



It's a good idea to use your programming language as the basis for PDL. You can then generate a code skeleton mixed with narrative text as you develop the design.

A program design language may be a simple transposition of a language such as Ada, C, or Java. Basic PDL syntax should include constructs for component definition, interface description, data declaration, block structuring, condition constructs, repetition constructs, and I/O constructs. It should be noted that PDL can be extended to include keywords for multitasking and/or concurrent processing, interrupt handling, interprocess synchronization, and many other features. The application design for which PDL is to be used should dictate the final form for the design language. The format and semantics for some of these PDL constructs are presented in the example that follows.

To illustrate the use of PDL, we consider a procedural design for the *SafeHome* security function discussed in earlier chapters. The system monitors alarms for fire, smoke, burglar, water, and temperature (e.g., furnace breaks while homeowner is away during winter), produces an alarm bell, and calls a monitoring service, generating a voice-synthesized message. In the PDL that follows, we illustrate some of the important constructs noted in earlier sections.

Recall that PDL is *not* a programming language. The designer can adapt as required without worry of syntax errors. However, the design for the monitoring software would have to be reviewed (do you see any problems?) and further refined before code could be written. The following PDL⁸ provides an elaboration of the procedural design for an early version of an alarm management component.

```

component alarmManagement;
The intent of this component is to manage control panel switches and input from sensors by
type and to act on any alarm condition that is encountered.
  set default values for systemStatus (returned value), all data items
  initialize all system ports and reset all hardware
  check controlPanelSwitches (cps)
    if cps = "test" then invoke alarm set to "on"
    if cps = "alarmOff" then invoke alarm set to "off"
    .
    .
    .
  default for cps = none
  reset all signalValues and switches
  do for all sensors
    invoke checkSensor procedure returning signalValue
    if signalValue > bound [alarmType]
      then phone.message = message [alarmType]
      set alarmBell to "on" for alarmTimeSeconds

```

⁸ The level of detail represented by the PDL is defined locally. Some people prefer a more natural language-oriented description while others prefer something that is close to code.

```

    set system status = "alarmCondition"
  parbegin
    invoke alarm procedure with "on", alarmTimeSeconds;
    invoke phone procedure set to alarmType, phoneNumber
  parend
  else skip
endif
enddofor
end alarmManagement

```

Note that the designer for the alarm management component has used the construct `parbegin . . . parend` that specifies a parallel block. All tasks specified within the `parbegin` block are executed in parallel. In this case, implementation details are not considered.

SOFTWARE TOOLS



Program Design Language

Objective: Although the vast majority of software engineers who use PDL or pseudocode develop a version that is adapted from the programming language that they intend to use for implementation, a number of PDL tools do exist.

Mechanics: In some cases, the tools reverse engineer existing source code (a sad reality in a world where some programs have absolutely no documentation at all). Others allow a designer to create PDL with an automated assist.

Representative Tools⁹

PDL/81, developed by Caine, Farber, and Gordon (<http://www.cfg.com/pdl81/lpd.html>), supports

the creation of designs using a defined version of PDL.

DocGen, distributed by Software Improvement Group (<http://www.software-improvers.com/DocGen.htm>), is a reverse engineering tool that generates PDL-like documentation from Ada and C code.

PowerPDL, developed by Iconix (<http://www.iconixsw.com/SpecSheets/PowerPDL.html>), allows a designer to create PDL based designs and then translates pseudocode into the forms that can generate other design representations.

11.5.4 Comparison of Design Notation

Design notation should lead to a procedural representation that is easy to understand and review. In addition, the notation should enhance “code to” ability so that code does, in fact, become a natural by-product of design. Finally, the design representation must be easily maintainable so that design always represents the program correctly.

A natural question that arises in any discussion of design notation is: What notation is really the best, given the attributes noted above? Any answer to this question is subjective and open to debate. However, it appears that program design language offers the best combination of characteristics. PDL may be embedded directly into source listings, improving documentation and making design maintenance less dif-

⁹ Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

ficult. Editing can be accomplished with any text editor or word-processing system, automatic processors already exist, and the potential for “automatic code generation” is good.

However, it does not follow that other design notation is necessarily inferior to PDL or is “not good” in specific attributes. The pictorial nature of activity diagrams and flowcharts provides a perspective on control flow that many designers prefer. The precise tabular content of decision tables is an excellent tool for table-driven applications. And many other design representations (e.g., Petri nets), not presented in this book, offer their own unique benefits. In the final analysis, the choice of a design tool may be more closely related to human factors than to technical attributes.

11.6 SUMMARY

The component-level design action encompasses a sequence of tasks that slowly reduces the level of abstraction with which software is represented. Component-level design ultimately depicts the software at a level of abstraction that is close to code.

Two different views of component-level design may be taken, depending on the nature of the software to be developed. The object-oriented view focuses on the elaboration of design classes that come from both the problem and infrastructure domain. The conventional view refines three different types of components or modules: control modules, problem domain modules, and infrastructure modules. In both cases, basic design principles and concepts that lead to high-quality software are applied. When considered from a process viewpoint, component-level design draws on reusable software components and design patterns that are pivotal elements of component-based software engineering.

Object-oriented component-level design is class-based. A number of important principles and concepts guide the designer as classes are elaborated. Principles such as the Open-Closed Principle and the Dependency Inversion Principle, and concepts such as coupling and cohesion guide the software engineer in building testable, implementable, and maintainable software components. To conduct component-level design in this context, classes are elaborated by specifying messaging details, identifying appropriate interfaces, elaborating attributes and defining data structures to implement them, describing processing flow within each operation, and representing behavior at a class or component level. In every case, design iteration (refactoring) is an essential activity.

Conventional component-level design requires the representation of data structures, interfaces, and algorithms for a program module in sufficient detail to guide in the generation of programming language source code. To accomplish this, the designer uses one of a number of design notations that represent component-level detail in either graphical, tabular, or text-based formats.

Structured programming is a procedural design philosophy that constrains the number and type of logical constructs used to represent algorithmic detail. The intent

of structured programming is to assist the designer in defining algorithms that are less complex and therefore easier to read, test, and maintain.

- [AMB02] Ambler, S., "UML Component Diagramming Guidelines," available at <http://www.modelingstyle.info/>, 2002.
- [BEN02] Bennett, S., S. McRobb, and R. Farmer, *Object-Oriented Analysis and Design*, 2nd ed., McGraw-Hill, 2002.
- [BOH66] Bohm, C., and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *CACM*, vol. 9, no. 5, May 1966, pp. 366–371.
- [CAI75] Caine, S. and K. Gordon, "PDL—A Tool for Software Design," in *Proc. National Computer Conference*, AFIPS Press, 1975, pp. 271–276.
- [DIJ65] Dijkstra, E., "Programming Considered as a Human Activity," in *Proc. 1965 IFIP Congress*, North-Holland Publishing Co., 1965.
- [DIJ72] Dijkstra, E., "The Humble Programmer," 1972 ACM Turing Award Lecture, *CACM*, vol. 15, no. 10, October, 1972, pp. 859–866.
- [DIJ76] Dijkstra, E., "Structured Programming," in *Software Engineering, Concepts and Techniques*, (J. Buxton et al., eds.), Van Nostrand-Reinhold, 1976.
- [HUR83] Hurley, R. B., *Decision Tables in Software Engineering*, Van Nostrand-Reinhold, 1983.
- [LET01] Lethbridge, T., and R. Laganieri, *Object-Oriented Software Engineering: Practical Software Development Using UML and Java*, McGraw-Hill, 2001.
- [LIS88] Liskov, B., "Data Abstraction and Hierarchy," *SIGPLAN Notices*, vol. 23, no. 5, May 1988.
- [MAR00] Martin, R., "Design Principles and Design Patterns," downloaded from <http://www.objectmentor.com>, 2000.
- [OMG01] *OMG Unified Modeling Specification*, Object Management Group, version 1.4, September, 2001.
- [WAR98] Warmer, J., and A. Klepp, *Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1998.

EXERCISES AND POINTS TO PONDER

- 11.1.** What is a guard-condition, and when is it used?
- 11.2.** Why are control components necessary in conventional software and generally not required in object-oriented software?
- 11.3.** Describe the OCP in your own words. Why is it important to create abstractions that serve as an interface between components?
- 11.4.** Is it reasonable to say that problem domain components should never exhibit external coupling? If you agree, what types of components would exhibit external coupling?
- 11.5.** Describe the DIP in your own words. What might happen if a designer depends too heavily on concretions?
- 11.6.** Select three components that you have developed recently and assess the types of cohesion that each exhibits. If you had to define the primary benefit of high cohesion, what would it be?
- 11.7.** Select three components that you have developed recently and assess the types of coupling that each exhibits. If you had to define the primary benefit of low coupling, what would it be?
- 11.8.** Do some research and develop a list of typical categories for infrastructure components.
- 11.9.** The term *component* is sometimes a difficult one to define. First provide a generic definition, and then provide more explicit definitions for OO and conventional software. Finally, pick three programming languages with which you are familiar and illustrate how each defines a component.

- 11.10.** The terms *public* and *private attributes* are often used in component-level design work. What do you think each means and what design concepts do they try to enforce?
- 11.11.** Select a small portion of an existing program (approximately 50–75 source lines). Isolate the structured programming constructs by drawing boxes around them in the source code. Does the program excerpt have constructs that violate the structured programming philosophy? If so, redesign the code to make it conform to structured programming constructs. If not, what do you notice about the boxes that you've drawn?
- 11.12.** What is a persistent data source?
- 11.13.** Are stepwise refinement and factoring the same thing? If not, how do they differ?
- 11.14.** Develop (1) an elaborated design class; (2) interface descriptions; (3) an activity diagram for one of the operations within the class; (4) a detailed statechart diagram for one of the *Safe-Home* classes that we have discussed in earlier chapters.
- 11.15.** Do a bit of research and describe three or four OCL constructs or operators that have not been discussed in Section 11.4.
- 11.16.** What is the role of interfaces in a class-based component-level design?

Design principles, concepts, guidelines, and techniques for object-oriented design classes and components are discussed in many books on object-oriented software engineering and OO analysis and design. Among the many sources of information are Bennett and his colleagues [BEN02], Larman (*Applying UML and Patterns*, Prentice-Hall, 2001), Lethridge and Laganieri [LET01], and Nicola and her colleagues (*Streamlined Object Modeling: Patterns, Rules and Implementation*, Prentice-Hall, 2001), Schach (*Object-Oriented and Classical Software Engineering*, fifth edition, McGraw-Hill, 2001), Dennis and his colleagues (*Systems Analysis and Design: An Object-Oriented Approach with UML*, Wiley, 2001), Graham (*Object-Oriented Methods: Principles and Practice*, Addison-Wesley, 2000), Richter (*Designing Flexible Object-Oriented Systems with UML*, Macmillan, 1999), Stevens and Pooley (*Using UML: Software Engineering with Objects and Components*, revised edition, Addison-Wesley, 1999), and Riel (*Object-Oriented Design Heuristics*, Addison-Wesley, 1996).

The design by contract concept is a useful design paradigm. Books by Mitchell and McKim (*Design by Contract by Example*, Addison-Wesley, 2001) and Jezequel and his colleagues (*Design Patterns and Contracts*, Addison-Wesley, 1999) cover this topic in some detail. Metsker (*Design Patterns Java Workbook*, Addison-Wesley, 2002) and Shalloway and Trott (*Design Patterns Explained: A New Perspective on Object-Oriented Design*, Addison-Wesley, 2001) consider the impact of patterns on the design of software components. Design iteration is essential for the creation of high-quality designs. Fowler (*Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999) provides useful guidance that can be applied as a design evolves.

The work of Linger, Mills, and Witt (*Structured Programming—Theory and Practice*, Addison-Wesley, 1979) remains a definitive treatment of the subject. The text contains a good PDL as well as detailed discussions of the ramifications of structured programming. Other books that focus on procedural design issues for traditional systems include those by Robertson (*Simple Program Design*, third edition, Course Technology, 2000), Farrell (*A Guide to Programming Logic and Design*, Course Technology, 1999), Bentley (*Programming Pearls*, second edition, Addison-Wesley, 1999), and Dahl (*Structured Programming*, Academic Press, 1997).

Relatively few recent books have been dedicated solely to component-level design. In general, programming language books address procedural design in some detail but always in the context of the language that is introduced by the book. Hundreds of titles are available.

A wide variety of information sources on component-level design are available on the Internet. An up-to-date list of World Wide Web references that are relevant to component-level design can be found at the SEPA Web site:

<http://www.mhhe.com/pressman>.

CHAPTER

12

PERFORMING USER INTERFACE DESIGN

KEY CONCEPTS

- accessibility
- design steps
- golden rules
- help facilities
- interface
 - analysis
 - consistency
 - evaluation
 - models
- internationalization
- object elaboration
- patterns
- task analysis
- usability
- workflow analysis

The blueprint for a house (its architectural design) is not complete without a representation of doors, windows, and utility connections for water, electricity, and telephone (not to mention cable TV). The “doors, windows, and utility connections” for computer software make up the interface design of a system.

Interface design focuses on three areas of concern: (1) the design of interfaces between software components, (2) the design of interfaces between the software and other nonhuman producers and consumers of information (i.e., other external entities), and (3) the design of the interface between a human (i.e., the user) and the computer. In this chapter we focus exclusively on the third interface design category—*user interface design*.

In the preface to his classic book on user interface design, Ben Shneiderman [SHN90] states:

Frustration and anxiety are part of daily life for many users of computerized information systems. They struggle to learn command language or menu selection systems that are supposed to help them do their job. Some people encounter such serious cases of computer shock, terminal terror, or network neurosis that they avoid using computerized systems.

The problems to which Shneiderman alludes are real. It is true that graphical user interfaces, windows, icons, and mouse picks have eliminated many of the most

QUICK LOOK

What is user interface design? It is the process of designing the visual and interactive aspects of a computer program, following a set of design principles that identify the objects and actions and their relationships that form the basis for a user interface. **Who does it?** A software engineer designs the user interface by applying an iterative process that draws on widely accepted design principles. **Why is it important?** If software is difficult to use, if it forces you into mistakes, or if it frustrates your efforts to accomplish your goals, you won't like it, regardless of the computational power it exhibits or the functionality it offers. The inter-

face has to be right because it molds a user's perception of the software.

What are the steps? User interface design begins with the identification of user, task, and environmental requirements. Once user tasks have been identified, user scenarios are created and analyzed to define a set of interface objects and actions. These form the basis for the creation of wireframes that depict graphical design and placement of icons, definition of descriptive text, and the design of user aids such as windows, scroll bars, and mouse and minor menu items. The design is then prototyped and ultimately implemented in the system model, and the result is evaluated for usability.

What is the work product? User scenarios, wireframes, and screen layouts are generated. The prototype is "test driven" by the users and feedback from the test drive is used for the next iteration of the prototype.

horrific interface problems. But even in a "Windows world," we all have encountered user interfaces that are difficult to learn, hard to use, confusing, counterintuitive, unforgiving, and in many cases, totally frustrating. Yet, someone spent time and energy building each of these interfaces, and it is not likely that the builder created these problems purposely.

User interface design has as much to do with the study of people as it does with technology issues. Who is the user? How does the user learn to interact with a new computer-based system? How does the user interpret information produced by the system? What will the user expect of the system? These are only a few of the many questions that must be asked and answered as part of user interface design.

12.1 THE GOLDEN RULES

In his book on interface design, Theo Mandel [MAN97] coins three "golden rules":

1. Place the user in control.
2. Reduce the user's memory load.
3. Make the interface consistent.

These golden rules actually form the basis for a set of user interface design principles that guide this important software design action.

12.1.1 Place the User in Control

During a requirements-gathering session for a major new information system, a key user was asked about the attributes of the windows-oriented graphical interface. "What I really would like," said the user solemnly, "is a system that reads my mind. It knows what I want to do before I need to do it and makes it very easy for me to get it done. That's all, just that."

My first reaction was to shake my head and smile, but I paused for a moment. There was absolutely nothing wrong with the user's request. She wanted a system that reacted to her needs and helped her get things done. She wanted to control the computer, not have the computer control her.

Most interface constraints and restrictions that are imposed by a designer are intended to simplify the mode of interaction. But for whom? In many cases, the designer might introduce constraints and limitations to simplify the implementation of the interface. The result may be an interface that is easy to build, but frustrating to use.

Mandel [MAN97] defines a number of design principles that allow the user to maintain control:

Define interaction modes in a way that does not force a user into unnecessary or undesired actions. An interaction mode is the current state of the interface. For example, if *spell check* is selected in a word-processor menu, the software moves to a spell-checking mode. There is no reason to force the user to remain in spell-checking mode if the user desires to make a small text edit along the way. The user should be able to enter and exit the mode with little or no effort.

Provide for flexible interaction. Because different users have different interaction preferences, choices should be provided. For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, or voice recognition commands. But every action is not amenable to every interaction mechanism. Consider, for example, the difficulty of using keyboard commands (or voice input) to draw a complex shape.

Allow user interaction to be interruptible and undoable. Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else (without losing the work that had been done). The user should also be able to “undo” any action.

Streamline interaction as skill levels advance and allow the interaction to be customized. Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a “macro” mechanism that enables an advanced user to customize the interface to facilitate interaction.

Hide technical internals from the casual user. The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology. In essence, the interface should never require that the user interact at a level that is “inside” the machine (e.g., a user should never be required to type operating system commands from within application software).

Design for direct interaction with objects that appear on the screen. The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing. For example, an application interface that allows a user to “stretch” an object (scale it in size) is an implementation of direct manipulation.

“I had always wished that my computer would be as easy to use as my telephone. My wish has come true. I no longer have to use my telephone.”
Bjarne Stroustrup (originator of C++)

12.1.2 Reduce the User's Memory Load

The more a user has to remember, the more error-prone interaction with the system will be. It is for this reason that a well-designed user interface does not tax the user's memory. Whenever possible, the system should "remember" pertinent information and assist the user with an interaction scenario that assists recall. Mandel [MAN97] defines design principles that enable an interface to reduce the user's memory load:

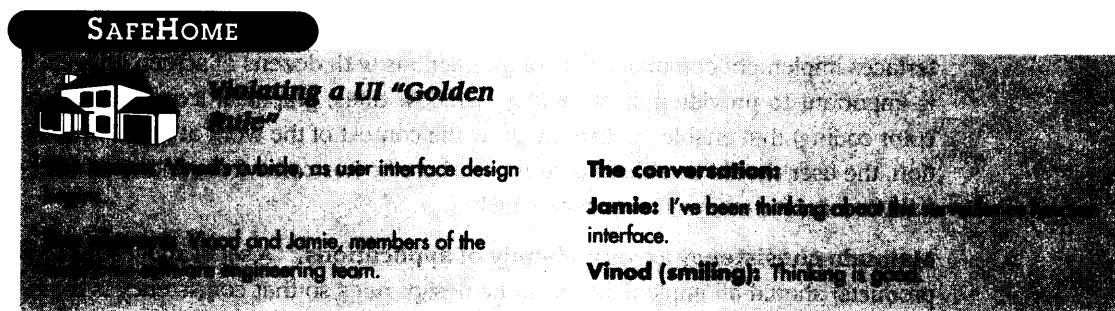
Reduce demand on short-term memory. When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions and results. This can be accomplished by providing visual cues that enable a user to recognize past actions, rather than having to recall them.

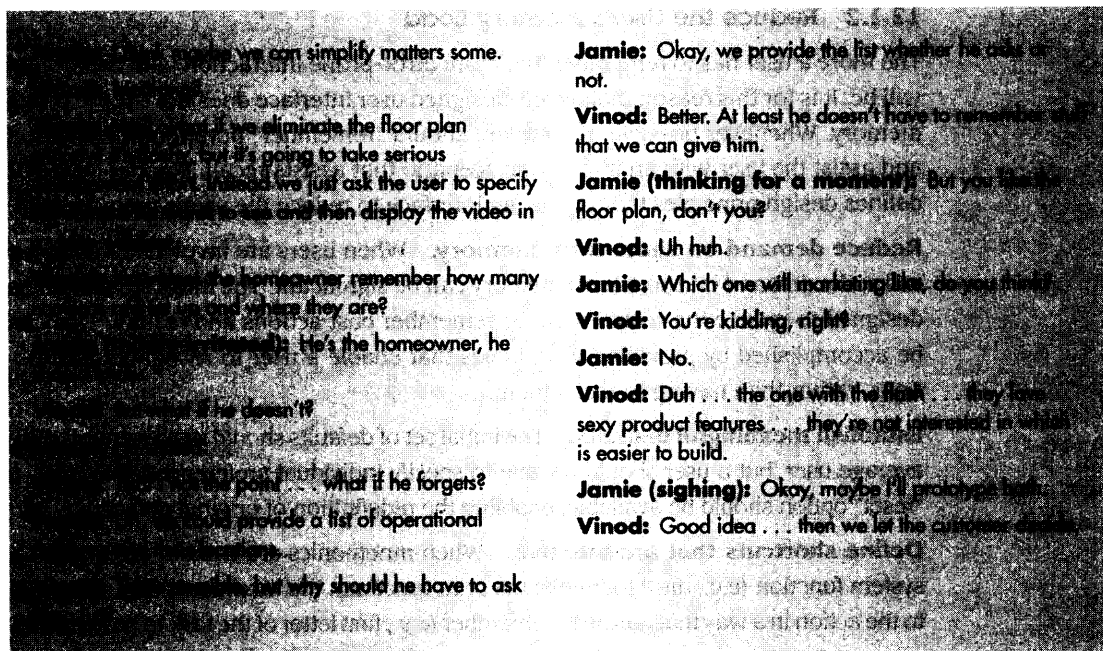
Establish meaningful defaults. The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences. However, a "reset" option should be available, enabling the redefinition of original default values.

Define shortcuts that are intuitive. When mnemonics are used to accomplish a system function (e.g., alt-P to invoke the print function), the mnemonic should be tied to the action in a way that is easy to remember (e.g., first letter of the task to be invoked).

The visual layout of the interface should be based on a real world metaphor. For example, a bill payment system should use a check book and check register metaphor to guide the user through the bill paying process. This enables the user to rely on well-understood visual cues, rather than memorizing an arcane interaction sequence.

Disclose information in a progressive fashion. The interface should be organized hierarchically. That is, information about a task, an object, or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick. An example, common to many word-processing applications, is the underlining function. The function itself is one of a number of functions under a *text style* menu. However, every underlining capability is not listed. The user must pick underlining, and then all underlining options (e.g., single underline, double underline, dashed underline) are presented.





12.1.3 Make the Interface Consistent

The interface should present and acquire information in a consistent fashion. This implies that (1) all visual information is organized according to a design standard that is maintained throughout all screen displays, (2) input mechanisms are constrained to a limited set that is used consistently throughout the application, and (3) mechanisms for navigating from task to task are consistently defined and implemented. Mandel [MAN97] defines a set of design principles that help make the interface consistent:

"Things that look different should act different. Things that look the same should act the same."
 Larry Markus

Allow the user to put the current task into a meaningful context. Many interfaces implement complex layers of interactions with dozens of screen images. It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand. In addition, the user should be able to determine where he has come from and what alternatives exist for a transition to a new task.

Maintain consistency across a family of applications. A set of applications (or products) should all implement the same design rules so that consistency is maintained for all interaction.

If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so. Once a particular interactive sequence has become a de facto standard (e.g., the use of alt-S to save a file), the user expects this in every application she encounters. A change (e.g., using alt-S to invoke scaling) will cause confusion.

The interface design principles discussed in this and the preceding sections provide basic guidance for a software engineer. In the sections that follow, we examine the interface design process itself.



Usability

In an insightful paper on usability, Larry Constantine [CON95] asks a question that has significant bearing on the subject: "What do users want, anyway?" He answers this way: "What users really want are good tools. All software systems, from operating systems and languages to data entry and decision support applications, are just tools. End users want from the tools we engineer for them much the same as we expect from the tools we use. They want systems that are easy to learn and that help them do their work. They want software that doesn't slow them down, that doesn't trick or confuse them, that does make it easier to make mistakes or harder to finish the job."

Constantine argues that usability is not derived from aesthetics, state-of-the-art interaction mechanisms, or built-in interface intelligence. Rather, it occurs when the architecture of the interface fits the needs of the people who will be using it.

A formal definition of usability is somewhat illusive. Donahue and his colleagues [DON99] define it in the following manner: "Usability is a measure of how well a computer system . . . facilitates learning; helps learners remember what they've learned; reduces the likelihood of errors; enables them to be efficient, and makes them satisfied with the system."

The only way to determine whether "usability" exists within a system you are building is to conduct usability assessment or testing. Watch users interact with the system and answer the following questions [CON95]:

- Is the system usable without continual help or instruction?
- Do the rules of interaction help a knowledgeable user to work efficiently?
- Do interaction mechanisms become more flexible as users become more knowledgeable?
- Has the system been tuned to the physical and social environment in which it will be used?
- Is the user aware of the state of the system? Does the user know where she is at all times?
- Is the interface structured in a logical and consistent manner?
- Are interaction mechanisms, icons, and procedures consistent across the interface?
- Does the interaction anticipate errors and help the user correct them?
- Is the interface tolerant of errors that are made?
- Is the interaction simple?

If each of these questions is answered yes, it is likely that usability has been achieved.

Among the many measurable benefits derived from a usable system are [DON99] increased sales and customer satisfaction, competitive advantage, better reviews in the media, better word of mouth, reduced support costs, improved end-user productivity, reduced training costs, reduced documentation costs, reduced likelihood of litigation from unhappy customers.

INFO

12.2 USER INTERFACE ANALYSIS AND DESIGN

The overall process for analyzing and designing a user interface begins with the creation of different models of system function (as perceived from the outside). The human- and computer-oriented tasks that are required to achieve system function

WebRef

An excellent source of UI design information can be found at www.useit.com.

are then delineated; design issues that apply to all interface designs are considered; tools are used to prototype and ultimately implement the design model; and the result is evaluated by end-users for quality.

12.2.1 Interface Analysis and Design Models

Four different models come into play when a user interface is to be analyzed and designed. A human engineer (or the software engineer) establishes a *user model*, the software engineer creates a *design model*, the end-user develops a mental image that is often called the user's *mental model* or the *system perception*, and the implementers of the system create a *implementation model*. Unfortunately, each of these models may differ significantly. The role of interface designer is to reconcile these differences and derive a consistent representation of the interface.



The user model establishes the profile of end-users of the system. To build an effective user interface, "all design should begin with an understanding of the intended users, including profiles of their age, sex, physical abilities, education, cultural or ethnic background, motivation, goals and personality" [SHN90]. In addition, users can be categorized as



Even a novice user wants short-cuts; even knowledgeable, frequent users sometimes need guidance. Give them what they need.

Novices. No syntactic knowledge¹ of the system and little semantic knowledge² of the application or computer usage in general.

Knowledgeable, intermittent users. Reasonable semantic knowledge of the application but relatively low recall of syntactic information necessary to use the interface.

Knowledgeable, frequent users. Good semantic and syntactic knowledge that often leads to the "power-user syndrome," that is, individuals who look for shortcuts and abbreviated modes of interaction.

A design model of the entire system incorporates data, architectural, interface, and procedural representations of the software. The requirements specification may establish certain constraints that help define the user of the system, but the interface design is often only incidental to the design model.³

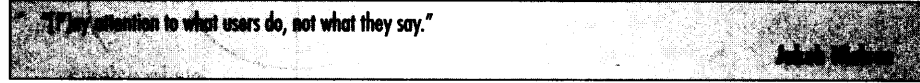
The user's *mental model* (system perception) is the image of the system that end-users carry in their heads. For example, if the user of a particular page layout system

- 1 In this context, *syntactic knowledge* refers to the mechanics of interaction that is required to use the interface effectively.
- 2 *Semantic knowledge* refers to the underlying sense of the application—an understanding of the functions that are performed, the meaning of input and output, and the goals and objectives of the system.
- 3 This is not the way things should be. In many cases, user interface design is as important as architectural and component-level design.

**KEY
POINT**

The user's mental model shapes how the user perceives the interface and whether the UI meets the user's needs.

were asked to describe its operation, the system perception would guide the response. The accuracy of the description will depend upon the user's profile (e.g., novices would provide a sketchy response at best) and overall familiarity with software in the application domain. A user who understands page layout fully but has worked with the specific system only once might actually be able to provide a more complete description of its function than the novice who has spent weeks trying to learn the system.



The implementation model combines the outward manifestation of the computer-based system (the look and feel of the interface), coupled with all supporting information (books, manuals, videotapes, help files) that describe system syntax and semantics. When the implementation model and the user's mental model are coincident, users generally feel comfortable with the software and use it effectively. To accomplish this "melding" of the models, the design model must have been developed to accommodate the information contained in the user model, and the implementation model must accurately reflect syntactic and semantic information about the interface.

The models described in this section are "abstractions of what the user is doing or thinks he is doing or what somebody else thinks he ought to be doing when he uses an interactive system" [MON84]. In essence, these models enable the interface designer to satisfy a key element of the most important principle of user interface design: *Know the user, know the tasks.*

12.2.2 The Process

The analysis and design process for user interfaces is iterative and can be represented using a spiral model similar to the one discussed in Chapter 3. Referring to Figure 12.1, the user interface analysis and design process encompasses four distinct framework activities [MAN97]:

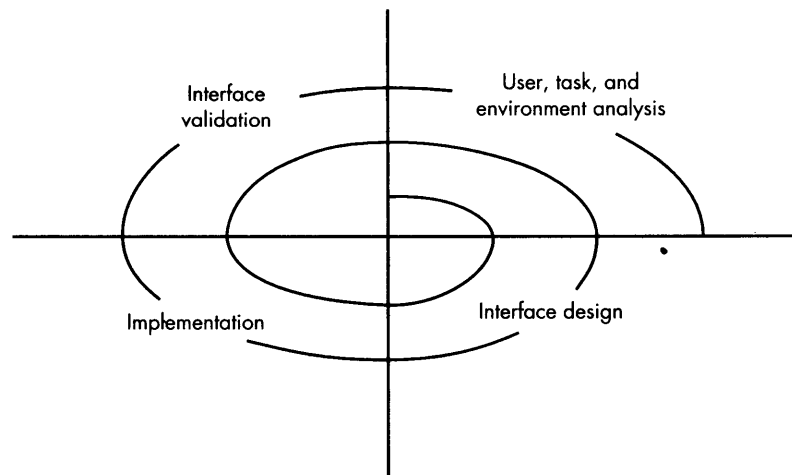
1. User, task, and environment analysis and modeling.
2. Interface design.
3. Interface construction (implementation).
4. Interface validation.

The spiral shown in Figure 12.1 implies that each of these tasks will occur more than once, with each pass around the spiral representing additional elaboration of requirements and the resultant design. In most cases, the construction activity involves prototyping—the only practical way to validate what has been designed.

Interface analysis focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system

FIGURE 12.1

The user interface design process



are recorded; and different user categories are defined. For each user category, requirements are elicited. In essence, the software engineer attempts to understand the system perception (Section 12.2.1) for each class of users.

"It's better to design the user experience than rectify it."

Jon Mosch

Once general requirements have been defined, a more detailed task analysis is conducted. Those tasks that the user performs to accomplish the goals of the system are identified, described, and elaborated (over a number of iterative passes through the spiral). Task analysis is discussed in more detail in Section 12.3.

The analysis of the user environment focuses on the physical work environment. Among the questions to be asked are:

What do we need to know about the environment as we begin UI design?

- Where will the interface be located physically?
- Will the user be sitting, standing, or performing other tasks unrelated to the interface?
- Does the interface hardware accommodate space, light, or noise constraints?
- Are there special human factors considerations driven by environmental factors?

The information gathered as part of the analysis activity is used to create an analysis model for the interface. Using this model as a basis, the design activity commences.

The goal of interface design is to define a set of interface objects and actions (and their screen representations) that enable a user to perform all defined tasks in a

manner that meets every usability goal defined for the system. Interface design is discussed in more detail in Section 12.4.

The construction activity normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, user interface development tools (see sidebar in Section 12.4) may be used to complete the construction of the interface.

Validation focuses on (1) the ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements; (2) the degree to which the interface is easy to use and easy to learn; and (3) the users' acceptance of the interface as a useful tool in their work.


As we have already noted, the activities described in this section occur iteratively. Therefore, there is no need to attempt to specify every detail (for the analysis or design model) on the first pass. Subsequent passes through the process elaborate task detail, design information, and the operational features of the interface.

12.3 INTERFACE ANALYSIS⁴

A key tenet of all software engineering process models is this: *you better understand the problem before you attempt to design a solution*. In the case of user interface design, understanding the problem means understanding (1) the people (end-users) who will interact with the system through the interface; (2) the tasks that end-users must perform to do their work, (3) the content that is presented as part of the interface, and (4) the environment in which these tasks will be conducted. In the sections that follow, we examine each of these elements of interface analysis with the intent of establishing a solid foundation for the design tasks that follow.

12.3.1 User Analysis

Earlier we noted that each user has a mental image or system perception of the software that may be different from the mental image developed by other users. In addition, the user's mental image may be vastly different from the software engineer's design model. The only way that a designer can get the mental image and the design model to converge is to work to understand the users themselves as well as how these people will use the system. Information from a broad array of sources can be used to accomplish this:

 **How do we learn what the user wants from the UI?**

User Interviews. The most direct approach, interviews involve representatives from the software team who meet with end-users to better understand their needs,

⁴ It is reasonable to argue that this section could be placed in Chapter 8, since requirements analysis issues are discussed there. It has been positioned here because interface analysis and design are intimately connected to one another, and the boundary between the two is often fuzzy.

SAFEHOME



Use-Cases for UI Design

The scene: Vinod's cubicle, as user

Design continues...

Who's who: Vinod and Jamie, members of the SafeHome software engineering team.

The conversation:

Jamie: I pinned down our marketing contact and had her write a use-case for the surveillance interface.

Vinod: From who's point of view?

Jamie: The home owner's, who else is there?

Vinod: There's also the system administrator role. Even if it's the homeowner playing the role, it's a different point of view. The "administrator" sets the system up, configures stuff, toys out the floor plan, places the cameras . . .

Jamie: All I had marketing do was play the role of a homeowner who wants to see video.

Vinod: That's okay. It's one of the major behaviors of the surveillance function interface. But we're going to have to examine the system administration behavior as well.

Jamie (irritated): You're right.

Jamie leaves to find the marketing person. She returns a few hours later.

Jamie: I was lucky. I found our marketing contact and we worked through the administrator use-case together.

Basically, we're going to define "administration" as one function that's applicable to all other SafeHome functions. Here's what we came up with.

(Jamie shows the informal use-case to Vinod.)

Informal use-case: I want to be able to set or edit the system layout at any time. When I set up the system, I select an administration function. It asks me whether I want to do a new set-up, or whether I want to edit an existing set-up. If I select a new set-up, the system displays a drawing screen that will enable me to draw the floor plan onto a grid. There will be icons for walls, windows, and doors so that drawing is easy. I just stretch the icons to their appropriate lengths. The system will display the lengths in feet or meters (I can select the measurement system). I can select from a library of sensors and cameras and place them on the floor plan. I get to label each, or the system will do automatic labeling. I can establish settings for sensors and cameras from appropriate menus. If I select edit, I can move sensors or cameras, add new ones or delete existing ones, edit the floor plan, and edit the setting for cameras and sensors. In every case, I expect the system to do consistency checking and to help me avoid mistakes.

Vinod (after reading the scenario): Okay, there are probably some useful design patterns or reusable components for GUIs for drawing programs. I'll betcha 50 bucks we can implement some or most of the administrator interface using them.

Jamie: Agreed. I'll check it out.

Task elaboration. In Chapter 9, we discussed stepwise elaboration (also called *functional decomposition* or *stepwise refinement*) as a mechanism for refining the processing tasks that are required for software to accomplish some desired function. Task analysis for interface design uses an elaborative approach to assist in understanding the human activities the user interface must accommodate.

Task analysis can be applied in two ways. As we have already noted, an interactive, computer-based system is often used to replace a manual or semi-automated activity. To understand the tasks that must be performed to accomplish the goal of the activity, a human engineer⁵ must understand the tasks that humans currently

⁵ In many cases, the tasks described in this section are performed by a software engineer. Ideally, this person has had some training in human engineering and user interface design.



Task elaboration is quite useful, but it can also be dangerous. Just because you have elaborated a task, do not assume that there isn't another way to do it, and that the other way will be tried when the UI is implemented.

perform (when using a manual approach) and then map these into a similar (but not necessarily identical) set of tasks that are implemented in the context of the user interface. Alternatively, the human engineer can study an existing specification for a computer-based solution and derive a set of user tasks that will accommodate the user model, the design model, and the system perception.

Regardless of the overall approach to task analysis, a human engineer must first define and classify tasks. We have already noted that one approach is stepwise elaboration. For example, assume that a small software company wants to build a computer-aided design system explicitly for interior designers. By observing an interior designer at work, the engineer notices that interior design comprises a number of major activities: furniture layout (note the use-case discussed earlier), fabric and material selection, wall and window coverings selection, presentation (to the customer), costing, and shopping. Each of these major tasks can be elaborated into subtasks. For example, using information contained in the use-case, furniture layout can be refined into the following tasks: (1) draw a floor plan based on room dimensions; (2) place windows and doors at appropriate locations; (3a) use furniture templates to draw scaled furniture outlines on floor plan; (3b) use accent templates to draw scaled accents on floor plan. (4) move furniture outlines and accent outlines to get best placement; (5) label all furniture and accent outlines; (6) draw dimensions to show location; (7) draw perspective rendering view for customer. A similar approach could be used for each of the other major tasks.

Subtasks 1–7 can each be refined further. Subtasks 1–6 will be performed by manipulating information and performing actions within the user interface. On the other hand, subtask 7 can be performed automatically in software and will result in little direct user interaction.⁶ The design model of the interface should accommodate each of these tasks in a way that is consistent with the user model (the profile of a “typical” interior designer) and system perception (what the interior designer expects from an automated system).

Object elaboration. Rather than focusing on the tasks that a user must perform, the software engineer examines the use-case and other information obtained from the user and extracts the physical objects that are used by the interior designer. These objects can be categorized into classes. Attributes of each class are defined, and an evaluation of the actions applied to each object provide the designer with a list of operations. For example, the furniture template might translate into a class called **Furniture** with attributes that might include **size, shape, location** and others. The interior designer would *select* the object from the **Furniture** class, *move* it to a position on the floor plan (another object in this context), *draw* the furniture outline, and so forth. The tasks *select, move, and draw* are operations. The user interface analysis



Although object elaboration is useful, it should not be used as a standalone approach. The user's voice must be considered during task analysis.

⁶ However, this may not be the case. The interior designer might want to specify the perspective to be drawn, the scaling, the use of color and other information. The use-case related to drawing perspective renderings would provide the information we need to address this task.

model would not provide a literal implementation for each of these operations. However, as the design is elaborated, the details of each operation are defined.

Workflow analysis. When a number of different users, each playing different roles, makes use of a user interface, it is sometimes necessary to go beyond task analysis and object elaboration and apply *workflow analysis*. This technique allows a software engineer to understand how a work process is completed when several people (and roles) are involved. Consider a company that intends to fully automate the process of prescribing and delivering prescription drugs. The entire process⁷ will revolve around a Web-based application that is accessible by physicians (or their assistants), pharmacists, and patients. Workflow can be represented effectively with a UML swimlane diagram (a variation on the activity diagram).

We consider only a small part of the work process: the situation that occurs when a patient asks for a refill. Figure 12.2 presents a swimlane diagram that indicates the tasks and decisions for each of the three roles noted above. This information may have been elicited via interview or from use-cases written by each actor. Regardless, the flow of events (shown in the figure) enable the interface designer to recognize three key interface characteristics:

1. Each user implements different tasks via the interface; therefore, the look and feel of the interface designed for the patient will be different from the one defined for pharmacists or physicians.
2. The interface design for pharmacists and physicians must accommodate access to and display of information from secondary information sources (e.g., access to inventory for the pharmacist and access to information about alternative medications for the physician).
3. Many of the activities noted in the swimlane diagram can be further elaborated using task analysis and/or object elaboration (e.g., *fills prescription* could imply a mail-order delivery, a visit to a pharmacy, or a visit to a special drug distribution center).

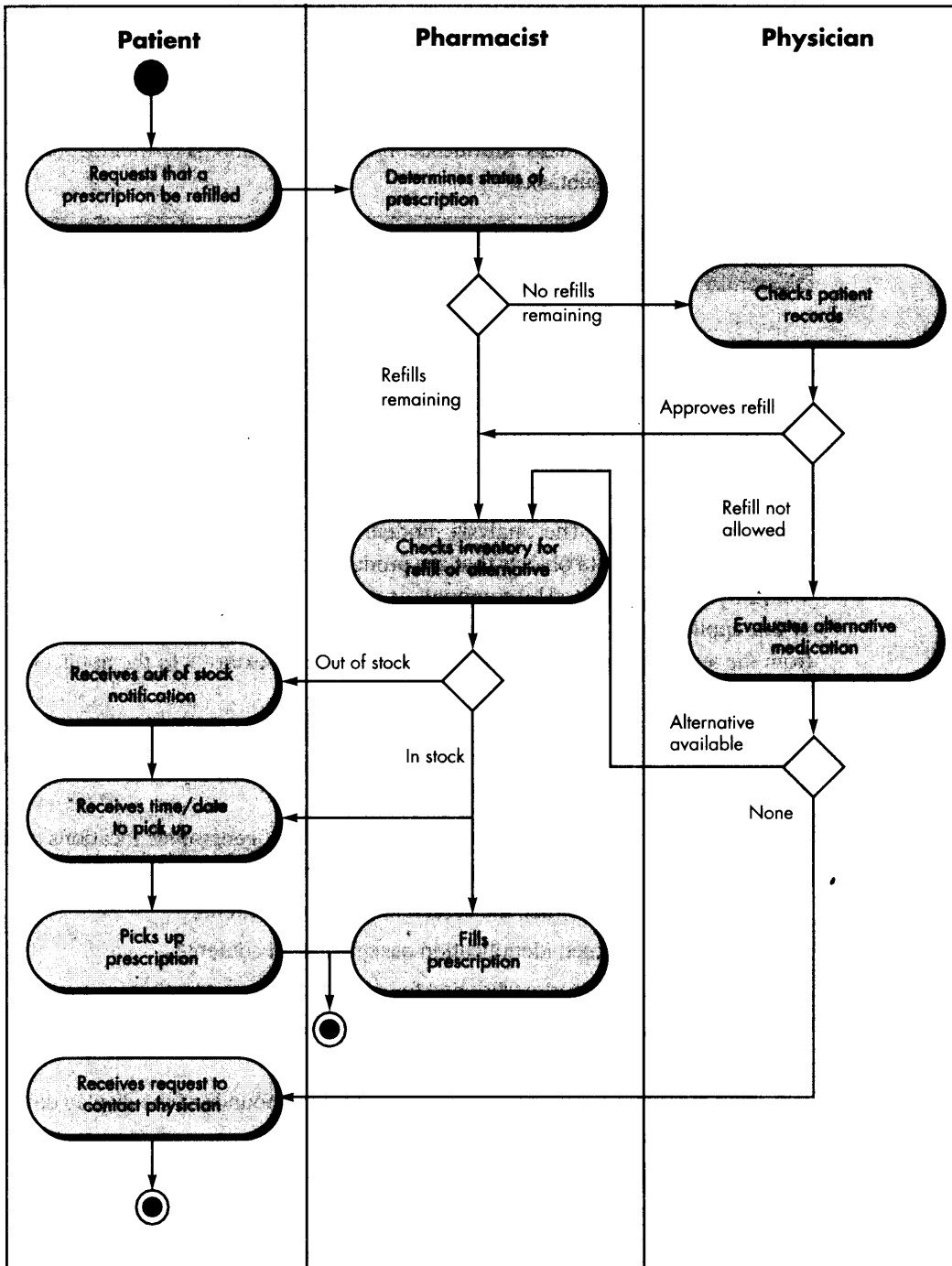
Hierarchical representation. As the interface is analyzed, a process of elaboration occurs. Once workflow has been established, a task hierarchy can be defined for each user type. The hierarchy is derived by a stepwise elaboration of each task identified for the user. For example, consider the user task *requests that a prescription be refilled*. The following task hierarchy is developed:

Request that a prescription be refilled

- *Provide identifying information*
 - *Specify name*

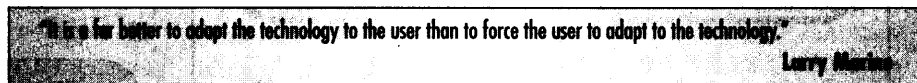
⁷ This example has been adapted from [HAC98].

FIGURE 12.2 Swimlane diagram for prescription refill function



- *Specify userid*
- *Specify PIN and password*
- *Specify prescription number*
- *Specify date refill is required*

To complete the *request that a prescription be refilled* tasks, three subtasks are defined. One of these subtasks, *provide identifying information*, is further elaborated in three additional sub-subtasks.



12.3.3 Analysis of Display Content

The user tasks identified in the preceding section lead to the presentation of a variety of different types of content. For modern applications, display content can range from character-based reports (e.g., a spreadsheet), graphical displays (e.g., a histogram, a 3-D model, a picture of a person), or specialized information (e.g., audio or video files). The analysis modeling techniques discussed in Chapter 8 identify the output data objects that are produced by an application. These data objects may be (1) generated by components (unrelated to the interface) in other parts of the application; (2) acquired from data stored in a database that is accessible from the application; or (3) transmitted from systems external to the application in question.

During this interface analysis step, the format and aesthetics of the content (as it is displayed by the interface) are considered. Among the questions that are asked and answered are:

How do we determine the format and aesthetics of content displayed as part of the UI?

- Are different types of data assigned to consistent geographic locations on the screen (e.g., photos always appear in the upper right hand corner)?
- Can the user customize the screen location for content?
- Is proper on-screen identification assigned to all content?
- How is a large report partitioned for ease of understanding?
- Will mechanisms be available for moving directly to summary information for large collections of data.
- Will graphical output be scaled to fit within the bounds of the display device that is used?
- How will color be used to enhance understanding?
- How will error messages and warnings be presented to the user?

As each of these (and other) questions are answered, the requirements for content presentation are established.

12.3.4 Analysis of the Work Environment

Hackos and Redish [HAC98] discuss the importance of work environment analysis when they state:

People do not perform their work in isolation. They are influenced by the activity around them, the physical characteristics of the workplace, the type of equipment they are using, and the work relationships they have with other people. If the products you design do not fit into the environment, they may be difficult or frustrating to use.

In some applications the user interface for a computer-based system is placed in a “user-friendly location” (e.g., proper lighting, good display height, easy keyboard access), but in others (e.g., a factory floor or an airplane cockpit) lighting may be sub-optimal, noise may be a factor, a keyboard or mouse may not be an option, display placement may be less than ideal. The interface designer may be constrained by factors that mitigate against ease of use.

In addition to physical environmental factors, the work place culture also comes into play. Will system interaction be measured in some manner (e.g., time per transaction or accuracy of a transaction)? Will two or more people have to share information before an input can be provided? How will support be provided to users of the system? These and many related questions should be answered before the interface design commences.

12.4 INTERFACE DESIGN STEPS

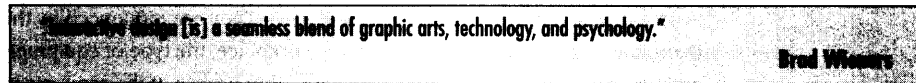
Once interface analysis has been completed, all tasks (or objects and actions) required by the end-user have been identified in detail, and the interface design activity commences. Interface design, like all software engineering design, is an iterative process. Each user interface design step occurs a number of times, each elaborating and refining information developed in the preceding step.

Although many different user interface design models (e.g., [NOR86], [NIE00]) have been proposed, all suggest some combination of the following steps:

1. Using information developed during interface analysis (Section 12.3), define interface objects and actions (operations).
2. Define events (user actions) that will cause the state of the user interface to change. Model this behavior.
3. Depict each interface state as it will actually look to the end-user.
4. Indicate how the user interprets the state of the system from information provided through the interface.

In some cases, the interface designer may begin with sketches of each interface state (i.e., what the user interface looks like under various circumstances) and then work backward to define objects, actions, and other important design information. Regardless of the sequence of design tasks, the designer must (1) always follow the

golden rules discussed in Section 12.1, (2) model how the interface will be implemented, and (3) consider the environment (e.g., display technology, operating system, development tools) that will be used.



12.4.1 Applying Interface Design Steps

An important step in interface design is the definition of interface objects and the actions that are applied to them. To accomplish this, use-cases are parsed in much the same way as described in Chapter 8. That is, a description of a use-case is written. Nouns (objects) and verbs (actions) are isolated to create a list of objects and actions.

Once the objects and actions have been defined and elaborated iteratively, they are categorized by type. Target, source, and application objects are identified. A *source object* (e.g., a report icon) is dragged and dropped onto a *target object* (e.g., a printer icon). The implication of this action is to create a hard-copy report. An *application object* represents application-specific data that are not directly manipulated as part of screen interaction. For example, a mailing list is used to store names for a mailing. The list itself might be sorted, merged, or purged (menu-based actions), but it is not dragged and dropped via user interaction.

When the designer is satisfied that all important objects and actions have been defined (for one design iteration), screen layout is performed. Like other interface design activities, *screen layout* is an interactive process in which graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and definition of major and minor menu items is conducted. If a real world metaphor is appropriate for the application, it is specified at this time, and the layout is organized in a manner that complements the metaphor.

To provide a brief illustration of the design steps noted previously, we consider a user scenario for the *SafeHome* system (discussed in earlier chapters). A preliminary use-case (written by the homeowner) for the interface follows:

Preliminary use-case: I want to gain access to my *SafeHome* system from any remote location via the Internet. Using browser software operating on my notebook computer (while I'm at work or traveling), I can determine the status of the alarm system; arm or disarm the system; reconfigure security zones; and view different rooms within the house via preinstalled video cameras.

To access *SafeHome* from a remote location, I provide an identifier and a password. These define levels of access (e.g., all users may not be able to reconfigure the system) and provide security. Once validated, I can check the status of the system and change status by arming or disarming *SafeHome*. I can reconfigure the system by displaying a floor plan of the house, viewing each of the security sensors, displaying each currently configured zone, and modifying zones as required. I can view the interior of the house via strategically placed video cameras. I can pan and zoom each camera to provide different views of the interior.

Based on this use-case, the following homeowner tasks, objects, and data items are identified:

- *accesses* the *SafeHome* system
- *enters* an **ID** and **password** to allow remote access
- *checks* **system status**
- *arms* or *disarms* *SafeHome* system
- *displays* **floor plan** and **sensor locations**
- *displays* **zones** on floor plan
- *changes* **zones** on floor plan
- *displays* **video camera locations** on **floor plan**
- *selects* **video camera** for viewing
- *views* **video images**
- *pans* or *zooms* the **video camera**

Objects (boldface) and actions (italics) are extracted from this list of homeowner tasks. The majority of objects noted are application objects. However, **video camera location** (a source object) is dragged and dropped onto **video camera** (a target object) to create a **video image** (a window that contains the video display).

A preliminary sketch of the screen layout for video monitoring is created (Figure 12.3).⁸ To invoke the video image, a video camera location icon, C, located in the floor plan displayed in the monitoring window, is selected. In this case, a camera location in the living room, LR, is then dragged and dropped onto the video camera icon in the upper left-hand portion of the screen. The video image window appears, displaying streaming video from the camera located in the living room (LR). The zoom and pan control slides are used to control the magnification and direction of the video image. To select a view from another camera, the user simply drags and drops a different camera location icon into the camera icon in the upper left-hand corner of the screen.

The layout sketch shown would have to be supplemented with an expansion of each menu item within the menu bar, indicating what actions are available for the video monitoring mode (state). A complete set of sketches for each homeowner task noted in the user scenario would be created during the interface design.

12.4.2 User Interface Design Patterns

Sophisticated graphical user interfaces have become so common that a wide variety of user interface design patterns has emerged. As we noted earlier in this book, a



Although automated tools can be useful in developing layout prototypes, sometimes a pencil and paper are all that are needed.

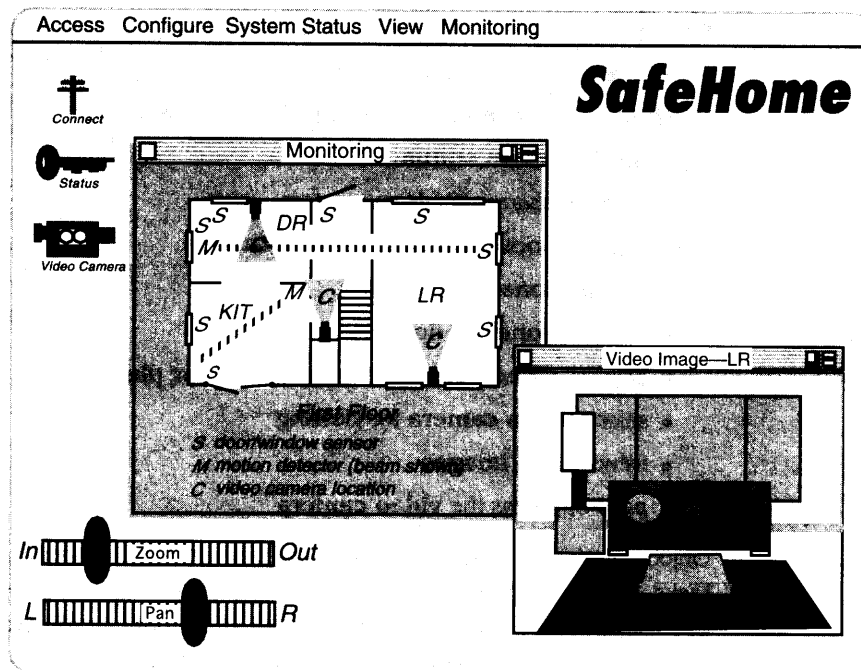
WebRef

A wide variety of UI design patterns have been proposed. For pointers to a variety of pattern sites, visit www.lispatterns.org.

⁸ Note that this differs somewhat from the implementation of these features in earlier chapters. This might be considered a first draft design and represents one alternative that might be considered.

FIGURE 12.3

Preliminary screen layout



design pattern is an abstraction that prescribes a design solution to a specific, well-bounded design problem. Each of the example patterns (and all patterns within each category) presented in the sidebar would also have a complete component-level design, including design classes, attributes, operations, and interfaces.



User Interface Patterns

Hundreds of UI patterns have been proposed over the past decade. Tidwell [TID02] and vanWelie [WEL01] provide taxonomies⁹ of user interface design patterns that can be organized into 10 categories. Example patterns within each of these categories are presented in this sidebar.

Whole UI. Provides design guidance for top-level structure and navigation.

Pattern: *top-level navigation*

Brief description: Provides a top-level menu, often coupled with a logo or identifying graphic, that

enables direct navigation to any of the system's major functions.

Page layout. Addresses the general organization of pages (for Web sites) or distinct screen displays (for interactive applications).

Pattern: *card stack*

Brief description: Provides the appearance of a stack of tabbed cards, each selectable with a mouse click and each representing specific subfunctions or content categories.

INFO

⁹ Full patterns descriptions (along with dozens of other patterns) can be found at [TID02] and [WEL01].

Forms and input. Considers a variety of design techniques for completing form-level input.

Pattern: *fill-in-the-blanks*

Brief description: Allow alphanumeric data to be entered in a "text box."

Tables. Provide design guidance for creating and manipulating tabular data of all kinds.

Pattern: *sortable table*

Brief description: Displays a long list of records that can be sorted by selecting a toggle mechanism for any column label.

Direct data manipulation. Addresses data editing, modification, and transformation.

Pattern: *bread crumbs*

Brief description: Provides a full navigation path when the user is working with a complex hierarchy of pages or display screens.

Navigation. Assists the user in navigating through hierarchical menus, Web pages, and interactive display screens.

Pattern: *edit-in-place*

Brief description: Provides simple text editing capability for certain types of content in the location that it is displayed.

Searching. Enables content-specific searches through information maintained within a Web site or contained by

persistent data stores that are accessible via an interactive application.

Pattern: *simple search*

Brief description: Provides the ability to search a Web site or persistent data source for a simple data item described by an alphanumeric string.

Page elements. Implement specific elements of a Web page or display screen.

Pattern: *wizard*

Brief description: Takes the user through a complex task one step at a time, providing guidance for the completion of the task through a series of simple window displays.

E-commerce. Specific to Web sites, these patterns implement recurring elements of e-commerce applications.

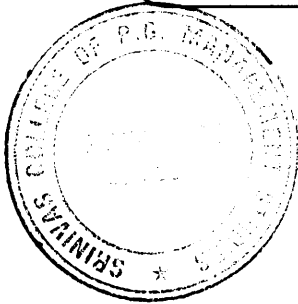
Pattern: *shopping cart*

Brief description: Provides a list of items selected for purchase.

Miscellaneous. Patterns that do not easily fit into one of the preceding categories. In some cases, these patterns are domain dependent or occur only for specific classes of users.

Pattern: *progress indicator*

Brief description: Provides an indication of progress when an operation is under way.



A comprehensive discussion of user interface patterns is beyond the scope of this book. The interested reader should see [DUY02], [BOR01], [WEL01], and [TID02] for further information.

12.4.3 Design Issues

As the design of a user interface evolves, four common design issues almost always surface: system response time, user help facilities, error information handling, and command labeling. Unfortunately, many designers do not address these issues until relatively late in the design process (sometimes the first inkling of a problem doesn't occur until an operational prototype is available). Unnecessary iteration, project delays, and customer frustration often result. It is far better to establish each as a design issue to be considered at the beginning of software design, when changes are easy and costs are low.

"A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools."

Douglas Adams

Response time. System response time is the primary complaint for many interactive applications. In general, system response time is measured from the point at which the user performs some control action (e.g., hits the return key or clicks a mouse) until the software responds with the desired output or action.

System response time has two important characteristics: length and variability. If system response is too long, user frustration and stress is the inevitable result. *Variability* refers to the deviation from average response time, and, in many ways, it is the most important response time characteristic. Low variability enables the user to establish an interaction rhythm, even if response time is relatively long. For example, a 1-second response to a command will often be preferable to a response that varies from 0.1 to 2.5 seconds. When variability is significant, the user is always off balance, always wondering whether something “different” has occurred behind the scenes.

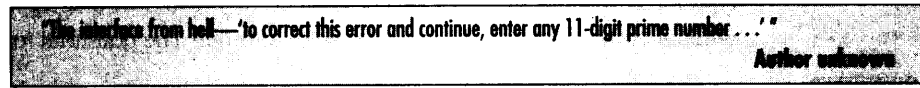
Help facilities. Almost every user of an interactive, computer-based system requires help now and then. In some cases, a simple question addressed to a knowledgeable colleague can do the trick. In others, detailed research in a multivolume set of “user manuals” may be the only option. In most cases, however, modern software provides on-line help facilities that enable a user to get a question answered or solve a problem without leaving the interface.

A number of design issues [RUB88] must be addressed when a help facility is considered:

- Will help be available for all system functions and at all times during system interaction? Options include help for only a subset of all functions and actions or help for all functions.
- How will the user request help? Options include a help menu, a special function key, or a HELP command.
- How will help be represented? Options include a separate window, a reference to a printed document (less than ideal), or a one- or two-line suggestion produced in a fixed screen location.
- How will the user return to normal interaction? Options include a return button displayed on the screen, a function key, or control sequence.
- How will help information be structured? Options include a “flat” structure in which all information is accessed through a keyword, a layered hierarchy of information that provides increasing detail as the user proceeds into the structure, or the use of hypertext.

Error handling. Error messages and warnings are “bad news” delivered to users of interactive systems when something has gone awry. At their worst, error messages and warnings impart useless or misleading information and serve only to increase user frustration: There are few computer users who have not encountered an

error of the form: "Application XXX has been forced to quit because an error of type 1023 has been encountered." Somewhere, an explanation for error 1023 must exist; otherwise, why would the designers have added the identification? Yet, the error message provides no real indication of what went wrong or where to look to get additional information. An error message presented in this manner does nothing to assuage user anxiety or to help correct the problem.



In general, every error message or warning produced by an interactive system should have the following characteristics:

What characteristics should a "good" error message have?

- The message should describe the problem in language the user can understand.
- The message should provide constructive advice for recovering from the error.
- The message should indicate any negative consequences of the error (e.g., potentially corrupted data files) so that the user can check to ensure that they have not occurred (or correct them if they have).
- The message should be accompanied by an audible or visual cue. That is, a beep might be generated to accompany the display of the message, or the message might flash momentarily or be displayed in a color that is easily recognizable as the "error color."
- The message should be nonjudgmental. That is, the wording should never place blame on the user.

Because no one really likes bad news, few users will like an error message no matter how well designed. But an effective error message philosophy can do much to improve the quality of an interactive system and will significantly reduce user frustration when problems do occur.

Menu and command labeling. The typed command was once the most common mode of interaction between users and system software and was commonly used for applications of every type. Today, the use of window-oriented, point and pick interfaces has reduced reliance on typed commands, but many power-users continue to prefer a command-oriented mode of interaction. A number of design issues arise when typed commands or menu labels are provided as a mode of interaction:

- Will every menu option have a corresponding command?
- What form will commands take? Options include a control sequence (e.g., alt-P), function keys, or a typed word.
- How difficult will it be to learn and remember the commands? What can be done if a command is forgotten?

- Can commands be customized or abbreviated by the user?
- Are menu labels self-explanatory within the context of the interface?
- Are submenus consistent with the function implied by a master menu item?

As we noted earlier in this chapter, conventions for command usage should be established across all applications. It is confusing and often error-prone for a user to type alt-D when a graphics object is to be duplicated in one application and alt-D when a graphics object is to be deleted in another. The potential for error is obvious.

WebRef

Guidelines for developing accessible software can be found at www-3.ibm.com/able/guidelines/software/accesssoftware.html.

Application accessibility. As computing applications become ubiquitous, software engineers must ensure that interface design encompasses mechanisms that enable easy access for those with special needs. *Accessibility* for users (and software engineers) who may be physically challenged is an imperative for moral, legal, and business reasons. A variety of accessibility guidelines (e.g., [W3C03])—many designed for Web applications but often applicable to all types of software—provide detailed suggestions for designing interfaces that achieve varying levels of accessibility. Others (e.g., [APP03], [MIC03]) provide specific guidelines for “assistive technology” that addresses the needs of those with visual, hearing, mobility, speech, and learning impairments.

Internationalization. Software engineers and their managers invariably underestimate the effort and skills required to create user interfaces that accommodate the needs of different locales and languages. Too often, interfaces are designed for one locale and language and then jury-rigged to work in other countries. The challenge for interface designers is to create “globalized” software. That is, user interfaces should be designed to accommodate a generic core of functionality that can be delivered to all who use the software. Localization features enable the interface to be customized for a specific market.

A variety of internationalization guidelines (e.g., [IBM03]) are available to software engineers. These guidelines address broad design issues (e.g., screen layouts may differ in various markets) and discrete implementation issues (e.g., different alphabets may create specialized labeling and spacing requirements). The *Unicode* standard [UNI03] has been developed to address the daunting challenge of managing dozens of natural languages with hundred of characters and symbols.

SOFTWARE TOOLS



User Interface Development

Objective: These tools enable a software engineer to create a sophisticated GUI with relatively little custom software development. The tools provide access to reusable components and make the creation of an interface a matter of selecting from predefined capabilities that are assembled using the tool.

Mechanics: Modern user interfaces are constructed with a set of reusable components that are coupled with some custom components developed to provide specialized features. Most user interface development tools enable a software engineer to create an interface using “drag and drop” capability. That is, the developer selects from many

predefined capabilities (e.g., forms builders, interaction mechanisms, command processing capability) and places these capabilities within the content of the interface to be created.

Representative Tools¹⁰

Macromedia Authorware, developed by macromedia Inc. (www.macromedia.com/software/), has been designed for the creation of e-learning interfaces and environments. Makes use of sophisticated construction capabilities.

Motif Common Desktop Environment, developed by The Open Group (www.osf.org/tech/desktop/cde/), is an integrated graphical user interface for open systems desktop computing. It delivers a single, standard graphical interface for the management of data, files, and applications.

PowerDesigner/PowerBuilder, developed by Sybase (www.sybase.com/products/internetappdevtools), is a comprehensive set of CASE tools that include many capabilities for designing and building GUIs.

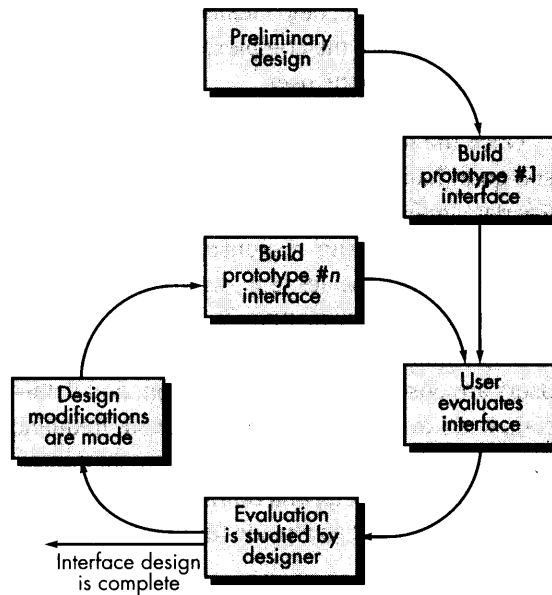
12.5 DESIGN EVALUATION

Once an operational user interface prototype has been created, it must be evaluated to determine whether it meets the needs of the user. Evaluation can span a formality spectrum that ranges from an informal "test drive," in which a user provides impromptu feedback to a formally designed study that uses statistical methods for the evaluation of questionnaires completed by a population of end-users.

The user interface evaluation cycle takes the form shown in Figure 12.4. After the design model has been completed, a first-level prototype is created. The prototype is

FIGURE 12.4

The interface design evaluation cycle



¹⁰ Tools noted here do not represent an endorsement, but rather a sampling of tools in this category. In most cases, tool names are trademarked by their respective developers.

evaluated by the user,¹¹ who provides the designer with direct comments about the efficacy of the interface. In addition, if formal evaluation techniques are used (e.g., questionnaires, rating sheets), the designer may extract information from these data (e.g., 80 percent of all users did not like the mechanism for saving data files). Design modifications are made based on user input, and the next level prototype is created. The evaluation cycle continues until no further modifications to the interface design are necessary.

The prototyping approach is effective, but is it possible to evaluate the quality of a user interface before a prototype is built? If potential problems can be uncovered and corrected early, the number of loops through the evaluation cycle will be reduced and development time will shorten. If a design model of the interface has been created, a number of evaluation criteria [MOR81] can be applied during early design reviews:

1. The length and complexity of the written specification of the system and its interface provide an indication of the amount of learning required by users of the system.
2. The number of user tasks specified and the average number of actions per task provide an indication of interaction time and the overall efficiency of the system.
3. The number of actions, tasks, and system states indicated by the design model imply the memory load on users of the system.
4. Interface style, help facilities, and error handling protocol provide a general indication of the complexity of the interface and the degree to which it will be accepted by the user.

Once the first prototype is built, the designer can collect a variety of qualitative and quantitative data that will assist in evaluating the interface. To collect qualitative data, questionnaires can be distributed to users of the prototype. Questions can be (1) simple yes/no response, (2) numeric response, (3) scaled (subjective) response, (4) Likert scales (e.g., strongly agree, somewhat agree), (5) percentage (subjective) response, or (6) open-ended.

If quantitative data are desired, a form of time study analysis can be conducted. Users are observed during interaction, and data—such as number of tasks correctly completed over a standard time period, frequency of actions, sequence of actions, time spent “looking” at the display, number and types of errors, error recovery time, time spent using help, and number of help references per standard time period—are collected and used as a guide for interface modification.

¹¹ It is important to note that experts in ergonomics and interface design may also conduct reviews of the interface. These reviews are called *heuristic evaluations* or *cognitive walkthroughs*.