.

The representative tasks should also be used as a sort of checklist to make sure the system is complete. If you can't work through each task with the current definition of the system, then the definition needs to be improved.

1.5 Think About It

No aviation firm would design and build a new jet airliner without first doing an engineering analysis that predicted the plane's performance. The cost of construction and the risk of failure are too high. Similarly, the costs of building a complete user interface and testing it with enough users to reveal all its major problems are unacceptably high. Although interface design

hasn't yet reached the level of sophistication of aircraft engineering, there are several structured approaches you can take to discover the strengths and weakness of an interface before building it.

One method is to count keystrokes and mental operations (decisions) for the tasks the design is intended to support. This will allow you to estimate task times and identify tasks that take too many steps. The procedures for this approach, called GOMS analysis, along with average times for things like decisions, keystrokes, mouse movements, etc. have been developed in considerable detail. We'll summarize the method later in the book.

Another method is to use a technique called the cognitive walkthrough to spot places in the design where users might make mistakes. Like GOMS modelling, the cognitive walkthrough analyzes users' interactions with the interface as they perform specific tasks. We'll also explain how to do cognitive walkthroughs later in the book.

1.6 Create a Mock-Up or Prototype

After thinking through the paper description of the design, it's time to build something more concrete that can be shown to users and that can act as a more detailed description for further work. In the early stages of a simple design, this concrete product might be as simple as a series of paper sketches showing the interface while a user steps through one of the representative tasks. A surprising amount of information can be gleaned by showing the paper mock-up to a few users. The mock-up may even reveal hidden misunderstandings among members of the design team.

For further analysis, the design can be prototyped using a system such as HyperCard, Dan Bricklin's Demo Package, or any of an increasing number of similar prototyping tools. It may even be possible to build a prototype using the User Interface Management System (UIMS) that will be the foundation of the final product. This approach can be especially productive, not only because it reduces the amount of work needed to create the production system, but also because interface techniques tested in a stand-alone prototyping system may be difficult to duplicate in the production UIMS.

The entire design doesn't need to be implemented at this stage. Initial efforts should concentrate on parts of the interface needed for the representative tasks. Underlying system functionality, which may still be under development, can be emulated using "Wizard of Oz" techniques. That is, the designer or a colleague can perform the actions that the system can't, or the system can be preloaded with appropriate responses to actions that a user might take. (The design team needs to take care that users and

management aren't misled into thinking the underlying system is finished.)

1.7 Test the Design With Users

No matter how much analysis has been done in designing an interface, experience has shown that there will be problems that only appear when the design is tested with users. The testing should be done with people whose background knowledge and expectations approximate those of the system's real users. The users should be asked to perform one or more of the representative tasks that the system has been designed to support. They should be asked to "think aloud," a technique described in more detail in Chapter 5.

Videotape the tests, then analyze the videotapes for time to complete the task, actual errors, and problems or surprises that the user commented on even if they didn't lead to errors. The user's thinking-aloud statements will provide important clues to why the errors were made.

1.8 Iterate

The testing with users will always show some problems with the design. That's the purpose of testing: not to prove the interface, but to improve it. The designer needs to look at the test results, balance the costs of correction against the severity of each problem, then revise the interface and test it again. Severe problems may even require a re-examination of the tasks and users.

One thing to keep in mind during each iteration is that the features of an interface don't stand alone. Revising a menu to resolve a problem that occurs with one task may create problems with other tasks. Some of these interactions may be caught by reanalyzing the design without users, using techniques like the cognitive walkthrough. Others may not show up without user testing.

[Hypertopic p. 8]

When should the iterations stop? If you've defined specific usability objectives (see hypertopic on Managing the Design Process), then iteration should be stopped when they are met. Otherwise, this will often be a management decision that balances the costs and benefits of further improvement against the need to get the product to market or, in in-house projects, into use.

1.9 Build the Design

The key guideline in building the interface is to build it for change. If you've been using a UIMS for prototyping, then you're already close to a finished product. If you've been using some other prototyping system, now is the time to switch to a UIMS or, perhaps, to an object-oriented programming environment. Try to anticipate minor changes with easily changed variables. For example, if you have to write your own display routine for a specialized menu, don't hardcode parameters such as size, color, or number of items. And try to anticipate major changes with code that is cleanly modular. If a later revision of the design requires that your specialized menu be replaced by some more generic function, the code changes should be trivial. These sound like ordinary guidelines for good programming, and indeed they are. But they are especially important for the user interface, which often represents more than half the code of a commercial product.

1.10 Track the Design

A fundamental principle of this book is that interface designers should not be a special group isolated from the rest of the system development effort. If this principle is to hold, then the designer must have contact with users after the design hits the street. In fact, it's easy to argue that this should be the case in any organization, because continued awareness of users and their real needs is a key requirement for a good designer.

One way to put designers in contact with users is to rotate them into temporary duty on the customer hotline. Another important point of contact for large systems is user group meetings. Managers also take advantage of these opportunities to see how real users react to the products they are selling.

Besides helping to answer the obvious question of whether the system is doing what it's designed to do, interactions with users can also yield surprises about other applications that have been found for the product, possibly opening up new market opportunities. This information can feed back into the design process as improved task descriptions for the next revision and better understanding on the part of the designer.

1.11 Change the Design

In today's computer market there are few if any software products that can maintain their sales without regular upgrades. No matter how well the

product is initially designed to fit its task and users, it will probably be inadequate in a few years. Tasks and users both change. Work patterns change because of the product itself, as well as because of other new hardware and software products. Users gain new skills and new expectations. Designers need to stay abreast of these changes, not only by watching the workplace in which their products are installed, but also by watching for developments in other parts of society, such as other work situations, homes, and the entertainment industry. The next revision of the design should be a response not only to problems but also to opportunities.

HyperTopic: Managing the Design Process

Task-Oriented vs. Waterfall Design

The traditional "waterfall" model of software design starts with a requirements analysis step that is performed by systems analysts who are usually not the interface designers. These requirements are transformed into system specifications, and eventually the hardware, underlying software, and user interface are designed to meet those specifications.

The waterfall model has proven to be a poor approach to software that has an important user interface component. As this chapter describes, the successful interface designer needs a deep understanding of the user's task and how the task fits into the rest of the user's work. That understanding can't be derived from a set of abstract specifications. Further, our experience has shown that several design iterations are essential in producing an effective interface. The traditional waterfall model simply doesn't allow those iterations.

The Design Team

Because the task-centered design methodology spreads the activities of interface design throughout the software design and life cycle, the interface can't be produced or analyzed at one point by a group of interface specialists. The job of building a good interface has to be taken on by the team that designs the product as a whole.

The design team needs to be composed of persons with a variety of skills who share several common characteristics. They

need to care about users, they need to have experience with both bad and good interfaces, and they need to be committed to and optimistic about creating an effective system. The team should include representatives from the entire range of interface-related areas: programmers, technical writers, training package developers, and marketing specialists. The team might include a user-interface analyst, but that's not essential. A shared commitment to interface quality, along with appropriate opportunities to interact with real users, will produce high quality interfaces for all but the most complex or critical interfaces.

Responsibility

Responsibility for the entire interface effort should be centralized. In particular, the designers who create the software shouldn't sign off on their product and hand it off to an entirely separate group that creates the manuals, who then hand off to another group that handles training. All of these activities need to be coordinated, and the only way to achieve that is through central management.

Usability Objectives

Serious corporate management efforts may require you to produce specific numbers that quantify usability. Usability objectives are target values for things such as speed to perform representative tasks and number of errors allowable. These can be used to motivate designers and support resource allocation decisions. The target values can be selected to beat the competition or to meet the functional needs of well-defined tasks.

(For more information on management, see Appendix M.)

Related exercise 1.1 ▷ see page 153

2 Getting to Know Users and Their Tasks

To get a good interface you have to figure out who is going to use it to do what. You may think your idea for a new system is so wonderful that everyone will want it, though you can't think of a really specific example, and that it will be useful in some way to people, even though you can't say just how. But history suggests you will be wrong. Even systems that turned out to be useful in unexpected ways, like the spreadsheet, started out by being useful in some expected ways.

Example: The Illusory Customers

There was a startup here in Boulder a few years back that wanted to build a system to help people build intelligent tutoring systems. They raised a bundle of money, brought a lot of smart people in, and went to work. But they didn't stop to figure out *exactly who* would use the system to do *exactly what*. The concept seemed too good, in those palmy days of AI madness, to require that kind of pedestrianism. The lack of specificity created some problems internally, since it was hard to make good decisions about what aspects of the new system were important. But the trouble became critical when the time came to line up test users, people who would try out the software and provide feedback to the developers. There were no takers! Not only were there no people waiting to fork out cash for the tool, there weren't even people who would take it for nothing. And this wasn't because the work wasn't quality. People just didn't want to do what the tool was supposed to help them do.

The company didn't just roll over; they searched around for something that people did want to do that they could do with something like the tool that had been built. Not surprisingly this didn't work out and the company folded its tents when the money ran out.

You may not have needed selling on this point. "Everybody" knows you have to do some kind of requirements analysis. Yes, but based on what, and in what form? Our advice is to insist that your requirements be grounded in information about real, individual people and real tasks that they really want to perform. Get soft about this and the illusions start to creep in and

before you know it you've got another system that everybody wants except people you can actually find.

HyperTopic: Contracts and Requirements

"Fortunately I don't have to worry about this. I work on contract stuff and all the requirements have been spelled out for me before I start."

Not so fast! It's one thing to meet the requirements of a contract and another thing to build a good system. Are anybody's interests really served if you build a system that meets spec but is a failure? That's what is likely to happen if you work to requirements that have not been grounded in reality, even if it's not your fault. Being selfish, will you get repeat business, or will you get a good reputation from work like this?

Clayton did once talk with a contract developer who assured him that on his current job the success of the system was really not an issue under any conceivable future (no possibility of repeat business, etc.) He has also heard of third-world "development" projects in which the real game is to please the bureaucrat who turns the tap on the (U.S.-supplied) "development" money, rather than to make something work. But life is too valuable to spend on activities like these. Find something to do that matters.

2.1 Getting in Touch With Users

So here's what to do. The first step is to find some real people who would be potential users of what you are going to build. If you can't find any you need to worry a lot. If you can't find them now, where will they come from when it's time to buy? When you have found some, get them to spend some time with you discussing what they do and how your system might fit in. Are they too busy to do this? Then they'll probably be too busy to care about your system after it exists. Do you think the idea is a real winner, and they will care if you explain it to them? Then buy their time in some way. Find people in your target group who are technology nuts and who'll talk with you because you can show them technology. Or go to a professional meeting and offer a unique T-shirt to people who'll talk with you (yes, there are people whose time is too expensive for you to buy for money who will work with you for a shirt or a coffee mug).

HyperTopic: Generic and Designer Users

"I don't have to bother about this stuff. My system will be a tool for any UNIX user, no matter what they are working on. There's no special user population I'm trying to support."

Unfortunately experience shows that many ideas that are supposed to be good for everybody aren't good for anybody. Why not check by finding somebody and making sure it's good at least for them?

"Well, I'm somebody and my tool is good for me."

Two points here. First, is it *really* good for you? Do you actually *use* it? Never work on something if you ought to be a user for it but you aren't. It's amazing how often this principle is violated. Second, there are lots of reasons why things often seem more useful to their designers than they do to other people. A big one is that the designer builds up his or her understanding of the thing over a long time and with a lot of thought. Usually the user wants to understand it right away, and often can't (Bruce Tognazzini makes this point very well in "Tog on Interface" Reading, MA: Addison Wesley, 1992, p. 8).

2.2 Learning About the Users' Tasks

Once you have some people to talk with, develop *concrete, detailed examples* of tasks they perform or want to perform that your system should support. Here's how this went for Clayton and colleagues in a recent project, disguised to avoid embarrassing anybody.

The aim was to develop a system for modelling traffic: street layout, traffic volumes, accidents and the like. The system was to provide a flexible, graphical interface that would make it easy to tweak the model and examine the results. We had good access to potential users, because the project was sponsored by an organization that included people who currently use an existing model that the new one was to replace.

But there was a delicate issue here, that you will often face. The particular people providing money for the project were not the users themselves, but a staff organization whose mission was to look after the needs of the users. Sounds OK? It's not: it meant that our direct contact was not with the people who really know firsthand what the problems are but people who are supposed to know the problems secondhand, a very different thing.

Fortunately we knew what we wanted and were able to arrange a series of meetings with real users.

In these meetings we developed a list of twelve things the users would actually want to do with the system. They were specific, as for example:

> Change the speed limit on Canyon Boulevard eastbound between Arapahoe and 9th. Calculate projected traffic flows on Arapahoe west of 6th assuming Canyon speeds between 25 and 55 in increments of 5 mph.

Notice a few things about this example.

It says what the user wants to do but does not say how the user would do it.

As stated, this task does not make any assumptions about the nature of the modelling tool or its interface. Therefore it can be used to compare different design alternatives for the system and interface in a fair way. If we said "change the speed limit by selecting a new value from a menu" we would have prejudged the right way for the user to perform this part of the task.

It is very specific.

Not only does it say exactly what the user wants to do, it actually specifies particular streets. What's the point of this? It means that we can fill out this description of the task with any other details that may become relevant in evaluating designs. In fact, it forces us to do this. For example, if the model needs to divide streets into separate pieces for purposes of analysis, and the user then needs to select a number of pieces to do an analysis for a stretch of street, we can see in detail how this would work out for the real Canyon Boulevard. We can't avoid the problem as we could if we were thinking of any old generic street.

Dennis Wixon has an example that makes this point (In M. Rudissill, T. McKay, C. Lewis, and P.G. Polson (Eds.), "Human-Computer Interaction Design: Success Cases, Emerging Methods, and Real-World Context." Morgan Kaufman. In press.). Wixon and colleagues were developing an interface for a file management system. It passed lab tests with flying colors, but bombed as soon as customers got it. The problem was that it had a scrolling file list that was (say) twenty characters wide, but the file names customers used were very long, and in fact often identical for more than

twenty characters (the names were made up by concatenating various qualifiers, and for many names the first several qualifiers would be the same.) Customers were not amused by needing to select from a scrolling list of umpty-ump identical entries that stood for different files. And this wasn't an oddball case, it was in fact typical. How had it been missed in the lab tests? Nobody thought it would matter what specific file names you used for the test, so of course they were all short.

It describes a complete job.

Note that the task doesn't just say to fiddle the speed limit on Canyon, or just to calculate projections for Arapahoe. It says the user wants to do both. This means that in seeing how this task plays out for a particular design of the interface we are forced to consider how features of the interface work together, not just how reasonable they may look separately.

We once evaluated a phone-in bank service that had done well in lab tests. People had no trouble checking their balances, finding out if checks had cleared, and the like. But when we looked at what was involved in finding if a check had cleared *and then* looking at a balance, we found big problems. The interface was supposed to support this kind of transition between functions, but the test tasks used in the lab had not required them.

↩ phone-in bank

Describing complete jobs is a key feature of our approach, and it's different from the usual way of doing things. Usually requirements lists are just that: lists of little things the system has to do. Meeting this kind of requirement does little to ensure that users can do a real job without going crazy: it just ensures that they can do all the *parts* of a real job.

An important angle on the complete job issue is seeing where inputs come from and where outputs go. In the example problem, where does the model the user is going to tweak come from? How does he or she obtain it? If there aren't good answers to these questions the system will be no good in practice, even if it does the tweaking and calculating parts very well.

Clayton worked on an early business graphics package whose sales were disappointing. Customer visits (done after ship, not before development when they should have been done) showed that the problem was that the package worked well when users had numbers to type in to make a graph, but badly when numbers were already in the system and needed to be extracted from a file. One user had temporarily hired typists to transcribe numbers from one screen, where a data file was displayed, onto another screen where the graph package was running. He was not eager to continue this arrangement. The majority of uses we saw were of this get-data-from-a-file kind,

so the system was unsuited to requirements even though it did a good job on making the graph itself, the part of the whole job on which the designers concentrated.

So you want to choose tasks that represent complete jobs, and you want to be sure to scrutinize the edges of the tasks. Where does stuff come in from? Where does it go? What has to happen next?

The tasks should say who the users are.

In the highway example we were dealing with a small, close-knit group of users, so we didn't specify in our example tasks *who* would be doing what: we took it as given. Probably we should have worried about this more, and probably you should. The success of a design can be influenced strongly by what users know, how the tasks supported by the system fit into other work they have to do, and the like. So you need to get a fix on these things early in design.

The design of Oncocin, a sophisticated cancer therapy advisor, illustrates what's at stake (Musen, M.A., Fagan, L.M., and Shortliffe, E.H. "Graphical specification of procedural knowledge for an expert system. In J.A. Hendler [Ed.], "Expert Systems: The User Interface." Norwood, NJ: Ablex, 1988, p. 15). Earlier experience with doctor users had shown that they are willing to invest very little time in becoming familiar with a new system. A system to be used directly by doctors therefore has to be different from one to be used by medical technicians, who can be told they *have* to learn a new tool. The Oncocin designers needed to decide up front whether their users would be doctors or would be technicians working in support of doctors. They went for direct use by doctors. The interface they came up with is as much as possible a faithful copy of the paper forms for specifying therapy that doctors were already using.

So what should you say about users in specifying your tasks? If possible, name names. This allows you to get more information if it becomes relevant, just as saying it's Arapahoe Avenue allows you to bring in more detail about the task if you need to. Beyond that you should note characteristics of the users that you already know will be important, such as what their job is and what expertise they have.

In choosing the sample tasks for the traffic modelling system we were guided by two objectives. First, we wanted to be sure that we had examples that illustrated the kinds of support that we as designers thought the system should provide. That is, we had some general ideas about what the system was supposed to be good for, and we tried to find tasks that were examples of

these. But second, we needed to reflect the interests of potential users in the examples. So we tried to find tasks that illustrated proposed functionality in the context of work users really wanted to do.

In the process some possible functions for the system dropped out. We had envisioned that the system might include some optimization functions that would manipulate the model to find the best of some range of alternatives with respect to some measure of quality. Users had no interest in this. They preferred to solve such problems by framing and evaluating alternatives themselves rather than having the system do this.

This is not an uncommon conflict, and one without a simple resolution. We thought, and still think, that users would eventually come to want optimization functions once more pressing modelling needs were met, so we didn't want to just take the users' word on this. But we put optimization on the back burner to be pushed again in the future.

This illustrates a key point about input from users: users are *not* always right. They cannot anticipate with complete accuracy how they will use new technology. As a designer your job is to build a system that users will want when it gets here, not to build the system users say they want as they see things today. You may well have insight into the future that users lack. But you need to be very careful about this. If you are like most designers, an outsider in the domain of work you are trying to support, you have to recognize that users know a whole lot more about what they are doing than you do. If you can't get users interested in your hot new idea, however hard you try to draw them into it, you're probably missing something.

HyperTopic: Participatory Design

In our discussion we have been assuming a split between the roles of designers and users that has been traditional in U.S. system design. Designers are not users; they gather information from users and reflect it in systems they build; they give these systems to users who use them or not. There is an alternative approach, pioneered by workers in Scandinavia, that rejects this structure. In *participatory design*, systems are designed by designers and users working together: a slogan is *designing with* rather than *designing for*. The power to make the system be one way rather than another is not reserved for the designers, as it is in most U.S. design practice, but rather is shared by designers and users working in collaboration.

The contrast between participatory design and standard U.S. practice reflects deep differences in political and philosophical

outlook between Europe and the U.S.. Most European countries give workers very broad influence on working conditions, which are much more strictly regulated than in the U.S.. In many countries workers must have seats on the boards of directors of companies, and workers must be consulted on the introduction of new technology in the workplace. With this view of workers it is natural to think that workers should have a direct hand in shaping the systems they have to use. By contrast the prevailing view in the U.S. is that management should just decide what systems are good for productivity and then impose them on workers, whose views basically don't matter.

Many people in the U.S., if they think about these matters at all, assume that the U.S. way of doing things must be right. What is your reaction when you hear that in some European countries it is illegal to make someone work in a room without a window? Does that seem silly, or maybe wrongheaded, because of the restriction in places on freedom of enterprise? What do you think about it when you also learn that a number of countries in Europe have higher per capita productivity than the U.S., and that the U.S. is steadily losing export position, especially in advanced industries, and holding its own primarily in commodity exports like farm produce? For a fascinating, disturbing, and constructive look at these and related issues, read the book "The Competitive Advantage of Nations," by Michael Porter (New York: Free Press, 1990).

You can find a lengthy discussion of participatory design in the journal Human-Computer Interaction, (Floyd, C., Mehl, W.-M., Reisin, F.-M., Schmidt, G., and Wolf, G. "Out of Scandinavia: Alternative approaches to software design and system development." Human-Computer Interaction, 4 (1989), pp. 252–350), and a short and sweet discussion in an article by Jeanette Blomberg and Austin Henderson of Xerox in the CHI'90 proceedings (Blomberg, A.L. and Henderson, A. "Reflections on participatory design." In Proc. CHI'90 Conference on Human Factors in Computer Systems. New York: ACM, 1990, pp. 353–359). Blomberg and Henderson stress three defining attributes of participatory design: the goal of the activity is to improve the worklife of users (not, for example, to demonstrate how neat object-oriented technology is); the activity is collaborative, with all goals and decisions actively negotiated and not imposed; the

activity is iterative, in that ideas are tested and adjusted by seeing how they play out in practice.

It's pretty easy to see how one could approach participatory design in in-house projects, though it would not be easy to get your organization actually to do it. For in-house projects it will be true at some level that designers and users share the same goals, though in the U.S. context these goals may not have much to do with the quality of worklife. But U.S. organizations usually have job demarcations which make it hard to get participatory design going. Usually, if my job is to be a user it is not to be a designer or to work on designs. That's the province of "experts" employed for the purpose, even though they have no idea what the real problems are that need to be solved.

For commercial projects there are further challenges. At bottom, your objective as a commercial developer may really not be to improve the quality of somebody else's work life, but rather to make money for yourself. So you don't have the right goal to begin with.

There are two ways to go from here. One is to forget about the defining goals of participatory design and go through the motions of it in service of your selfish goals. The idea is that you hope to produce a better design, and hence make more money, by engaging users in collaboration focussed on the user's goals. To draw users in to making the considerable investment of time they would have to make to work with you, you would offer them a piece of the action.

Developing new technology in close partnership with potential users like this is a good idea in many industries for lots of reasons not restricted to the peculiarities of user interfaces. As Michael Porter recounts, the modern printing press was developed by a new technology company that was supported by some of its potential users, big English newspapers. Such a relationship gets you the information you need to make your system really useful, and hence successful, as well as developing an early market.

Another response to the mismatch of your goal of making money and the participatory design goal of improving the quality of work life is to change your goal. Will money actually make you happy? Of course not. Will improving somebody's work life make you happy? Maybe so, if the work involved is itself

> something worthwhile, or if you just take satisfaction in doing something well. Even if you can't get this unselfish the logic of the situation is that you may do better all around, including monetarily, if you really care about the people who will use your system and what they do than if you only care about the money.

2.3 Using the Tasks in Design

[traffic modelling, pp. 13*ff*]

Back to the traffic modelling system and our sample tasks. What did we do with them after we got them? Taking a look at their fate may clarify what the tasks should be like, as well as helping to persuade you that it's worth defining them.

Our first step was to write up descriptions of all the tasks and circulate them to the users (remember, we're back in us-versus-them mode, with designers and users clearly different teams.) We included queries for more information where we felt the original discussion had left some details out. And we got corrections, clarifications, and suggestions back which were incorporated into the written descriptions.

scenario ↪

We then roughed out an interface design and produced a *scenario* for each of the sample tasks. A scenario spells out what a user would have to do and what he or she would see step-by-step in performing a task using a given system. The key distinction between a scenario and a task is that a scenario is design-specific, in that it shows how a task would be performed if you adopt a particular design, while the task itself is design-independent: it's something the user wants to do regardless of what design is chosen. Developing the scenarios forced us to get specific about our design, and it forced us to consider how the various features of the system would work together to accomplish real work. We could settle arguments about different ways of doing things in the interface by seeing how they played out for our example tasks.

Handling design arguments is a key issue, and having specific tasks to work with really helps. Interface design is full of issues that look as if they could be settled in the abstract but really can't. Unfortunately, designers, who often prefer to look at questions in the abstract, waste huge amounts of time on pointless arguments as a result.

For example, in our interface users select graphical objects from a palette and place them on the screen. They do this by clicking on an object in the palette and then clicking where they want to put it. Now, if they want to

place another object of the same kind should they be made to click again on the palette or can they just click on a new location? You can't settle the matter by arguing about it on general grounds.

You can settle it by looking at the *context* in which this operation actually occurs. if the user wants to adjust the position of an object after placing it, and you decide that clicking again somewhere places a new object, and if it's legal to pile objects up in the same place, then you have trouble. how will you select an object for purposes of adjustment if a click means "put another object down"? on the other hand, if your tasks don't require much adjustment, but do require repeated placement of the same kind of object, you're pushed the other way. our tasks seemed to us to require adjustment more than repeated placement, so we went the first way.

This example brings up an important point about using the example tasks. it's important to remember that they are *only examples*. Often, as in this case, a decision requires you to look beyond the specific examples you have and make a judgement about what will be common and what will be uncommon. You can't do this just by taking an inventory of the specific examples you chose. You can't defend a crummy design by saying that it handles all the examples, any more than you can defend a crummy design by saying it meets any other kind of spec.

We represented our scenarios with *storyboards*, which are sequences of sketches showing what the screen would show, and what actions the user would take, at key points in each task. We then showed these to the users, stepping them through the tasks. Here we saw a big gain from the use of the sample tasks. They allowed us to tell the users what they really wanted to know about our proposed design, which was what it would be like to use it to do real work. A traditional design description, showing all the screens, menus, and so forth, out of the context of a real task, is pretty meaningless to users, and so they can't provide any useful reaction to it. Our scenarios let users see what the design would really give them.

"This sample task idea seems crazy. What if you leave something out? And won't your design be distorted by the examples you happen to choose? And how do you know the design will work for anything *other* than your examples?" There is a risk with any spec technique that you will leave something out. In choosing your sample tasks you do whatever you would do in any other method to be sure the important requirements are reflected. As noted above, you treat the sample tasks as examples. Using them does not relieve you of the responsibility of thinking about how other tasks would be handled. But it's better to be sure that your design can do a good job on at least some real tasks, and that it has a good chance of working on other

tasks, because you've tried to design for generality, than to trust exclusively in your ability to design for generality. It's the same as that point about users: if a system is supposed to be good for *everybody* you'd better be sure it's good for *somebody*.

HyperTopic: Integrating Task-Centered Design and Traditional Requirements Analysis

If you're working for a small company or developing small projects for a few internal users at a large firm, the task-centered design approach may be all you need. But for larger projects, you'll probably have to work within the structure of an established software engineering procedure. How to apply task-centered principles within that procedure will vary depending on the software engineering approach used at your company. But we can give some general guidelines that are especially useful in the early stages of development.

Most large software projects are developed using some version of the "waterfall method." The basic waterfall method assumes that a piece of software is produced through a clearly defined series of steps, or "phases":

- Requirements analysis
- Specification
- Planning
- Design
- Implementation
- Integration
- Maintenance.

In its strictest version, this method states that each phase must be completed before the next phase can begin, and that there's no chance (and no reason) to return to an earlier phase to redefine a system as its being developed.

Most software engineering specialists today realize that this approach is unrealistic. It was developed in the era of punch cards and mainframes, so it doesn't have a real place for considerations of interactive systems. Even in the mainframe era it was less than successful, because the definition of what's required inevitably changes as the system is developed.

Various modifications to the phases of the waterfall method and their interaction have been proposed. However, it's not unusual to find productive software development environments that still incorporate many steps of the method, partly for historical reasons and partly because the approach helps to define responsibilities and costs for various activities within a large software project. With some effort, the task-centered design approach can supplement the early stages of a waterfall environment.

Requirements Analysis

The waterfall method's initial "Requirements Analysis" phase describes the activity of defining the precise needs that the software must meet. These needs are defined in terms of the users and the their environment, with intentionally no reference to how the needs will actually be met by the proposed system.

This is exactly the same approach as we suggest for describing representative tasks: define what the user needs to do, not how it will be done. The difference is that the representative tasks in task-centered design are complete, real, detailed examples of things users actually need to do. The requirements produced by traditional software engineering, on the other hand, are abstract descriptions of parts of those representative tasks.

This is an important distinction, and we want to emphasize it most strongly:

> Task-centered design focuses on *real*, *complete*, *representative* tasks. Traditional requirements analysis looks at *abstract*, *partial* task elements.

Here's an example. For a document processing system, a representative task might be to produce this book. Not to produce "a book," but to produce "version 1 of Task-Centered Design, by Lewis and Rieman." That representative task supplements the detailed partial tasks collected in traditional requirements analysis, which might include things such as "key in text" and "check spelling" and "print the document."

So if you're doing a traditional requirements analysis, you need to supplement it by collecting some representative tasks. The two approaches complement each other nicely. The traditional approach helps to ensure that all important functions of

the system are recognized, while the representative tasks in the task-centered approach provide an integrated picture of those functions working together.

Specification

In the traditional "Specifications" phase of software engineering, the requirements are used to produce a description of the system that includes the details needed by the software designers and implementers. The customers — the end users — can then sign off on this document, and the software team can begin to plan and design the actual system. This sounds like great stuff from a management point of view, but it practice it often falls apart. Users aren't experts at reading specifications documents, and they have trouble imagining how the system will actually perform. Various alternatives to written specifications have been proposed, including prototypes and a more iterative approach to design, both of which fit nicely into the task-centered design approach.

However, even if you're still doing written specifications, the representative tasks can be of value. Include those tasks, with some details about how they will be performed, in the specification document. The big win here is that the customers will be able to understand this part of the specifications. It will also force the specifications writer to consider a complete task, which may catch problems that could be missed when single functions are considered individually.

Notice that the description of the proposed software hasn't quite reached the stage where you could do a complete "scenario," as we have defined it. Many of the details, such as the names of menu items, the distribution of functionality among dialog boxes, etc., remain to be defined. But a high-level overview of the interactions can be described, and doing this well is a test of your understanding of the users' needs.

[scenario p. 20]

Planning, Design, and Beyond

From this point on, the strict waterfall method and the task-centered design approach take very different paths. Many of the principles we describe can be used in doing the first pass at sys-

tem and interface design, but inherent in the task-centered approach is the need for iteration: it's very rare that the first design of an interface is a complete success. Several iterations of testing and redesign are required, and that may well involve jumping backward through the phases of the waterfall method, something that's traditionally not allowed. Fortunately, the strict forward-moving method is seldom adhered to today. Most development environments recognize the need for some iteration, and that should make it possible to accommodate the general spirit of the task-centered approach.

Related exercise 2.1 ▷ see page 154

3 Creating the Initial Design

The foundation of good interface design is *intelligent borrowing*. That is, you should be building your design on other people's good work rather than coming up with your own design ideas. Borrowing is important for three distinct reasons. First, given the level of quality of the best user interfaces today, it's unlikely that ideas you come up with will be as good as the best ideas you could borrow. Second, there's a good chance, if you borrow from the right sources, that many of your users will already understand interface features that you borrow, whereas they'd have to invest in learning about features you invent. Finally, borrowing can save you tremendous effort in design and implementation and often in maintenance as well.

HyperTopic: "Won't I get sued if I follow your advice?"

As you read through this chapter you'll realize that much of the borrowing we recommend is not only allowed, it's actually encouraged by the developers of the original system. Apple Computer, for example, provides style guides, software toolkits, and other support for developers who want to produce Macintosh programs that look and act similar to all the other Macintosh programs.

In addition, there are a lot of other things you can copy without infringing on the rights of other companies. Unfortunately, there's no clear and simple rule that defines those rights. Appendix L gives an overview of the legal principles and provides a list of "boundary markers," examples of things that can and cannot be legally copied under the current laws. The development process we recommend is to keep those boundary markers in mind, rough out your interface, and then talk over your decisions with your company's attorney. If there's a central feature of the interface you're worried about, such as picking up an entire interface metaphor, you may want to get legal advice a little earlier.

[Appendix L pp. 133*ff*]

Because the law is always changing (and this area of law is especially unsettled) there are some other things you need to do. One is to read the trade journals and keep yourself abreast of the current state of the law. Another is that your business plans should take into account the possibility that, no matter how careful you are, you may become involved in a lawsuit.

3.1 Working Within Existing Interface Frameworks

The first borrowing you should do is to work within one of the existing user interface frameworks, such as Macintosh, Motif or Windows. The choice may have already been made for you: in in-house development your users may have PCs and already be using Windows, or in commercial development it may be obvious that the market you are trying to reach (you've already found out a lot about who's in the market, if you're following our advice) is UNIX-based. If you want to address several platforms and environments you should adopt a framework like XVT that has multi-environment support.

The advantages of working in an existing framework are overwhelming, and you should think more than twice about participating in a project where you won't be using one. It's obvious that if users are already familiar with Windows there will be big gains for them if you go along. But there are also big advantages to you, as mentioned earlier.

You'll get a *style guide* that describes the various interface features of the framework, such as menus, buttons, standard editable fields and the like. The style guide will also provide at least a little advice on how to map the interface requirements of your application onto these features, though the guides aren't an adequate source on this. This information saves you a tremendous amount of work: you can, and people in the old days often did, waste huge amounts of time designing scroll bars or ways to nest menus. Nowadays these things have been done for you, and better than you could do them yourself.

You also get *software tools* for implementing your design. Not only does the framework have an agreed design for menus and buttons, but it also will have code that implements these things for you. We'll discuss these implementation aids in a later chapter.

HyperTopic: One-of-a-Kind Hardware

> "We can't use any of the existing frameworks because of our special hardware, which we have to use because of military specs/the customer's hardware base/the new psychotelekinetic interaction device we're supporting."
>
> Make sure your schedule allows for the huge amount of extra work you are buying into, and make sure you are being paid enough. Also think about where your next job is coming from: the record of sustained success for onesy developments is not good, because of all the added costs. Try really hard to find a

way to accommodate that psychotelekinetic device as an add-on to one of the standard environments.

3.2 Making Use of Existing Applications

The next copying you ought to do is to find good existing applications that already provide some of the functionality you need and plan to incorporate those applications into your system. Do users need some database capabilities, and some ability to do calculations of the data they are dealing with in your application? They almost always do. You will not be able to develop your own Dbase, or your own Excel, that will be nearly as good, or as powerful, as existing products that have been shaped by intense and extended competition in the commercial market. Even if you could you and your users are still behind: many of them already know how to use Excel, and they gain nothing by having to learn about your version of a spreadsheet instead.

Example: Monte Carlo Modelling Systems

Suppose you want to do financial projections, such as you might do with a spreadsheet, but you need to represent uncertainty in the numbers you use: next year's sales should be somewhere around $1.5M, but they could be as low as about $1M or as high as $2M. Costs should be around $1.2M, but they could be as high as $1.4M or as low as $1M. So what's the gross profit picture for next year? You could use a regular spreadsheet by running various what-if's with different numbers, but if you have a few more parameters that gets tedious and error-prone. And suppose you think that sales and costs are probably correlated? Monte Carlo simulation is a modelling technique that lets you specify statistical distributions for parameters, and correlations between them, and estimates the distribution for resulting quantities by sampling from the distributions you provide.

Suppose you wanted to sell a Monte Carlo tool to the business world. You could try to build your own system from the ground up, but you'd be wrong. The right thing to do, as Crystal Ball and @Risk have done, is to build and sell a spreadsheet add-on. Modern spreadsheets, and the systems in which they run, permit fairly easy communication between the spreadsheet and your code. This is a huge win, for the following reasons:

- The spreadsheet provides for you many functions which you would otherwise have to implement yourself, including data entry and specification of computations in the model.
- It provides these functions in a form many users already understand. In fact the same add-on can be made to work with more than one of the popular spreadsheets, so users can choose whichever spreadsheet they already know most about.
- The spreadsheet provides many functions that aren't part of what you *have to* do in your application but are significant enhancements that you get for nothing. For example, modern spreadsheets offer a wide range of graphical presentations of data.
- Chances are users will want to transfer data between your package and their spreadsheet anyway. That'll be much easier with the add-on than if you build a separate package.
- The spreadsheet can handle most if not all of the various environmental dependencies for you, such as drivers for various kinds of displays and printers. There is no way that you could afford to put as much coverage of environment options into your package as the spreadsheet developers, with their huge market, can afford to put in theirs.

3.3 Copying Interaction Techniques From Other Systems

Another kind of borrowing is copying specific interaction techniques from existing systems. If the style guides were good enough you might not have to do this, but the fact is the only way to get an adequate feel for how various interface features should be used, and how different kinds of information should be handled in an interface, is to look at what other applications are doing. The success of the Macintosh in developing a consistent interface style early in its life was based on the early release of a few programs whose interfaces served as models of good practice. An analogous consensus for the IBM PC doesn't really exist even today, but as it forms it is forming around prominent Windows applications like Excel or Word.

It follows from the need to borrow from other applications that you can't be a good designer without becoming familiar with leading applications. You have to seek them out, use them, and analyze them.

The key to "intelligent" borrowing, as contrasted with borrowing pure and simple, is knowing *why* things are done the way they are. If you know why an application used a tool palette rather than a menu of functions, then you have a chance of figuring out whether you want to have a palette or a menu. If you don't know why, you don't know whether the same or a different choice makes sense for you.

Bill Atkinson's MacPaint program was one of the standard-setting early Macintosh programs, and it used a tool palette, a box on the side of the window containing icons. The icons on the palette stand for various functions like "enter text", "move view window", "draw a rectangle", and the like. Why was this a good design choice, rather than just listing these functions in a pull-down menu? In fact, some similar functions are listed in a pulldown menu called "goodies". So should you have a palette for what you are doing or not?

Here are some of the considerations:

- Operations on menus usually do not permit or require graphical specification of parameters, though they can require responding to queries presented in a dialog box. So an operation like "draw a rectangle", in which you would click on the corners of the intended rectangle, would be odd as a menu item.

- A palette selection actually enters a *mode*, a special state in which things happen differently, and keeps you there. This doesn't happen when you make a menu selection. For example, if you select the tool for drawing rectangles from a palette, your mouse clicks get interpreted as corners of rectangle until you get out of rectangle drawing mode. If you selected "draw a rectangle" from a menu, assuming the designer hadn't been worried about the point above, you'd expect to be able to draw just one rectangle, and you'd have to go back to the menu to draw another one.

  So a tool palette is appropriate when it's common to want to do a lot of one kind of thing rather than switching back and forth between different things.

- Modes are generally considered bad. An example which influenced a lot of thinking was the arrangement of input and command modes in early text editors, some of which are still around. In one of these editors what you typed would be interpreted either as text to be included in your document, if you were in input mode, or as a command, if you were in command mode. There were two big problems. First, if you

forgot what mode you were in you were in trouble. Something you intended as text, if typed in command mode, could cause the system to do something dreadful like erase your file. Second, even if you remembered what mode you were in, you had the bother of changing modes when you needed to switch between entering text and entering commands.

- But modes controlled by a tool palette are considered OK because:
  - there is a change in the cursor to indicate what mode you are in, so it's harder to forget;
  - you only get into a mode by choosing a tool, so you know you've done it;
  - it's easy to get out of a mode by choosing another tool.

- In a tool palette, tools are designated by icons, that is little pictures, whereas in menus the choices are indicated by text. There are two subissues here. First, for some operations, like drawing a rectangle, it's easy to come up with an easily interpretable and memorable icon, and for others it's not. So sometimes icons will be as good as or better than text, and sometimes not. Second, icons are squarish while text items are long and thin. This means icons *pack* differently on the screen: you can have a bunch of icons close together for easy viewing and picking, while text items on a menu form a long column which can be hard to view and pick from.

So... this tells you that you should use a tool palette in your application if you have operations that are often repeated consecutively, and you can think of good icons for them, and they require mouse interaction after the selection of the operation to specify fully what the operation does.

Depending on the style guide you are using, you may or may not find a good, full discussion of matters like this. One of the places where experience will pay off the most for you, and where talking with more experienced designers will be most helpful, is working out this kind of rationale for the use of various interface features in different situations.

HyperTopic: Some Kinds of Why's

Analyzing the why's of interface features is complicated and detailed, as the example showed. But it's possible to identify some broad kinds of arguments that crop up often.

Geometrical and Movement Arguments

Small targets are harder (and slower) to hit with a mouse than big targets; long mouse movements are slower than short ones; icons pack differently from text strings; more keystrokes take longer to type; switching between mouse and keyboard is slow.

Memory Arguments

It is easier to recognize something when you see it (for example on a menu) than it is to recall it from scratch (for example in typing in a command); it is hard to remember much information from one step in a process to another (so, for example, having help information available at the same time as the user carries out an operation is a good idea; more generally, information that is used together should be presented together); the interface should present key information, such as the current mode, rather than requiring the user to remember it.

Problem-Solving Arguments

Interface features should help the user to select operations that are relevant to their goals, by labelling the operations in ways that match the way the user thinks about his or her task; the user needs to know what an operation has actually done (the UNIX approach of saying nothing unless something goes wrong is useless if you are a learner and do not already know what the commands do); users will make mistakes, especially if they are exploring options, so give them a way to back out.

Attention Arguments

Information presented with a big change in the display is more likely to be read; information presented close to where the user is looking is more likely to be read; auditory signals cannot be ignored as easily as visual signals (this is a two-edged sword; sometimes you want to be able to ignore things).

Convention arguments

If you do things the way your users are familiar with, they will be happier; conventional ways of using features have stood the

test of time, but any innovation you make has not and thus may suffer from hard-to-anticipate problems.

Diversity Arguments

Different users have different preferences for interaction styles, and some users have physical limitations that make it difficult or impossible to use certain features. A blind user, for example, can work with textual menus using a device that translates on-screen text to audible speech, but graphical icons can't be translated by the device. A person with impaired motor control may be able to type but may not have the fine hand control needed to use the mouse, while a person with the use of only one hand might have problems with two-key combinations and prefer the mouse and pulldown menus. For all of these users, providing more than one access to a function may be essential.

All of these statements are abstract. It can be hard to see how or whether they apply to a given design problem without some experience in applying them. Spend some time looking at a good interface and seeing if you can make sense of its features in terms of these ideas.

Example: Copying in the Traffic Modelling System

[traffic modelling, pp. 13*ff*]

We did all of the above kinds of copying in the traffic modelling system. To begin with we incorporated a statistics package called S+ bodily: users needed to do statistical treatments and plots, and S+ already had implemented more than they would need. We were working in UNIX so it was fairly easy to run S+ as a separate process, send requests to it and bring back answers from it or have it present its output directly to users. We also adapted an existing package for diagram building called AgentSheets so that users could build their model in diagram form by pointing and clicking on graphical model elements. This was not a simple copy because AgentSheets needed to be extended slightly.

AgentSheets is implemented in LISP, and we proposed to do other development in LISP, so we needed LISP-compatible software for other interface features that AgentSheets did not

provide. We chose Garnet, a LISP-based user interface support package developed at Carnegie-Mellon University (CMU). Garnet is intended to provide very flexible support for people who want to create their own interface features, rather than adopting an existing framework like MOTIF. Since we didn't really need to create any new features it was not really a good choice for our purposes, but at the time we were doing the work we did not have LISP support for other options.

↩ Garnet

Our experience with Garnet is a good example of the costs of not copying enough. We had to implement interface features in Garnet that would be provided as standard parts of a system like MOTIF today. (Of course this is not a reflection on Garnet but on us: we were trying to use it for purposes for which it was not intended. Garnet was and is a very powerful tool for exploring new interface techniques such as demonstrational interfaces.)

Since Garnet was not intended to support any particular interface style it did not have a style guide. This was not a problem, because we simply copied stylistic features from other systems, especially the Macintosh. Our users (we knew exactly who they were) had no experience with *any* graphical user interface, so we considered ourselves free to use Mac-style interaction even though we were implementing on a UNIX platform. But this was probably a bad copying decision, which we would not have made had it been clearer at the time how much acceptance MOTIF would get. One of the ways designers today are fortunate is that these choices have become clearer.

Against all of this background we could concentrate on a few areas of the design for which we did not find a clear precedent. One was allowing users to specify model inputs by typing in numbers or by designating a prepared file; another was how to control the execution of the model so as to explore various possible combinations of input values (recall the speed-limit study mentioned in the previous chapter); another was how to capture model results in a form that could be fed into further processing, so as to prepare a graph comparing results from different runs, for example.

[speed limits, pp. 13*ff*]

We didn't need to do anything very clever about these things. We represented data, whether model input or output, as graphical objects that could be connected to other model elements. These objects could be opened, exposing their contents for edit-

ing, and the opened object contained a file browser that could be used optionally to select a file from which values could be read or into which values could be placed for later use. Execution control was done by specifying input parameters and values to be used for them, along with other specifications of a run, in a dialog box.

3.4 When You Need to Invent

At some point in most projects you'll probably feel that you've done all the copying that you can, and that you've got design problems that really call for new solutions. Here are some things to do.

- Think again about copying. Have you really beaten the bushes enough for precedents? Make another try at locating a system that does the kind of thing you need. Ask more people for leads and ideas.
- Make sure the new feature is really important. Innovation is risky and expensive. It's just not worth it for a small refinement of your design. The new feature has to be central.
- Apply the whole process of this book to the design. This means you can't expect to come up with a good innovation just by thinking about it, any more than you can come up with a good interface just by thinking about it. Be careful and concrete in specifying the requirements for the innovation, that is, the context in which it must work. Rough out some alternatives. Analyze them, as discussed in the next chapter. Then try them out with users, as discussed in the chapter after that.

Example: Tog's One-or-more Button

Bruce Tognazzini has a great example of process of innovation in "Tog on Interface" (Reading, MA: Addison Wesley, 1992), a book you should read, especially, but not only, if you are a Mac person. He recounts the design of a kind of button that requires the user to turn on at least one of them. Repeated try-outs with users were completely crucial to success.

Related exercise 3.1 ▷ see page 156

HyperTopic: The Good Old Days, When Designers Were Designers

If you look at design case studies in the literature you are likely to be misled about what's involved in good design. Many of the most interesting case studies, such as those for the Xerox Star, come from a good while ago (Smith, D.C., Irby, C., Kimball, R., Verplank, W., and Harslem, E. "Designing the Star user Interface." Byte, 7:4 (April 1982), pp. 242–282). They tell you about designing something that was totally revolutionary in its day. Virtually every feature of the interface was an innovation, so virtually every feature was subjected individually to intensive design study including user testing. These studies tell you about the heroic age of design.

But you are probably not creating something totally revolutionary. In fact, as we've been advising, you should be trying hard not to, in most situations.

Even contemporary case study reports can be misleading. These reports usually focus on the innovations, because that's where the news and interest are. This means you don't really learn how to get your job done from these studies.

Related exercise 3.3 ▷ see page 157

3.5 Graphic Design Principles[1]

The graphic design of an interface involves decisions about issues such as where to put things on the screen, what size and font if type to use, and what colors will work best. For these questions as for other, more substantive design issues, intelligent borrowing should be your first approach. But that often leaves you with a lot of decisions still to be made. Here are a few principles of graphic design that will not only make your interface more attractive, but will also make it more usable. Each principle is accompanied by a description of *why* it's important, so you'll be able to consider the tradeoffs when there's a conflict between two principles or between a design principle and a borrowed technique.

[1]Erroneously labeled 3.4 in the original.

The Clustering Principle:

Organize the screen into visually separate blocks of similar controls, preferably with a title for each block.

"Controls," as we use the word here, include menus, dialog boxes, on-screen buttons, and any other graphic element that allows the user to interact with the computer. Modern *WIMP* (Windows-Icons-Menus-Pointer) systems are a natural expression of the Clustering Principle. Similar commands should be on the same menu, which places them in close proximity visually and gives them a single title. Large numbers of commands related to a given area of functionality may also show up in a dialog box, again a visually defined block.

But the same principle should apply if you are designing a special control screen with many buttons or displays visible, perhaps a touch-screen interface. The buttons for a given function should be grouped together, then visually delineated by color, or a bounding box, or surrounding space ("white space"). The principle should also be applied within WIMP systems when you design a dialog box: If there is a subgroup of related functions, put them together in the box.

There are two important reasons for the clustering principle. First, it helps users search for the command they need. If you're looking for the "Print setup" menu, it's easier to find if it's in a box or menu with the label "Printer" then if it's one of hundreds of command buttons randomly distributed on the top of the screen. Second, grouping commands together helps the user acquire a conceptual organization for facts about the program. It's useful to know, for example, that Bold, Italic, and Outline are all one kind of font modification, while Times Roman, Palatino, and Courier are another kind. (That distinction, common to most PC-based word processors, doesn't hold for many typesetting systems, where users have to acquire a different conceptual organization.)

The Visibility Reflects Usefulness Principle:

Make frequently used controls obvious, visible, and easy to access; conversely, hide or shrink controls that are used less often.

This is a principle that WIMP systems implement with dialog boxes and, in many recent systems, with "toolbars" of icons for frequently used functions. The reasoning behind this principle is that users can quickly search a small set of controls, and if that set contains the most

frequently used items, they'll be able to find and use those controls quickly. A more extended search, through dialog boxes, for example, is justified for controls that are used infrequently.

The Intelligent Consistency Principle:
Use similar screens for similar functions.

This is similar to intelligent borrowing, but in this case you're borrowing from one part of your design and applying it to another part. The reasoning should be obvious: Once users learn where the controls are on one screen (the "Help" button, for example), they should be able to apply that knowledge to other screens within the same system.

This approach lets you make a concentrated effort to design just a few attractive, workable screens, then modify those slightly for use in other parts of the application.

Be careful to use consistency in a meaningful way, however. Screens shouldn't look alike if they actually do significantly different things. A critical error warning in a real-time system should produce a display that's very different from a help screen or an informational message.

The Color As a Supplement Principle:
Don't rely on color to carry information; use it sparingly to emphasize information provided through other means.

Color is much easier to misuse than to use well. Different colors mean different things to different people, and that relationship varies greatly from culture to culture. Red, for example, means danger in the U.S., death in Egypt, and life in India. An additional problem is that some users can't distinguish colors: about 7 percent of all adults have some form of color vision deficiency.

A good principle for most interfaces is to design them in black and white, make sure they are workable, then add minimal color to the final design. Color is certainly useful when a warning or informational message needs to stand out, but be sure to provide additional cues to users who can't perceive the color change.

Unless you're an experienced graphic designer, minimal color is also the best design principle for producing an attractive interface. Try to stick with grays for most of the system, with a small amount of bright color in a logo or a label field to distinguish your product. Remember that many users can — and frequently do — revise the color of their

windows, highlighting, and other system parameters. Build a product that will work with that user input, not one that fights it.

The Reduced Clutter Principle:
Don't put "too much" on the screen.

This loosely defined principle is a good checkpoint to confirm that your design reflects the other principles listed above. If only the most highly used controls are visible, and if controls are grouped into a small number of visual clusters, and if you've used minimal color, then the screen should be graphically attractive.

This is also a good principle to apply for issues that we haven't dealt with specifically. Type size and font, for example: the Reduced Clutter Principle would suggest that one or two type styles are sufficient. Don't try to distinguish each menu by its own font, or work with a large range of sizes. Users typically won't notice the distinction, but they will notice the clutter.

Related exercise 3.2 ▷ see page 156

4 Evaluating the Design Without Users

Throughout this book we've emphasized the importance of bringing users into the interface design process. However, as a designer you'll also need to evaluate the evolving design when no users are present. Users' time is almost never a free or unlimited resource. Most users have their own work to do, and they're able to devote only limited time to your project. When users do take time to look at your design, it should be as free as possible of problems. This is a courtesy to the users, who shouldn't have to waste time on trivial bugs that you could have caught earlier. It also helps build the users' respect for you as a professional, making it more likely that they will give the design effort serious attention.

A second reason for evaluating a design without users is that a good evaluation can catch problems that an evaluation with only a few users may not reveal. The numbers tell the story here: An interface designed for a popular personal computer might be used by thousands of people, but it may be tested with only a few dozen users before beta release. Every user will have a slightly different set of problems, and the testing won't uncover problems that the few users tested don't have. It also won't uncover problems that users might have after they get more experience. An evaluation without users won't uncover all the problems either. But doing both kinds of evaluation significantly improves the chances of success.

In this chapter we describe three approaches to evaluating an interface in the absence of users. The first approach is the cognitive walkthrough, a task-oriented technique that fits especially well in the context of task-centered design. The second approach is action analysis, which allows a designer to predict the time that an expert user would need to perform a task, and which forces the designer to take a detailed look at the interface. The third approach is heuristic evaluation, a kind of check-list approach that catches a wide variety of problems but requires several evaluators who have some knowledge of usability problems.

We'll describe these techniques and show how each one applies to the analysis of a single interface. The interface we'll look at is the "Chooser" in an early version of the Apple Macintosh operating system. The Chooser lets the user select printers and printer options. For our task-oriented evaluations, the task will be to turn on background printing.

The Chooser is an interesting example because it's part of a system that was designed with usability and simplicity as paramount goals. Nonetheless, we'll see that it has some potential problems. Some of those problems have been corrected in later versions of the Mac operating system; others haven't,

possibly because the structure of the interface is too deeply embedded in the functionality of the system, or possibly because the changes would be too disruptive to existing users. (See the end of Appendix M for more thoughts on upgrading systems after they are in the field.)

[§M.4, p. 149]

Take a few minutes before you read the rest of this chapter to look over the following description of the Chooser and write down any problems you find.

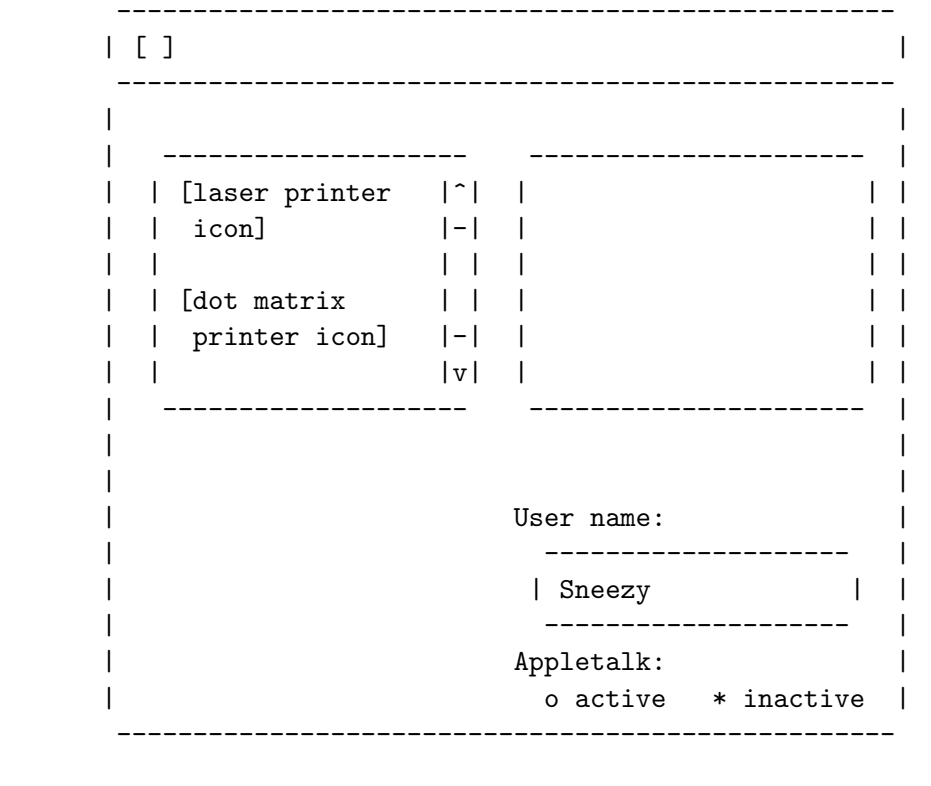
Example: Selecting Background Printing with the Mac Chooser

We've presented this example in the rough format that a system designer might use to describe a suggested system to his colleagues. This is the point in an interface design when you should be using the techniques described in this chapter, either alone or as a group — don't wait until after the system is implemented!

```
-------------------------------------------------------
START

User is working with word processor.  Task is to turn on
background printing.  Screen shows the current
application window and the following menubar of pulldown
menus:

    ----------------------------------------
     @ File Edit Search Font Utilities
    ----------------------------------------

("@" stands for the apple icon.)

-------------------------------------------------------
```

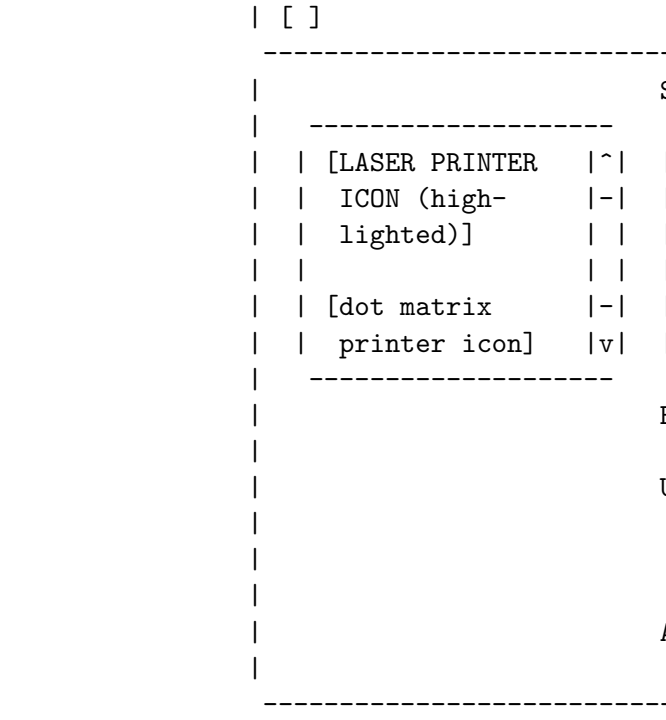
```
------------------------------------------------------------
ACTION 1:  Pull down the apple menu.

Apple pulldown menu looks like this:

      | @ |
       ------------------
      | About UltraWord  |
      | ---------------- |
      | Alarm Clock      |
      | Calculator       |
      | Chooser          |
      | Control Panel    |
      | Find File        |
      | Keycaps          |
      | Scrapbook        |
       ------------------

------------------------------------------------------------
```

```
------------------------------------------------------
ACTION 2:  Select "Chooser" from the menu.

A dialog box appears:
```




```
------------------------------------------------------
```

```
------------------------------------------------------
ACTION 3:  Click on current printer type, which is Laser.

The laser printer icon highlights and new things appear
in the dialog box:
```




```
------------------------------------------------------

------------------------------------------------------
ACTION 4:  Click the On button under background printing.

  The On button highlights and the Off button unhighlights.
------------------------------------------------------

------------------------------------------------------
ACTION 5:  Click the Close Box in the upper left window.

  The screen appears as it did at startup.
------------------------------------------------------
```

4.1 Cognitive Walkthroughs

The cognitive walkthrough is a formalized way of imagining people's thoughts and actions when they use an interface for the first time. Briefly, a walkthrough goes like this: You have a prototype or a detailed design description of the interface, and you know who the users will be. You select one of the tasks that the design is intended to support. Then you try to tell a believable story about each action a user has to take to do the task. To make the story believable you have to motivate each of the user's actions, relying on the user's general knowledge and on the prompts and feedback provided by the interface. If you can't tell a believable story about an action, then you've located a problem with the interface.

Example: A Quick Cognitive Walkthrough

Here's a brief example of a cognitive walkthrough, just to get the feel of the process. We're evaluating the interface to a personal desktop photocopier. A design sketch of the machine shows a numeric keypad, a "Copy" button, and a push button on the back to turn on the power. The machine automatically turns itself off after 5 minutes inactivity. The task is to copy a single page, and the user could be any office worker. The actions the user needs to perform are to turn on the power, put the original on the machine, and press the "Copy" button.

In the walkthrough, we try to tell a believable story about the user's motivation and interaction with the machine at each action. A first cut at the story for action number one goes something like this: The user wants to make a copy and knows that the machine has to be turned on. So she pushes the power button. Then she goes on to the next action.

But this story isn't very believable. We can agree that the user's general knowledge of office machines will make her think the machine needs to be turned on, just as she will know it should be plugged in. But why shouldn't she assume that the machine is already on? The interface description didn't specify a "power on" indicator. And the user's background knowledge is likely to suggest that the machine is normally on, like it is in most offices. Even if the user figures out that the machine is off, can she find the power switch? It's on the back, and if the machine is on the user's desk, she can't see it without getting up. The switch doesn't have any label, and it's not the kind of

switch that usually turns on office equipment (a rocker switch is more common). The conclusion of this single-action story leaves something to be desired as well. Once the button is pushed, how does the user know the machine is on? Does a fan start up that she can hear? If nothing happens, she may decide this isn't the power switch and look for one somewhere else.

That's how the walkthrough goes. When problems with the first action are identified, we pretend that everything has been fixed and we go on to evaluate the next action (putting the document on the machine).

You can see from the brief example that the walkthrough can uncover several kinds of problems. It can question assumptions about what the users will be thinking ("why would a user think the machine needs to be switched on?"). It can identify controls that are obvious to the design engineer but may be hidden from the user's point of view ("the user wants to turn the machine on, but can she find the switch?"). It can suggest difficulties with labels and prompts ("the user wants to turn the machine on, but which is the power switch and which way is on?"). And it can note inadequate feedback, which may make the users balk and retrace their steps even after they've done the right thing ("how does the user know it's turned on?").

The walkthrough can also uncover shortcomings in the current specification, that is, not in the interface but in the way it is described. Perhaps the copier design really was "intended" to have a power-on indicator, but it just didn't get written into the specs. The walkthrough will ensure that the specs are more complete. On occasion the walkthrough will also send the designer back to the users to discuss the task. Is it reasonable to expect the users to turn on the copier before they make a copy? Perhaps it should be on by default, or turn itself on when the "Copy" button is pressed.

Walkthroughs focus most clearly on problems that users will have when they first use an interface, without training. For some systems, such as "walk-up-and-use" banking machines, this is obviously critical. But the same considerations are also important for sophisticated computer programs that users might work with for several years. Users often learn these complex programs incrementally, starting with little or no training, and learning new features as they need them. If each task-oriented group of features can pass muster under the cognitive walkthrough, then the user will be able to progress smoothly from novice behavior to productive expertise.

One other point from the example: Notice that we used some features of the task that were implicitly pulled from a detailed, situated understanding of the task: the user is sitting at a desk, so she can't see the power switch. It would be impossible to include all relevant details like this in a written specification of the task. The most successful walkthroughs will be done by designers who have been working closely with real users, so they can create a mental picture of those users in their actual environments.

Now here's some details on performing walkthroughs and interpreting their results.

4.1.1 Who should do a walkthrough, and when?

If you're designing a small piece of the interface on your own, you can do your own, informal, "in your head" walkthroughs to monitor the design as you work. Periodically, as larger parts of the interface begin to coalesce, it's useful to get together with a group of people, including other designers and users, and do a walkthrough for a complete task. One thing to keep in mind is that the walkthrough is really a tool for developing the interface, not for validating it. You should go into a walkthrough expecting to find things that can be improved. Because of this, we recommend that group walkthroughs be done by people who are roughly at the same level in the company hierarchy. The presence of high-level managers can turn the evaluation into a show, where the political questions associated with criticizing someone else's work overshadow the need to improve the design.

4.1.2 What's needed before you can do a walkthrough?

You need information about four things. (1) You need a description or a prototype of the interface. It doesn't have to be complete, but it should be fairly detailed. Things like exactly what words are in a menu can make a big difference. (2) You need a task description. The task should usually be one of the representative tasks you're using for task-centered design, or some piece of that task. (3) You need a complete, written list of the actions needed to complete the task with the interface. (4) You need an idea of who the users will be and what kind of experience they'll bring to the job. This is an understanding you should have developed through your task and user analysis.

[§1.1 & §1.2 pp. 1*ff*]

HyperTopic: Common Mistakes in Doing a Walkthrough

In teaching people to use the walkthrough method, we've found that there are two common misunderstandings. We'll point those out here, so you'll be on guard to avoid them.

First, many evaluators merge point (3) above into the walkthrough evaluation process. That is, they don't know how to perform the task themselves, so they stumble through the interface trying to discover the correct sequence of actions — and then they evaluate the stumbling process.

There may be times when that approach is useful, but it misses an important part of the walkthrough method. In the walkthrough, you should *start with a correct list of the individual actions* needed to complete the given task: Click button "File", Type in "Smith Letter", Click button "OK," etc. If you have to explore the interface to identify those actions, fine — but that's not the walkthrough. The walkthrough begins when you have the list of actions in hand. The reason for this approach is that, in the best of all worlds, the user should identify and perform the optimal action sequence. So the walkthrough looks at exactly that sequence. If the walkthrough shows that the user may have trouble identifying or performing one of the actions, your primary interest is not what the user will do when that problem arises — what you really want to know is the fact that there is a problem here, which needs to be corrected.

The second point that many first-time users of the walkthrough method miss is to that the walkthrough *does not test real users on the system.* Of course, watching real users try out the interface can be a valuable approach, and we discuss it in detail in Chapter 5. But the walkthrough is an evaluation tool that helps you and your co-workers apply your design expertise to the evaluation of the interface. Because you can imagine the behavior of entire classes of users, the walkthrough will often identify many more problems than you would find with a single, unique user in a single test session.

4.1.3 What should you look for during the walkthrough?

You've defined the task, the users, the interface, and the correct action sequence. You've gathered a group of designers and other interested folk

together. Now it's time to actually *do the walkthrough*.

In doing the walkthrough, you try to tell a story about why the user would select each action in the list of correct actions. And you critique the story to make sure it's believable. We recommend keeping four questions in mind as you critique the story:

1. Will users be trying to produce whatever effect the action has?
2. Will users see the control (button, menu, switch, etc.) for the action?
3. Once users find the control, will they recognize that it produces the effect they want?
4. After the action is taken, will users understand the feedback they get, so they can go on to the next action with confidence?

Here are a few examples — "failure stories" — of interfaces that illustrate how the four questions apply.

The first question deals with what the user is thinking. Users often aren't thinking what designers expect them to think. For example, one portable computer we've used has a slow-speed mode for its processor, to save battery power. Assume the task is to do some compute-intensive spreadsheet work on this machine, and the first action is to toggle the processor to high-speed mode. Will users be trying to do this? Answer: Very possibly not! Users don't expect computers to have slow and fast modes, so unless the machine actually prompts them to set the option, many users may leave the speed set at its default value — or at whatever value it happened to get stuck in at the computer store.

The second question concerns the users' ability to locate the control — not to identify it as the right control, but simply to notice that it exists! Is this often a problem? You bet. Attractive physical packages commonly hide "ugly" controls. One of our favorite examples is an office copier that has many of its buttons hidden under a small door, which has to be pressed down so it will pop up and expose the controls. If the task is, for example, to make double-sided copies, then there's no doubt that users with some copier experience will look for the control that selects that function. The copier in question, in fact, has a clearly visible "help" sheet that tells users which button to push. But the buttons are hidden so well that many users have to ask someone who knows the copier where to find them. Other interfaces that take a hit on this question are graphic interfaces that require the user to hold some combination of keys while clicking or dragging with a mouse,

and menu systems that force the users to go through several levels to find an option. Many users will never discover what they're after in these systems without some kind of training or help.

[vocabulary problem, see also pp. 71, 123]

The third question involves identifying the control. Even if the users want to do the right thing and the control is plainly visible, will they realize that this is the control they're after? An early version of a popular word processor had a table function. To insert a new table the user selected a menu item named "Create Table." This was a pretty good control name. But to change the format of the table, the user had to select a menu item called "Format Cells." The designers had made an analogy to spreadsheets, but users weren't thinking of spreadsheets — they were thinking of tables. They often passed right over the correct menu item, expecting to find something called "Format Table." The problem was exacerbated by the existence of another menu item, "Edit Table," which was used to change the table's size.

Notice that the first three questions interact. Users might not want to do the right thing initially, but an obvious and well labeled control could let them know what needs to be done. A word processor, for example, might need to have a spelling dictionary loaded before the spelling checker could run. Most users probably wouldn't think of doing this. But if a user decided to check spelling and started looking through the menus, a "load spelling dictionary" menu item could lead them to take the right action. Better yet, the "check spelling" menu item could bring up a dialog box that asked for the name of the spelling dictionary to load.

The final question asks about the feedback after the action is performed. Generally, even the simplest actions require some kind of feedback, just to show that the system "noticed" the action: a light appears when a copier button is pushed, an icon highlights when clicked in a graphical interface, etc. But at a deeper level, what the user really needs is evidence that whatever they are trying to do (that "goal" that we identified in the first question) has been done, or at least that progress has been made. Here's an example of an interface where that fails. A popular file compression program lets users pack one or more files into a much smaller file on a personal computer. The program presents a dialog box listing the files in the current directory. The user clicks on each file that should be packed into the smaller file, then, after each file, clicks the "Add" button. But there's no change visible after a file has been added. It stays in the list, and it isn't highlighted or grayed. As a result, the user isn't sure that all the files have been added, so he or she may click on some files again — which causes them to be packed into the smaller file twice, taking up twice the space!

4.1.4 What do you do with the results of the walkthrough?

Fix the interface! Many of the fixes will be obvious: make the controls more obvious, use labels that users will recognize (not always as easy as it sounds), provide better feedback. Probably the hardest problem to correct is one where the user doesn't have any reason to think that an action needs to be performed. A really nice solution to this problem is to eliminate the action — let the system take care of it. If that can't be done, then it may be possible to re-order the task so users will start doing something they know needs doing, and then get prompted for the action in question. The change to the "spelling dictionary" interaction that we described is one example. For the portable computer speed problem, the system might monitor processor load and ask if the user wanted to change to low speed whenever the average load was low over a 20 minute period, with a similar prompt for high speed.

Example: Cognitive Walkthrough of Setting Mac Background Printing

The task is to turn on background printing. The interface and action sequence are described in the box at the beginning of the chapter. We'll consider users who have some experience with this computer, but who have never set the background printing option.

The walkthrough analysis proceeds action by action.

The first action is to pull down the apple menu. (We might also aggregate the first two actions, saying "select Chooser from the apple menu." The analysis should reveal pretty much the same things.) We predict that users want to find a menu item that lets them select background printing — that is, they want to do the right thing. The Apple menu is clearly visible, although it might not be clear that it's a menu to a first-time user. And since our users have used the Mac before, it's reasonable that they might look for a "system" function here. However they might also think background printing is a "print" option, and go looking under the "Print..." menu item under "File." This seems like it could be a problem. The low-level feedback is OK — a menu drops. But it would be better if the menu had some items in it that clearly had to do with printing.

The second action is to select "Chooser" from the menu. The user is still trying to find the "printer" options. Will they realize that "Chooser" is the right menu item? It seems like they could just as reasonably select "Control Panel." Once again, we have to fault the labeling. Once the action is performed, however, the printer icons in the dialog box at least suggest that this is the right tactic. But notice: still nothing visible about "background printing."

Third action is click on the current printer type, which is laser printer. Why would the user want to do this? We can't come up with any good rationalization. Some users may not even see that the action is available, since icons in a scrolling window aren't a common Mac interface technique. But the feedback, assuming the user does click the icon, is finally a success: here's the background printing toggle we were looking for (assuming the user sees it below the list of printer names).

Fourth action is to click "On" for background printing. This seems to be problem free. It's exactly what the user wants to do, the control is clearly visible, and the feedback — the button becomes filled in — is informative.

Final action is to close the Chooser window by clicking the close box in the upper left corner. We've been calling it a "dialog box," but it's actually a window filled with controls. This seems odd. Users with Mac experience will probably think that they need to click the "OK" button, but there isn't one. And the close box is in the diagonally opposite corner from where the "OK" button should be, so it may be especially hard to find. This also raises a feedback question. Users expect dialog boxes to put options into effect when "OK" is clicked. Since there's no "OK" button, users may wonder if just closing the window activated the options.

Summary

There's a real start-up problem with this task. The user wants to set "background printing" but he or she has to go through three levels of controls that don't seem very closely related to that goal: the apple menu, the "chooser" menu item, and the printer type. It seems like all of these things need to be revised to make the path more obvious. A better solution might be

to put the background printing toggle into a place where the user might already be looking: make it part of the dialog box that comes up when "Print" is selected from a "File" menu. The other noticeable problem with the interface is the use of a window instead of a dialog box with an "OK" button.

Related exercise 4.1 ▷ see page 158

Credits and Pointers: Cognitive Walkthrough

The cognitive walkthrough was developed by Clayton Lewis, Peter Polson, Cathleen Wharton, and John Rieman. A description of the theoretical foundations of the method can be found in:

– Polson, P.G., Lewis, C., Rieman, J., and Wharton, C. "Cognitive walkthroughs: A method for theory-based evaluation of user interfaces." International Journal of Man-Machine Studies, 36 (1992), pp. 741–773.

The method has been tested in several situations. A summary and discussion of those tests, along with additional examples of interface success and failure stories, is given in:

– Wharton, C., Rieman, J., Lewis, C., and Polson, P. "The cognitive walkthrough: A practitioner's guide." In J. Nielsen and R.L. Mack (Eds.), Usability Inspection Methods, New York: John Wiley & Sons (1994).

4.2 Action Analysis

Action analysis is an evaluation procedure that forces you to take a close look at the sequence of actions a user has to perform to complete a task with an interface. In this book we'll distinguish between two flavors of action analysis. The first, "formal" action analysis, is often called "keystroke-level analysis" in HCI work. The formal approach is characterized by the extreme detail of the evaluation. The detail is so fine that, in many cases, the analysis

can predict the time to complete tasks within a 20 percent margin of error, even before the interface is prototyped. It may also predict how long it will take a user to learn the interface. Unfortunately, formal action analysis isn't easy to do.

The second flavor of action analysis is what we call the "back of the envelope" approach. This kind of evaluation won't provide detailed predictions of task time and interface learnability, but it can reveal large-scale problems that might otherwise get lost in the forest of details that a designer is faced with. As its name implies, the back-of-the-envelope approach doesn't take a lot of effort.

Action analysis, whether formal or back-of-the-envelope, has two fundamental phases. The first phase is to decide what physical and mental steps a user will perform to complete one or more tasks with the interface. The second phase is to analyze those steps, looking for problems. Problems that the analysis might reveal are that it takes too many steps to perform a simple task, or it takes too long to perform the task, or there is too much to learn about the interface. The analysis might also reveal "holes" in the interface description — things that the system should be able to do but can't. And it could be useful in writing or checking documentation, which should describe the facts and procedures that the analysis has shown the user needs to know.

4.2.1 Formal Action Analysis

The formal approach to action analysis has been used to make accurate predictions of the time it takes a skilled user to complete tasks. To predict task times, the times to perform each small step of the task, physical or mental, are estimated, and those times are totalled. Most steps take only a fraction of a second. A typical step is a keystroke, which is why the formal approach is often called "keystroke-level analysis."

The predictions of times for each small step are found by testing hundreds of individual users, thousands of individual actions, and then calculating average values. These values have been determined for most of the common actions that users perform with computer interfaces. We summarize those values in the tables below. If an interface control isn't in the table, it might be possible to extrapolate a reasonable value from similar devices, or user testing might have to be done for the new control.

The procedure for developing the list of individual steps is very much like programming a computer. The basic task is divided into a few subtasks, like subroutines in a computer program. Then each of those subtasks is

broken into smaller subtasks, and so on until the description reaches the level of the fraction-of-a-second operations listed in the table. The end result is a hierarchical description of a task and the action sequence needed to accomplish it.

Table: Average times for computer interface actions[2]

Physical Movements	
Enter one keystroke on a standard keyboard Ranges from .07 second for highly skilled typists doing transcription, to .2 second for an average 60-wpm typist, to over 1 second for a bad typist. Random sequences, formulas, and commands take longer than plain text.	.28 second
Use mouse to point at object on screen May be slightly lower — but still at least 1 second — for a small screen and a menu. Increases with larger screens, smaller objects.	1.5 second
Move hand to pointing device or function key Ranges from .21 second for cursor keys to .36 second for a mouse.	.3 second
Visual Perception	
Respond to a brief light Varies with intensity, from .05 second for a bright light to .2 second for a dim one.	.1 second
Recognize a 6-letter word	.34 second
Move eyes to new location on screen (saccade)	.23 second

Table continues . . .

[2][Based on detailed information in Judith Reitman Olson and Gary M. Olson, "The growth of cognitive modeling in human-computer interaction since GOMS," Human-Computer Interaction, 5 (1990), pp. 221–265. Many values given in this table are averaged and rounded.]

Table: Average times for computer interface actions (cont.)

Mental Actions	
Retrieve a simple item from long-term memory A typical item might be a command abbreviation ("dir"). Time is roughly halved if the same item needs to be retrieved again immediately.	1.2 second
Learn a single "step" in a procedure May be less under some circumstances, but most research shows 10 to 15 seconds as a minimum. None of these figures include the time needed to get started in a training situation.	25 seconds
Execute a mental "step" Ranges from .05 to .1 second, depending on what kind of mental step is being performed.	.075 second
Choose among methods Ranges from .06 to at least 1.8 seconds, depending on complexity of factors influencing the decision.	1.2 second

Example: Formal action analysis

Here's an example of a formal action analysis, roughly in the style described by David Kieras (see Credits and Pointers).

The top-level goal is to turn on background printing. Our top-level analysis of the task is:

Method to accomplish goal of turning on background printing

Step 1. Accomplish goal of making printer dialog box visible.

Step 2. Accomplish goal of setting printer controls.

Step 3. Accomplish goal of putting away printer dialog box.

Step 4. Report goal accomplished

Here are some points to note about this first level. It doesn't include any actual, physical or mental actions. It is a "method" (the human version of a subroutine), and it's described in terms of goals that need to be accomplished. As we'll see, each of

these goals will also be achieved with "methods." Also note that the three-step process is one example of a very common, generic Mac sequence: get a dialog box, set its controls, put it away. We could replace the word "printer" with "file" in Steps 1–3 and the procedure would work for saving a file. That says something good about the interface: it's consistent, so the user can learn one procedure and apply it to accomplish many things. Finally, note that there's an explicit step to report that the goal has been accomplished. This is parallel to the "return" call of a computer subroutine. In some procedures we might also make verifying the goal an explicit step.

Next we have to expand each step into methods. Step 1 expands into selecting from the menu. We'll skip the details of that and go on to expanding Step 2:

Method to accomplish goal of setting printer controls

Step 1. Accomplish goal of specifying printer type.

Step 2. Accomplish goal of setting background printing control.

Step 3. Report goal accomplished.

The methods we're describing are an explicit description of how we believe the user will conceptualize the task. Here we've had to make a decision, a judgement call, about what the user thinks. We've decided to break setting the printer controls into two steps, each with its own method. But another analyst might argue that the user will think of the task at this level as a single method or procedure, a single series of actions with the single goal of setting the background printing option in the dialog box. The results of the analysis could be somewhat different if that approach were taken.

We'll stay with the two-method approach and expand its first step. As it happens, there are two ways to specify the printer type. The user can click on the laser printer icon, as shown in the action sequence at the beginning of the chapter. Or, the user can type the first letter of the icon name, "Laser Printer." The formal analysis requires that we write a "selection rule," an if-then statement, that describes what conditions the user will consider to choose between the two techniques.

Selection rule for goal of specifying printer type.

```
IF hand is on mouse
THEN accomplish goal of specifying printer with mouse.
IF hands are on the keyboard
THEN accomplish goal of specifying printer with keyboard
Report goal accomplished
```

Now we'll expand the first of those options.

Method for accomplishing goal of specifying printer with mouse.

Step 1. Recall current printer type.

Step 2. Match type to name/icon in list.

Step 3. Move to and click on icon.

Step 4. Verify that icon is selected.

Step 5. Report goal accomplished.

This is the lowest level of the hierarchy for this action. It describes low-level physical and mental actions at roughly the "keystroke" level. We can assign times to those actions as follows:

Recall current printer type: 1.2 second
: This is just recalling an item from long-term memory

Match type to name/icon list: 1 second
: This is a combination of moving the eyes, reading the name of the icon, and performing the mental step of matching. It would take longer if there were more than two icons to choose from.

Move to and click on icon: 1.5 second
: Standard mouse movement time. We know from the selection rule that the hand is already on the mouse.

System time to highlight icon: .1 second
: We've just guessed at this.

Verify that icon is selected: .2 second
: This is a combination of recognizing the change in intensity and making the mental step of interpreting it.

Report goal accomplished. .1 second
: A simple mental step.

This totals to 4 seconds. The selection rule that preceded the actions would add at least 1 second more. If you have a Mac handy and actually time yourself, you'll probably do the task a little faster. That's largely because you've already decided on a method, recalled your printer type, and have it's icon in mind (which will speed the matching process). But the method predicts that, on the average for a skilled user, setting the printer type will take about 5 seconds.

Points Raised by the Analysis

To complete the formal analysis we'd have to go down to the keystroke level for each of the steps in the process. We'd find that the full task would take on the order of 10 to 15 seconds on a fast Mac with a hard drive, but it could take 30 seconds or more on one of the early Macs with only a floppy drive. Most of the system time goes into opening and closing the dialog box. Time-wise, any small amount that might be shaved off the 8 to 10 seconds of user activity would be overshadowed by the savings that could be achieved by providing faster hardware.

But the analysis can highlight some problems in areas other than task time. We noted that evaluators might differ on whether "setting printer type" should be a separate goal or merged into the goal of setting background printing. If the evaluators have trouble making that decision, it's a good bet that users will have a related difficulty. The analysis doesn't really suggest a solution here, only a potential problem.

Another question that the analysis highlights is the need for a selection rule to decide between keyboard and mouse selection of the printer type. Is the rule we've proposed a good one? If the user has just used the mouse to choose from the menu, will the keyboard option ever be appropriate? This requires some further thought, since dialog boxes with selectable lists are a common feature in the interface. The selection technique should be consistent, but it should also be quick. Do the occasions when the keyboard will be used justify a 1 second decision penalty every time a list appears?

The analysis might also call into question the large number of steps needed to do this fairly simple task. Of course, as you can probably see, any analysis at this level produces a "large" num-

ber of steps, so we can't be sure there's a problem here without comparing the task to some other tasks of similar complexity. A related issue is the task's long-term memory requirements. The analysis makes it clear that the user must recall the printer type. Will the user know the printer type?

Summary

The formal analysis takes a lot of work, and it's not clear that it was worth it in this case, at least not for the timing results. But looking closely at the actions did reveal a few problems that go deeper than timing, including the way the user should think about setting printer type, the question of multiple selection methods for lists, and the amount of knowledge required to do this apparently trivial task.

A full-blown formal analysis of a complex interface is a daunting task. The example and the size of the time values in the table should give some idea of why this is so. Imagine you want to analyze two designs for a spreadsheet to see which is faster on a given task. The task is to enter some formulas and values, and you expect it to take the skilled user on the order of 10 minutes. To apply the formal action analysis approach you'll have to break the task down into individual actions, most of which take less than a second. That comes out to around 1000 individual actions, just to analyze a single 10-minute task! (There will probably be clusters of actions that get repeated; but the effort is still nontrivial.)

A further problem with formal analysis is that different analysts may come up with different results, depending on how they see the task hierarchy and what actions they predict a user will take in a given situation. (Will the user scan left, then down the spreadsheet? Down then left? Diagonally? The difference might be seconds, swamping other details.) Questions like this may require user testing to settle.

Because it's so difficult, we think that formal action analysis is useful only in special circumstances — basically, when its high cost can be justified by a very large payoff. One instance where this was the case was the evaluation of a proposed workstation for telephone operators (see the article by Gray et al listed in Credits and Pointers, below). The phone company contracting the action analysis calculated that a savings of a few seconds in a procedure performed by thousands of operators over hundreds of thousands of calls would more than repay the months of effort that went into the evaluation.

Another place formal action analysis can be effective is for segments of the interface that users will access repeatedly as part of many tasks. Some examples of this are choosing from menus, selecting or moving objects in a graphics package, and moving from cell to cell within a spreadsheet. In each of these examples, a savings of a few tenths of a second in an interaction might add up to several minutes during an hour's work. This could justify a detailed analysis of competing designs.

In most cases, however, a few tenths of a second saved in performing an action sequence, and even a few minutes saved in learning it, are trivial compared to the other aspects of the interface that we emphasize in this book. Does the interface (and the system) do what the user needs, fitting smoothly into the rest of the user's work? Can the user figure out how the interface works? Does the interface's combination of controls, prompts, warning messages, and other feedback allow the user to maintain a comfortable "flow" through a task? If the user makes an error, does the system allow graceful recovery? All of these factors are central, not only to productivity but also to the user's perception of the system's quality. A serious failure on any of these points isn't going to be countered by shaving a few seconds off the edges of the interface.

4.2.2 Back-of-the-Envelope Action Analysis

The back-of-the-envelope approach to action analysis foregoes detailed predictions in an effort to get a quick look at the big picture. We think this technique can be very valuable, and it's easy to do. Like the formal analysis, the back-of-the-envelope version has two phases: list the actions, then think about them. The difference is that you don't need to spend a lot of time developing a detailed hierarchical breakdown of the task. You just list a fairly "natural" series of actions and work with them.

A process that works well for listing the actions is to imagine you are explaining the process to a typical user. That means you aren't going to say, "take your hand off the keyboard and move it to the mouse," or "scan down the menu to the 'chooser' item." You'll probably just say, "select 'chooser' from the apple menu." You should also put in brief descriptions of mental actions, such as "remember your password," or "convert the time on your watch to 24-hour time."

Once you have the actions listed there are several questions you can ask about the interface:

- Can a simple task be done with a simple action sequence?

- Can frequent tasks be done quickly?
- How many facts and steps does the user have to learn?
- Is everything in the documentation?

You can get useful answers to all these questions without going into fraction-of-a-second details. At the action level you'd use in talking to a user, *every action takes at least two or three seconds.* Selecting something from a menu with the mouse, entering a file name, deciding whether to save under a new name or the old one, remembering your directory name — watch over a user's shoulder sometime, or videotape a few users doing random tasks, and you'll see that combined physical and mental time for any one of these actions is a couple of seconds on a good day, three or four or even ten before morning coffee. And you'll have to start measuring in minutes whenever there's any kind of an error or mistake.

By staying at this level of analysis, you're more likely to keep the task itself in mind, along with the user's work environment, instead of getting lost in a detailed comparison of techniques that essentially do the same thing. For example, you can easily use the back-of-the-envelope results to compare your system's proposed performance with the user's ability to do the same task with typewriters, calculators, and file cabinets.

This kind of action analysis is especially useful in deciding whether or not to add features to an interface, or even to a system. Interfaces have a tendency to accumulate features and controls like a magnet accumulates paperclips in a desk drawer. Something that starts out as a simple, task-oriented action sequence can very quickly become a veritable symphony of menus, dialog boxes, and keystrokes to navigate through the options that various people thought the system should offer. Often these options are intended as "time savers," but the user ends up spending an inordinate amount of time just deciding which time saver to use and which to ignore. (One message you should take away from the table of action times is that it takes real time to decide between two ways of doing something.)

A few quick calculations can give you ammunition for convincing other members of a development team which features should or should not be added. Of course, marketing arguments to the contrary may prevail: it often seems that features sell a program, whether or not they're productive. But it's also true that popular programs sometimes become so complicated that newer, simpler programs move in and take over the low end of the market. The newcomers may even eventually displace the high-functionality leaders.

(An example of this on a grand scale is the effect of personal computers on the mainframe market.)

Example: Back-of-the-envelope action analysis

Here's an example of an interface where a quick, informal action analysis shows real problems, which were gradually alleviated as the interface evolved. (We'll spare you from reading a second action analysis of the Mac chooser.) The interface is the controls to a 35-mm camera. When these cameras first became popular in the 1950s, the action sequence needed to take a picture went something like this:

- take light meter out of pocket
- make sure film speed is set correctly on light meter
  - remember: what speed of film is in the camera
- aim light meter at scene
- read light value from needle on light meter
- adjust calculator dial on light meter to light value
- the calculator shows several possible combinations of f-stop and shutter speed, all of which give the correct exposure. So...
  - remember: big f-stop number means small lens opening
  - remember: small lens opening means more depth of field
  - remember: big shutter speed number means short exposure
  - remember: short exposure means moving things aren't blurred
  - decide: which combination of f-stop and speed to use
- set the f-stop on the camera lens
- set the shutter speed on the camera speed dial
- remove the lens cap
- cock the shutter
- aim the camera
- focus the camera

  - remember: if lens aperture is small, depth of field is shallow
- press the shutter release
- advance the film

(Whew!)

Starting with the action list shown above, the camera designers could make rough estimates, or observe one or two users with a mock-up, and find that most actions in the list would take at least a couple of seconds. Some might take several seconds. These ballpark results would predict that it could take on the order of 30 seconds to take a photograph, and that's for a skilled user!

The analysis should make it clear that the interface would be hard to operate for several reasons:

- It takes too long to take a picture.
- There are too many actions to be learned.
- Some of the actions have to be "understood" — for example, what *is* depth of field? This makes the interface harder to learn and slower to use.
- Every action is an opportunity for error, and almost any error can ruin the final result.

The analysis would also show that more than half the time was spent metering the light and setting the exposure, a very system-oriented activity that many camera users don't want to think about.

We don't know if there was ever a back-of-the-envelope action analysis done on 35-mm cameras, but the development of the camera interface can be broadly described as an ongoing effort to combine or eliminate the steps described above, with the exposure-setting functions being the first to go. The ultimate realization of this effort is the modern "point-and-shoot" pocket 35-mm. The camera reads the film speed off the film cassette, light-meters the scene, selects a combination of shutter speed and f-stop, and turns on a flash if needed. (We didn't even mention the half-dozen or more additional actions needed to do flash photography before electronics!) Focusing with the point-and-shoot camera is automatic, and a motor cocks the shutter and

winds the film. All that's left for the user is to aim and press the shutter release.

However, another look at the example illustrates some of the shortcomings of any action analysis, formal or informal. The modern, fully automated 35-mm camera isn't the only kind of camera people buy today. Many people buy cameras that are not fully automated or that let the user override the automation. Some of those purchasers are experienced photographers who believe they can get better or more varied results manually. Some are people who just like to have a lot of controls on their cameras. Whatever the reason, these contextual and market factors won't show up in an action analysis. But they can be discovered by designers who spend time getting to know the users.

Related exercise 4.2 ▷ see page 159

Credits and Pointers: Action analysis

The quantitative, engineering-style analysis of actions was initially developed by Stuart Card, Thomas Moran, and Alan Newell, based on theories of cognitive psychology that Newell produced with Herbert Simon. The Card, Moran, and Newell approach is referred to as GOMS modelling (for Goals, Operators, Methods, and Selection). It's described in detail in:

> Card, S., Moran, T., and Newell, A. "The Psychology of Human-Computer Interaction." Hillsdale, New Jersey: Lawrence Erlbaum, 1983.

Important extensions to the work, including verifications and additions to the list of action times, have been made by a number of researchers. An excellent overview of this work is given in:

> Olson, Judith Reitman, and Olson, Gary M. "The growth of cognitive modeling in human-computer interaction since GOMS." Human-Computer Interaction, 5 (1990), pp. 221–265.

The detailed action analysis on telephone operator workstations is described in:

> Gray, W.D., John, B.E., Stuart, R., Lawrence, D., Atwood, M.E. "GOMS meets the phone company: Analytic modeling applied to real-world problems." Proc. IFIP Interact'90: Human-Computer Interaction. 1990, pp. 29–34.

The back-of-the-envelope analysis we describe draws on observations by David Kieras on the procedures for doing formal action analysis and interpreting its results. Kieras has worked to make the GOMS method easier to learn and more reliable. If you need to learn more about formal action analysis, a good place to start would be:

> Kieras, D.E. "Towards a practical GOMS model methodology for user interface design." In M. Helander (Ed.), "Handbook of Human-Computer Interaction" Amsterdam: Elsevier Science (North-Holland), 1988.

4.3 Heuristic Analysis

Heuristics, also called guidelines, are general principles or rules of thumb that can guide design decisions. As soon as it became obvious that bad interfaces were a problem, people started proposing heuristics for interface design, ranging from short lists of very general platitudes ("be informative") to a list of over a thousand very detailed guidelines dealing with specific items such as menus, command names, and error messages. None of these efforts has been strikingly successful in improving the design process, although they're usually effective for critiquing favorite bad examples of someone else's design. When the short lists are used during the design process, however, a lot of problems get missed; and the long lists are usually too unwieldy to apply. In addition, all heuristics require that an analyst have a fair amount of user interface knowledge to translate the general principles into the specifics of the current situation.

Recently, Jacob Nielsen and Rolf Molich have made a real breakthrough in the use of heuristics. Nielsen and Molich have developed a short list of general heuristics, and more importantly, they've developed and tested a procedure for using them to evaluate a design. We give the details of that procedure below, but first we want to say something about why heuristics,

which are not necessarily a task-oriented evaluation technique, can be an important part of task-centered design.

The other two evaluation methods described in this chapter, the cognitive walkthrough and action analysis, are task-oriented. That is, they evaluate an interface as applied to a specific task that a user would be doing with the interface. User testing, discussed in chapter 6, is also task oriented. Task-oriented evaluations have some real advantages. They focus on interface problems that occur during work the user would actually be doing, and they give some idea of the importance of the problems in the context of the job. Many of the problems they reveal would only be visible as part of the sequence of actions needed to complete the task. But task-oriented evaluations also have some shortcomings. The first shortcoming is coverage: There's almost never time to evaluate every task a user would perform, so some action sequences and often some controls aren't evaluated. The second shortcoming is in identifying cross-task interactions. Each task is evaluated standing alone, so task-oriented evaluations won't reveal problems such as command names or dialog-box layouts that are done one way in one task, another way in another.

Task-free evaluation methods are important for catching problems that task-oriented methods miss. Both approaches should be used as the interface develops. Now, here's how the heuristic analysis approach works.

Nielsen and Molich used their own experience to identify nine general heuristics (see table, below), which, as they noted, are implicit or explicit in almost all the lists of guidelines that have been suggested for HCI. Then they developed a procedure for applying their heuristics. The procedure is based on the observation that no single evaluator will find every problem with an interface, and different evaluators will often find different problems. So the procedure for heuristic analysis is this: Have several evaluators use the nine heuristics to identify problems with the interface, analyzing either a prototype or a paper description of the design. Each evaluator should do the analysis alone. Then combine the problems identified by the individual evaluators into a single list. Combining the individual results might be done by a single usability expert, but it's often useful to do this as a group activity.

Table: Nielsen and Molich's Nine Heuristics
Simple and natural dialog Simple means no irrelevant or rarely used information. Natural means an order that matches the task.
Speak the user's language Use words and concepts from the user's world. Don't use system-specific engineering terms.
Minimize user memory load Don't make the user remember things from one action to the next. Leave information on the screen until it's not needed.
Be consistent Users should be able to learn an action sequence in one part of the system and apply it again to get similar results in other places.
Provide feedback Let users know what effect their actions have on the system.
Provide clearly marked exits If users get into part of the system that doesn't interest them, they should always be able to get out quickly without damaging anything.
Provide shortcuts Shortcuts can help experienced users avoid lengthy dialogs and informational messages that they don't need.
Good error messages Good error messages let the user know what the problem is and how to correct it.
Prevent errors Whenever you write an error message you should also ask, can this error be avoided?

The procedure works. Nielsen and Molich have shown that the combined list of interface problems includes many more problems than any single evaluator would identify, and with just a few evaluators it includes most of the major problems with the interface. Major problems, here, are problems that confuse users or cause them to make errors. The list will also include less critical problems that only slow or inconvenience the user.

How many evaluators are needed to make the analysis work? That depends on how knowledgeable the evaluators are. If the evaluators are experienced interface experts, then 3 to 5 evaluators can catch all of the "heuristically identifiable" major problems, and they can catch 75 percent of the

total heuristically identifiable problems. (We'll explain what "heuristically identifiable" means in a moment.) These experts might be people who've worked in interface design and evaluation for several years, or who have several years of graduate training in the area. For evaluators who are also specialists in the specific domain of the interface (for example, graphic interfaces, or voice interfaces, or automated teller interfaces), the same results can probably be achieved with 2 to 3 evaluators. On the other hand, if the evaluators have no interface training or expertise, it might take as many as 15 of them to find 75 percent of the problems; 5 of these novice evaluators might find only 50 percent of the problems.

We need to caution here that when we say "all" or "75 percent" or "50 percent," we're talking only about "heuristically identifiable" problems. That is, problems with the interface that actually violate one of the nine heuristics. What's gained by combining several evaluator's results is an increased assurance that if a problem can be identified with the heuristics, then it will be. But there may still be problems that the heuristics themselves miss. Those problems might show up with some other evaluation method, such as user testing or a more task-oriented analysis.

Also, all the numbers are averages of past results, not promises. Your results will vary with the interface and with the evaluators. But even with these caveats, the take-home message is still very positive: Individual heuristic evaluations of an interface, performed by 3 to 5 people with some expertise in interface design, will locate a significant number of the major problems.

To give you a better idea of how Nielsen and Molich's nine heuristics apply, one of the authors has done a heuristic evaluation of the Macintosh background printing controls (see Box).

Example: One Evaluator's Heuristic Analysis of the Mac Background Printing Controls

Remember, for an effective heuristic analysis, *several* evaluators should check out the interface independently and combine their notes into a single problem list.

Since heuristic analysis is usually a task-free approach, the Mac background printing controls would probably be presented to the evaluator as part of an overall description of the Chooser, or perhaps of the Mac system interface as a whole. If just the Chooser were being checked, the design description might specify how the dialog box would be brought up through the Apple menu (shown in Steps 1 and 2) and it might have sketches of how the box would look before and after a printer was specified (shown in Steps 3 and 4). It would also give some description of how to use the controls in the dialog box.

[Mac printing, see Example pp. 52*ff*]

Here's my evaluation. This piece of the interface is small enough that I can just go through the nine heuristics one at a time and look for possible problems anywhere in the design. (For a larger design, I might want to step through the nine heuristics several times, once for each manageable chunk of the interface.)

The first heuristic is "simple and natural dialog." OK, the idea of this design seems to be that I select what kind of printer I'm going to use, then specify some options. I guess that will work, but why do I have to select the kind of printer and say which one? Couldn't I tell it which one and let the system figure out whether it's a laser or dot-matrix printer?

Second heuristic: "speak the user's language." I can see an effort to do this. They've called it the "Chooser" instead of "Output options." But an even better name might be "Printer" — parallel to "Alarm Clock" and "Calculator." I don't know what "Appletalk" is, though. Is there some more common name for this? And why don't the printers have names like "central office" or "library," indicating where they are?

[related to vocabulary problem, pp. 51, 123]

Heuristic: "Minimize user memory load." This heuristic primarily applies to things the user has to notice in one step and remember one or more steps later. Everything stays pretty visible here, so the memory load is low. But I do have to remember what kind of printer I'm using and what its name is.

Heuristic: "Be consistent." I don't think I've seen a scrolling list of clickable icons in any other dialog box. Do the icons add anything here? Maybe there should just be a scrolling list of text entries.

Heuristic: "Provide feedback." Design seems good to me. Something happens whenever I do anything. Of course, there's no evidence of what I've done after the dialog box goes away. But that seems OK here. I don't mind having some options set "behind the scenes," as long as I can check them whenever I want.

Heuristic: "Provide clearly marked exits." Well, looking at the dialog box, I see that it has a "close box" in the upper left. I kind of expect an "OK" and a "Cancel" button in a dialog box. Maybe I should have noted this under consistency with the rest of the interface? Anyway, it might confuse some people.

Heuristic: "Provide shortcuts." I suppose the usual Mac menu-selection shortcuts work — like typing "M" to select "Mary's printer." Should there be other shortcuts? I dunno. Depends on whether people need to do something really often, like toggling AppleTalk or Background printing. Need some more task analysis here.

Heuristic: "Good error messages." Current design doesn't describe any error messages. I guess I'd like users to get an error if they selected a printer that wasn't connected to the computer. Need to check on that. Also, what if they turn off background printing while something is in the queue?

Heuristic: "Prevent errors." Same comments as for error messages, except it would be better if the user wasn't given options that wouldn't work. So if there isn't an ImageWriter connected, they shouldn't be able to select it — it should probably be grayed out, or not there at all.

Summary

Based on what I've noticed in the heuristic analysis, I'd like to see some changes. I'd want the menu item to be called "Printer," and I'd want the dialog box to give me a list of currently available printer locations to choose from. The system should figure out what kind of printer it is. Also, I'd use an "OK" and a "Cancel" button instead of a window close box.

(Some discussion with the designers would probably raise the point that selecting a printer type effects how a document is formatted, even if a printer isn't connected. I'd have to argue that this really isn't "speaking the user's language." I think most users would expect to find formatting options under the format menu in the application.)

Related exercise 4.3 ▷ see page 159

Credits and Pointers: Heuristic Analysis

The heuristic analysis technique was developed by Jacob Nielsen and Rolf Molich. Tests of the technique's effectiveness are described in:

- Nielsen, J. and Molich, R. "Heuristic evaluation of user interfaces." Proc. CHI'90 Conference on Human Factors in Computer Systems. New York: ACM, 1990, pp. 249–256.
- Nielsen, J. "Finding usability problems through heuristic evaluation." Proc. CHI'92 Conference on Human Factors in Computer Systems. New York: ACM, 1992, pp. 373–380.

More detailed descriptions of the nine heuristics are given in:

- Molich, R. and Nielsen, J. "Improving a human-computer dialogue: What designers know about traditional interface design." Communications of the ACM, 33 (March 1990), pp. 338–342.

– Nielsen, J. "Usability Engineering." San Diego, CA: Academic Press, 1992.

4.4 Chapter Summary and Discussion

[Summaries pp. 53, 61, 73]

We've looked at three methods for evaluating an interface without users. If you scan through the "Summary" sections at the ends of the examples, you'll see that each method uncovered different problems. The cognitive walkthrough identified problems with finding the controls and suggested that they be moved to the Print dialog box. A one-person heuristic analysis noted the same problem and some others, and suggested that the controls should be relabeled. The formal action analysis suggested a vague question about the printer-type selection and made it clear that there were a lot of actions needed to do this simple task. A back-of-the-envelope action analysis might have given similar results, without details of how long the task took.

Even the combined result of the three techniques didn't catch all the problems. That's partly because there are kinds of problems that none of these methods will catch, and partly because no individual analysis is perfect. One example of something none of the methods caught, although both the cognitive walkthrough and the heuristic analysis noted related problems, is that "Chooser" and "Control Panel" have similar functions (setting options about the system) and also similar names (starting with C, two o's in the middle). It's likely that users will sometimes slip and pick the wrong one even when they know which one they're after, especially since items are alphabetized in the Apple menu. This problem could surface in user testing, or at least in beta testing. Another problem, which another heuristic analyst might have caught, is that the dialog box doesn't let users know which printer type is selected when it first opens. A user who isn't sure what kind of printer is being used may change to the wrong type, after which printing may not work at all.

Here's a recommendation on how to use the various methods in the design process. In general, the cognitive walkthrough will give you the best understanding of problems it uncovers, and we recommend thinking through the interface in walkthrough terms as you develop it. When a substantial part of the interface is complete, heuristic analysis is a good way to catch additional problems — but you need several evaluators to do it well. Back-of-the-envelope action analysis makes a good "sanity check" as the interface becomes more complex, and it's also useful early in system design to help

decide whether a system's features will pay back the effort of learning and using them. Formal action analysis is probably only appropriate for very special cases, as we described in that section. All the methods need to be combined with user testing, which we discuss in the next chapter.

5 Testing The Design With Users

You can't really tell how good or bad your interface is going to be without getting people to use it. So as your design matures, but before the whole system gets set in concrete, you need to do some user testing. This means having real people try to do things with the system and observing what happens. To do this you need people, some tasks for them to perform, and some version of the system for them to work with. Let's consider these necessities in order.

5.1 Choosing Users to Test

The point of testing is to anticipate what will happen when real users start using your system. So the best test users will be people who are representative of the people you expect to have as users. If the real users are supposed to be doctors, get doctors as test users. If you don't, you can be badly misled about crucial things like the right vocabulary to use for the actions your system supports. Yes, we know it isn't easy to get doctors, as we noted when we talked about getting input from users early in design. But that doesn't mean it isn't important to do. And, as we asked before, if you can't get any doctors to be test users, why do you think you'll get them as real users?

If it's hard to find really appropriate test users you may want to do some testing with people who represent some approximation to what you really want, like medical students instead of doctors, say, or maybe even premeds, or college-educated adults. This may help you get out some of the big problems (the ones you overlooked in your cognitive walkthrough because you knew too much about your design and assumed some things were obvious that aren't). But you have to be careful not to let the reactions and comments of people who aren't really the users you are targeting drive your design. Do as much testing with the right kind of test users as you can.

HyperTopic: Ethical Concerns in Working with Test Users

> Serving as a test user can be very distressing, and you have definite responsibilities to protect the people you work with from distress. We have heard of test users who left the test in tears, and of a person in a psychological study of problem solving who was taken away in an ambulance under a sedation because of not being able to solve what appeared to be simple logic puzzles. There's no joke here.

Another issue, which you also have to take seriously, is embarrassment. Someone might well feel bad if a video of them fumbling with your system were shown to someone who knew them, or even if just numerical measures of a less-than-stellar performance were linked with their name.

The first line of defense against these kinds of problems is voluntary, informed consent. This means you avoid any pressure to participate in your test, and you make sure people are fully informed about what you are going to do if they do participate. You also make clear to test users that they are free to stop participating at any time, and you avoid putting any pressure on them to continue, even though it may be a big pain for you if they quit. You don't ask them for a reason: if they want to stop, you stop.

Be very careful about getting friends, co-workers, or (especially) subordinates to participate in tests. Will these people really feel free to decline, if they want to? If such people are genuinely eager to participate, fine, but don't press the matter if they hesitate even (or especially) if they give no reason.

During the test, monitor the attitude of your test users carefully. You will have stressed that it is your system, not the users, that is being tested, but they may still get upset with themselves if things don't go well. Watch for any sign of this, remind them that they aren't the focus of the test, and stop the test if they continue to be distressed. We are opposed to any deception in test procedures, but we make an exception in this case: an "equipment failure" is a good excuse to end a test without the test user feeling that it is his or her reaction that is to blame.

Plan carefully how you are going to deal with privacy issues. The best approach is to avoid collecting information that could be used to identify someone. We make it a practice not to include users' faces in videos we make, for example, and we don't record users' names with their data (just assign user numbers and use those for identification). If you will collect material that could be identified, such as an audio recording of comments in the test user's voice, explain clearly up front if there are any conditions in which anyone but you will have access to this material. Let the user tell you if he or she has any objection, and abide by what they say.

A final note: taking these matters seriously may be more than

a matter of doing the right thing for you. If you are working in an organization that receives federal research funds you are obligated to comply with formal rules and regulations that govern the conduct of tests, including getting approval from a review committee for any study that involves human participants.

5.2 Selecting Tasks for Testing

In your test you'll be giving the test users some things to try to do, and you'll be keeping track of whether they can do them. Just as good test users should be typical of real users, so test tasks should reflect what you think real tasks are going to be like. If you've been following our advice you already have some suitable tasks: the tasks you developed early on to drive your task-centered design.

You may find you have to modify these tasks somewhat for use in testing. They may take too long, or they may assume particular background knowledge that a random test user won't have. So you may want to simplify them. But be careful in doing this! Try to avoid any changes that make the tasks easier or that bend the tasks in the direction of what your design supports best.

Example: Test Users and Tasks for the Traffic Modelling System

We didn't have to modify our tasks for the traffic modelling system, because we used the same people as test users that we had worked with to develop our target tasks. So they had all the background that was needed. If we had not had access to these same folks we would have needed to prepare briefing materials for the test users so that they would know where Canyon and Arapahoe are and something about how they fit into the surrounding street grid.

[traffic modelling, pp. 13*ff*]

If you base your test tasks on the tasks you developed for task-centered design you'll avoid a common problem: choosing test tasks that are too fragmented. Traditional requirements lists naturally give rise to suites of test tasks that test the various requirements separately. Remember the phone-in bank system we discussed in Chapter 2? It had been thoroughly tested, but only using tests that involved single services, like checking a

[phone-in bank, p. 15]

balance or transferring funds, but not combinations of services like checking a balance and then transferring funds contingent on the results of the check. There were big problems in doing these combinations.

5.3 Providing a System for Test Users to Use

The key to testing early in the development process, when it is still possible to make changes to the design without incurring big costs, is using mockups in the test. These are versions of the system that do not implement the whole design, either in what the interface looks like or what the system does, but do show some of the key features to users. Mockups blur into *prototype*s, with the distinction that a mockup is rougher and cheaper and a prototype is more finished and more expensive.

The simplest mockups are just pictures of screens as they would appear at various stages of a user interaction. These can be drawn on paper or they can be, with a bit more work, created on the computer using a tool like HyperCard for the Mac or a similar system for Windows. A test is done by showing users the first screen and asking them what they would do to accomplish a task you have assigned. They describe their action, and you make the next screen appear, either by rummaging around in a pile of pictures on paper and holding up the right one, or by getting the computer to show the appropriate next screen.

[HyperCard §6.3, pp.103*ff*]
[Windows §6.4, pp.107*ff*]

This crude procedure can get you a lot of useful feedback from users. Can they understand what's on the screens, or are they baffled? Is the sequence of screens well-suited to the task, as you thought it would be when you did your cognitive walkthrough, or did you miss something?

To make a simple mockup like this you have to decide what screens you are going to provide. Start by drawing the screens users would see if they did everything the best way. Then decide whether you also want to "support" some alternative paths, and how much you want to investigate error paths. Usually it won't be practical for you to provide a screen for every possible user action, right or wrong, but you will have reasonable coverage of the main lines you expect users to follow.

During testing, if users stay on the lines you expected, you just show them the screens they would see. What if they deviate, and make a move that leads to a screen you don't have? First, you record what they wanted to do: that's valuable data about a discrepancy between what you expected and what they want to do, which is why you are doing the test. Then you can tell them what they would see, and let them try to continue, or you can tell them to make another choice. You won't see as much as you would if you

had the complete system for them to work with, but you will see whether the main lines of your design are sound.

HyperTopic: Some Details on Mockups

What if a task involves user input that is their free choice, like a name to use for a file? You can't anticipate what the users will type and put it on your mockup screens. But you can let them make their choice and then say something like, "You called your file 'eggplant'; in this mockup we used 'broccoli'. Let's pretend you chose 'broccoli' and carry on."

What if it is a feature of your design that the system should help the user recover from errors? If you just provide screens for correct paths you won't get any data on this. If you can anticipate some common errors, all you have to do is to include the error recovery paths for these when you make up your screens. If errors are very diverse and hard to predict you may need to make up some screens on the fly, during the test, so as to be able to show test users what they would see. This blends into the "Wizard of Oz" approach we describe later.

Some systems have to interact too closely with the user to be well approximated by a simple mockup. For example a drawing program has to respond to lots of little user actions, and while you might get information from a simple mockup about whether users can figure out some aspects of the interface, like how to select a drawing tool from a palette of icons, you won't be able to test how well drawing itself is going to work. You need to make more of the system work to test what you want to test.

The thing to do here is to get the drawing functionality up early so you can do a more realistic test. You would not wait for the system to be completed, because you want test results early. So you would aim for a prototype that has the drawing functionality in place but does not have other aspects of the system finished off.

In some cases you can avoid implementing stuff early by faking the implementation. This is the *Wizard of Oz* method: you get a person to emulate unimplemented functions and generate the feedback users should see. John Gould at IBM did this very effectively to test design alternatives for a speech transcription system for which the speech recognition component was not yet ready. He built a prototype system in which test users' speech was piped to a fast typist, and the typist's output was routed back to the test users'

screen. This idea can be adapted to many situations in which the system you are testing needs to respond to unpredictable user input, though not to interactions as dynamic as drawing.

If you are led to develop more and more elaborate approximations to the real system for testing purposes you need to think about controlling costs. Simple mockups are cheap, but prototypes that really work, or even Wizard of Oz setups, take substantial implementation effort.

↪ 'Version 0' prototype
[VB §6.4, pp.107*ff*]
[HyperCard §6.3, pp.103*ff*]

Some of this effort can be saved if the prototype turns out to be just part of the real system.As we will discuss further when we talk about implementation, this is often possible. A system like Visual Basic or Hypercard allows an interface to be mocked up with minimal functionality but then hooked up to functional modules as they become available. So don't plan for throw-away prototypes: try instead to use an implementation scheme that allows early versions of the real interface to be used in testing.

5.4 Deciding What Data to Collect

Now that we have people, tasks, and a system, we have to figure out what information to gather. It's useful to distinguish *process data* from *bottom-line* data. Process data are observations of what the test users are doing and thinking as they work through the tasks. These observations tell us what is happening step-by-step, and, we hope, suggests *why* it is happening. Bottom-line data give us a summary of *what* happened: how long did users take, were they successful, how many errors did they make.

It may seem that bottom-line data are what you want. If you think of testing as telling you how good your interface is, it seems that how long users are taking on the tasks, and how successful they are, is just what you want to know.

[§5.6, pp. 88*ff*]

We argue that process data are actually the data to focus on first. There's a role for bottom-line data, as we discuss in connection with Usability Measurement below. But as a designer you will mostly be concerned with process data. To see why, consider the following not-so-hypothetical comparison.

Suppose you have designed an interface for a situation in which you figure users should be able to complete a particular test task in about a half-hour. You do a test in which you focus on bottom-line data. You find that none of your test users was able to get the job done in less than an hour. You know you are in trouble, but what are you going to do about it? Now suppose instead you got detailed records of what the users actually did. You see that their whole approach to the task was mistaken, because they didn't use the

frammis reduction operations presented on the third screen. Now you know where your redesign effort needs to go.

We can extend this example to make a further point about the information you need as a designer. You know people weren't using frammis reduction, but do you know why? It could be that they understood perfectly well the importance of frammis reduction, but they didn't understand the screen on which these operations were presented. Or it could be that the frammis reduction screen was crystal clear but they didn't think frammis reduction was relevant.

Depending on what you decide here, you either need to fix up the frammis reduction screen, because it isn't clear, or you have a problem somewhere else. But you can't decide just from knowing that people didn't use frammis reduction.

To get the why information you really want, you need to know what users are thinking, not just what they are doing. That's the focus of the thinking-aloud method, the first testing technique we'll discuss.

5.5 The Thinking Aloud Method

The basic idea of thinking aloud is very simple. You ask your users to perform a test task, but you also ask them to talk to you while they work on it. Ask them to tell you what they are thinking: what they are trying to do, questions that arise as they work, things they read. You can make a recording of their comments or you can just take notes. You'll do this in such a way that you can tell what they were doing and where their comments fit into the sequence.

You'll find the comments are a rich lode of information. In the frammis reduction case, with just a little luck, you might get one of two kinds of comments: "I know I want to do frammis reduction now, but I don't see anyway to do it from here. I'll try another approach," or "Why is it telling me about frammis reduction here? That's not what I'm trying to do." So you find out something about *why* frammis reduction wasn't getting done, and whether the frammis reduction screen is the locus of the problem.

Example: Vocabulary Problems

One of Clayton's favorite examples of a thinking-aloud datum is from a test of an administrative workstation for law offices. This was a carefully designed, entirely menu-driven system intended for use by people without previous computing experience.

The word "parameter" was used extensively in the documentation and in system messages to refer to a quantity that could take on various user-assigned values. Test users read this word as "perimeter", a telling sign that the designers had stepped outside the vocabulary that was meaningful to their users.

Finding this kind of problem is a great feature of thinking-aloud. It's hard as a designer to tell whether a word that is perfectly meaningful and familiar to you will be meaningful to someone else. And its hard to detect problems in this area just from watching mistakes people make.

You can use the thinking-aloud method with a prototype or a rough mock-up, for a single task or a suite of tasks. The method is simple, but there are some points about it that repay some thought. Here are some suggestions on various aspects of the procedure. This material is adapted from Lewis, C. "Using the thinking-aloud method in cognitive interface design," IBM Research Report RC 9265, Yorktown Heights, NY, 1982.

5.5.1 Instructions

The basic instructions can be very simple: "Tell me what you are thinking about as you work." People can respond easily to this, especially if you suggest a few categories of thoughts as examples: things they find confusing, decisions they are making, and the like.

There are some other points you should add. Tell the user that you are not interested in their secret thoughts but only in what they are thinking about their task. Make clear that it is the system, not the user, that is being tested, so that if they have trouble it's the system's problem, not theirs. You will also want to explain what kind of recording you will make, and how test users' privacy will be protected (see discussion of ethics in testing earlier in this chapter).

[ethics hypertopic, p. 77]

5.5.2 The Role of the Observer

Even if you don't need to be available to operate a mockup, you should plan to stay with the user during the test. You'll do two things: prompt the user to keep up the flow of comments, and provide help when necessary. But you'll need to work out a policy for prompting and helping that avoids distorting the results you get.

It's very easy to shape the comments users will give you, and what they do in the test, by asking questions and making suggestions. If someone has missed the significance of some interface feature a word from you may focus their attention right on it. Also, research shows that people will make up an answer to any question you ask, whether or not they have any basis for the answer. You are better off, therefore, collecting the comments people offer spontaneously than prodding them to tell you about things you are interested in.

But saying nothing after the initial instructions usually won't work. Most people won't give you a good flow of comments without being pushed a bit. So say things that encourage them to talk, but that do not direct what they should say. Good choices are "Tell me what you are thinking" or "Keep talking". Bad choices would be "What do you think those prompts about frammis mean?" or "Why did you do that?"

On helping, keep in mind that a very little help can make a huge difference in a test, and you can seriously mislead yourself about how well your interface works by just dropping in a few suggestions here and there. Try to work out in advance when you will permit yourself to help. One criterion is: help only when you won't get any more useful information if you don't, because the test user will quit or cannot possibly continue the task. If you do help, be sure to record when you helped and what you said.

A consequence of this policy is that you have to explain to your test users that you want them to tell you the questions that arise as they work, but that you won't answer them. This seems odd at first but becomes natural after a bit.

5.5.3 Recording

There are plain and fancy approaches here. It is quite practical to record observations only by taking notes on a pad of paper: you write down in order what the user does and says, in summary form. But you'll find that it takes some experience to do this fast enough to keep up in real time, and that you won't be able to do it for the first few test users you see on a given system and task. This is just because you need a general idea of where things are going to be able to keep up. A step up in technology is to make a video record of what is happening on the screen, with a lapel mike on the user to pick up the comments. A further step is to instrument the system to pick up a trace of user actions, and arrange for this record to be synchronized in some way with an audio record of the comments. The advantage of this approach is that it gives you a machine readable record of user actions that

can be easier to summarize and access than video.

A good approach to start with is to combine a video record with written notes. You may find that you are able to dispense with the video, or you may find that you really want a fancier record. You can adapt your approach accordingly. But if you don't have a video setup don't let that keep you from trying the method.

5.5.4 Summarizing the Data

The point of the test is to get information that can guide the design. To do this you will want to make a list of all difficulties users encountered. Include references back to the original data so you can look at the specifics if questions arise. Also try to judge why each difficulty occurred, if the data permit a guess about that.

5.5.5 Using the Results

Now you want to consider what changes you need to make to your design based on data from the tests. Look at your data from two points of view. First, what do the data tell you about how you *thought* the interface would work? Are the results consistent with your cognitive walkthrough or are they telling you that you are missing something? For example, did test users take the approaches you expected, or were they working a different way? Try to update your analysis of the tasks and how the system should support them based on what you see in the data. Then use this improved analysis to rethink your design to make it better support what users are doing.

Second, look at all of the errors and difficulties you saw. For each one make a judgement of how important it is and how difficult it would be to fix. Factors to consider in judging importance are the costs of the problem to users (in time, aggravation, and possible wrong results) and what proportion of users you can expect to have similar trouble. Difficulty of fixes will depend on how sweeping the changes required by the fix are: changing the wording of a prompt will be easy, changing the organization of options in a menu structure will be a bit harder, and so on. Now decide to fix all the important problems, and all the easy ones.

Related exercise 5.1 ▷ see page 160

HyperTopic: The Two-Strings Problem and Selecting Panty Hose

Thinking aloud is widely used in the computer industry nowadays, and you can be confident you'll get useful results if you use it. But it's important to understand that test users can't tell you everything you might like to know, and that some of what they will tell you is bogus. Psychologists have done some interesting studies that make these points.

Maier had people try to solve the problem of tying together two strings that hung down from the ceiling too far apart to be grabbed at the same time (Maier, N.R.F. "Reasoning in humans: II. The solution of a problem and its appearance in consciousness." Journal of Comparative Psychology, 12 (1931), pp. 181–194). One solution is to tie some kind of weight to one of the strings, set it swinging, grab the other string, and then wait for the swinging string to come close enough to reach. It's a hard problem, and few people come up with this or any other solution. Sometimes, when people were working, Maier would "accidentally" brush against one of the strings and set it in motion. The data showed that when he did this people were much more likely to find the solution. The point of interest for us is, what did these people say when Maier asked them how they solved the problem? They did *not* say, "When you brushed against the string that gave me the idea of making the string swing and solving the problem that way," even though Maier knows that's what really happened. So they could not and did not tell him what feature of the situation really helped them solve the problem.

Nisbett and Wilson set up a market survey table outside a big shopping center and asked people to say which of three pairs of panty hose they preferred, and why (Nisbett, R.E., and Wilson, T.D. "Telling more than we can know: Verbal reports on mental processes." Psychological Review, 84 (1977), pp. 231–259). Most people picked the rightmost pair of the three, giving the kinds of reasons you'd expect: "I think this pair is sheerer" or "I think this pair is better made." The trick is that the three pairs of panty hose were *identical*. Nisbett and Wilson knew that given a choice among three closely-matched alternatives there is a bias to pick the last one, and that that bias was the real basis for people's choices. But (of course) nobody *said* that's why

they chose the pair they chose. It's not just that people couldn't report their real reasons: when asked they made up reasons that seemed plausible but are wrong.

What do these studies say about the thinking-aloud data you collect? You won't always hear why people did what they did, or didn't do what they didn't do. Some portion of what you do hear will be wrong. And, you're especially taking a risk if you ask people specific questions: they'll give you some kind of an answer, but it may have nothing to do with the facts. Don't treat the comments you get as some kind of gospel. Instead, use them as input to your own judgment processes.

5.6 Measuring Bottom-Line Usability

We argue that you usually want process data, not bottom-line data, but there are some situations in which bottom-line numbers are useful. You may have a definite requirement that people be able to complete a task in a certain amount of time, or you may want to compare two design alternatives on the basis of how quickly people can work or how many errors they commit. The basic idea in these cases is that you will have people perform test tasks, you will measure how long they take and you will count their errors.

Your first thought may be to combine this with a thinking-aloud test: in addition to collecting comments you'd collect these other data as well. Unfortunately this doesn't work as well as one would wish. The thinking-aloud process can affect how quickly and accurately people work. It's pretty easy to see how thinking-aloud could slow people down, but it has also been shown that sometimes it can speed people up, apparently by making them think more carefully about what they are doing, and hence helping them choose better ways to do things. So if you are serious about finding out how long people will take to do things with your design, or how many problems they will encounter, you really need to do a separate test.

Getting the bottom-line numbers won't be too hard. You can use a stopwatch for timings, or you can instrument your system to record when people start and stop work. Counting errors, and gauging success on tasks, is a bit trickier, because you have to decide what is an error and what counts as successful completion of a task. But you won't have much trouble here either as long as you understand that you can't come up with perfect criteria for these things and use your common sense.

5.6.1 Analyzing the Bottom-Line Numbers

When you've got your numbers you'll run into some hard problems. The trouble is that the numbers you get from different test users will be different. How do you combine these numbers to get a reliable picture of what's happening?

Suppose users need to be able to perform some task with your system in 30 minutes or less. You run six test users and get the following times:

20 min
15 min
40 min
90 min
10 min
5 min

Are these results encouraging or not? If you take the average of these numbers you get 30 minutes, which looks fine. If you take the *median*, that is, the middle score, you get something between 15 and 20 minutes, which looks even better. Can you be confident that the typical user will meet your 30-minute target?

The answer is no. The numbers you have are so variable, that is, they differ so much among themselves, that you really can't tell much about what will be "typical" times in the long run. Statistical analysis, which is the method for extracting facts from a background of variation, indicates that the "typical" times for this system might very well be anywhere from about 5 minutes to about 55 minutes. Note that this is a range for the "typical" value, not the range of possible times for individual users. That is, it is perfectly plausible given the test data that if we measured lots and lots of users the average time might be as low as 5 min, which would be wonderful, but it could also be as high as 55 minutes, which is terrible.

There are two things contributing to our uncertainty in interpreting these test results. One is the small number of test users. It's pretty intuitive that the more test users we measure the better an estimate we can make of typical times. Second, as already mentioned, these test results are very variable: there are some small numbers but also some big numbers in the group. If all six measurements had come in right at (say) 25 minutes, we could be pretty confident that our typical times would be right around there. As things are, we have to worry that if we look at more users we might get a lot more 90-minute times, or a lot more 5-minute times.

It's the job of statistical analysis to juggle these factors — the number of people we test and how variable or consistent the results are — and give

us an estimate of what we can conclude from the data. This is a big topic, and we won't try to do more than give you some basic methods and a little intuition here.

Here's a cookbook procedure for getting an idea of the range of typical values that are consistent with your test data.

1. Add up the numbers. Call this result "sum of x". In our example this is 180.

2. Divide by the n, the number of numbers. The quotient is the average, or mean, of the measurements. In our example this is 30.

3. Add up the squares of the numbers. Call this result "sum of squares" In our example this is 10450.

4. Square the sum of x and divide by n. Call this "foo". In our example this is 5400.

5. Subtract foo from sum of squares and divide by $n - 1$. In our example this is 1010.

6. Take the square root. The result is the "standard deviation" of the sample. It is a measure of how variable the numbers are. In our example this is 31.78, or about 32.

7. Divide the standard deviation by the square root of n. This is the "standard error of the mean" and is a measure of how much variation you can expect in the typical value. In our example this is 12.97, or about 13.

8. It is plausible that the typical value is as small as the mean minus two times the standard error of the mean, or as large as the mean plus two times the standard error of the mean. In our example this range is from $30 - (2 \times 13)$ to $30 + (2 \times 13)$, or about 5 to 55.

What does "plausible" mean here? It means that if the real typical value is outside this range, you were very unlucky in getting the sample that you did. More specifically, if the true typical value were outside this range you would only expect to get a sample like the one you got 5 percent of the time or less.

Experience shows that usability test data are quite variable, which means that you need a lot of it to get good estimates of typical values. If you pore over the above procedure enough you may see that if you run four times

as many test users you can narrow your range of estimates by a factor of two: the breadth of the range of estimates depends on the square root of the number of test users. That means a lot of test users to get a narrow range, if your data are as variable as they often are.

What this means is that you can anticipate trouble if you are trying to manage your project using these test data. Do the test results show we are on target, or do we need to pour on more resources? It's hard to say. One approach is to get people to agree to try to manage on the basis of the numbers in the sample themselves, without trying to use statistics to figure out how uncertain they are. This is a kind of blind compromise: on the average the typical value is equally likely to be bigger than the mean of your sample, or smaller. But if the stakes are high, and you really need to know where you stand, you'll need to do a lot of testing. You'll also want to do an analysis that takes into account the cost to you of being wrong about the typical value, by how much, so you can decide how big a test is really reasonable.

HyperTopic: Measuring User Preference

Another bottom-line measure you may want is how much users like or dislike your system. You can certainly ask test users to give you a rating after they finish work, say by asking them to indicate how much they liked the system on a scale of 1 to 10, or by choosing among the statements, "This is one of the best user interfaces I've worked with", "This interface is better than average", "This interface is of average quality", "This interface is poorer than average", or "This is one of the worst interfaces I have worked with." But you can't be very sure what your data will mean. It is very hard for people to give a detached measure of what they think about your interface in the context of a test. The novelty of the interface, a desire not to hurt your feelings (or the opposite), or the fact that they haven't used your interface for their own work can all distort the ratings they give you. A further complication is that different people can arrive at the same rating for very different reasons: one person really focuses on response time, say, while another is concerned about the clarity of the prompts. Even with all this uncertainty, though, it's fair to say that if lots of test users give you a low rating you are in trouble. If lots give you a high rating that's fine as far as it goes, but you can't rest easy.

5.6.2 Comparing Two Design Alternatives

If you are using bottom-line measurements to compare two design alternatives, the same considerations apply as for a single design, and then some. Your ability to draw a firm conclusion will depend on how variable your numbers are, as well as how many test users you use. But then you need some way to compare the numbers you get for one design with the numbers from the others.

The simplest approach to use is called a *between-groups experiment*. You use two groups of test users, one of which uses version A of the system and the other version B. What you want to know is whether the typical value for version A is likely to differ from the typical value for version B, and by how much. Here's a cookbook procedure for this.

1. Using parts of the cookbook method above, compute the means for the two groups separately. Also compute their standard deviations. Call the results m_a, m_b, s_a, s_b. You'll also need to have n_a and n_b, the number of test users in each group (usually you'll try to make these the same, but they don't have to be.)

2. Combine s_a and s_b to get an estimate of how variable the whole scene is, by computing
$$s = \frac{\sqrt{n_a \times s_a^2 + n_b \times s_b^2}}{n_a + n_b - 2}$$

3. Compute a combined standard error:
$$s_e = s \times \sqrt{1/n_a + 1/n_b}$$

4. Your range of typical values for the difference between version A and version B is now:
$$m_a - m_b \pm 2 \times s_e$$

Another approach you might consider is a *within-groups experiment*. Here you use only one group of test users and you get each of them to use both versions of the system. This brings with it some headaches. You obviously can't use the same tasks for the two versions, since doing a task the second time would be different from doing it the first time, and you have to worry about who uses which system first, because there might be some advantage or disadvantage in being the system someone tries first. There are ways around these problems, but they aren't simple. They work best for very

simple tasks about which there isn't much to learn. You might want to use this approach if you were comparing two low-level interaction techniques, for example. You can learn more about the within-groups approach from any standard text on experimental psychology (check your local college library or bookstore).

HyperTopic: Don't Push Your Statistics Too Far

The cookbook procedures we've described are broadly useful, but they rest on some technical assumptions for complete validity. Both procedures assume that your numbers are drawn from what is called a "normal distribution," and the comparison procedure assumes that the data from both groups come from distributions with the same standard deviation. But experience has shown that these methods work reasonably well even if these assumptions are violated to some extent. You'll do OK if you use them for broad guidance in interpreting your data, but don't get drawn into arguments about exact values.

As we said before, statistics is a big topic. We don't recommend that you need an MS in statistics as an interface designer, but it's a fascinating subject and you won't regret knowing more about it than we've discussed here. If you want to be an interface researcher, rather than a working stiff, then that MS really would come in handy.

5.7 Details of Setting Up a Usability Study

The description of user testing we've given up to this point should be all the background you need during the early phases of a task-centered design project. When you're actually ready to evaluate a version of the design with users, you'll have to consider some of the finer details of setting up and running the tests. This section, which you may want to skip on the first reading of the chapter, will help with many of those details.

5.7.1 Choosing the Order of Test Tasks

Usually you want test users to do more than one task. This means they have to do them in some order. Should everyone do them in the same order, or should you scramble them, or what? Our advice is to choose one sensible order, starting with some simpler things and working up to some

more complex ones, and stay with that. This means that the tasks that come later will have the benefit of practice on the earlier one, or some penalty from test users getting tired, so you can't compare the difficulty of tasks using the results of a test like this. But usually that is not what you are trying to do.

5.7.2 Training Test Users

Should test users hit your system cold or should you teach them about it first? The answer to this depends on what you are trying to find out, and the way your system will be used in real life. If real users will hit your system cold, as is often the case, you should have them do this in the test. If you really believe users will be pre-trained, then train them before your test. If possible, use the real training materials to do this: you may as well learn something about how well or poorly it works as part of your study.

5.7.3 The Pilot Study

You should always do a pilot study as part of any usability test. A *pilot study* is like a dress rehearsal for a play: you run through everything you plan to do in the real study and see what needs to be fixed. Do this twice, once with colleagues, to get out the biggest bugs, and then with some real test users. You'll find out whether your policies on giving assistance are really workable and whether your instructions are adequately clear. A pilot study will help you keep down variability in a bottom-line study, but it will avoid trouble in a thinking-aloud study too. Don't try to do without!

5.7.4 What If Someone Doesn't Complete a Task?

If you are collecting bottom-line numbers, one problem you will very probably encounter is that not everybody completes their assigned task or tasks within the available time, or without help from you. What do you do? There is no complete remedy for the problem. A reasonable approach is to assign some very large time, and some very large number of errors, as the "results" for these people. Then take the results of your analysis with an even bigger grain of salt than usual.

5.7.5 Keeping Variability Down

As we've seen, your ability to make good estimates based on bottom-line test results depends on the results not being too variable. There are things

you can do to help, though these may also make your test less realistic and hence a less good guide to what will happen with your design in real life. Differences among test users is one source of variable results: if test users differ a lot in how much they know about the task or about the system you can expect their time and error scores to be quite different. You can try to recruit test users with more similar background, and you can try to brief test users to bring them close to some common level of preparation for their tasks.

Differences in procedure, how you actually conduct the test, will also add to variability. If you help some test users more than others, for example, you are asking for trouble. This reinforces the need to make careful plans about what kind of assistance you will provide. Finally, if people don't understand what they are doing your variability will increase. Make your instructions to test users and your task descriptions as clear as you can.

5.7.6 Debriefing Test Users

We've stressed that it's unwise to ask specific questions during a thinking aloud test, and during a bottom-line study you wouldn't be asking questions anyway. But what about asking questions in a debriefing session after test users have finished their tasks? There's no reason not to do this, but don't expect too much. People often don't remember very much about problems they've faced, even after a short time. Clayton remembers vividly watching a test user battle with a text processing system for hours, and then asking afterwards what problems they had encountered. "That wasn't too bad, I don't remember any particular problems," was the answer. He interviewed a real user of a system who had come within one day of quitting a good job because of failure to master a new system; they were unable to remember any specific problem they had had. Part of what is happening appears to be that if you work through a problem and eventually solve it, even with considerable difficulty, you remember the solution but not the problem.

There's an analogy here to those hidden picture puzzles you see on kids' menus at restaurant: there are pictures of three rabbits hidden in this picture, can you find them? When you first look at the picture you can't see them. After you find them, you can't help seeing them. In somewhat the same way, once you figure out how something works it can be hard to see why it was ever confusing.

Something that might help you get more info out of questioning at the end of a test is having the test session on video so you can show the test user the particular part of the task you want to ask about. But even if you do

this, don't expect too much: the user may not have any better guess than you have about what they were doing.

Another form of debriefing that is less problematic is asking for comments on specific features of the interface. People may offer suggestions or have reactions, positive or negative, that might not otherwise be reflected in your data. This will work better if you can take the user back through the various screens they've seen during the test.

Related exercise 5.2 ▷ see page 160

6 User Interface Management and Prototyping Systems

In this chapter we'll talk about some of the software tools that are essential for building the prototypes and the final version of your system. "Essential" is a strong word, but in a competitive market it's the right one. Successful application developers rely on these tools to get products to market faster, with better interfaces, more robust behavior, and fewer potential legal problems. This is true not only for big companies with internationally marketed products, but also for small local consulting firms and in-house development teams. To compete with the winners, you'll need to take a similar approach.

The software tools we'll describe have a variety of names and provide a variety of functionality. The simplest are "toolkits," libraries of program subroutines that create and run the many on-screen objects of a graphical user interface, such as windows, buttons, and menus. These can save some programming time and maintain consistency with existing applications, but they still leave most of the work to the programmer. A giant step up from toolkits are user interface management systems (UIMS). A UIMS gives you a toolkit and a programming environment for using it. With the better systems, the environment is largely visual, so instead of writing C or Pascal code that specifies the numerical coordinates of each button in a dialog box (i.e., 140,29,170,80 for its corners), you can just drag a button out of your design palette and drop it on the screen where it looks good.

Some UIMS packages are limited environments that provide very fast development of simple programs or early prototypes. Others may require more work to get started, but they allow you to start with a prototype and iterate it into a full-fledged, highly functional application. A survey in 1991 showed that over 40 percent of programmers in commercial environments were using some sort of UIMS with more power than a toolkit, spending roughly 40 percent of their application development time on the code that went into their applications' user interfaces. Toolkit users, by comparison, spent 60 percent of their time on the interface. In both cases, the interface accounted for about half of the total application code. (Brad Meyers & Mary Beth Rosson, "Survey on user interface programming," Proc. CHI'92 Conference on Human Factors in Computer Systems. New York: ACM, 1992, pp. 195–202.) The savings can be even greater if you can design your application as an embedded system that uses functionality provided by other programs the user already owns, a technique we'll describe in the section on Microsoft Windows.

[Windows §6.4, pp.107*ff*]

The UIMS approach does more than save programming time. It also helps you build a better interface, by maintaining consistency within the application and across applications under a common operating system, and by making it easier to rapidly iterate through the implement-and-test cycle. An additional advantage is that a UIMS can provide answers to the difficult question of what you can legally copy. The fundamental purpose of a UIMS is to help programmers develop interfaces rapidly, by copying controls that users already know. That goal is shared by programmers, users, and the vendor of the underlying operating system. The UIMS is sold by a company that has legal rights to the interface techniques you want to use, and purchasing the UIMS gives you clearly defined rights to sell the systems you develop.

Here's the plan for the rest of this chapter. The next section introduces some programming concepts that underlie the UIMS approach. That's followed by three sections describing specific interface building packages that are representative of the systems you're likely to encounter.

Related exercise 6.1 ▷ see page 161

6.1 Concepts

6.1.1 Object-Oriented Programming

Object-oriented programming is a technique for making programs easier to write, easier to maintain, and more robust. The *objects* of object-oriented programming are blocks of code, not necessarily on-screen objects, although the on-screen objects usually are implemented with program objects. The program objects have several important characteristics: They are defined hierarchically, with each object being an *instance* of a *class* of similar objects. (A class is also defined in a single block of code.) For example, the blocks of code describing the File menu and the Edit menu would be two instances of the class of menus, and that class could itself be a *subclass* of the class of labels. Objects *inherit* the behavior and characteristics defined higher in the hierarchy, unless that inheritance is specifically overridden by the object's code. Inheritance would allow the programmer to change the font of all the menu objects by making a single change to the class definition. Each object also has *private data* that defines its own characteristics and maintains information about its current state. Objects communicate with each other by sending *messages*. An object's response to a message is part of the behavior that the object inherits or can override.

Object-oriented programming requires an object-oriented programming language, such as C++ or CLOS, that supports the object features in addition to the basic functionality provided by most modern languages. There are differences between object-oriented languages, and not all of them support all of the features described in the previous paragraph. But they do all provide a programming structure that's an excellent match to the needs of user-interface programming. If you need another menu, you just create another instance of the menu class — a single line of code. Then you fill in the details unique to that menu: what are the names of the menu items, where is it located on the screen, and what messages should each item send when it is selected. Additional code objects implement the functions of the program: sorting, searching, whatever. These code objects, sometimes called *handlers*, are invoked by messages sent from menu items and other user controls, as well as from other code objects. If you need to modify the program's behavior, the class hierarchies let you make sweeping changes consistently and easily, while the private data allows you to change the behavior of an individual object without fear of unexpected side-effects.

6.1.2 Event-Driven Programs

The traditional paradigm for computer programs is sequential: the user starts the program and the program takes control, prompting the user for input as needed. It's possible to write a highly interactive program (for example, a word processor) using the sequential programming paradigm, but it isn't easy. The resulting programs are often very modal: an input mode, an edit mode, a print mode, etc. The programmer has to anticipate every sequence of actions the user might want to take, and modes restrict those sequences to a manageable set.

For a simpler and more natural approach, modern interactive systems use an event-driven paradigm. Events are messages the user, or the system, sends to the program. A keystroke is an event. So is a mouse-click. Incoming e-mail or an empty paper tray on the printer might cause the system to generate an event. The core of every event-driven program is a simple loop, which waits for an event to take place, responds appropriately to that event, and waits for another. For example, the event loop of a word processor would notice a keystroke event, display the character on the screen, and then wait for another event. If it noticed a mouse-double-click event, it would select the word the mouse was pointing at. If it noticed a mouse-click in a menu, it would take whatever action the menu item specified. Early interactive systems actually required the programmer to write the event loop, but in

a modern UIMS environment the programmer just needs to build objects (menus, windows, code, etc.) and specify how they send or respond to messages.

6.1.3 Resources

Resources, for our purposes, are interface-specific information such as menu titles or button positions, which are stored so they can be easily changed without affecting the underlying program functionality. (A system may also treat the main program code as a resource.)

On some systems you might change the resources by editing values in a text file; on others you might need to use a special resource editor. But it typically won't require recompiling the application itself, and it won't require access to the source code. During prototyping, a few simple changes to resources might dramatically improve an interface, with no "real" programming. When the system is ready to ship, resources can be changed so the product can be used in countries with a different language.

6.1.4 Interapplication Communication

Interapplication communication describes a situation in which two programs, running at the same time, exchange data. For example, a word processing program might send some numbers to a spreadsheet, which could calculate their average and send it back to the word processor. That would allow the word processor to have the calculating power of the spreadsheet, without duplicating the code. Communication between applications is a common technique on minicomputers (the world of UNIX), but it's only recently been implemented in personal computer operating systems. That's partly because early personal computers could only run one application at a time, and partly because, unlike the command-driven software that forms the basis of most minicomputer applications, interactive graphical systems can't easily be adapted to respond to commands from other programs.

Two major personal computer software environments, Microsoft Windows (currently version 3) and Apple's Macintosh System (version 7), support interapplication communication. They provide a communications pathway between applications, and they specify standard data formats for the interaction. As we write this, Windows seems to have the lead both in functionality and number of third-party applications that another application can access. On either the Mac or Windows, you should look for places

where interapplication communications can help you avoid rebuilding systems that the user already has and knows.

HyperTopic: Roll Your Own UIMS for Unique Environments

"All right," you say, "These UIMS products sound great for programming on personal computers or Unix machines, but I'm building the interface to a walk-up-and-use airport baggage check, using all custom hardware. What do I do?"

Here are two suggestions. First, invest in a simple UIMS on a personal computer, something at the level of Apple's HyperCard, and use that to prototype your system. Evaluation of the prototype won't uncover all the problems that could arise with the real system, but it's a quick and inexpensive way to get started in the right direction.

Second, develop in UIMS style for the dedicated hardware. Don't go overboard on this — if you decide to write your own object-oriented cross-compiler, your competition could have their baggage-check system on the market before you even have a prototype. But the application you develop can incorporate the basic ideas of event-oriented programming, modularized code, and table-lookup for resource information. The extra time taken up front to design this system will certainly be paid back as you iterate the prototype, and it will be paid back again when it's time to write a new version.

6.2 OSF/Motif in X-Windows — Toolboxes in the Trenches

The X-Windows system is a software product that allows programs to be run on computer networks, with the interaction taking place at the user's machine while the main program is running on some other computer. In this approach, the user's local system is called the *server*, responding to the interface-related needs of the *client* program on the remote machine. The user's machine typically has less power than the client that's running the main program. It may even be an "X Terminal," a mouse/monitor/keyboard unit with just enough computing power to buffer i/o and rapidly display graphics. X-Windows was originally developed for academic networking as part of MIT's project Athena, but it has since been adopted as a standard by several major computer manufacturers.

Motif was developed by the Open Software Foundation (OSF) as a standard graphical user interface substrate for X-Windows. Motif consists of a library of C language subroutines that define a toolbox of interface objects and techniques for combining them into a program. The toolbox allows a programmer to create programs that are event-driven, object-oriented, and adaptable through on external resource files. There are full-featured UIMS packages for X-Windows, such as the Garnet system that was discussed in Chapter 3. However, many programmers choose to use the Motif toolkit within their standard C programming environment. We describe that approach in this section to provide a baseline for appreciating more sophisticated approaches. It's not an approach we recommend.

[Garnet, p. 35]

All the graphical objects in Motif are referred to as *widgets*, which are defined as an object-oriented hierarchy. A typical class in the hierarchy is labels, which has buttons as a subclass, which in turn has individual buttons as instances. The options for each class and instance, such as color, text string, and behavior, are specified as resources in the program's resource database. The database itself is created at program startup by reading in a series of text files that describe the program, the hardware, and the user's preferences.

Widgets within a program are also arranged hierarchically, but in a different sense: The main window of the program is the top widget of the hierarchy, a menu within it is at the next level, and buttons within that window are at the next. Many of the parameters at lower levels are inherited or calculated from the higher levels. Dragging the window changes its location parameters, for example, and this change propagates through the hierarchy to the controls within the window.

The runtime behavior of a Motif program is event-driven, controlled by an event loop that is supplied as part of the toolkit. The system watches for events such as mouse-clicks, cursors moving in or out of windows, and keystrokes. When an event occurs in a widget, the system checks the translation table, a part of the widget's resource list that tells what action to take, if any. Like most other things in the resource database, the translation table can be changed by the program as it runs. This allows the behavior of a control to reflect the current situation. If the translation table indicates that some action is to be taken, the system generates a "callback" to the client program, telling it what needs to be done.

Writing a Motif program is like writing a typical C program, with some additions. You might start by writing a module that defines the program's basic functions, such as search and sort for a database. This can be compiled and debugged independently with a simple command-line interface. The

Motif interface code can be written as a separate module.

One part of the Motif code you'll need to create will be the resource files that define all the initial parameters of the on-screen objects. Since this isn't a visual programming environment, each parameter has to be specified textually. The label class of widget has about 45 parameters that can be specified in the resource file. Specifying the controls this way is time-consuming, but it's not as bad as it sounds. A class's parameters are inherited by objects below it, so only parameters unique to a class or instance need to be specified. Also, consistency among similar items can be achieved by using wildcard specifications. For example, the resource specification:

```
hello-world.controls.quitButton.XmNhighlightColor:blue
```

sets the highlight color field for a single widget in the "controls" module of the "hello-world" program. The specification:

```
hello-world.main*XmNhighlightColor:blue
```

sets the same parameter for all widgets in the module.

In addition to the resource files, you might employ the Motif User Interface Language (UIL — like it or not, it seems that any three-word phrase in programming documentation gets reduced to an acronym). The UIL approach allows the programmer to redefine the widget hierarchy external to the compiled C program, something that can't be done with the resource files.

After defining the resources, you'll have to write the C program that will become the heart of the system. This code will start with header information that references the standard Motif files, including at least 30 libraries of standard widget classes. It will also define the callback procedures that link interface actions to the functions defined in the main module. Then it will specify a series of actions necessary to get the program running: initialize the Motif toolkit, open a network connection to the server, define a top-level widget for the hierarchy, read specifications from the database and define all the widgets in the window, and then display all the widgets. Finally, the code will turn control over to an event loop, supplied as one of the Motif toolbox routines. To produce a simple "hello world" program you might have to write on the order of 100 lines of C code.

6.3 Rapid Prototyping in HyperCard

The Apple II was Apple Computer's first big success. Indeed, it was the first big success of any personal computer outside the hobby market. Every

Apple II included a BASIC language interpreter for users who wanted to write their own programs. When Apple introduced the Macintosh, many potential users were disappointed that it came without a standard programming language. HyperCard, introduced a few years later, is a low cost, low effort programming environment that answers some of those users' concerns. It goes well beyond BASIC in terms of the ease with which a graphical user interface can be created.

Programs written in the HyperCard environment are usually called "stacks" because they present a stack-of-index-cards metaphor. The HyperCard application itself is required not only to write but also to run a stack. Every new Macintosh comes with HyperCard Player, a version of the application that will run any stack but allows only limited programming.

HyperCard provides an object-oriented visual programming environment in which the user can create and customize three main classes of object: buttons and text fields, which are on-screen controls; and cards, which are individual windows that contain the buttons and fields. A HyperCard program for a personal phone list could be implemented as a stack of 26 cards, each with text fields containing names and phone numbers, and with buttons on each card labelled A through Z that take the user to the appropriate card. The phone list is exactly the kind of program HyperCard was designed for, and it would be supported by a number of built-in functions that the user can access through the standard run-time menus, such as "go to next card" and "find a string of text." Since all the cards are the same except for text content, they could even be implemented as a single class, called the "background."

The phone-list application can be programmed entirely by interacting with the visual programming environment — selecting from menus, filling in dialog boxes, dragging buttons and fields into place. For more sophisticated applications, HyperCard provides a Pascal-like language called HyperTalk. HyperTalk procedures (handlers) are activated by messages from user controls and other procedures. For example, you might want to add a feature to the phone stack that would find all the names with a given area code. You could write a HyperTalk procedure that prompts the user for the area code, searches for that code in the area-code field of each card, and collects all the names into a new field. You'd associate that code with a new button, "Find Folks by Area Code," either by actually writing the code "in" the button itself (command-option-click the button to open it up), or by writing the code in the stack as a message handler and having the button send a message when clicked.

HyperCard is an interpreted language, so procedures can be tested as

soon as you write them, with no compilation. But they don't run exceptionally fast, and they provide only limited access to the Macintosh operating system toolbox, which is needed to create standard interface objects such as general-purpose dialog boxes and scrolling windows. (Pulldown menus can be created with HyperTalk commands.) You can overcome these limitations by writing external functions and commands (XFCNs and XCMDs). These are procedures written in a programming language such as C or Pascal and incorporated into a HyperCard stack as code resources. They can be called from a HyperTalk procedure as if they were built-in HyperTalk functions.

We've found HyperCard to be very valuable for prototyping simple interfaces. An idea can be roughed out and shown to other designers in less than a day, and building enough functionality for early user testing won't take much longer. The system has two shortcomings. First, unless you use XCMDs and put in a lot of effort, the programs you develop look like HyperCard stacks, not like Macintosh programs. Even with external commands, menus and dialog boxes never become true objects in the visual programming environment. This makes HyperCard a good environment for prototyping things with buttons and text, like automatic teller machines, but not so good for prototyping Macintosh applications.

The second problem with HyperCard is that it wasn't intended to be a general purpose, full functionality programming environment. It's very good for simple prototypes or small applications that fit the stack-of-cards metaphor, but if you try to push beyond its limits you'll soon find yourself with a large, unwieldy program in an environment that has little support for modularity and versioning. In the worst case, you'll also be juggling functionality between XCMDs and HyperTalk. Together these problems are an invitation to some pretty flakey code.

HyperCard and similar programs are simple design tools, something like an artist's sketchpad, that you should have available and use when appropriate. For extended testing and final program delivery on the Mac, you will usually want a more powerful UIMS, either a visual environment or a programming language with object-oriented extensions supporting the target system's full toolbox of interface objects.

Example: Experiences with HyperCard

We've been using HyperCard for several years for prototyping and small applications. A typical project was an early, computer-based version of the cognitive walkthrough, which prompted users with detailed questions about an interface. The evaluator would click a Yes or a No button, or type in some text. The

clever part of the application was that it could sometimes tell from the answer of one question that other questions could be skipped.

One of Clayton's graduate students roughed out the design in HyperCard, and we did a round of user testing with that prototype. The tests showed promise, and we decided to turn the prototype into a slicker application that we could give out for more feedback. John started to incorporate the changes suggested by user testing into the prototype, but he almost immediately decided to scrap the prototype entirely and rewrite everything, still working in HyperCard. The problems with the prototype weren't the fault of the original programmer. It's just that fast prototyping — brainstorming on-line — can lead to very sloppy code. That's a lesson that applies to other prototyping systems as well, although having the code distributed among various buttons, fields, and cards exacerbates the situation in HyperCard.

The basic rewrite in HyperCard only took about a week, plus a few more days for testing. That illustrates another fact about prototyping systems: if you've built a prototype once, duplicating it (neatly) in the same system can be trivial. However, duplicating the prototype in a different system can be very *non*-trivial, because the new system typically doesn't support the same interaction techniques as the prototype.

Part of the redevelopment time was spent writing XCMDs in C to get around HyperCard's shortcomings. One of the routines changed upper to lower case, something HyperTalk could do but not fast enough. Another found commas in text that the user had typed in and changed them to vertical bars ("|"), because we were using some built-in HyperCard routines that mistook the user's commas for field delimiters. The bars had to be changed back into commas whenever the text was redisplayed for the user.

We gave out several copies of the walkthrough stack for comments. A couple of users reported that they couldn't get the stack to run. It turned out that they were running an earlier version of HyperCard. That illustrates another potential problem: HyperCard and some other UIM systems don't deliver standalone applications. The behavior of the code you deliver may depend on the version of the user's software. It may also depend on the options that the user has set for that software.

We stopped work on the walkthrough stack after we simpli-

fied the walkthrough process, but HyperCard might have been an adequate vehicle for final program delivery if the project had continued. The forms-oriented view that HyperCard supports was well suited to the simple walkthrough program. However, another project we started in HyperCard, the ChemTrains graphical programming language, outgrew the environment's capabilities after just a few weeks of work. Even though later versions of HyperCard have fixed some of the problems we had, our overall experience with the system recalls similar experiences with interpreted BASIC on other personal computers: projects get started very fast, but they soon bog down because of program size, poor support for modularity, and performance limitations.

6.4 Windows, the Shared-Code Approach, and Visual Basic

Microsoft Windows is an operating system level product that supports graphical user interfaces on the Intel 80x86 platform (PC's, AT's, clones, etc.). Version 3 of Windows is one of the most popular software packages ever developed. Half a million copies were shipped in the first six weeks after its introduction in 1990, and over a million copies per month were selling at one time. Windows NT, a version of the system that runs in 32-bit mode on Intel and other hardware, now makes the Windows environment available outside of the PC world. The huge potential market for application software that runs under Windows has naturally attracted the attention of many product developers, including several who have developed state-of-the-art UIMS packages.

The popularity of Windows is certainly due in part to the availability of inexpensive but powerful compatible hardware, the 80x86 machines from dozens of competitive vendors. But Microsoft has maintained its leading position in this market by continually improving the software's functionality. Windows today provides a sophisticated event-oriented user interface, memory management, the ability to run several programs at the same time, and various ways for those programs to communicate. The last two features make it possible for you to write applications that effectively share the code of other applications that the user already has. Here are some of the interapplication communication protocols that Windows supports:

Dynamic Link Libraries (DLL). These are toolbox routines to draw and run various controls, or to take procedural actions like sorting data. Many of them come with Windows, and you can inexpensively license additional routines from independent software developers and distribute them with an application. So if you need a gas gauge display in your application and Windows doesn't have one, you can license the code from someone else. You can also write your own DLL in a language such as C.

Dynamic Data Exchange (DDE, soon to be replaced by DDEML). This is the fundamental protocol for interapplication communication. It allows your program (the client) to send data to another application (the server), asking that application to execute a menu item or a macro. (Note that these are different definitions for client and server than X-Windows uses; in fact, they're almost the opposite.) It also lets your program receive data sent by another application. This is the technique you'd use if you wanted to have a spreadsheet do calculations and return the result. Of course, the spreadsheet has to recognize DDE communication, but competitive major applications typically will.

Object Linking and Embedding (OLE). Object linking and embedding are two related techniques that allow the user to create documents containing objects created and maintained by different applications. For example, these features allow a user to include a "live" drawing in a word processing document. Clicking on the drawing will bring up the graphics application that created the drawing, with the drawing loaded and ready to edit. If the drawing is only linked, then displaying it is always handled by the graphics program, even when it's displayed in the word processor window. If it's embedded, then the word processing program maintains a static copy of the display, so the graphics program doesn't have to be running.

HyperTopic: Robustness in the Shared-Code Approach

Writing a program that calls on another application's existing functionality is a powerful approach, but John's experience with macros and programmed links between systems from different vendors suggests that you should keep in mind the potential for problems with long-term reliability. Here are some specific points to consider:

- Can the user accidentally edit files (blank forms or macros, perhaps) that are critical to your program's behavior?

- Can the user set options in a server application that will affect your program's behavior? Is the user *required* to set those options before the system will run?
- Can the user clearly distinguish between bugs in your program (not that there are any!) and bugs in one of the server programs? And is your customer support staff ready to track down those problems so the user isn't left between two vendors, both saying, "It's *your* fault!"
- If one of the server applications is upgraded, will it still run the macros you've written?
- If operating system upgrades require it, will the vendors of the server applications all upgrade their products promptly?

The shared-code approach has a lot of things going for it. But while taking advantage of its power, you should be especially conscious of one of Nielsen and Molich's heuristics: "Prevent errors." [heuristics, §4.3, p. 67] [N&M, p. 69]

If you decide to use interapplication communication in Windows, you'll need to know what applications your users already have available, and you'll need reference manuals that describe the interapplication communication conventions for those applications. For example, what's the syntax for the DDE command to average data in the spreadsheet, and how do you specify which data to average? You'll also need to build the code and the interface that's unique to your program, as well as the "glue" that links everything together. To simplify that part of the job there are several good UIMS packages for Windows programmers. One of these is Microsoft's Visual Basic.

Visual Basic combines a visual interface editor with an event-driven programming language. Creating an interface is a matter of creating windows ("forms"), menus, buttons, and other controls interactively. You click on the control you want in a palette, drag it into place, and specify its characteristics in a dialog box. The effect of each control is defined with Basic code written into the control's code window, and controls send messages to each other and to handlers that you write to define the program's functionality. Programs you distribute need to include a run-time Basic support module, implemented as a DLL file. The system can be extended with DLL code that you write yourself or license from third parties.

All this sounds very much like HyperCard, and conceptually it is. But Visual Basic is intended as a full-fledged program development environment, and it provides significant support for that goal. Menus, windows, dialog boxes, and all other common Windows controls can be created using visual programming. Additional controls licensed as third-party DLL routines are also fully integrated into the visual environment. Many of these controls are included in Professional Toolkit for Visual Basic, also sold by Microsoft. Taken together, these features allow a program written in Visual Basic to look and act like any other program in the Windows environment.

Modular code in Visual Basic is supported by allowing multiple files with several levels of variable scope: local to a procedure (the default), shared by all procedures in a form or module, or global for access by any procedure. The technique of run-time support through a DLL file is common in Windows, and Visual Basic helps you create an Installer for your application to ensure that all necessary files are in place and up-to-date on the user's machine. Interapplication communication is also supported. You can call other DLL's, send and receive messages from other programs using the DDE protocol, and make other programs act as OLE servers (although the program you write can only be a client).

Industry response to Visual Basic has been very positive. Developers are using it to prototype applications, then deciding to deliver the final application in the same language. There is some concern with speed, but this can often be addressed by recoding critical routines as a DLL. For programmers who aren't comfortable with the event-oriented paradigm, other UIMS packages offer interface-building functionality similar to Visual Basic, but with a more procedural language.

HyperTopic: What to Look for in a UIMS — and Where

There are many UIMS packages available, and more will be introduced in the future. Here's a checklist of features you should watch for if you're shopping for a UIMS or trying to convince your manager that one is needed. You should decide which items are critical for your project.

Features To Watch For

Action Logging

Built-in features that let you capture a trace of the user's actions are useful for some phases of usability testing, as well as for

tracking down intermittent bugs during beta testing. You'd like the trace to be at a fairly high level ("pulled down Edit menu"), not just a list of keystrokes and X,Y positions of mouse clicks. Macro packages running on top of your application may also give you this capability.

Big Program Support

Look for programming language and environment features that support good programming practices: modularity (multiple files, separate compilation, version support), scoped and typed variables, a good debugger, etc. Make sure the speed of the generated executables will meet your needs.

Code Generation

Does the UIMS generate executable code or does it produce a file of source code to be compiled with your usual compiler? The first approach is more convenient, while the second gives you a little more flexibility. Keep in mind, though, that automatically generated source code is often unmaintainable by human programmers.

Extensibility of the Interface

Can you add new controls and widgets to the programming environment, and will they have the same ease of programming as the original control set? Equally important, what extensions are available?

Extensibility of the Program

Can the program you generate call routines written in other languages?

4GL programming

Some UIM systems are part of a more extensive 4th generation programming language that simplifies development of the main program as well as the interface. This is especially likely for database programming applications.

Operating-System Specific Techniques

The UIMS should support any special interface techniques that users of other programs in the operating system have come to expect. A UIMS on the Macintosh, for example, should support the Macintosh clipboard, and it should generate applications that can be started by double-clicking on their data files. A UIMS for Windows should support OLE and help you build an application installer.

Prototype to Final Application Capability

You'd like to begin and end development with a single UIMS.

Stand-alone Application Generation

It's more convenient if the UIMS generates applications that don't require the user to purchase a separate run-time support package.

Style Guide Support

Some UIM systems will automatically check or enforce interface style guide provisions, such as "every dialog box must have an OK button," or "no more than 7 items in a menu."

Vendor Support and Longevity

Does the UIMS vendor have a history of maintaining the package and responding to upgrades in the operating system? Will the vendor still be in business when your application becomes a success?

Visual Programming With Structure

Can you lay out the interface visually? Can you specify constraints on that layout, preferably either with drawing tools ("align left") or programmable constraints ("all buttons of this class must be 50 pixels high").

Where To Find Out More

Here are some of the resources you can use to find out about the UIMS and prototyping systems available for your computing environment.

First, talk to people in your organization. If someone already has a system they're happy with, then you'll have the benefit of their expertise if you choose the same system. It will also make it easier for applications developed in different parts of your organization to communicate and to have similar interfaces. (This is the "borrowing" principle again.)

Next, start browsing through trade journals. Find a library with back issues of the "____ World" magazine (fill in the name of your computer system), and skim the tables of contents for the last couple of years. Look especially for review articles — you may find several prototyping systems compared and contrasted. In any case, look at the recent ads and send for information on systems that seem interesting. Try out the library's on-line or paper index to journal articles, but be sure to supplement any "hits" you find there with additional browsing in the stacks.

Another place to look is a large bookstore, especially a technical or university bookstore. The store will usually have a larger selection of magazines than most libraries, and it may also have a good selection of up-to-date technical "how-to" books ("Grover Cleveland's Six Easy Steps to Programming in UltraSystem 6.0," and the like). These books are of such narrow and short-term value that libraries can seldom afford them.

Still another resource is programmers outside your organization. If you belong to a user group, talk to people there (and look in back issues of the user group magazine). If you have access to a network, watch the discussions on-line, or post a query.

Related exercise 6.2 ▷ see page 161

7 The Extended Interface

The "extended interface" goes beyond the system's basic controls and feedback to include manuals, on-line help, training packages, and customer support. These are all resources that users rely on to accomplish tasks with a system.

How important is the extended interface? Isn't it true that "nobody reads manuals?" On the contrary, a survey we did recently showed that users in all kinds of jobs, from secretaries to scientists to programmers, will turn to external information sources when they don't know how to do something with a computer system. The source they turn to varies: it may be the manual, it may be the local system support people, it may be a phone support line. But users almost never know everything there is to know about the applications they use, and when they need to do something new, they look to the extended interface for help. (For details of the survey see Rieman, J. "The diary study: A workplace-oriented tool to guide laboratory studies," Proc. InterCHI'93 Conference on Human Factors in Computer Systems. New York: ACM, 1993. pp. 321–326.)

Of course, the usefulness of external support varies, not only with the individual but also with the situation. For a walk-up-and use system, such as an information kiosk in a museum or a flight-insurance sales machine in an airport, the on-line interface is the whole ball game. But for a complex application aimed at a professional market, such as Mathematica or a sophisticated desk-top publishing system, external resources will be important while users are first learning the package, and they will continue to be important as a reference to seldom-used functions. In short, the use and importance of the extended interface depends on the task and the user. To accommodate this dependency, the development of the extended interface should be integrated into the task-centered design process.

This integrated design approach should begin with your very first interactions with users. You should be on the lookout for situations where manuals or other external resources would be used, as well as for situations where that kind of support would be inappropriate. Those observations will give you the background needed to rough out the design, imagining task scenarios in which users will work with both the on-line system and external support. The techniques we've described for evaluating the system in the absence of users can also be applied to the entire system, especially to manuals and on-line help. Later in the design process, when you've built a prototype of the system and the external resources, task-centered user testing with the complete package can predict much more than testing in a

barren, unsupported environment that most users will never encounter.

That said, we should raise a flag of caution:

Don't rely on external support to make up for a bad on-line interface!

It's true that most users will look in a manual or ask for help if they can't figure out the system. But they don't especially want to do this, and it's not a very productive use of their time. Worse, there's a good chance that they won't find the answer they're looking for, for reasons we'll describe later in this chapter. So, you should strive for an interface that comes as close as possible to being walk-up-and-use, especially for the core functionality that's defined by your representative tasks. External support should be there as a fallback for users who stray from the common paths that you've tried to clearly define, and as a resource for users who are ready to uncover paths that you may have intentionally hidden from novices (such as macro packages or user-programmable options).

The rest of this chapter gives guidelines for some common forms of external support: manuals, training, etc. This is a traditional division, but it's not necessarily one you should adhere to. Once again, look at your users, look at the task, and determine what would be the best way to support the on-line interface.

HyperTopic: The Integrated Design Team

Integrating the development of the entire interface requires that the developers of all parts of the extended interface work together as a unified design team, with common goals and reporting to the same manager. This is in sharp contrast to a more traditional corporate approach that divides responsibility into software development, manual writing, and customer support. If you're in a situation where integrated development isn't an option, at least try to get individual representatives of the different areas to attend each other's meetings.

7.1 Manuals

For many users, the manual is the most readily available source of information outside the on-line interface. For other users, the first place to go for information is the on-site support person or another, more expert user —

but those experts gained much of their knowledge from manuals. In general, a good manual is the most important extension to the on-line interface.

To help understand what a "good" manual is, it's useful to imagine doing a cognitive walkthrough on a manual and noting points where the manual could fail. First, the user may be looking for information that isn't in the manual. Second, the manual may contain the information but the user may not be able to find it. Third, the user may not be able to recognize or understand the information as presented.

A task-centered design of the manual can help overcome all of these problems. If you understand the user's tasks and the context in which they're performed, you'll be able to include the information the user will look for in the manual. This will be primarily task-oriented descriptions of how to use the system itself, but it may also include descriptions of how your system interacts with other software, or comments about operations users might want to perform that aren't yet supported. Knowledge of the users will help you present this information in terms that make sense to them, rather than in system-oriented terms that make sense to the software designers. To repeat one of Nielsen and Molich's heuristics, you should "speak the user's language." [N&M, p. 69]

The problem of users not being able to find information that's in the manual is probably the most difficult to address. Speaking the user's language will help some, and keeping the manual "lean," including only material that's relevant, will help even more. Indeed, brevity is the touchstone of an important approach to documentation, the "minimal manual," which we discuss under the heading of training. But the problem goes deeper than that, and we describe a further solution in the section on indexing.

Deciding on the top-level organization for your manual should be another place where the principles of task-centered design come into play. Using a manual is just like using the on-line interface: people can transfer the skills they already have to the system you're designing. When you're getting to know the users and their tasks, look at manuals they're comfortable with, and watch how those manuals are used. Make sure to look at the manuals users actually rely on, which are often task-oriented books written by third parties to fill the gaps left by inadequate system documentation. Design the manual for your system so it fits the patterns of seeking and using information that users find effective.

In the spirit of borrowing, the HyperTopic in this section shows a default manual organization that should be familiar to anyone who has worked with larger applications on personal or minicomputers. The manual has three parts. First, there's a section that describes common procedures in

a narrative style. This section is organized around the user's tasks. Then there's a detailed description of each command, a section that's organized more in terms of the system. Finally there's a "Super Index," a section specifically designed to overcome the "can't find it" problem that users so often have with computer manuals. We might have also included a reference card, if users expected one. As a more convenient alternative, however, we'll assume that brief reference information is included as on-line help.

The next few paragraphs describe each of the sections in the default manual organization. Keep in mind that this organization wouldn't be appropriate for every system. It would be overkill for a VCR, for example. However, many of the principles that apply to the design, such as brevity and speaking the user's language, will apply to manuals of any size.

Example: Organization of a Manual for a Large Application

The UltraProgram Manual

Detailed Task Instructions This section gives step-by-step instructions for each of the major tasks you can complete using UltraProgram. You can use these instructions as a detailed tutorial, or you can scan the headings to get an overview of a procedure.

Command Reference Here you'll find descriptions of how each menu item and dialog box in UltraProgram works. Look here if you don't understand a command's options, or if you can't figure out why it has the effect it does.

Super Index Look here if you know what you want to do, but don't know what command you should use. This is your "traveler's dictionary" that translates your phrases into the ones we use in the rest of the manual. For example, if you look up "form letters" or "address lists" or "document assembly," they'll all refer you to the sections on "mail merge" (the term used in UltraProgram).

7.1.1 The Detailed Task Instructions

The Detailed Task Instructions give the user explicit, step-by-step instructions for performing each of the major tasks the interface supports. This section will support different needs for different users at different times.

Some users will work through each step of each task in order to learn the system. More adventurous users may just glance through the Detailed Task Instructions to get an overview of how a task is performed, then investigate the system on their own, referring to the instructions only when problems arise. For both novices and experienced users, the section will be used as a reference throughout the useful life of the system.

Two questions to consider in writing the Detailed Task Instructions are what information the section should include and how that information should be organized. If you've been following the task-centered design process, the question of what to include should be easy to answer: The instructions should give step-by-step instructions for performing each of the representative tasks that have been the focus of the design effort. Additional tasks may be added if the representative tasks don't cover the entire system. Training for each task should cover everything a user needs to know for that task, with the exception of things that your intended user population already knows. The cognitive walkthrough will help uncover the details of what the user needs to know, and your early user analysis should describe the knowledge users already have.

The top-level outline of the Detailed Task Instructions section will simply be a list of the tasks. For each task, the instructions should give a brief conceptual overview of the task and the subprocedures used to accomplish it, then present sequential, step-by-step instructions for each subprocedure. The following example gives the overview for the mail-merge instructions of a word processor. That overview would be followed by the step-by-step details of each of the three major subprocedures needed to accomplish the task.

Example: Sample Overview Section of a Detailed Task Description

> Section 3: Mail Merge
>
> You can use the mail merge feature of UltraProgram to send identical or similar form letters to many addressees.
>
> Imagine that you want to send a short letter to three customers, telling them that their account is overdue, by how many days. The detailed steps in this section show you how to:
>
> 1. create a file containing the basic letter,
> 2. create a file containing addresses and overdue information,
> 3. merge the two files to create the finished letters.

Notice that the sample overview is very, very brief. It's tempting to put a lot of detail into the overview, both to help the user understand the upcoming detailed steps and to call out useful options that the current task doesn't exercise. However, detail is exactly what the user does *not* need in the overview. If you fill the overview with technical terms and commentary on options, it will be meaningless techno-babble to the novice user. Even our simple mail merge overview won't mean much to a user who has never done something similar on another system. A novice's mental overview of an operation will slowly emerge out of an understanding of the details; the best your written overview can do is point them in the right direction.

The overview is also brief because the task itself is simple. If your representative tasks are complex, you should break them into simpler subtasks for the purpose of the manual. For example, don't give detailed instructions for doing a mail merge controlled by keyboard macros that load the files and insert today's date.

Another workable approach is to include a description of advanced options in a subsection at the end of the step-by-step task description. If you do this, be sure to put a pointer to that section into the overview, so users already familiar with similar systems can find it without working through details that don't interest them.

For each task, it's good to have a complete example, one involving actual file names, dialog box selections, etc. A more abstract description, such as, "Next, type in the file name," will inevitably leave some readers confused about the details abstracted. Showing the details of the example task will be much briefer and clearer than trying to explain the same information.

Within the step-by-step description of each task, the important principle is to be as brief as possible while still supplying the information the user needs. Remember that users already know some things, and the on-line interface provides additional information. Each step of the instructions should just fill in the gaps. Here again it's useful to think in cognitive walkthrough terms. Is the user trying to do the right thing? If it's not clear, the instructions should make it so. Is the action obviously available? If it isn't, the instructions should explain where it is. Will the user realize that the action moves them along the intended path? If not, clarify why it does. And finally, be sure to periodically describe feedback, so the user knows they're following the instructions correctly.

HyperTopic: Writing Tips for Manuals

Be brief. Briefness is central to a usable manual. In longer manuals, users often can't find the information they need because it's buried in a mass of irrelevant text. Here are a few suggestions on how to achieve briefness:

- Write a section, then go back later and delete repetitive or unnecessary text.
- Cut introductions, overviews, and summaries to the bone. Expert manual users often skip these sections, while novices just bog down in them.
- Don't be historical. Users don't care about the design history.
- Don't be philosophical. Users don't care about the design rationale.
- Don't give technical details. Users don't care about the internal mechanisms.
- Don't market. They've bought the product, don't try to sell them on it again.

Present an overview and a sequence of steps for each task. This is the approach we discuss in the text. Remember to keep the overview brief, and work through a simple but real task. A follow-up section on advanced features related to the task is optional.

Speak the user's language. You've heard this one before, but it bears repeating. You're writing a user manual, not a technical description for other software designers.

Find a light-handed editor. A good editor can catch awkward sentences and gaps in your presentation, as well as correcting errors in grammar and spelling that reduce the perceived quality of your product. A "light-handed" editor helps you maintain your personal writing style, which should reflect your knowledge of how users communicate.

Find an editor who is also a user. An editor who has a background similar to your user population can easily notice descriptions that are incomplete or hard to understand. An editor with no technical background will either limit the editing to simple grammar and spelling ("copy editing") or ask for too much additional detail in the descriptions.

Use a general and an internal style guide. A style guide gives answers to the hundreds of writing questions for which there is no "right" answer, such as whether to write "disks" or "diskettes," or whether to call your system "UltraProgram" or "the UltraProgram." A general style guide, such as the "Chicago Manual of Style," gives default answers to questions that commonly arise in books. An *internal style guide* is one you develop within your project or company, to cover points not in the general guide, as well as points where you want to disagree with the general guide. With an evolving internal style guide, you and your editor can make a decision on a style question once, then go on to more important things.

Use graphics sparingly and professionally. Except for screen dumps, graphics (cartoons, photos, abstract designs) usually don't add much to a manual. If you need an illustration, such as a drawing of hardware to show where switches are located, get a professional artist to do a clean, simple drawing.

Borrow an attractive and functional visual design. When you're deciding on the manual's visual design, such as the appearance of chapter headings, the font for the text, and the layout of tables, you should once again take a good look at several manuals that users find workable. You can also call in a professional designer to fine-tune things like type size and font; but borrowing from existing designs is the best way to start.

7.1.2 The Command Reference

The Command Reference section of a manual includes detailed descriptions of each command, including seldom-used information such as keyboard equivalents, hidden options, maximum and minimum sizes of data objects,

etc. This information will be most useful to more experienced users who are pushing your system to its limits. It will also be turned to by system administrators who are trying to understand problems that arise in systems they support. Many users, however, will never need the Command Reference section. It shouldn't be required to complete the reference tasks on which your task-centered design has focussed, and it is the first section you should eliminate for smaller systems.

Because the Command Reference is a system-oriented view of the interface, it may be the easiest section for you, the designer, to write. You still need to write for the user, but now you're writing for a user with more experience, so the presentation can be more technical. It's often effective to use a standard, tabular format for each command, which might list the command name, its effect, a description of options and parameters, and any warnings or limitations. Keep the entire description as brief as possible.

The organization of the Command Reference section is problematic. With command-line interfaces, the section is typically alphabetical by command name. For graphical interfaces, one option is hierarchical, following the menu hierarchy: look under File to find New, then look within that section to learn about the New dialog box. This is workable with shallow hierarchies (File...New...dialog box), but it becomes cumbersome with more deeply nested menus (File...Save...New...dialog box). For such systems, an alphabetical organization may again be more useful.

7.1.3 The Super Index

[See also pp. 51, 71 for examples]

In the mid 1980s, a group of researchers at Bell Communications Research (Bellcore) noticed that it seemed almost impossible to choose the "right" names for commands. A system feature that one user called Copy, another wanted to call Move, while the designer thought it should be called Paste. This came to be known as the *vocabulary problem*. To determine how serious the problem was, the Bellcore researchers surveyed several groups of people, hundreds of individuals, asking for the names of common household objects and computer procedures. The results were very disheartening in those days of command-line interfaces.

The surveys showed that a computer system designer's first choice for a word to describe a system feature — the "armchair design" for a command name — had roughly one chance in ten of matching the word first assigned to the same feature by a randomly selected user. If the designer called the feature Paste, then nine out of ten users would expect it to be called something else. (G.W. Furnas, T.K. Landauer, L.M. Gomez, and

S.T. Dumais. "The vocabulary problem in human-system communication," Communications of the ACM, 30 (Nov. 1987), pp. 964–971.)

But the problem was worse than a simple indictment of armchair design. Even if the word most commonly selected by users was assigned to the system feature, there would still only be about one chance in five that a randomly chosen user would select that word. The survey of names for common objects showed that this wasn't simply a problem related to the newness of computer technology. People just didn't use the same words for things, although of course they would recognize other people's words.

Quite simply, the basic result meant that there was no "right" name for a command. Whatever word was chosen, 80 percent of the users would probably expect the command to have some other name.

Graphical user interfaces have partially overcome the vocabulary problem by asking users to *recognize* commands in menus, rather than *recall* them and type them in. A user who expects a Copy menu item can usually recognize that the Move menu item has the same effect, although more complex concepts still cause problems. Within a manual, however, the vocabulary problem remains very real. A common complaint about manuals is, "I can't find anything in them."

We saw an example of this recently while watching some workers in their normal office setting. Two users were trying to find a word processor's "overstrike" feature, which they wanted to use to mark a block of text in a document. They checked all the menus, looked in the manual, and even looked in a third-party manual. They couldn't find the entry for "overstrike" in the index or table of contents of either manual, although they were convinced they had seen the feature somewhere in the program. Finally they gave up and decided to make do with a feature that forced them to backspace and overstrike each character individually. The feature they were looking for but never found was indeed available. It could be found in a dialog box under the font menu, and it was listed in the index. It was called "strikethru."

As this example illustrates, the vocabulary problem can seriously reduce a manual's usefulness. The typical manual's index has very few entry points to each concept or command. There will be the item itself, possibly listed hierarchically under some broader heading (such as Copy under Edit). And there will be a few "see" or "see also" references that point to the same concept ("Clipboard, see also Copy"). But there won't be entries for synonyms unless those synonyms have actually been used in the text. That means that the user who is looking under "Move" may never find the section on "Copy."

The Super Index helps overcome the vocabulary problem for manuals. This index makes use of a further finding of the Bellcore researchers. A single term is clearly inadequate as an entry point into the manual, but several well chosen terms can significantly increase the effectiveness of the index.

The first step in creating the Super Index, then, is to apply one of the techniques of task-centered design: borrowing. Look at other packages in the market you're entering and determine what terms they use for various operations. You should already be using the most common of those terms as command names in your own system. Add the other terms into the index as "see" references: "Paste, see Copy" For terms that have other meanings within your system, use see also: "Move, 24, 67, 92; see also Copy."

For larger manuals it's worth taking a second step toward creating a Super Index. Survey potential users to determine what terms they would use for system features. The survey should describe each feature without giving its name (for example, "what do you call it when text is taken away from one place and put back in another?"). Ask users to give three to six synonyms for each operation. The number of users surveyed doesn't have to be large; even a half a dozen will make a big difference, but a larger number will be better. The five to ten most common synonyms for each system feature should be added to the index as "see" or "see also" references. The results of a small experiment done by the Bellcore researchers suggested that an index based on a small user survey might make as much as a four-fold improvement in the ability of other users to find items in the manual.

Related exercise 7.1 ▷ see page 163

7.2 On-Line Help

On-line help is the great unfulfilled promise of computer applications. Surely the power of a computer to search, reorganize, customize, and even animate information should make it possible to help the user far more effectively than with printed paper, a centuries-old technology. Well, maybe someday it will. But except for a very few products that have required extensive effort to develop, such as Bellcore's "SuperBook" and the Symbolics Lisp Machine's "Document Examiner," the attempts to deliver large volumes of information on-line have generally been shown to be no more effective than traditional books and manuals. This is the case even though the on-line systems offer hypertext features such as word search and linking to related

topics. Without these features, on-line text may well be less effective. For basic on-line help systems the message is this: If you need to present more than a brief description of a system feature, don't rely on on-line help.

The ineffectiveness of lengthy on-line help files is probably the result of several factors: Text on a computer screen is usually less readable than printed text. Less text is presented on the screen than on the printed page. It's much easier to get lost while navigating through screens of text than while thumbing through pages of a book. Screens don't update as quickly as pages turn. You can't circle a word in on-line text with your pencil, or dog-ear a page. The help window or screen often covers the part of the interface that the user has questions about. And, people haven't practiced reading on-line text for most of their life, as they have with printed text. The combined effect of these problems over-balances all the advantages that are associated with on-line text.

But while on-line help isn't the place to put the entire manual, it does have potential uses. It's an excellent place to put brief facts, "one liners," that users need frequently or are likely to forget. A prime example is the definitions of function keys or keyboard equivalents to mouse menu items. On many systems, these mappings are shown on the mouseable menus, an effective form of on-line help for users who are interested. Another example is a list of command options for a command-oriented system. Users often know exactly what they want to do in these systems, but forget the exact spelling or syntax of a command. A *simple* display of that information can save the user the trouble of digging out a manual. For these and other simple facts, on-line help is usually better than a reference card, which is easily lost or misplaced.

The most common failure of on-line help is to provide too much information. It's not that users could never apply the extra information, but that they often won't be able to find what's immediately relevant within the extra text. The solution is to cut the on-line text to a bare minimum. As a rule of thumb, a line of information is useful. A paragraph of text is questionable, although a table with several lines may work. An entire screen should usually be relegated to the manual.

To go a little beyond the rule of thumb, you can apply a slightly modified version of the cognitive walkthrough to on-line help. Imagine a user with a problem that the proposed on-line help could solve. Ask yourself: Will the user think of looking in on-line help? Will the user be able to find the information that would solve the current problem? And if the user finds the right information, will it be obvious that it answers the user's question? The last two points recall the vocabulary problem, which applies in spades

to on-line help systems. These systems can't easily be scanned or browsed, they usually don't have an index, and reading speed and comprehension deteriorate for on-screen text.

Related exercise 7.2 ▷ see page 163

7.3 Training

In this section we'll use the word "training" to include both classroom instruction and tutorials, on-line or otherwise, that users can do on their own. Many of the ideas about training for computer systems were developed before personal computers were widely available, when users had to be trained in the most basic computer concepts and activities. Today most users have experience with many kinds of computers, including PC's, games, telephone interfaces, automated teller machines, and so forth. Designing training for these users is both easier and harder than working from the ground up. It's easier because less training is needed, especially if you've borrowed key interface techniques from programs the user already knows. But it's harder because you have to decide what topics the training should cover, if any.

Once again, the task-centered design process should have already provided an answer to the question of what the training should cover. It should cover your representative tasks, just like the manual. In fact, the manual described in this chapter should be an effective training device for users who want to work through it. It provides exactly the information that's needed, and it's organized around the appropriate tasks. The manual also provides an appropriate fallback for users who want to explore the system without guidance.

However, users differ widely in their approaches to learning a new system. Some users, and some managers, prefer a classroom training situation, or at least a section of the documentation that is specifically designed as a tutorial, not just part of the reference manual. If you decide to support the needs of those users, the tasks described in the manual still make a good starting point. Unlike the manual, however, the training package needs to be structured in a way that will force the user to become actively involved in working with the system.

A "minimal manual" version of your basic manual can serve as an effective centerpiece for a training program. The minimal manual, an idea suggested and tested by John Carroll at IBM, goes a step beyond the brevity we recommend for the basic manual. The minimal manual is intentionally incomplete. It couples a carefully crafted lack of details with clear task goals,

forcing the user to investigate the on-line interface to discover how it works. (J.M. Carroll. "The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill." Cambridge, Mass.: MIT Press, 1990.)

Several studies by Carroll and other researchers have shown that the minimalist approach yields dramatically shorter learning times than traditional, fully guided training. The approach is also supported by several things that psychology has discovered about human learning (see HyperTopic). Some authors even recommend that the only manual supplied with a system should be a minimal manual. We think that view underestimates the value of a more complete manual for long-term reference, but we do recommend minimal training documents. And we echo the minimalist call for the briefest possible presentations of information, in training, manuals, and throughout the external interface.

HyperTopic: Some Psychological Facts About Learning

Experimental psychologists have been studying learning and memory objectively for over a hundred years. They still can't tell you how to transfer knowledge from one person to another with the ease of copying files onto a new disk, but they have discovered some things that help predict what kinds of training will help the user learn things faster and remember them better.

In this section we describe some of the more powerful learning effects that have been discovered. You should, however, take this information with a grain of salt — maybe even with a whole saltshaker. All of these effects have been repeatedly validated in laboratory experiments, but learning in the real world involves a complex mixture of influences, many of which aren't well understood because they are too hard to study in the laboratory.

So, use these facts as ideas to help you improve your training program, but keep in mind that they are only a small part of a much larger story.

People learn facts best if they have to work with them. There are lots of ways to make learners "work with" what they're learning. You can describe two procedures, then let them decide which is correct for a given situation. You can ask them to figure out on their own what the system does (this is called "discovery" learning). You can ask them to think about how the procedures they are learning would apply to their own work. You can teach them facts briefly,

then later ask them to recall what they've learned. The important thing is to do something more than simply presenting the step-by-step procedures and expecting those to be memorized.

Skills improve with practice. Like a lot of psychological discoveries, this seems obvious. But it may not be obvious how broadly applicable it is. Essentially any skill, from typing to memorizing numbers to skiing to solving math problems, can be made faster and more accurate with the right kind of practice.

Practice requires feedback to be effective. The practice that is most effective for learning is practice in which learners are encouraged to try things and then given immediate feedback as to whether they've done the right thing or not. This kind of practice, incidentally, is a lot of work for the learner.

Spaced practice is more effective. Practicing the same simple exercise over and over (for example, saving a file) isn't as effective as doing the exercise once, then doing several other things, then doing the original exercise again. In a training program, a set of skills should be introduced, then the instruction should go on to other things. Later, the learner should be put in a situation where the first set of skills have to be recalled and used again.

Things that shouldn't be learned should be varied. If you present a series of exercises involving files named File-A, File-B, File-C, File-D, and File-E, then the users may go away thinking that all files have to be named "File" plus a letter. (This is something we've actually seen happen, before basic computer knowledge was more widespread.) More subtly, if your system has several screens available but the training exercises all happen to be done while one screen is showing, then users will be more likely to remember what they've learned when that screen is visible in the future — even if they realize that the same facts apply to the other screens.

People can only work on learning a few things at a time. This factor most often becomes a problem when the learning environment itself requires more attention than the material

that should be learned. An on-screen tutorial that requires the user to learn several keystrokes to navigate through it is one bad example. Another is a tutorial that frequently requires the user to find and load special files from floppy disks.

People can't run (or dance) before they walk. This is as much common sense as psychology, but it interacts with the previous item about learning only a few things at a time. If learners don't have a good grasp of basic skills, such as file loading, keyboard and mouse usage, menu selection, and using the "undo" menu, then they won't be able to concentrate on learning more sophisticated skills. An effective way to teach computer skills, where possible, is to teach basic skills one day and send users back to their office for a week of practice. Then bring them back for a day of advanced training. (What's "basic" and "advanced" depends on the user and the system, of course.)

7.4 Customer-Support Phone Lines

Phone support is a huge cost for a successful product, and a good motivator for more attention to usability. Do the arithmetic by multiplying out two five-minute phone calls per customer by the number of customers: for 100,000 customers that's a million minutes on the phone, or about five years of working days. If you cut that back to one call by improving your design you're saving a lot of money directly, to say nothing of the value of increased customer satisfaction and productivity.

In setting up an effective customer support line, the three keys are training, tracking, and — once again — task-centered design. Training should be obvious. The people answering the phone should have the answers, or know where to get them, and they should have some basic training in phone communication techniques. Tracking refers to the user's questions, *not* the users themselves. You want to know what problems the users are having, even if they are problems that seem to be "the users' fault." Those are the problems you should be addressing in the next upgrade of your system.

And again, task-centered design. If your company already has products on the market, then you have some experience with the kinds of problems users will have. If not, you can probably imagine the categories of questions

that will arise: how-to questions, software bugs, compatibility problems (with other applications, or hardware, or the operating system), questions about upgrades, etc. Imagine some specific questions in each of these categories, and then get together with someone else in your design group and act out the interaction that an imaginary user would have with your phone support technician. This is a walkthrough-like technique that can suggest modifications to both the phone-support system and the supported product, and it can also be used to train the support technicians.

Communication between a knowledgeable technician and a less-experienced user will always be problematic, and it's even harder over the phone than in person. The HyperTopic in this section gives some suggestions for improving communication. There's nothing surprising about these guidelines; any user who has dealt with on-line help would have similar ideas. Keeping in touch with the user community after your product is on the market will suggest additional improvements.

As a final suggestion concerning phone support technicians: be one! Customer calls are a great source of feedback from users. One project at Chemical Abstracts, which sells information services for chemists, has all members of the development team, including the project manager, rotate through the office that handles customer calls. This way everybody finds out what customers are trying to do, and what problems they encounter, first hand.

Related exercise final ▷ see page 164

HyperTopic: Suggestions for User-Centered Phone-in Help

Speak the user's language. (Obviously!)

Be polite, cheerful, and positive. Don't be arrogant, and don't make users feel like the problems are their fault.

Give frequent feedback so the user knows that the conversation is going in the right direction. An example of phone interactions where this is typically done well is the practice of reading back order information for credit-card catalog orders.

If there's information that the user will have to provide, such as a software version or license number, make sure it's recorded where the user can find it. The start-up screen is a possibility, or under an "About this program" menu, with the information also recorded in the manual in case the system won't run.

Avoid asking for information that isn't related to the user's problem. Asking for the software license number is OK. Users will understand that only licensed software is supported. But it's wasting the user's time to ask where the package was purchased (corporate users often won't know), or what is the user's zip or area code.

Make it possible for the user to call back and get the same technician for follow-up help. After explaining the problem in detail to a disembodied voice named "Janet," the user doesn't want to call back 15 minutes later and be routed to "Carl," who not only doesn't know the problem but hasn't even heard of "Janet."

Transferring the user's call to other support people is always dangerous. Something that will help is a phone system that can support a brief three-way connection during which the technician eases the transition. Otherwise the user is put on the spot, having to explain the problem again, as well as explaining why they've been transferred.

For users, the worst-case scenario is one in which you can't transfer them at all and they have to redial, especially if they have to punch their way through a touch-tone dialog and come back into the end of a hold queue.

L What Can You Borrow? A Quick Introduction to Copyrights and Related Legal Stuff, as of 1994

If you read computer magazines or the business press, you know that many of the big software companies have recently been embroiled in legal battles with each other over claims of copyright violation. That fact reflects the current state of the copyright law: it's very confused. Copyright law was developed primarily to protect printed matter, and courts are still struggling to decide how to apply that law to the computer software, especially user interfaces.

Because the law is unclear and rapidly changing, we can't give you firm advice on exactly what you can borrow from other interfaces. But we can give you some background that will help illuminate the issues the courts are struggling with. And, although we aren't patent or copyright attorneys, we can define what we perceive as fairly clear boundary markers: things you certainly can borrow and things you certainly cannot. Our practical, procedural advice is that you should build your interface with those boundary markers in mind, then work with your company legal staff to check out the gray areas.

L.1 Background

We'll start with the background. In the United States there are four main legal techniques for protecting "intellectual property," which is what the law calls novel ideas that people invent or create. The four techniques are trade secrets, trademarks, patents, and copyrights. All are intended to encourage creativity and commerce by giving people some kind of ownership in the fruits of their intellectual labor and by making their creations available to the public.

Trade secrets are simply things that a company doesn't reveal about its product. Most software code is protected this way. The source code — the human-readable Fortran or C or Pascal — doesn't come with the package when you buy a program for your personal computer, so you can't copy parts of it and use them to build a different program. Various laws allow the company to prohibit employees and former employees from passing secrets on to other companies. Obviously, trade secrets don't protect the parts of the interface that users can see, but the underlying implementation is usually secret.

Trademarks don't protect software itself. Instead, they protect the graphics or words that a company uses to identify its products, so other companies can't confuse customers by using similar identifiers for their products. It takes time and effort to acquire rights to a new trademark, and protection can be lost if the public starts to think of the trademark as a generic term. This happened to both "aspirin" and "nylon," for example. To prevent this, major companies will aggressively encourage you to identify ownership of their trademarks if you use them in your documentation. This is the reason for notices of the form "BigGraph is a trademark of BigCompany, Inc." that you often see in computer publications. Trademarks sometimes appear as a part of an interface, such as the stylized apple in the Macintosh menu bar, which is a trademark of Apple Computer.

Patents give inventors exclusive rights to machines or processes they've invented, for a period of 17 years. Patents are relatively difficult and expensive to obtain, and before 1981 the patentability of software was uncertain. Today software patents are fairly common, including some covering interface features. For example, features of a program called "Zoomracks" were patented, and that patent had to be licensed by Apple Computer when it produced HyperCard. Patents protect an inventor's ideas from more than copying: they also protect the ideas from being used by someone who reinvents them. This means you can infringe a patent without ever having seen the inventor's work! Some of the requirements for a patent are that the idea be substantially innovative and original, that the inventor disclose the idea in sufficient detail that other people can apply it when the patent protection expires, and that the exact boundaries of the new idea — the "claims" of the patent — be spelled out.

Copyrights are currently the most important form of intellectual property protection that you'll deal with as an interface designer. Copyrights were originally developed to prevent unauthorized copying of printed text, graphics, and audio-visual recordings. Their protection has been extended to computer software, in part because the physical process of copying software is similar to copying text or recordings, and in part because the availability of patents for software was uncertain. A copyright is easy to acquire. As soon as you create something, you (or your employer) own copyright to it. If you publish what you've created, then you should register it with the US Copyright Office and put a notice on copies that you distribute: "Copyright 1995 by Joe Programmer." Failure to do those things doesn't completely destroy your rights, however. A copyright gives protection for the life of the author plus 50 years, or for 75 years if it's owned by a company.

L.2 What's Covered by Copyright

You should assume that all commercial software is protected by copyright, unless it comes with a specific statement giving you rights to copy it. But exactly what protection does that copyright provide?

Copyrights protect the "form" of an idea's expression, but not the idea itself. For example, we, the authors, hold the copyright to this book. If you want to copy the chapter describing task-centered design, you need our permission (which we give, with restrictions, in the shareware notice). But if you want to use the task-centered design process, our copyright doesn't stop you. In fact, you could write your own description of task-centered design and publish it, even copyright it. You could do those things because the copyright on this book only protects the way we've described the process: the words, the graphics, the overall structure of the presentation. Unlike a patent, it doesn't give us any rights to the idea.

The distinction between what's the "idea" and what's the "expression" is one of the important questions that hasn't been clearly answered for copyright protection of interfaces. The copyright statute provides very little guidance, so the courts have to decide each individual case by analogy to previous court decisions, most of which don't deal with software. One principle the courts rely on is that functional parts of the design qualify as ideas, not as expressions. So, for example, selecting from a menu is functional. That's an idea. Maybe it could be patented, but it can't be copyrighted. On the other hand, the graphical details and perhaps the order of items in the menu could be copyrighted — unless they were specifically designed to improve the menu's usability (that makes them ideas), or unless they are the only way of making this menu work (that makes the idea "merge" with the expression). You can see that this is a slippery issue.

There's a second important question about which the law is still unclear. If we go back to the printed-text foundations of copyright, we find that our copyright on this book prevents other authors from copying an entire section, or translating part of the book into another language, or outlining it, or producing a very close paraphrase. But it's obvious that the copyright doesn't stop anyone from using the same individual words and phrases we've used in some completely different order. If you want to publish a sentence with "task" or "design" or "it's obvious that" in it, you can. Individual words, common phrases, and even longer, anonymous texts that everyone knows (such as, "Why does the chicken cross the road. . . "), are in the public domain. Similar principles apply to interface copyrights, but it isn't yet clear what's a word-sized or phrase-sized chunk of an interface, or what's in the

public domain, or what's a translation or a paraphrase. The courts have made some rulings, but the answer will change as technology and the law evolve.

Both of these issues — ideas versus expressions, and public domain elements — are critical to the "look and feel" lawsuits that have seen so much coverage in the recent press. Is the desktop metaphor a functional idea, or is it one expression of an idea? And even if it is an idea, is it in the public domain? Some courts have recently taken the approach of breaking down an interface into its component elements and analyzing those separately for copyright violation, which may spell the death of the overall "look and feel" protection. But the issue is still undecided.

L.3 Practical Boundary Markers

With that as background, here are some of the boundary markers we promised. They define the current state of interface law as we see it, from a practitioner's point of view, not from an attorney's. They should help you decide what can or can't be copied as you design an interface.

1. Things you certainly can copy (unless the rights have been sold).

    - Anything produced by your company.
    - Things you've written earlier for your current company.
    - Things recommended in the style guide for the system you're developing under.
    - Things supplied as examples/prototypes in a commercial toolkit, programming language, or user-interface management system (often these require you to give notice, such as "parts of this interface are copyrighted by WallyWindows UIMS").

2. Things you can probably copy — but watch the news and check with your attorney.

    - "Common" ideas, such as windows as the boundaries of concurrent work sessions, menus for selecting commands, an arrow-shaped pointer controlled by a mouse, graphical icons to represent software objects. The problem here is making sure it's a common ("public domain") idea, not one developed by some company and stolen by a few others.

- Sequences or arrangements of menu items, commands, screens, etc., IF the original program clearly orders the sequence to improve usability (i.e., if it's alphabetical, or most-common-item first), or IF there is only one or a very few other ways that it could be arranged (i.e., if there are only two items in a menu, then there are only two ways they can be arranged).
- Icons ideas, commands, menu items, or other words that are obvious choices for the function they represent, so usability might be reduced if other words or graphics were used. An example might be the word "print" to stand for printing, or a computer-mouse icon to select mouse options (but don't copy the mouse graphic itself — draw your own).
- Things you can probably not copy — again, watch the news and check with your attorney.
- Sequences or arrangements of menu items, commands, screens, etc., if you're only copying the sequence order because it will make it easier for users of someone else's existing program to use your new program.
- Icons, commands, menu items, or other words that are not an obvious choice to describe their function, even if they would make your program more usable for users of the original program. An example might be a database print command labelled "DataDump," or a mouse-options icon showing a cute little mouse with a checklist.

3. Things you can certainly not copy (unless you get permission).

- Things you've written earlier for a different company.
- An entire interface from another company's program, even if you implement it with all new code.
- An entire detailed screen from another company's program.
- Source code or object code from another company's program, even if you translate it into a different language.
- Trademarks from other companies. If you need to use someone else's trademark, such as "Unix^{TM}" in documentation, be sure you credit the owner: "Unix is a trademark of Unix Systems Laboratories, Inc."

- Patented features. Unfortunately, there's no easy way to discover what's patented.
- Exact bitmaps of icons, words, or fonts.
- Graphic details that define an interface's aesthetic look.

L.4 Strategy

As we noted earlier, the development process we recommend is to keep these boundary markers in mind, develop your interface, and then talk over your decisions briefly with your company's legal staff. If there's a central feature of the interface you're worried about, such as picking up an entire interface metaphor, you may want to get legal advice a little earlier.

If you're an individual developer or a small start-up company, the same strategy should work, with a couple of modifications. First, attorneys are expensive. You should have one, but make sure you have a client-attorney relationship in which the attorney provides guidance and you do the legwork of checking out other programs, negotiating for license agreements, etc. Second, when you're making decisions such as whether to incorporate your business, how much of a contingency fund to maintain, and how much to charge for your product, you should consider the possibility that your system might unknowingly violate a patent or a copyright.

L.5 Some Philosophical Observations

The current state of the law makes the design of good interfaces unnecessarily difficult. This is largely because of recent court cases that extend copyright protection to functional interfaces. Historically, the court cases and the statutory law that defined copyright were developed largely for the protection of published text and entertainment (novels, music, films) that are typically used *once* by each user. This law is fundamentally inappropriate for protecting functional interfaces, although perhaps not for computer games, for several specific reasons:

1. Copyright is explicitly prohibited from protecting ideas. But new ideas — functionality — is exactly what the law should be protecting and encouraging. Such ideas are the express domain of patent protection.

2. The effect of current copyright law is to give companies a monopoly on functionality that was created, not by them, but by the users of their products. It is users' efforts in learning a random arrangement

of controls that makes the arrangement valuable when it is later incorporated into another program. Once again, patent succeeds where copyright fails: a patent can only protect useful novelty created by the inventor, not random elements that other people's acceptance might someday render useful.

3. The burden of determining which parts of a copyrighted interface are protected is placed entirely on the designer of a new program. This, too, differs notably from patent, where the inventor's claims list identifies specific issues of concern.

4. The copyright protection period of 75 years for a corporate author is unreasonably long. In the fast-changing world of software, it effectively gives the creator an absolute monopoly, as compared to the limited monopoly that both patents and copyrights are intended to confer.

On the balance, then, patent appears to be more appropriate than copyright for the protection of functional interfaces. This is not surprising, since patent law was developed to support the innovation and distribution of functional ideas. However, patents also have their problems, including difficulty and expense of acquisition, difficulty of discovery, and a 17-year protection period that may be too long for the pace of software development. These issues would benefit from thoughtful legislative attention, since their continuing evolution in the courts is discouraging software development. We urge designers to investigate the issues on their own and to support efforts for rational legislation.

Credits and Pointers:

The Communications of the ACM is a good journal to watch for recent news and practitioner-oriented discussions of intellectual property issues related to interfaces. Three recent articles of interest are by Paul Heckel, supporting patents for software; the League of Programming Freedom, arguing against patent restrictions on software; and Pamela Samuelson, reviewing recent copyright decisions:

– Heckel, Paul. "Debunking the Software Patent Myths." Commun. ACM 35:6, (June 1992), pp. 121–140.

– The League for Programming Freedom. "Against software patents." Commun. ACM 35:1, (Jan. 1992), pp. 17–22, 121.
– Samuelson, Pamela. "Updating the copyright look and feel lawsuits." Commun. ACM 35:9, (Sept. 1992), pp. 25–31.

An in-depth discussion of copyright issues, concluding that copyright can evolve into an effective protection for computer software, is presented in:

– Clapes, A.L. "Software, Copyright, and Competition." Quorum Books, New York, 1989.

For a layman's overview of general copyright and trademark law (not as applied to software), see:

– Andorka, Frank H. "A Practical Guide to Copyrights and Trademarks." Pharos Books, New York, 1989.

For those interested in a lawyer's point of view, one place to look is the Software Law Journal. An article summarizing the issues we discuss, current as of the end of 1992, is:

– Gollhofer, R.A. "Copyright protection of computer software: what is it and how did we get it." Software Law Journal 5 (Dec. 1992), pp. 695–713.

M Managing User Interface Development

We've covered the key methods required for developing good user interfaces. We've seen in a general way how these pieces fit into an overall process. But how do you make this process happen in your organization?

In this chapter we'll write as if you are the manager of your development group: if you get to make the decisions, here's how you ought to make them. If you aren't the manager you ought to read this stuff anyway. You may be able to influence the people you work with, or you may be able to recognize that you need to find a job in a better organization.

M.1 Staffing

Of course you want the right people in your group, but what makes people right for user interface development does not seem to be obvious to a lot of managers. We'll start with two don'ts.

Don't get high-powered technical people who think computers and the things they can make them do are the most interesting and important things in life, and who think people are an unfortunate, but temporary, faltering in the march of evolution. Clayton once worked with a user interface designer who said that if he could get the results of user testing of his designs immediately and at no cost he would ignore them, because he *knew* his interface designs were the best possible and if users didn't like them they were wrong. This guy was a great programmer, and popular with management for that reason. But he was hopeless as an interface designer because he was interested in his designs just as pieces of computing, not as things actual people could or could not do their actual work with.

On the other hand, *don't* get psychologists or human factors people who literally or figuratively want to wear white lab coats as symbols of their status as serious behavioral scientists. These folks may seem to be interested in people, but they really aren't: at any rate they're interested in them only as objects of study rather than as living beings trying to get things done. The key symptom to be concerned about: the person refuses to offer a judgement on anything without running a big experiment.

You have to be careful here, because some of the very best user interface designers are in fact psychologists. Psychologists know a lot about how to do user testing, and about how people solve problems, how they learn, and other matters crucial to good interface design. If they are willing to use this knowledge in the service of building useful systems, and stay focussed on that goal, they can be invaluable. But if doing a good study is more

important to them than building a good system, they can't help you much.

Another point of caution: a person who refuses to offer a judgement about anything without visiting some users may seem just as unhelpful as the person hung up on experiments. But in fact this may be just the person you want. There's no virtue in being free and easy with judgements just because a quick judgement lets people get on with the job. You want people who care enough about the success of your system to get the information needed to do things right. As we've stressed here all along, that information centers on users and their work.

More abstractly, the people you want are interested in the richness and detail of human life. They like to know what people do and how they do it, and what problems they encounter. They're more excited about seeing their system help somebody do real work than about the logic of their design. In our opinion this trait is more important than technical skills, whether in computing or psychology, because it's harder to acquire.

M.2 Organization

Traditional organizational structures often segregate people by technical specialty, so that planners, designers, programmers, writers, usability people, and quality control people often find themselves in different groups. We urge you to avoid this setup if you possibly can and aim for an integrated organization in which the people responsible for design, implementation, evaluation, and documentation of your user interface are fully integrated into your development group.

There are three crucial issues here. One is integration of design: you can't separate user interface design from specification of functions and from documentation without loss of quality. The following example gives Clayton's favorite illustration of this point.

Example: The Worst Interface Ever and How It Came About

When Clayton was working for a large computer company he got a call from a planner down south who wanted people to come and see a great new financial analysis product. He said it was better than the spreadsheet (a hot new concept at the time) and people around the company needed to come and see it.

Clayton and a bunch of other interested people showed up for what turned out to be a kind of mass user trial, with a room full of people sitting in pairs at terminals trying and failing to

make the thing work. He and his partner had trouble getting anywhere, and were especially baffled by what seemed to be unrepeatable errors. They would run into trouble, back up, and try to do the very same thing again, only to get different results. The developers were hovering around, looking puzzled and hurt, and Clayton called one over for consultation.

After some discussion the developer explained the problem. This system ran on a type of terminal in which most keys produced input that was buffered in the terminal, but some special keys, including the function keys, communicated immediately with the host computer. The ENTER key was one of these special keys, and the developers had the bright idea of using it as an extra function key. They arranged the system so that when you hit ENTER you got whatever function was associated with the last function key you had pressed. Clayton and his partner were getting baffling results because in fooling around between attempts at their task they hit different function keys and hence set up different bindings for ENTER. The developer was surprised they hadn't figured that out.

"Is there some way we can tell what we'll get if we use ENTER," Clayton asked? "I'm surprised you didn't figure that out either," said the developer. "See that list of function key bindings at the bottom of the screen? The function you'll get is the one that's missing from that list."

Also hovering in the room was a technical writer who came over to join the discussion. "I'm so delighted you're having all these problems," she said. "I keep trying to tell them there's no way in the world I can describe this so it seems sensible, but they won't listen. They say they know there are rough spots but I should just explain them in the manuals."

This is the type specimen of the "peanut butter theory of usability," in which usability is seen as a spread that can be smeared over any design, however dreadful, with good results if the spread is thick enough. If the underlying functionality is confusing, then spread a graphical user interface on it. (In fact, that was exactly the origin of this system: it was an existing financial modelling package to which a new user interface had been fitted.) If the user interface still has some problems, smear some manuals over it. If the manuals are still deficient, smear on some training which you force users to take.

> Of course the theory doesn't work, as this system showed so dramatically. The original design has to consider usability, and the problem of how to explain things to users has to be dealt with up front, not as an afterthought.
>
> The trial session was a great success for the guy who invited Clayton. He knew all along the system was a disaster, but he knew he would need help to kill it. He also knew no-one would come if he told them the truth about it. So he lied, lots of people came from around the company, and the project was quietly shelved.

The second issue is avoiding what we call "the doer-kibitzer split". In one common setup there are usability people in a support group empowered to review designs that the developers come up with, with or without testing them, and suggest usability improvements. Consistently this setup leads to the development of two bad attitudes. The developers come to see the usability people as a drag on their progress, outsiders who just sit and snipe while the developers try nobly to press on with the real work of the organization. The usability people complain that they get no respect, and that the developers are insisting on shipping rubbish that will bankrupt the company.

Some organizations have responded to this by allowing developers to choose freely whether or not to call on the usability people for advice, and whether or not to pay any attention to the advice they get. This takes some of the poison out of the air, though usability people can still feel like fifth wheels. But in our view it doesn't give usability the central focus it needs to have.

Another organizational variation is to entrust not just usability critiquing but all of user interface design to a separate support group. This again takes some of the poison out but it moves user interface design out of the center of power in development and off to one side. The main development group correctly views getting a good user interface as somebody else's job.

We think user interface development should be just as much a core responsibility of the main development group as any other aspect of function, implementation or performance. Members of the development group should be encouraged to respond professionally to that responsibility, rather than to pass it off to somebody else. Just as developers make it a point of professional pride to be knowledgeable about programming languages and tools so they should demand of themselves and their co-workers that they be

knowledgeable about their users and the work they do. Just as they hold themselves to high standards regarding good choices of data representations and algorithms they should set high standards for the fit between their user interface and user needs. All these things should be seen as parts of their professional contribution, all demanding professional knowledge and hard work of them, not of somebody else.

Are we saying everybody has to be a usability specialist? No, no more than everybody in a group has to be an algorithms specialist. In any group there are people who focus more on some aspects of the job and less on others. That will be as true for usability as it is for algorithms or anything else. What we think should be avoided is an organizational structure that puts up a barrier with usability on one side and other issues on the other, and with usability people on one side and everybody else on the other. That makes it too easy for most people in the organization to ignore their own responsibility for usability.

One argument you hear in favor of segregating usability people organizationally is that it supports their professional development. Since many usability people have training in psychology, rather than in computer science, the argument goes, you need to create an environment in which they work with other people with similar background and interests. This will keep them from feeling isolated in a sea of programmers. It may also create a career path for them, since there will need to be more senior usability people, managers of usability groups, and so on. Usability managers will recognize that the usability people are different from programmers and treat them better.

The problems this argument describes are real enough: usability people with poor background in computing do have a hard time making it. But segregation doesn't avoid the problems and in fact can make them worse by creating official second class citizens instead of unofficial ones. Individual usability people can hope to broaden their knowledge and be accepted as regular systems people, only with added value, if they are not trapped in a limited role. Similarly, regular systems people can develop usability skills, if usability isn't made out of bounds for them.

M.3 Resource Allocation

One of your key functions as a manager is to decide how much effort to spend on various aspects of development. This saddles you with one of the toughest problems in user interface development: when should you stop iterating your design?

If you demand a scientific answer to this question you probably won't be a very good manager: you're paid to make lots of decisions like this, without having a good basis for them. When is performance good enough? When is the bug rate low enough? You have to make calls like this on the basis of ideology and instinct, not science (6 sigma quality, a defect rate around 10^{-7}, isn't something companies aspire to because they have data showing it's the economic rate; they aspire to it because it says something about their view of themselves. It's a way they can be the best).

Well, you say, I *am* a good manager, and I still demand a scientific answer on when my user interface is good enough. We're still going to give you a hard time. Do you have scientific answers to those other questions, the ones about performance and bug rate? No? Then why do you insist on one for usability? We think it's because you don't want to be held responsible for usability: you want to pass the buck to some scientific decision process. You don't do this for performance and bug rates because you accept them as part of your territory as a professional. Like it or not, usability is part of your professional territory too and you will only get grief by trying to pretend it's not.

[See also §5.6, pp. 88*ff*]

If you're still interested after that sermon, we'll tell you how to get that scientific answer. Read the HyperTopic on quantitative usability targets.

HyperTopic: Quantitative Usability Targets

Warning! Unrecommended method!

[We'll expand this topic in a future version. For now, the idea is that you set quantitative usability goals for the product: times for test tasks, number of errors, rated test user satisfaction. Then you keep iterating until you get test results showing you have met your objectives. Sounds simple, but problems are many: how to handle statistical uncertainty, how to set the targets to begin with, how to avoid adjusting the targets when you don't live up to your ambitions.]

So if science won't help you, how do you decide about those iterations? Here are the main factors you'll be juggling.

How do you feel about the interface? Are you proud of it? Does it work smoothly on your sample tasks? If you don't feel good about your interface you have to do more work. Users will feel the same way.

Can you afford to work longer? Why not? If your answer to the last question was negative, meaning you think the interface is still crummy, you

may not be able to afford *not* to work longer. If the interface is probably good enough, you may still easily be able to afford more work on it, especially if other aspects of the system are slipping.

[See also note about 'version 0', p. 82]

You'll have an easier time with these questions and the decision that hangs on them if you've done some preparation at the time you planned out your project. First, you should have included in your original schedule at least two iterations, so you don't have to make any tough decisions before the design has a chance to be in reasonable shape. Second, you should have gotten interface design started early, and overlapped it with other development work. If you are lucky this means interface design will not be on the critical path for the project as a whole, meaning that you can take a little longer with it without extending the overall schedule. Third, you should have adopted software tools that minimize the time required to make interface changes.

Here's one more idea for taking some of the stress off the iteration decision. Make a plan that includes a short period, say a week or two, just for user interface improvements, as late in the schedule as you can tolerate (you have to worry about redoing pictures in manuals, for example, so changes can't be literally at the last moment.) Get everybody to agree to spend that time just on polishing up the user interface. If you do this then whenever you decide to stop iterating there'll still be a chance for some final finish work.

HyperTopic: What If Nobody's Willing to Hold Back the Product for Usability Work?

Peter Conklin of Digital, drawing on earlier work from Hewlett Packard, has developed a useful way to increase willingness to invest in product improvements of all kinds, including usability improvements, by getting people to think differently about ship dates and their significance (In M. Rudissill, T. McKay, C. Lewis, and P.G. Polson (Eds.), "Human-Computer Interaction Design: Success Cases, Emerging Methods, and Real-World Context." Morgan Kaufmann. In press.). The idea is to replace emphasis on *time to market* by emphasis on *time to break even*.

Many companies measure projects by how quickly they ship. A project that gets to market sooner is rated better than one that takes longer. Conklin points out that the point of getting to market is to make money, and so a more important target date

is the time the product recovers its development costs and starts to earn a profit: time to break even. Anything that increases the rate of product acceptance, that is, the growth of sales volume, will shorten time to break even, and if the increase in acceptance is big enough, time to break even may be shorter even if time to market is longer. It's smart to take time to produce a better product if the impact on acceptance is big enough.

Nothing in Conklin's approach makes decisions for you, but it does help the tone of group discussions. If you focus on time to market, any effort that delays shipment is bad, period. Somebody holding out for taking more time for usability improvements looks like they're just standing in the way of progress and making the whole project look bad. If I've implemented my routines on time, I'll resent giving the user interface people more time, because I'll be just as late as they are if the product slips. If you focus on time to break even, added development time can be good, if it's important enough (and there aren't overriding timing considerations, like an impending competitive release that could freeze you out). Anybody proposing added development has a clear shot at persuading everybody else in a rational discussion. If the user interface people manage to move up the time to break even, everybody looks good.

Here are a few last management suggestions.

- Use your system yourself. There's no other way to get an adequate feel for where it is. Don't sit and watch demos.
- Know the competition. Use their systems.
- Spend time with users. Get everybody in your group to do some of this. Become an expert in the users' application area, if you aren't one already.
- Be familiar with the concrete tasks that are being used to drive the design.
- Ask the designers what features of the design will ensure that it will work beyond the sample tasks.
- Ask the designers how they are using test results to shape the design.

- Ask the implementers how their approach supports easy modification of the user interface.
- Have a plan for getting feedback from real users.

HyperTopic: I can't get my management to do things right

Lots of usability people have tried to make careers of "educating" managers about the importance of usability, user testing, etc. etc. Don't waste time on this. If you are proposing concrete, practical work and management won't listen, quit and get a job working with people who are smart enough to do what's in their own interest. If you think your organization has a bright future without worrying about usability, and you want to stay, then don't you worry about usability either: get out of usability work.

M.4 Product Updates

No matter how wonderful your product is you'll have to upgrade it over time. There'll be changes in the platform and in user expectations that will affect some of your existing users as well as new users you hope will buy the product. This poses a big dilemma: how do you improve the user interface without turning off your loyal existing users, who have gotten used to the thing the way it is?

Marcy Telles of WordStar, a product which has faced this issue in a big way, argues that updating is harder than creating a new interface, because all the same problems arise with the added constraint of working around the existing interface ("Updating an older interface," Proc. CHI'90 Conference on Human Factors in Computer Systems. New York: ACM, 1990, pp. 243–247.) She recommends trying to work as much as possible by adding options to the existing interface, so that the habits of existing users will still work. In the case of WordStar it was possible to move to modern menu-based interface without disturbing the old keystroke commands very much. Telles also recommends discussing possible changes thoroughly with existing users, so you know what features of the old interface they are really dependent upon.

Exercises

These exercises will give you some experience in the steps that make up task-centered design. Each exercise includes a page limit for the report of your findings. That limit should force you to think about what you've found and report just the most important facts, instead of an endless list of points with no distinction as to importance. (A "1-page" limit is about 500 words.)

Forward

Exercise 0.1: Looking for the Interface

In your home or workplace find a simple machine, something with two to four controls and with no (!) internal computer (a simple toaster or an electric drill are possible candidates). Describe:

- the users the machine seems to be designed for;
- the tasks and subtasks the machine was evidently designed to support;
- the "interface" part of the machine;
- the part of the machine that is *not* the interface.

Limit your answer to one page (it may be less).

Chapter 1. Task-Centered Design.

Exercise 1.1: Task-Centered Design in Other Areas

We present the task-centered design process as an approach to producing better computer interfaces. A similar process could be used for any other field that produces artifacts (i.e., man-made things) for people to use in accomplishing tasks. Some examples are books, buildings, hand tools, and cookware. Where have you used parts of the task-centered design process in your own life or work? (If nowhere, then where could you have used them productively?)

Be specific about which steps in the task-centered design process you did or did not use.

Limit your answer to one page.

Chapter 2. Getting to Know Users and Their Tasks.

Exercise 2.1: Task and User Analysis

This exercise involves working with other people, so you should get started on it right away, so you'll have time to schedule meetings.

Here are two potential software products:

A backpacking checklist builder.
People who only backpack once or twice a year may spend so much time deciding what to take, running from store to store to pick it up, and packing it, that the trip is more work than fun — and something usually gets forgotten anyway. This software would produce simple pre-trip instructions and checklists that would alleviate these problems.

A cooking help system for folks who don't cook often.
Many single people cook only infrequently, so they don't have many ingredients on hand in the kitchen. When they do cook, planning and shopping is as much a part of their effort as following the recipe. This software would help the infrequent cook maintain a kitchen with the ingredients common to many meals, and would help plan meals that could be produced (or nearly so) with what was currently on hand.

Pick just one of these projects. *It should be the one you know the least about!* Find someone who fits the general description of the user (backpacker or infrequent cook) and spend an hour or so with them doing a task analysis. Then think about those results for a while and produce a rough description of the system's functions and interface. Find another (different) potential user and discuss the system you've described.

Focus your work on tasks and users, and don't get hung up on detailed or difficult details. For example, each of the projects could require large amounts of stored and frequently updated data; identify the need to store and update that data, but don't worry about exactly how it will be loaded into the computer.

Write a report describing what you've learned. The report should focus on tasks and users; it should not be a description of the system. It should include at least the following:

- a description of the users you interviewed;
- a description of the general characteristics of the system's users, based on your interviews;

- your understanding of the tasks and subtasks the interface will support;
- how your understanding evolved as the analysis proceeded.

Your answer should take about two pages.

Chapter 3. Creating the Initial Design

Exercise 3.1: Selecting Controls

Imagine that you are working for the software developer of the word processor you use most of the time. After surveying users' needs, the developer has decided to add a new feature: "QuickLists." QuickLists (which might look like the following list) have the following characteristics:

- They are intended to be used in text with minimal effort by users, especially novices.
- Their default indent and spacing is set automatically by the word processing program, which chooses these values to make the list look good with the preceding paragraph.
- The user can specify whether the items of the list are preceded by a special character (such as a bullet) or are sequentially numbered or lettered.

Where should the program incorporate the commands to apply this new feature and set the character that precedes each list item? On a menu? On the keyboard? In dialog boxes? Somewhere else? Give reasons for your answer.

(If your word processor already has a feature like QuickLists, then answer the same question for the following new feature: A company style checker, which checks memos, reports, and letters to make sure they follow the official company guidelines for format, spelling, and style. Assume that guidelines are predefined by the support staff at the user's company, so all the user has to do is specify whether the document being checked is a memo, report, or letter.)

Your answer should take about one page.

Exercise 3.2: Borrowed Colors

Find an interface that uses color. It can be on a computer or somewhere else (a stereo system, car, or appliance, for example). List at least three of the colors, note what they're used on, and explain how that use of color relies on ideas borrowed from other interfaces. For example, you might look at the dashboard of a car and comment: "low-oil warning, red; red is the color for stop lights, and you should stop your engine if the oil is low."

Try to find at least one use of color that is both supported and contradicted by established use, and explain why the designers might have chosen that color.

This assignment should take one page or less.

Exercise 3.3: Unpacking a Metaphor

In the "good old days" (see HyperTopic), many systems were designed using explicit metaphors — that is, new things on the computer were represented by things that users were expected to already know. A well-known example is the Macintosh "desktop" metaphor, which uses icons that look like sheets of paper to represent computer files, icons that look like paper folders to represent computer directories, and an icon that looks like a trash can to represent the delete operation. [HyperTopic, p. 37]

We don't advise you to create new metaphors, but it may be useful to understand existing metaphors and their value when you create new applications in an existing environment. This exercise should get you to think about how metaphors work.

Locate a system or software package that's very different from those you're familiar with. If you are primarily a PC user, you might see if someone will let you use a Macintosh for a while. If you've never used a "paint" or "draw" program, you might borrow one of those, or try one at a computer software store. If you have broad experience with many systems, you might try to locate a public-domain game that you've never played.

Explore the software for a while, using whatever method and resources you find most effective. Then analyze the program in terms of metaphor:

- What is the underlying metaphor of the program, if any?
- Are there any other obvious metaphors that are used within the program?
- Where are some of the points the metaphors fail? That is, what behavior would the metaphors lead you to expect that isn't supported?
- Did the metaphors help you discover how the program works? Did they help you remember the functions of the controls?

Limit your answer to two pages.

Chapter 4. Evaluating the Design Without Users

Exercise 4.1: Cognitive Walkthrough

Perform a cognitive walkthrough for the following situation:

System: The telephone answering machine or messaging system you have at home or at work.

Users: First-time users of the system, without training and without a manual. (If the system isn't on a separate machine, assume the user has the minimal documentation — maybe a template on the touch-tone pad, or a wallet card summarizing key commands.)

Task: Check for messages, find there are some (five), play them, replay the second one, and then delete all of them.

Note that "delete all five messages" may have different meanings depending on the system you are using. For some systems, it may mean positioning the cassette tape so the next messages will overwrite the old ones. Other systems might not require any action — maybe new messages automatically overwrite old ones, or maybe old ones aren't saved unless a special command is issued. In any case, assume your user wants to "delete" the messages because that's what he or she has always done with other answering machines.

What to write up:

- A *brief* description of the system — a sketch of an answering machine would supply most of the details, and you can include more information later in the walkthrough stories.
- The list of correct actions.
- A story for each action. It can be short, but it should touch on all four important points:
  - is the user trying to do the right thing?
  - is the correct action obviously available?
  - will the user connect the correct action to what they are trying to do?
  - after the correct action is performed, will feedback show it was the right thing to do?
- If there were problems, suggest how they could be fixed.

A note on strategy: Like the designer of a system, you are probably intimately familiar with this task on your own phone answering system. Try to use the walkthrough procedure to distance yourself from that familiarity.

This assignment may take two to four pages.

Exercise 4.2: Action Analysis for Knowledge Assessment

One possible outcome of an action analysis is a list of things the user needs to know in order to use an interface. (Notice the "remember" items in the back-of-the-envelope analysis of taking pictures with a 35-mm camera.)

Consider the word processor you are most familiar with. Do a back-of-the-envelope action analysis for entering a document of several paragraphs using that system. Assume a less-than-perfect typist, so corrections will have to be made. Use the analysis to identify the knowledge needed to perform the task. The knowledge may include:

- conceptual understanding of the interface (e.g., for a spreadsheet, the screen represents a grid of cells);
- low level details about how to enter text, select from menus, etc.;
- information about what commands are available, and where;
- expectations as to the interface's response to certain actions.

A complete listing of the knowledge would take more time than you need to spend on this assignment, so organize your answer by major areas of knowledge and give full details only for representative items.

Your answer should provide about one page for the action analysis and another page for the knowledge list.

Exercise 4.3: Heuristic Analysis

Locate an on-line library catalog, either at a university or at your local public library. Use Nielsen and Molich's heuristics to evaluate the interface. [N&M, p. 69]

Get together with three or four other students and combine your analyses into a single list of problems with the interface. Assign priorities to the problems, so the designers of the system would be able to decide which things were most important to change.

Your answer should take a page or two, depending on the quality of the interface.

If you want to do more: Do a quick cognitive walkthrough of the same interface, using what you believe to be a representative task. In a half page or less, compare the kinds of things discovered by heuristic analysis to the kinds of things discovered by the walkthrough. [CogWalk, §4.1 pp. 46*ff*]

Chapter 5. Testing the Design With Users

Exercise 5.1: Thinking Aloud

Ask two friends to be subjects of thinking-aloud studies. Choose a simple piece of software that your subjects have never used (a word processor, spreadsheet, or graphics program would be good). Take an hour or so to work with the program and devise a task that an experienced user could complete in about five minutes. Your novice users will probably require half an hour or more for the same task. Decide on "fallback" procedures — when will you give the subjects help and what will you say if they are seriously stuck.

Follow the thinking-aloud instructions given in Chapter 5, taking special care to emphasize that you are testing the interface, not the subjects' abilities. Review the HyperTopic on "Ethical Concerns" before you begin.

[§5.5, pp. 83*ff*]
[Hypertopic, p. 77]

Record each subject's behavior on video if possible, or on audio tape. Remember that it may sometimes be necessary to remind a subject to think aloud.

Analyze the tapes to discover problems with the interface. Your report should not exceed three pages. For each problem, consider whether it might occur with other users (why or why not?) and how it might be solved. Note any additional problems that your subjects did not have, but that occurred to you during the study. Comment briefly on the differences between the two subjects.

Exercise 5.2: Failures of User Testing

Think of a sophisticated program that you use regularly, a program that comes with a slick manual and has clearly been designed for usability — perhaps your default word processor or spreadsheet program. Identify the interface bug in that program that you find most annoying. Since you are a user and you have this problem, user testing should have identified it — isn't that right? So, why is the problem still there?

A half page should be enough to answer this.

Chapter 6. User Interface Management and Prototyping Systems

Exercise 6.1: Learning About Your System

This is an assignment that you will have to help define on your own: Find out what UIMS/prototyping systems are currently available for "your" system. "Your" system may be the networked computer system you use as a programmer, or the system used by the programmers you work with, or the personal computer you use at home or school.

This isn't an assignment you can do entirely on your own. You should talk to other people who are using the same system. Ask them what they use and how satisfied they are. Look in trade magazines for that system. Check out software stores and ads.

As you investigate what's available, keep in mind that deciding what to use should be a task-centered process. A software package that's good for the task of prototyping small applications may not be adequate for final development of large applications.

Write one to two pages listing the foremost packages with their strengths and weaknesses. Identify the sources of your opinions.

Exercise 6.2: Pushing the Envelope

This is another assignment that you have to define on your own. The general idea is to push your programming abilities beyond where they are today. But it's also important to push them in the right direction.

Here's what you should do. Identify a UIMS or prototyping system that you have never used. Then spend 3–6 hours (we'd suggest two morning or evening sessions) learning the basics of using it. When you're done, you should have created at least one "program."

Here are some examples of systems you might learn:

- If you've never programmed, you might try out HyperCard or some similar "visual programming" system that can be programmed using menus and on-screen graphical objects.
- If you've programmed in traditional languages such as Pascal, C, or BASIC, then you might also try out a visual system. Make it a game to see how far you can push the language without falling back on traditional programming techniques.
- If you're a nonprogrammer who's a regular user of spreadsheets, you might try to learn your spreadsheet's macro programming language.

- If you're familiar with most of the programmable applications on your own system, learn how to use a programming environment in another system — a Mac, or a PC, or even UNIX (budget more than 6 hours for this one!).

Whatever your level, there are two cautionary points:

1. Don't waste your time by trying to get started on this project alone. Ask for help from someone who's more advanced. Just be sure that, when you're done, you can create a program on your own.

2. The goal of this exercise is to give you experience in "smarter" programming, not just more programming. Do something more than demonstrating the skills you already have, no matter how good they are.

Write a page describing what you did.

Chapter 7. The Extended Interface

Exercise 7.1: A Mini-Manual

Choose a single, simple task that can be accomplished with a program you know. Writing a two-line memo with a word processor is an example, or entering a trip report on a spreadsheet, or drawing a party invitation in a paint program. Write the parts of a manual that would support your task, including (1) the detailed task instructions, (2) the command reference, and (3) the super index.

Note that this isn't the complete manual — it's just the excerpt necessary for a specific task. But you should keep the needs of the full manual in mind as you work. The length of the manual will depend on what you need to put into it.

After you've finished — and only then — compare what you've written to the program's manual. Write a page describing the differences and saying which way is best.

Exercise 7.2: Help!

Find a program that has on-line help. Consider the help system from a task-centered point of view. How would you improve the system, if at all? Write no more than a page.

Final Project

This project gives you a chance to use most of the design and evaluation techniques described in the book. It should take several weeks to complete.

F.1: The Design Assignment

Imagine that you have been asked to design the user interface to a "smart" home thermostat. The system will be targeted at middle-income home owners, for both new and existing homes. The marketing strategy will be to emphasize the cost-savings and environmental benefits of presetting the system to keep rooms warm only when they are likely to be in use.

A preliminary market survey has shown that homeowners are receptive to this idea, and a number of competing systems are already on the market. However, the marketing survey has also shown that existing systems are typically too complex and confusing for the homeowner to use effectively. The key to your system's success, therefore, will be its excellent user interface.

Here are the major requirements and constraints of the design:

- The control system will initially be marketed in cold-climate areas, so you should consider only heating functions, not air conditioning.
- Different heating methods (hot air, hot water, steam, electric) may have slightly different requirements. In a real design situation, you would discuss this issue with engineers. For the purposes of this project, act as your own engineering expert and consider only the kind of heating you have in your home.
- An important contribution to heating cost savings has been identified as "zoned" heating. For example, the bedrooms should be kept cool during the day when no one is using them, even if the kitchen or playroom is being heated during that period. Design the system to support three or more zones, and assume the method of heating will also support this.
- Your marketing and production departments have a rough idea of how much hardware you can afford to incorporate into the interface. It's assumed that you will need a simple microprocessor, ROM for program code, and nonvolatile RAM for storing settings. As a rough estimate, assume you have the processing power of a Macintosh or an IBM-PC.
- The physical interface is more limited than the underlying processor. Marketing has determined that homeowners don't want large, complex

controls on their walls. You must incorporate the controls onto a 4-inch by 6-inch control panel. In this space you can put whatever you want: buttons, dials, gauges, lcd screens, touch screens, stylus pads, color, etc. Stay with hand-and-eye interactions; don't propose voice systems.

- Each zone can have its own control panel, or you can combine all controls on a single master panel. (Marketing votes for a single panel, because that's cheaper and easier to install.)
- Each zone has its own temperature sensor. The heating system will be able to maintain a set temperature for each zone. However, changes to temperature won't occur instantly.
- Some of the tasks that marketing thinks the control system should support are:
  - presetting temperatures that the heating system should maintain for zones at various times during various days of the week (for example, keep bedrooms at 50 degrees in the daytime, except Sunday before 10 a.m.);
  - overriding preset settings for a few hours (for someone at home on a single-day holiday);
  - overriding preset settings for a few days (when no one is at home during vacation).

Your task and user analysis should confirm, deny, or expand on these.

F.2: Deliverables

Here's exactly what you should do and what you need to write up.

Sequence: The project is divided into three separate parts, each of which should be completed before the next is begun.

Page limits: Each of the three parts should take about two or three pages. You may want to include sketches of your design that will take up additional space.

Strategy: This interface will be used by nontechnical people, usually without any training. If your interface description covers ten pages, then it probably includes a lot of features that no one will ever discover or figure out. Try to keep the interface as *simple* as possible!

Attention Programmers! This is a *design* task. You should be able to do the entire project without writing a line of code. If you decide to implement a prototype, be sure you can explain why that was a better use of your time than doing further user testing with mockups.

Assignment 1. Task and User Analysis

You will probably consider yourself a potential user of this system, but it's always dangerous to rely on your own impressions. So you should interview at least two possible users, from different households, and write up the following:

1. A description of the users you interviewed.

2. The general characteristics you expect system users to have (age, education, etc.).

3. A description of five scenarios of system usage. For example, "The homeowner gets up at 3 a.m. with a sick child and decides to sit with the child in the living room. The living room is cold, and the homeowner wants to bring it quickly to an above-average temperature." These scenarios will be used in your task-centered design efforts. They should be chosen to cover the most important functionality of the interface.

4. A list of the basic tasks the system will support. This is a "sanity check" to make sure your task scenarios haven't left out anything critical.

Assignment 2. Initial Design and Cognitive Walkthrough

Produce an initial description of the interface and perform cognitive walkthroughs on it using three of the tasks you identified for task-centered design. Write up:

1. a description of your initial interface design.

2. a description of any problems discovered with the walkthroughs.

Assignment 3. Thinking-Aloud Study and Final Design

Modify your design to correct the problems discovered by the cognitive walkthroughs. Then do thinking-aloud studies with two potential users. Ask each user to accomplish one (or more, if you have time) of the tasks defined in the scenarios for task-centered design.

For example, you would first describe the "thinking-aloud" process to your subject, and emphasize that you are testing your interface, not their ability. Then, if you were using the scenario described above, you would present your user with a drawing or cardboard mock-up of your design and say: "Imagine yourself getting up at 3 a.m. with a sick child. You decide to sit with the child in the living room. But the living room is cold, so you want to bring the temperature quickly up to 75 degrees. Show me what you would do using this heat control system, and please let me know what you're thinking as you work through the problem."

Revise your design to avoid any problems discovered in the thinking-aloud studies.

Write up:

1. A brief description of the thinking-aloud studies, with special attention to any problems they discovered.

2. A description of your final interface design.

3. The "design rationale" — that is, your reasons for the important features of your design. The reasons should generally refer to the scenarios you've used in task-centered design.

Index

Page numbers in *slanted text* are definitions.