QUICK LOOK

What is the work product? User scenarios are created and screen layouts are generated. An interface prototype is developed and modified in an iterative fashion.

How do I ensure that I've done it right? The prototype is "test driven" by the users and feedback from the test drive is used for the next iterative modification of the prototype.

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 horrific interface problems. But even in a "Windows world," we all have encountered user interfaces that are difficult to learn, difficult to use, confusing, 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.

15.1 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 activity.

15.1.1 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 window-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 of 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:

How do we design interfaces 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 command (or voice input) to draw a complex shape.

Quote:

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

Douglas Adams

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.

15.1.2 Reduce the User's Memory Load

The more a user has to remember, the more error-prone will be the interaction with the system. 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:

How do we design interfaces that 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).

Quote:

"The interface from hell: 'Enter any 11-digit prime number to continue . . .' "

author unknown

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 of functions under a text style menu. However, every underlining capability is not listed. The user must pick underlining, then all underlining options (e.g., single underline, double underline, dashed underline) are presented.

15.1.3 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 are 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:

How do we design interfaces that are consistent?

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 he 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.

15.2 USER INTERFACE DESIGN

The overall process for 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 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 for quality.

15.2.1 Interface Design Models

An excellent source of GUI design guidelines, methods, and references can be found at **www.ibm.com/ibm/easy/**

Four different models come into play when a user interface is to be designed. The software engineer creates a *design model,* a human engineer (or the software engineer) establishes a *user model,* the end-user develops a mental image that is often called the *user's model* or the *system perception,* and the implementers of the system create a *system image* [RUB88]. 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.

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 to define the user of the system, but the interface design is often only incidental to the design model.[1]

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

- **Novices.** No *syntactic knowledge*[2] of the system and little *semantic knowledge*[3] 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.

Quote:

"USER, n.: The word computer professionals use when they mean 'idiot.' "

Dave Barry

The system perception (user's model) is the image of the system that end-users carry in their heads. For example, if the user of a particular word processor 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 word processors fully but has worked with the specific word processor 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.

KEY POINT

When the system image and the system perception coincide, the user can apply the application effectively.

The system image 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 system image and the system perception 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 system image 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

---

1 Of course, this is not as it should be. For interactive systems, the interface design is as important as the data, architectural, or component-level design.

2 In this context, *syntactic knowledge* refers to the mechanics of interaction that is required to use the interface effectively.

3 *Semantic knowledge* refers to an 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.

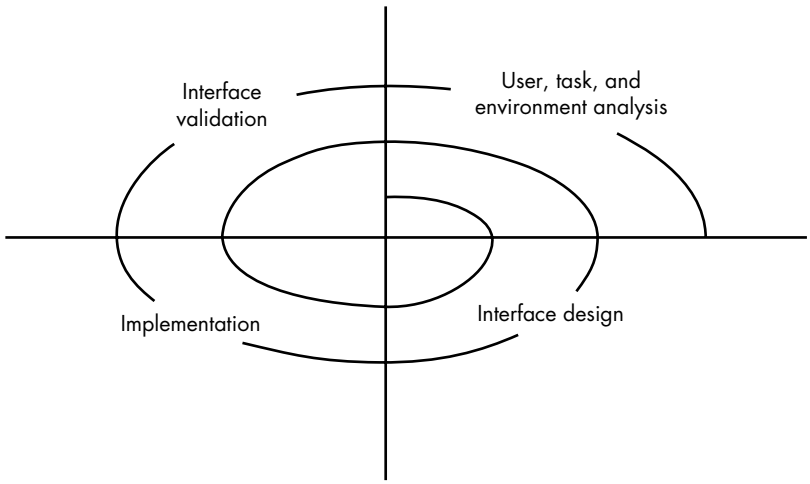

FIGURE 15.1
The user interface design process

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."

15.2.2 The User Interface Design Process

The design process for user interfaces is iterative and can be represented using a spiral model similar to the one discussed in Chapter 2. Referring to Figure 15.1, the user interface design process encompasses four distinct framework activities [MAN97]:

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

The spiral shown in Figure 15.1 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 implementation activity involves prototyping—the only practical way to validate what has been designed.

Quote:
"I never design a building before I've seen the site and met the people who will be using it."
Frank Lloyd Wright

The initial analysis activity focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system 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 15.2.1) for each class of users.

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 15.3.

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

- 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.

? What is the goal of user interface design?

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 15.4.

The implementation activity normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit (Section 15.5) 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.

15.3 TASK ANALYSIS AND MODELING

In Chapter 13, 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. Later in this book, we consider object-oriented analysis as a modeling approach for computer-based systems. *Task analysis* for interface design uses either an elaborative or object-oriented approach but applies this approach to human activities.

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-manual activity. To understand the tasks that must be performed to accomplish the goal of the

activity, a human engineer[4] must understand the tasks that humans currently 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.

Human tasks are defined and classified as part of task analysis. A process of elaboration is used to refine tasks. Alternatively, objects and actions are identified and refined.

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, 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, 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; (3) use furniture templates to draw scaled furniture outlines on floor plan; (4) move furniture outlines to get best placement; (5) label all furniture outlines; (6) draw dimensions to show location; (7) draw perspective 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. 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-oriented analysis techniques can be applied during task analysis. These are discussed in Chapter 21.

An alternative approach to task analysis takes an object-oriented point of view. The human engineer observes the physical objects that are used by the interior designer and the actions that are applied to each object. For example, the furniture template would be an object in this approach to task analysis. The interior designer would *select* the appropriate template, *move* it to a position on the floor plan, *trace* the furniture outline and so forth. The design model for the interface would not provide a literal implementation for each of these actions, but it would define user tasks that accomplish the end result (drawing furniture outlines on the floor plan).

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

15.4 INTERFACE DESIGN ACTIVITIES

Once task 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. The first interface design steps [NOR86] can be accomplished using the following approach:

? What steps do we perform to accomplish interface design?

1. Establish the goals[5] and intentions for each task.
2. Map each goal and intention to a sequence of specific actions.
3. Specify the action sequence of tasks and subtasks, also called a *user scenario,* as it will be executed at the interface level.
4. Indicate the state of the system; that is, what does the interface look like at the time that a user scenario is performed?
5. Define control mechanisms; that is, the objects and actions available to the user to alter the system state.
6. Show how control mechanisms affect the state of the system.
7. Indicate how the user interprets the state of the system from information provided through the interface.

Always following the golden rules discussed in Section 15.1, the interface designer must also consider how the interface will be implemented, the environment (e.g., display technology, operating system, development tools) that will be used, and other elements of the application that "sit behind" the interface.

15.4.1 Defining Interface Objects and Actions

XRef

A complete discussion of the grammatical parse can be found in Section 12.6.2.

An important step in interface design is the definition of interface objects and the actions that are applied to them. To accomplish this, the user scenario is parsed in much the same way as processing narratives were parsed in Chapter 12. That is, a description of a user scenario 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 is 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.

---

5 Goals include a consideration of the usefulness of the task, its effectiveness in accomplishing the overriding business objective, the degree to which the task can be learned quickly, and the degree to which users will be satisfied with the ultimate implementation of the task.

? What is screen layout and how is it applied?

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 an advanced version of the *SafeHome* system (discussed in earlier chapters). In the advanced version, *SafeHome* can be accessed via modem or through the Internet. A PC application allows the homeowner to check the status of the house from a remote location, reset the *SafeHome* configuration, arm and disarm the system, and (using an extra cost video option[6]) monitor rooms within the house visually. A preliminary user scenario for the interface follows:

XRef

The scenario described here is similar to use-cases described in Chapter 11.

Scenario: The homeowner wishes to gain access to the *SafeHome* system installed in his house. Using software operating on a remote PC (e.g., a notebook computer carried by the homeowner while at work or traveling), the homeowner determines the status of the alarm system, arms or disarms the system, reconfigures security zones, and views different rooms within the house via preinstalled video cameras.

To access *SafeHome* from a remote location, the homeowner provides 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, the user (with full access privileges) checks the status of the system and changes status by arming or disarming *SafeHome.* The user reconfigures 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. The user views the interior of the house via strategically placed video cameras. The user can pan and zoom each camera to provide different views of the interior.

Homeowner tasks:

- *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** (4 frames per second)
- *pans* or *zooms* the **video camera**

---

6 The video option enables the homeowner to place video cameras at key locations throughout a house and peruse the output from a remote location. Big Brother?

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 with video display).

A preliminary sketch of the screen layout for video monitoring is created (Figure 15.2). To invoke the video image, a **video camera location** icon, *C,* located in **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

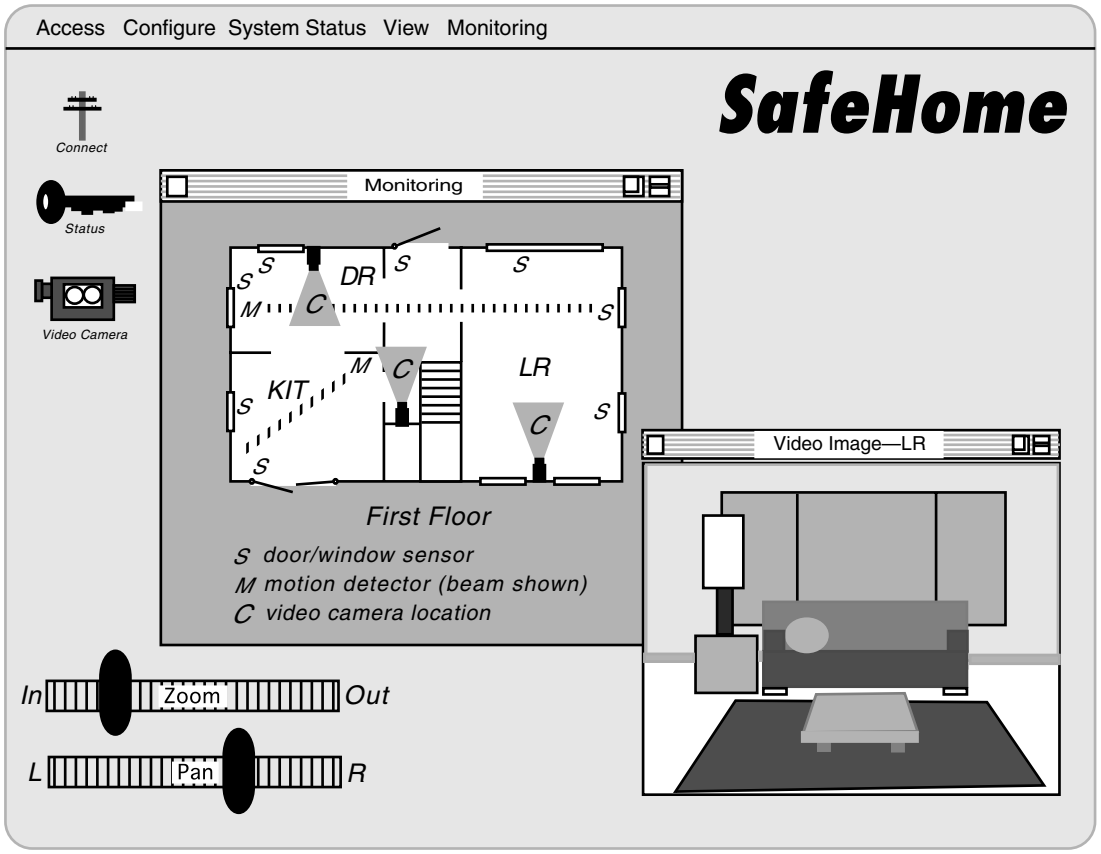

FIGURE 5.2 Preliminary screen layout

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.

15.4.2 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.

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 desired output or action.

System response time has two important characteristics: length and variability. If the *length* of system response is too long, user frustration and stress is the inevitable result. However, a very brief response time can also be detrimental if the user is being paced by the interface. A rapid response may force the user to rush and therefore make mistakes.

If variable response is unavoidable, be certain to provide some visual indication of progress, so that the user is aware of what is happening.

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 is preferable to a response that varies from 0.1 to 2.5 seconds. The user is always off balance, always wondering whether something "different" has occurred behind the scenes.

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 many cases, however, modern software provides on-line help facilities that enable a user to get a question answered or resolve a problem without leaving the interface.

Two different types of help facilities are encountered: integrated and add-on [RUB88]. An *integrated help facility* is designed into the software from the beginning. It is often context sensitive, enabling the user to select from those topics that are relevant to the actions currently being performed. Obviously, this reduces the time required for the user to obtain help and increases the "friendliness" of the interface. An *add-on help facility* is added to the software after the system has been built. In many ways, it is really an on-line user's manual with limited query capability. The user may have to search through a list of hundreds of topics to find appropriate guidance, often making many false starts and receiving much irrelevant information. There is little doubt that the integrated help facility is preferable to the add-on approach.

What design issues should be considered when we build a help facility?

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 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:

Spend twice as much effort and expend twice as many words on troubleshooting as you think you'll need for your help facility, and you'll probably get it about right.

SEVERE SYSTEM FAILURE -- 14A

Somewhere, an explanation for error 14A must exist; otherwise, why would the designers have added the identification? Yet, the error message provides no real indication of what is 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:

- The message should describe the problem in jargon that 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.

The typed command was once the most common mode of interaction between user 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 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?

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.

15.5 IMPLEMENTATION TOOLS

Once a design model is created, it is implemented as a prototype,[7] examined by users (who fit the user model described earlier) and modified based on their comments. To accommodate this iterative design approach, a broad class of interface design and prototyping tools has evolved. Called *user-interface toolkits* or *user-interface development systems* (UIDS), these tools provide components or objects that facilitate creation of windows, menus, device interaction, error messages, commands, and many other elements of an interactive environment.

CASE Tools
User Interface Design

Using prepackaged software components to create a user interface, a UIDS provides built-in mechanisms [MYE89] for

- managing input devices (such as a mouse or keyboard)
- validating user input
- handling errors and displaying error messages
- providing feedback (e.g., automatic input echo)
- providing help and prompts

7 It should be noted that in some cases (e.g., aircraft cockpit displays) the first step might be to simulate the interface on a display device rather than prototyping it.

- handling windows and fields, scrolling within windows
- establishing connections between application software and the interface
- insulating the application from interface management functions
- allowing the user to customize the interface

These functions can be implemented using either a language-based or graphical approach.

15.6 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 15.3. After the design model has been completed, a first-level prototype is created. The prototype is evaluated by the user, 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

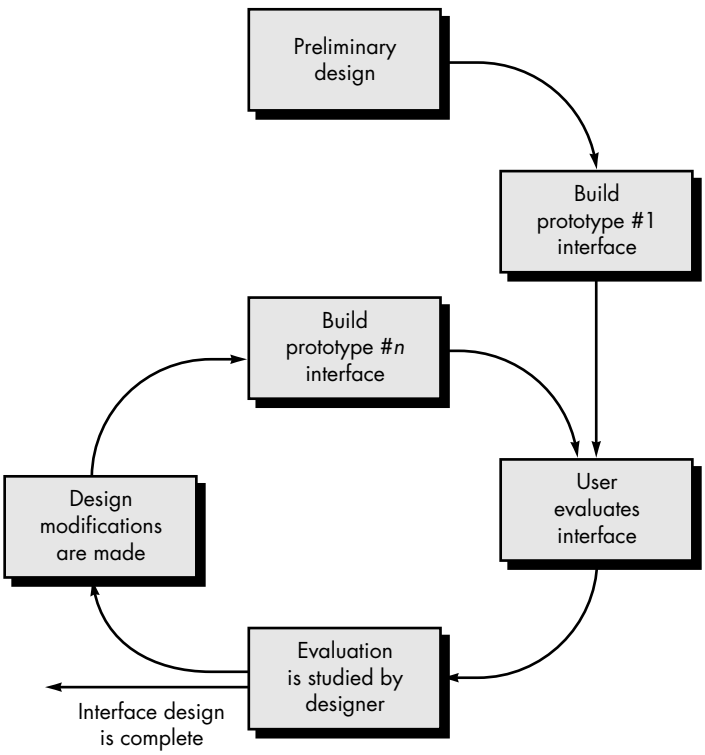

FIGURE 15.3 The interface design evaluation cycle

Quote:

"A computer terminal is not some clunky old television with a typewriter in front of it. It is an interface where the mind and body can connect with the universe and move bits of it about."

Douglas Adams

(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 all (1) simple yes/no response, (2) numeric response, (3) scaled (subjective) response, or (4) percentage (subjective) response. Examples are

User Interface

1. Were the icons self-explanatory? If not, which icons were unclear?
2. Were the actions easy to remember and to invoke?
3. How many different actions did you use?
4. How easy was it to learn basic system operations (scale 1 to 5)?
5. Compared to other interfaces you've used, how would this rate—top 1%, top 10%, top 25%, top 50%, bottom 50%?

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.

A complete discussion of user interface evaluation methods is beyond the scope of this book. For further information, see [LEA88] and [MAN97].

15.7 SUMMARY

The user interface is arguably the most important element of a computer-based system or product. If the interface is poorly designed, the user's ability to tap the computational power of an application may be severely hindered. In fact, a weak interface may cause an otherwise well-designed and solidly implemented application to fail.

Three important principles guide the design of effective user interfaces: (1) place the user in control, (2) reduce the user's memory load, and (3) make the interface consistent. To achieve an interface that abides by these principles, an organized design process must be conducted.

User interface design begins with the identification of user, task, and environmental requirements. Task analysis is a design activity that defines user tasks and actions using either an elaborative or object-oriented approach.

Once tasks have been identified, user scenarios are created and analyzed to define a set of interface objects and actions. This provides a basis for the creation of screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items. Design issues such as response time, command and action structure, error handling, and help facilities are considered as the design model is refined. A variety of implementation tools are used to build a prototype for evaluation by the user.

The user interface is the window into the software. In many cases, the interface molds a user's perception of the quality of the system. If the "window" is smudged, wavy, or broken, the user may reject an otherwise powerful computer-based system.

REFERENCES

[LEA88] Lea, M., "Evaluating User Interface Designs," *User Interface Design for Computer Systems,* Halstead Press (Wiley), 1988.

[MAN97] Mandel, T., *The Elements of User Interface Design,* Wiley, 1997.

[MON84] Monk, A. (ed.), *Fundamentals of Human-Computer Interaction,* Academic Press, 1984.

[MOR81] Moran, T.P., "The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems," *Intl. Journal of Man-Machine Studies,* vol. 15, pp. 3–50.

[MYE89] Myers, B.A., "User Interface Tools: Introduction and Survey, *IEEE Software,* January 1989, pp. 15–23.

[NOR86] Norman, D.A., "Cognitive Engineering," in *User Centered Systems Design,* Lawrence Earlbaum Associates, 1986.

[RUB88] Rubin, T., *User Interface Design for Computer Systems,* Halstead Press (Wiley), 1988.

[SHN90] Shneiderman, B., *Designing the User Interface,* 3rd ed., Addison-Wesley, 1990.

PROBLEMS AND POINTS TO PONDER

15.1. Describe the worst interface that you have ever worked with and critique it relative to the concepts introduced in this chapter. Describe the best interface that you have ever worked with and critique it relative to the concepts introduced in this chapter.

15.2. Develop two additional design principles that "place the user in control."

15.3. Develop two additional design principles that "reduce the user's memory load."

15.4. Develop two additional design principles that "make the interface consistent."

15.5. Consider one of the following interactive applications (or an application assigned by your instructor):

a. A desktop publishing system.
b. A computer-aided design system.
c. An interior design system (as described in Section 15.3.2).
d. An automated course registration system for a university.
e. A library management system.
f. An Internet-based polling booth for public elections.
g. A home banking system.
h. An interactive application assigned by your instructor.

Develop a design model, a user model, a system image, and a system perception for any one of these systems.

15.6. Perform a detailed task analysis for any one of the systems listed in Problem 15.5. Use either an elaborative or object-oriented approach.

15.7. Continuing Problem 15.6, define interface objects and actions for the application you have chosen. Identify each object type.

15.8. Develop a set of screen layouts with a definition of major and minor menu items for the system you chose in Problem 15.5.

15.9. Develop a set of screen layouts with a definition of major and minor menu items for the advanced *SafeHome* system described in Section 15.4.1. You may elect to take a different approach than the one shown for the screen layout in Figure 15.2.

15.10. Describe your approach to user help facilities for the task analysis design model and task analysis you have performed as part of Problems 15.5 through 15.8.

15.11. Provide a few examples that illustrate why response time variability can be an issue.

15.12. Develop an approach that would automatically integrate error messages and a user help facility. That is, the system would automatically recognize the error type

and provide a help window with suggestions for correcting it. Perform a reasonably complete software design that considers appropriate data structures and algorithms.

15.13. Develop an interface evaluation questionnaire that contains 20 generic questions that would apply to most interfaces. Have ten classmates complete the questionnaire for an interactive system that you all use. Summarize the results and report them to your class.

FURTHER READINGS AND INFORMATION SOURCES

Although his book is not specifically about human/computer interfaces, much of what Donald Norman (*The Design of Everyday Things,* reissue edition, Currency/Doubleday, 1990) has to say about the psychology of effective design applies to the user interface. It is recommended reading for anyone who is serious about doing high-quality interface design.

Dozens of books have been written about interface design over the past decade. However, books by Mandel [MAN97] and Shneiderman [SHN90] continue to provide the most comprehensive (and readable) treatments of the subject. Donnelly (*In Your Face: The Best of Interactive Interface Design,* Rockport Publications, 1998); Fowler, Stanwick, and Smith (*GUI Design Handbook,* McGraw-Hill, 1998); Weinschenk, Jamar, and Yeo (*GUI Design Essentials,* Wiley, 1997); Galitz (*The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques,* Wiley, 1996); Mullet and Sano (*Designing Visual Interfaces: Communication Oriented Techniques,* PrenticeHall, 1995); and Cooper (*About Face: The Essentials of User Interface Design,* IDG Books, 1995) have written treatments that provide additional design guidelines and principles as well as suggestions for interface requirements elicitation, design modeling, implementation, and testing.

Task analysis and modeling are pivotal interface design activities. Hackos and Redish (*User and Task Analysis for Interface Design,* Wiley, 1998) have written a book dedicated to these subjects and provide a detailed method for approaching task analysis. Wood (*User Interface Design: Bridging the Gap from User Requirements to Design,* CRC Press, 1997) considers the analysis activity for interfaces and the transition to design tasks. One of the first books to present the subject of scenarios in user-interface design has been edited by Carroll *(Scenario-Based Design: Envisioning Work and Technology in System Development,* Wiley, 1995). A formal method for design of user interfaces, based on state-based behavior modeling has been developed by Horrocks (*Constructing the User Interface with Statecharts,* Addison-Wesley, 1998).

The evaluation activity focuses on usability. Books by Rubin (*Handbook of Usability Testing: How to Plan, Design, and Conduct Effective Tests,* Wiley, 1994) and Nielson (*Usability Inspection Methods,* Wiley, 1994) address the topic in considerable detail.

The Apple Macintosh popularized easy to use and solidly designed user interfaces. The Apple staff (*MacIntosh Human Interface Guidelines,* Addison-Wesley, 1993) dis-

cusses the now famous (and much copied) Macintosh look and feel. One of the earliest among many books written about the Microsoft Windows interface was produced by the Microsoft staff (*The Windows Interface Guidelines for Software Design: An Application Design Guide,* Microsoft Press, 1995).

In a unique book that may be of considerable interest to product designers, Murphy (*Front Panel: Designing Software for Embedded User Interfaces,* R&D Books, 1998) provides detailed guidance for the design of interfaces for embedded systems and addresses safety hazards inherent in controls, handling heavy machinery, and interfaces for medical or transport systems. Interface design for embedded products is also discussed by Garrett (*Advanced Instrumentation and Computer I/O Design: Real-Time System Computer Interface Engineering,* IEEE, 1994).

A wide variety of information sources on user interface design and related subjects is available on the Internet. An up-to-date list of World Wide Web references that are relevant to interface design issues can be found at the SEPA Web site:
http://www.mhhe.com/engcs/compsci/pressman/resources/interface-design.mhtml

CHAPTER

16 COMPONENT-LEVEL DESIGN

KEY CONCEPTS

Component-level design, also called *procedural design,* occurs after data, architectural, and interface designs have been established. The intent is to translate the design model into operational software. But the level of abstraction of the existing design model is relatively high, and the abstraction level of the operational program is low. The translation can be challenging, opening the door to the introduction of subtle errors that are difficult to find and correct in later stages of the software process. In a famous lecture, Edsgar Dijkstra, a major contributor to our understanding of design, stated [DIJ72]:

> Software seems to be different from many other products, where as a rule higher quality implies a higher price. Those who want really reliable software will discover that they must find a means of avoiding the majority of bugs to start with, and as a result, the programming process will become cheaper . . . effective programmers . . . should not waste their time debugging—they should not introduce bugs to start with.

Although these words were spoken many years ago, they remain true today. When the design model is translated into source code, we must follow a set of design principles that not only perform the translation but also do not "introduce bugs to start with."

QUICK LOOK

What is it? Data, architectural, and interface design must be translated into operational software. To accomplish this, the design must be represented at a level of abstraction that is close to code. Component-level design establishes the algorithmic detail required to manipulate data structures, effect communication between software components via their interfaces, and implement the processing algorithms allocated to each component.

Who does it? A software engineer performs component-level design.

Why is it important? You have to be able to determine whether the program will work before you build it. The component-level design represents the software in a way that allows you to review the details of the design for correctness and consistency with earlier design representations (i.e., the data, architectural, and interface designs). It provides a means for assessing whether data structures, interfaces, and algorithms will work.

What are the steps? Design representations of data, architecture, and interfaces form the foundation for component-level design. The processing narrative for each component is translated into a procedural design model using a set of structured programming constructs. Graphical, tabular, or text-based notation is used to represent the design.

QUICK LOOK

What is the work product? The procedural design for each component, represented in graphical, tabular, or text-based notation, is the primary work product produced during component-level design.

How do I ensure that I've done it right? A design walkthrough or inspection is conducted. The design is examined to determine whether data structures, interfaces, processing sequences, and logical conditions are correct and will produce the appropriate data or control transformation allocated to the component during earlier design steps.

It is possible to represent the component-level design using a programming language. In essence, the program is created using the design model as a guide. An alternative approach is to represent the procedural design using some intermediate (e.g., graphical, tabular, or text-based) representation that can be translated easily into source code. Regardless of the mechanism that is used to represent the component-level design, the data structures, interfaces, and algorithms defined should conform to a variety of well-established procedural design guidelines that help us to avoid errors as the procedural design evolves. In this chapter, we examine these design guidelines.

16.1 STRUCTURED PROGRAMMING

Quote:

"When I'm working on a problem, I never think about beauty. I think only how to solve the problem. But when I have finished, if the solution is not beautiful, I know it is wrong."

R. Buckminster Fuller

The foundations of component-level design were formed in the early 1960s and were solidified with the work of Edsgar Dijkstra and his colleagues ([BOH66], [DIJ65], [DIJ76]). In the late 1960s, Dijkstra and others proposed the use of a set of constrained logical constructs from which any program could be formed. The constructs emphasized "maintenance of functional domain." That is, each construct had a predictable logical structure, was entered at the top and exited at the bottom, enabling a reader to follow procedural flow more easily.

The constructs are sequence, condition, and repetition. *Sequence* implements processing steps that are essential in the specification of any algorithm. *Condition* provides the facility for selected processing based on some logical occurrence, and *repetition* allows for looping. These three constructs are fundamental to *structured programming*—an important component-level design technique.

The structured constructs were proposed to limit the procedural design of software to a small number of predictable operations. Complexity metrics (Chapter 19) indicate that the use of the structured constructs reduces program complexity and thereby enhances readability, testability, and maintainability. The use of a limited number of logical constructs also contributes to a human understanding process that psychologists call *chunking.* To understand this process, consider the way in which you are reading this page. You do not read individual letters but rather recognize patterns or chunks of letters that form words or phrases. The structured constructs are

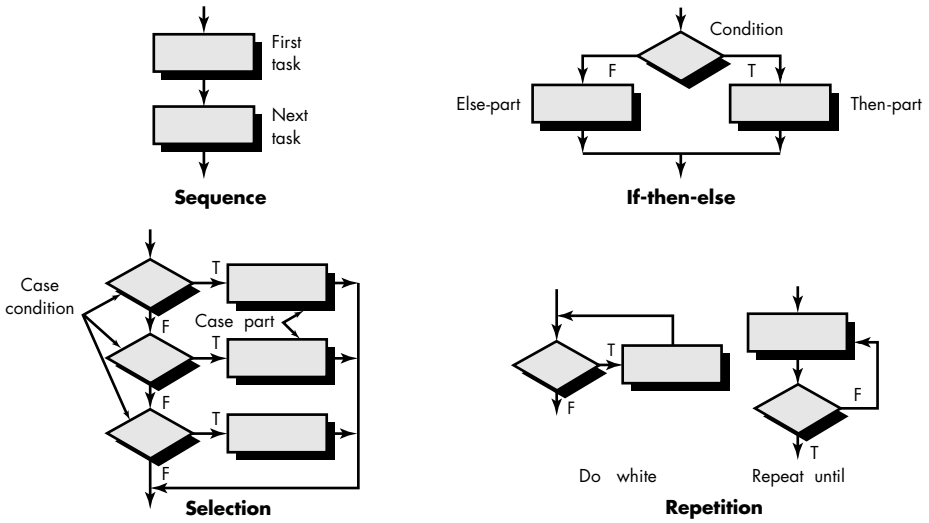

FIGURE 16.1
Flowchart constructs

KEY POINT
Structured programming provides a designer with useful logical patterns.

logical chunks that allow a reader to recognize procedural elements of a module, rather than reading the design or code line by line. Understanding is enhanced when readily recognizable logical patterns are encountered.

Any program, regardless of application area or technical complexity, can be designed and implemented using only the three structured constructs. It should be noted, however, that dogmatic use of only these constructs can sometimes cause practical difficulties. Section 16.1.1 considers this issue in further detail.

16.1.1 Graphical Design Notation

"A picture is worth a thousand words," but it's rather important to know which picture and which 1000 words. There is no question that graphical tools, such as the flowchart or box diagram, provide useful pictorial patterns that readily depict procedural detail. However, if graphical tools are misused, the wrong picture may lead to the wrong software.

A flowchart is quite simple pictorially. A box is used to indicate a processing step. A diamond represents a logical condition, and arrows show the flow of control. Figure 16.1 illustrates three structured constructs. The *sequence* is represented as two processing boxes connected by an line (arrow) of control. *Condition,* also called *if-then-else,* is depicted as a decision diamond that if true, causes *then-part* processing to occur, and if false, invokes *else-part* processing. *Repetition* is represented using two slightly different forms. The *do while* tests a condition and executes a loop task repetitively as long as the condition holds true. A *repeat until* executes the loop task first, then tests a condition and repeats the task until the condition fails. The selection (or select-case) construct shown in the figure is actually an extension of the

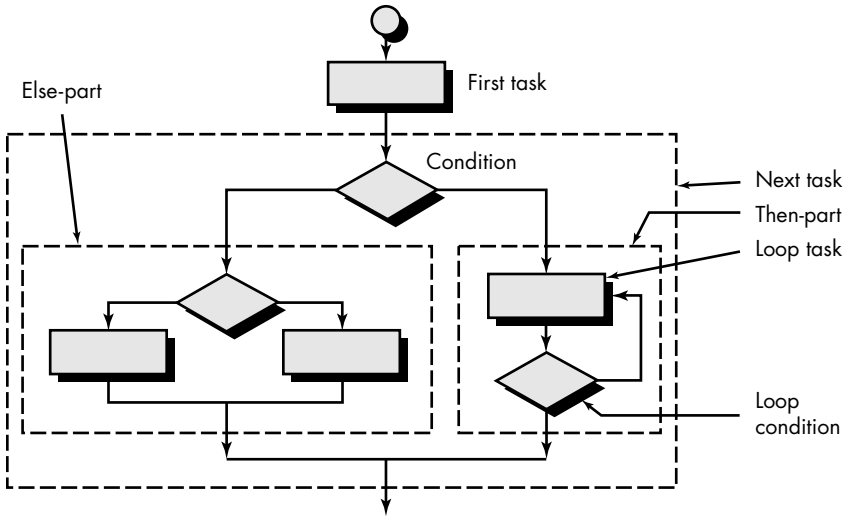

FIGURE 16.2 Nesting constructs

if-then-else. A parameter is tested by successive decisions until a true condition occurs and a case part processing path is executed.

Structured programming constructs should make it easier to understand the design. If using them without "violation" results in unnecessary complexity, it's OK to violate.

The structured constructs may be nested within one another as shown in Figure 16.2. Referring to the figure, repeat-until forms the then part of if-then-else (shown enclosed by the outer dashed boundary). Another if-then-else forms the else part of the larger condition. Finally, the condition itself becomes a second block in a sequence. By nesting constructs in this manner, a complex logical schema may be developed. It should be noted that any one of the blocks in Figure 16.2 could reference another module, thereby accomplishing procedural layering implied by program structure.

In general, the dogmatic use of only the structured constructs can introduce inefficiency when an escape from a set of nested loops or nested conditions is required. More important, additional complication of all logical tests along the path of escape can cloud software control flow, increase the possibility of error, and have a negative impact on readability and maintainability. What can we do?

Both the flowchart and box diagrams no longer are used as widely as they once were. In general, use them to document or evaluate design in specific instances, not to represent an entire system.

The designer is left with two options: (1) The procedural representation is redesigned so that the "escape branch" is not required at a nested location in the flow of control or (2) the structured constructs are violated in a controlled manner; that is, a constrained branch out of the nested flow is designed. Option 1 is obviously the ideal approach, but option 2 can be accommodated without violating of the spirit of structured programming.

Another graphical design tool, the *box diagram,* evolved from a desire to develop a procedural design representation that would not allow violation of the structured constructs. Developed by Nassi and Shneiderman [NAS73] and extended by Chapin [CHA74], the diagrams (also called *Nassi-Shneiderman charts, N-S charts,* or *Chapin charts*) have the following characteristics: (1) functional domain (that is, the scope of

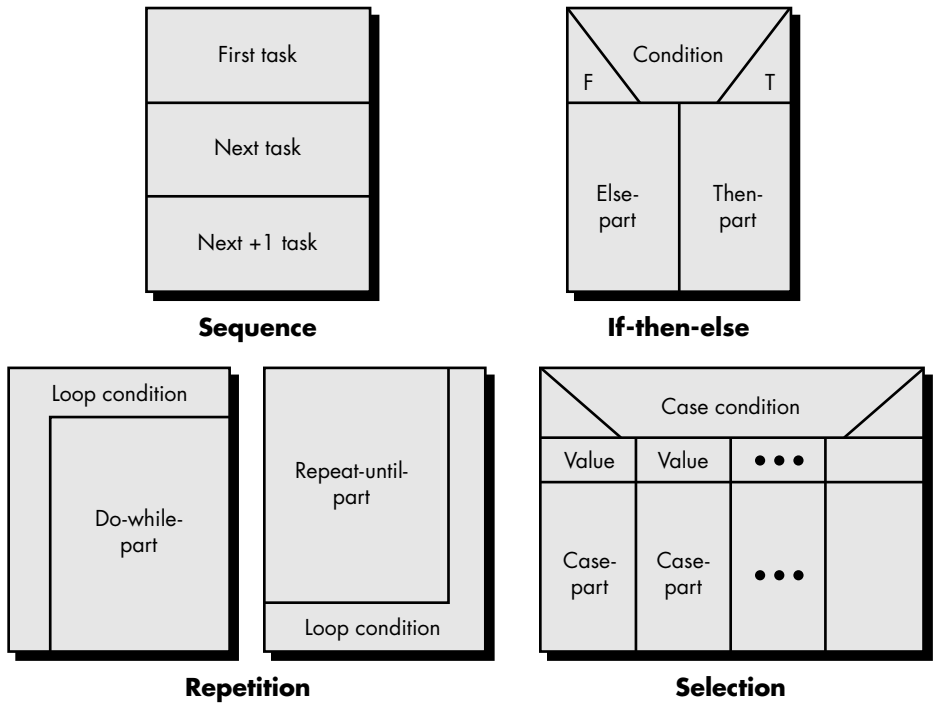

FIGURE 16.3
Box diagram constructs

repetition or if-then-else) is well defined and clearly visible as a pictorial representation, (2) arbitrary transfer of control is impossible, (3) the scope of local and/or global data can be easily determined, (4) recursion is easy to represent.

The graphical representation of structured constructs using the box diagram is illustrated in Figure 16.3. The fundamental element of the diagram is a box. To represent sequence, two boxes are connected bottom to top. To represent if-then-else, a condition box is followed by a then-part and else-part box. Repetition is depicted with a bounding pattern that encloses the process (do-while part or repeat-until part) to be repeated. Finally, selection is represented using the graphical form shown at the bottom of the figure.

Like flowcharts, a box diagram is layered on multiple pages as processing elements of a module are refined. A "call" to a subordinate module can be represented within a box by specifying the module name enclosed by an oval.

Use a decision table when a complex set of conditions and actions is encountered within a component.

16.1.2 Tabular Design Notation

In many software applications, a module may be required to evaluate a complex combination of conditions and select appropriate actions based on these conditions. Decision tables provide a notation that translates actions and conditions (described in a processing narrative) into a tabular form. The table is difficult to misinterpret and may

even be used as a machine readable input to a table driven algorithm. In a comprehensive treatment of this design tool, Ned Chapin states [HUR83]:

> Some old software tools and techniques mesh well with new tools and techniques of software engineering. Decision tables are an excellent example. Decision tables preceded software engineering by nearly a decade, but fit so well with software engineering that they might have been designed for that purpose.

Decision table organization is illustrated in Figure 16.4. Referring to the figure, the table is divided into four sections. The upper left-hand quadrant contains a list of all conditions. The lower left-hand quadrant contains a list of all actions that are possible based on combinations of conditions. The right-hand quadrants form a matrix that indicates condition combinations and the corresponding actions that will occur for a specific combination. Therefore, each column of the matrix may be interpreted as a processing *rule*.

? How do I build a decision table?

The following steps are applied to develop a decision table:

1. List all actions that can be associated with a specific procedure (or module).
2. List all conditions (or decisions made) during execution of the procedure.
3. Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions.
4. Define rules by indicating what action(s) occurs for a set of conditions.

	Rules								
Conditions	**1**	**2**	**3**	**4**					n
Condition #1	✓			✓	✓				
Condition #2		✓		✓					
Condition #3			✓		✓				
Actions									
Action #1	✓			✓	✓				
Action #2		✓		✓					
Action #3			✓						
Action #4			✓	✓	✓				
Action #5	✓	✓			✓				

FIGURE 16.4 Decision table nomenclature

To illustrate the use of a decision table, consider the following excerpt from a processing narrative for a public utility billing system:

> If the customer account is billed using a fixed rate method, a minimum monthly charge is assessed for consumption of less than 100 KWH (kilowatt-hours). Otherwise, computer billing applies a Schedule A rate structure. However, if the account is billed using a variable rate method, a Schedule A rate structure will apply to consumption below 100 KWH, with additional consumption billed according to Schedule B.

Figure 16.5 illustrates a decision table representation of the preceding narrative. Each of the five rules indicates one of five viable conditions (i.e., a T (true) in both fixed rate and variable rate account makes no sense in the context of this procedure; therefore, this condition is omitted). As a general rule, the decision table can be effectively used to supplement other procedural design notation.

16.1.3 Program Design Language

Program design language (PDL), also called *structured English* or *pseudocode,* is "a pidgin language in that it uses the vocabulary of one language (i.e., English) and the overall syntax of another (i.e., a structured programming language)" [CAI75]. In this chapter, PDL is used as a generic reference for a design language.

At first glance PDL looks like a modern programming language. The difference between PDL and a real programming language lies in the use of narrative text (e.g., English) embedded directly within PDL statements. Given the use of narrative text embedded directly into a syntactical structure, PDL cannot be compiled (at least not yet). However, PDL tools currently exist to translate PDL into a programming lan-

	Rules				
Conditions	**1**	**2**	**3**	**4**	**5**
Fixed rate acct.	T	T	F	F	F
Variable rate acct.	F	F	T	T	F
Consumption <100 kwh	T	F	T	F	
Consumption ≥100 kwh	F	T	F	T	
Actions					
Min. monthly charge	✓				
Schedule A billing		✓	✓		
Schedule B billing				✓	
Other treatment					✓

FIGURE 16.5 Resultant decision table

guage "skeleton" and/or a graphical representation (e.g., a flowchart) of design. These tools also produce nesting maps, a design operation index, cross-reference tables, and a variety of other information.

It's a good idea to use your programming language as the basis for the PDL. This will enable you to generate a code skeleton (mixed with narrative) as you perform component-level design.

A program design language may be a simple transposition of a language such as Ada or C. Alternatively, it may be a product purchased specifically for procedural design. Regardless of origin, a design language should have the following characteristics:

- A fixed syntax of keywords that provide for all structured constructs, data declaration, and modularity characteristics.
- A free syntax of natural language that describes processing features.
- Data declaration facilities that should include both simple (scalar, array) and complex (linked list or tree) data structures.
- Subprogram definition and calling techniques that support various modes of interface description.

A basic PDL syntax should include constructs for subprogram definition, interface description, data declaration, techniques for block structuring, condition constructs, repetition constructs, and I/O constructs. The format and semantics for some of these PDL constructs are presented in the section that follows.

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.

16.1.4 A PDL Example

To illustrate the use of PDL, we present an example of a procedural design for the *SafeHome* security system software introduced in earlier chapters. The system monitors alarms for fire, smoke, burglar, water, and temperature (e.g., furnace breaks while homeowner is away during winter) and 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 PDL defines an elaboration of the procedural design for the *security monitor* component.

```
PROCEDURE security.monitor;
INTERFACE RETURNS system.status;
TYPE signal IS STRUCTURE DEFINED
        name IS STRING LENGTH VAR;
        address IS HEX device location;
```

```
        bound.value IS upper bound SCALAR;
        message IS STRING LENGTH VAR;
END signal TYPE;
TYPE system.status IS BIT (4);
TYPE alarm.type DEFINED
        smoke.alarm IS INSTANCE OF signal;
        fire.alarm IS INSTANCE OF signal;
        water.alarm IS INSTANCE OF signal;
        temp.alarm IS INSTANCE OF signal;
        burglar.alarm IS INSTANCE OF signal;
TYPE phone.number IS area code + 7-digit number;
        •
        •
        •
initialize all system ports and reset all hardware;
CASE OF control.panel.switches (cps):
        WHEN cps = "test" SELECT
            CALL alarm PROCEDURE WITH "on" for test.time in seconds;
        WHEN cps = "alarm-off" SELECT
            CALL alarm PROCEDURE WITH "off";
        WHEN cps = "new.bound.temp" SELECT
        CALL keypad.input PROCEDURE;
        WHEN cps = "burglar.alarm.off" SELECT deactivate signal [burglar.alarm];
        •
        •
        •
        DEFAULT none;
ENDCASE
REPEAT UNTIL activate.switch is turned off
        reset all signal.values and switches;
        DO FOR alarm.type = smoke, fire, water, temp, burglar;
            READ address [alarm.type] signal.value;
            IF signal.value > bound [alarm.type]
            THEN  phone.message = message [alarm.type];
                        set alarm.bell to "on" for alarm.timeseconds;
                        PARBEGIN
                        CALL alarm PROCEDURE WITH "on", alarm.time in seconds;
                        CALL phone PROCEDURE WITH message [alarm.type], phone.number;
                        ENDPAR
            ELSE  skip
            ENDIF
        ENDFOR
ENDREP
END security.monitor
```

Note that the designer for the *security.monitor* component has used a new construct **PARBEGIN . . . ENDPAR** that specifies a parallel block. All tasks specified within the **PARBEGIN** block are executed in parallel. In this case, implementation details are not considered.

16.2 COMPARISON OF DESIGN NOTATION

In the preceding section, we presented a number of different techniques for representing a procedural design. A comparison must be predicated on the premise that any notation for component-level design, if used correctly, can be an invaluable aid in the design process; conversely, even the best notation, if poorly applied, adds little to understanding. With this thought in mind, we examine criteria that may be applied to compare 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.

The following attributes of design notation have been established in the context of the general characteristics described previously:

? What criteria can be used to assess design notation?

Modularity. Design notation should support the development of modular software and provide a means for interface specification.

Overall simplicity. Design notation should be relatively simple to learn, relatively easy to use, and generally easy to read.

Ease of editing. The procedural design may require modification as the software process proceeds. The ease with which a design representation can be edited can help facilitate each software engineering task.

Machine readability. Notation that can be input directly into a computer-based development system offers significant benefits.

Maintainability. Software maintenance is the most costly phase of the software life cycle. Maintenance of the software configuration nearly always means maintenance of the procedural design representation.

Structure enforcement. The benefits of a design approach that uses structured programming concepts have already been discussed. Design notation that enforces the use of only the structured constructs promotes good design practice.

Automatic processing. A procedural design contains information that can be processed to give the designer new or better insights into the correctness and quality of a design. Such insight can be enhanced with reports provided via software design tools.

Data representation. The ability to represent local and global data is an essential element of component-level design. Ideally, design notation should represent such data directly.

Logic verification. Automatic verification of design logic is a goal that is paramount during software testing. Notation that enhances the ability to verify logic greatly improves testing adequacy.

"Code-to" ability. The software engineering task that follows component-level design is code generation. Notation that may be converted easily to source code reduces effort and error.

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 admittedly 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 difficult. 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 flowcharts and box diagrams provide 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., see [PET81], [SOM96]), 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.

16.3 SUMMARY

The design process encompasses a sequence of activities that slowly reduces the level of abstraction with which software is represented. Component-level design depicts the software at a level of abstraction that is very close to code.

At the component level, the software engineer must represent data structures, interfaces, and algorithms 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.

REFERENCES

[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.

[CHA74] Chapin, N., "A New Format for Flowcharts," *Software—Practice and Experience,* vol. 4, no. 4 , 1974, pp. 341–357.
[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.
[LIN79] Linger, R.C., H.D. Mills, and B.I. Witt, *Structured Programming,* Addison-Wesley, 1979.
[NAS73] Nassi, I. and B. Shneiderman, "Flowchart Techniques for Structured Programming," *SIGPLAN Notices,* ACM, August 1973.
[PET81] Peters, L.J., *Software Design: Methods and Techniques,* Yourdon Press, 1981.
[SOM96] Sommerville, I., *Software Engineering,* 5th ed., Addison-Wesley, 1996.

PROBLEMS AND POINTS TO PONDER

16.1. 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?

16.2. All modern programming languages implement the structured programming constructs. Provide examples from three programming languages.

16.3. Why is "chunking" important during the component-level design review process?

Problems 16.4–16.11 may be represented using any one (or more) of the procedural design notations presented in this chapter. Your instructor may assign specific design notation to particular problems.

16.4. Develop a procedural design for components that implement the following sorts: Shell-Metzner sort; heapsort; BSST (tree) sort. Refer to a book on data structures if you are unfamiliar with these sorts.

16.5. Develop a procedural design for an interactive user interface that queries for basic income tax information. Derive your own requirements and assume that all tax computations are performed by other modules.

16.6. Develop a procedural design for a program that accepts an arbitrarily long text as input and produces a list of words and their frequency of occurrence as output.

16.7. Develop a procedural design of a program that will numerically integrate a function *f* in the bounds *a* to *b*.

16.8. Develop a procedural design for a generalized Turing machine that will accept a set of quadruples as program input and produce output as specified.

16.9. Develop a procedural design for a program that will solve the Towers of Hanoi problem. Many books on artificial intelligence discuss this problem in some detail.

16.10. Develop a procedural design for all or major portions of an LR parser for a compiler. Refer to one or more books on compiler design.

16.11. Develop a procedural design for an encryption/decryption algorithm of your choosing.

16.12. Write a one- or two-page argument for the procedural design notation that you feel is best. Be certain that your argument addresses the criteria presented in Section 16.2.

FURTHER READINGS AND INFORMATION SOURCES

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 include those by Robertson (*Simple Program Design,* Boyd and Fraser Publishing, 1994), Bentley *(Programming Pearls,* Addison-Wesley, 1986 and *More Programming Pearls,* Addison-Wesley, 1988), and Dahl, Dijkstra, and Hoare (*Structured Programming,* Academic Press, 1972).

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. The following books are representative of hundreds of titles that consider procedural design in a programming language context:

[ADA00] Adamson, T.A., K.C. Mansfield, and J.L. Antonakos, *Structured Basic Applied to Technology,* Prentice-Hall, 2000.

[ANT96] Antonakos, J.L. and K. Mansfield, *Application Programming in Structured C,* Prentice-Hall, 1996.

[FOR99] Forouzan, B.A. and R. Gilberg, *Computer Science: A Structured Programming Approach Using C++,* Brooks/Cole Publishing, 1999.

[OBR93] O'Brien, S.K. and S. Nameroff, *Turbo Pascal 7: The Complete Reference,* Osborne McGraw-Hill, 1993.

[WEL95] Welburn, T. and W. Price, *Structured COBOL: Fundamentals and Style,* 4th ed., Mitchell Publishers, 1995.

A wide variety of information sources on software design and related subjects is available on the Internet. An up-to-date list of World Wide Web references that are relevant to design concepts and methods can be found at the SEPA Web site: **http://www.mhhe.com/engcs/compsci/pressman/resources/comp-design.mhtml**

CHAPTER

17 SOFTWARE TESTING TECHNIQUES

KEY CONCEPTS

The importance of software testing and its implications with respect to software quality cannot be overemphasized. To quote Deutsch [DEU79],

> The development of software systems involves a series of production activities where opportunities for injection of human fallibilities are enormous. Errors may begin to occur at the very inception of the process where the objectives . . . may be erroneously or imperfectly specified, as well as [in] later design and development stages . . . Because of human inability to perform and communicate with perfection, software development is accompanied by a quality assurance activity.

Software testing is a critical element of software quality assurance and represents the ultimate review of specification, design, and code generation.

The increasing visibility of software as a system element and the attendant "costs" associated with a software failure are motivating forces for well-planned, thorough testing. It is not unusual for a software development organization to expend between 30 and 40 percent of total project effort on testing. In the extreme, testing of human-rated software (e.g., flight control, nuclear reactor monitoring) can cost three to five times as much as all other software engineering steps combined!

QUICK LOOK

What is it? Once source code has been generated, software must be tested to uncover (and correct) as many errors as possible before delivery to your customer. Your goal is to design a series of test cases that have a high likelihood of finding errors—but how? That's where software testing techniques enter the picture. These techniques provide systematic guidance for designing tests that (1) exercise the internal logic of software components, and (2) exercise the input and output domains of the program to uncover errors in program function, behavior. and performance.

Who does it? During early stages of testing, a software engineer performs all tests. However, as the testing process progresses, testing specialists may become involved.

Why is it important? Reviews and other SQA activities can and do uncover errors, but they are not sufficient. Every time the program is executed, the customer tests it! Therefore, you have to execute the program before it gets to the customer with the specific intent of finding and removing all errors. In order to find the highest possible number of errors, tests must be conducted systematically and test cases must be designed using disciplined techniques.

What are the steps? Software is tested from two different perspectives: (1) internal program logic is exercised using "white box" test case design tech- ▶