CHAPTER

13 DESIGN CONCEPTS AND PRINCIPLES

KEY CONCEPTS

The designer's goal is to produce a model or representation of an entity that will later be built. The process by which the design model is developed is described by Belady [BEL81]:

> [T]here are two major phases to any design process: diversification and convergence. Diversification is the *acquisition* of a repertoire of alternatives, the raw material of design: components, component solutions, and knowledge, all contained in catalogs, textbooks, and the mind. During convergence, the designer chooses and combines appropriate elements from this repertoire to meet the design objectives, as stated in the requirements document and as agreed to by the customer. The second phase is the gradual *elimination* of all but one particular configuration of components, and thus the creation of the final product.

Diversification and convergence combine intuition and judgment based on experience in building similar entities, a set of principles and/or heuristics that guide the way in which the model evolves, a set of criteria that enables quality to be judged, and a process of iteration that ultimately leads to a final design representation.

Software design, like engineering design approaches in other disciplines, changes continually as new methods, better analysis, and broader understanding

QUICK LOOK

What is it? Design is a meaningful engineering representation of something that is to be built. It can be traced to a customer's requirements and at the same time assessed for quality against a set of predefined criteria for "good" design. In the software engineering context, design focuses on four major areas of concern: data, architecture, interfaces, and components. The concepts and principles discussed in this chapter apply to all four.

Who does it? Software engineers design computer-based systems, but the skills required at each level of design work are different. At the data and architectural level, design focuses on patterns as they apply to the application to be built. At the interface level, human ergonomics often dictate our design approach. At the component level, a "programming approach" leads us to effective data and procedural designs.

Why is it important? You wouldn't attempt to build a house without a blueprint, would you? You'd risk confusion, errors, a floor plan that didn't make sense, windows and doors in the wrong place . . . a mess. Computer software is considerably more complex than a house; hence, we need a blueprint—the design.

What are the steps? Design begins with the requirements model. We work to transform this model into four levels of design detail: the data structure,

QUICK LOOK

the system architecture, the interface representation, and the component level detail. During each design activity, we apply basic concepts and principles that lead to high quality.

What is the work product? Ultimately, a Design Specification is produced. The specification is composed of the design models that describe data, architecture, interfaces, and components. Each is a work product of the design process.

How do I ensure that I've done it right? At each stage, software design work products are reviewed for clarity, correctness, completeness, and consistency with the requirements and with one another.

evolve. Software design methodologies lack the depth, flexibility, and quantitative nature that are normally associated with more classical engineering design disciplines. However, methods for software design do exist, criteria for design quality are available, and design notation can be applied. In this chapter, we explore the fundamental concepts and principles that are applicable to all software design. Chapters 14, 15, 16, and 22 examine a variety of software design methods as they are applied to architectural, interface, and component-level design.

13.1 SOFTWARE DESIGN AND SOFTWARE ENGINEERING

Quote:

"The most common miracles of software engineering are the transitions from analysis to design and design to code."

Richard Dué

Software design sits at the technical kernel of software engineering and is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and specified, software design is the first of three technical activities—design, code generation, and test—that are required to build and verify the software. Each activity transforms information in a manner that ultimately results in validated computer software.

Each of the elements of the analysis model (Chapter 12) provides information that is necessary to create the four design models required for a complete specification of design. The flow of information during software design is illustrated in Figure 13.1. Software requirements, manifested by the data, functional, and behavioral models, feed the design task. Using one of a number of design methods (discussed in later chapters), the design task produces a data design, an architectural design, an interface design, and a component design.

The *data design* transforms the information domain model created during analysis into the data structures that will be required to implement the software. The data objects and relationships defined in the entity relationship diagram and the detailed data content depicted in the data dictionary provide the basis for the data design activity. Part of data design may occur in conjunction with the design of software architecture. More detailed data design occurs as each software component is designed.

The *architectural design* defines the relationship between major structural elements of the software, the "design patterns" that can be used to achieve the requirements

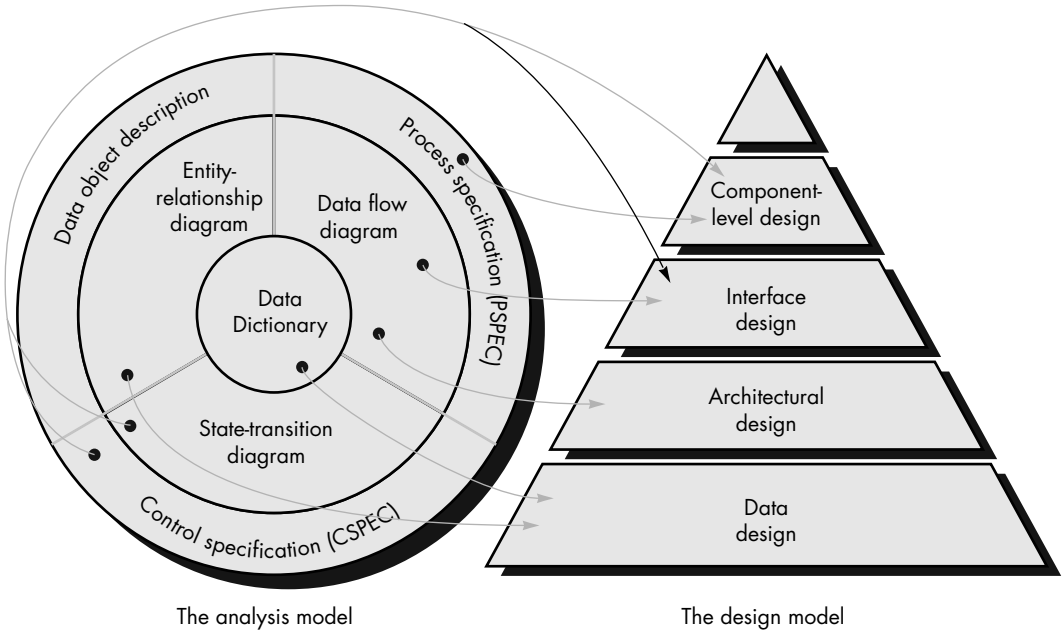

FIGURE 13.1 Translating the analysis model into a software design

that have been defined for the system, and the constraints that affect the way in which architectural design patterns can be applied [SHA96]. The architectural design representation—the framework of a computer-based system—can be derived from the system specification, the analysis model, and the interaction of subsystems defined within the analysis model.

The *interface design* describes how the software communicates within itself, with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, data and control flow diagrams provide much of the information required for interface design.

The *component-level design* transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the PSPEC, CSPEC, and STD serve as the basis for component design.

During design we make decisions that will ultimately affect the success of software construction and, as important, the ease with which software can be maintained. But why is design so important?

The importance of software design can be stated with a single word—*quality.* Design is the place where quality is fostered in software engineering. Design provides us with representations of software that can be assessed for quality. Design is the only way that we can accurately translate a customer's requirements into a finished software product or system. Software design serves as the foundation for all

the software engineering and software support steps that follow. Without design, we risk building an unstable system—one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in the software process, when time is short and many dollars have already been spent.

13.2 THE DESIGN PROCESS

Software design is an iterative process through which requirements are translated into a "blueprint" for constructing the software. Initially, the blueprint depicts a holistic view of software. That is, the design is represented at a high level of abstraction—a level that can be directly traced to the specific system objective and more detailed data, functional, and behavioral requirements. As design iterations occur, subsequent refinement leads to design representations at much lower levels of abstraction. These can still be traced to requirements, but the connection is more subtle.

13.2.1 Design and Software Quality

Throughout the design process, the quality of the evolving design is assessed with a series of formal technical reviews or design walkthroughs discussed in Chapter 8. McGlaughlin [MCG91] suggests three characteristics that serve as a guide for the evaluation of a good design:

Quote:

"To achieve a good design, people have to think the right way about how to conduct the design activity."

Katharine Whitehead

- The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
- The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

Each of these characteristics is actually a goal of the design process. But how is each of these goals achieved?

In order to evaluate the quality of a design representation, we must establish technical criteria for good design. Later in this chapter, we discuss design quality criteria in some detail. For the time being, we present the following guidelines:

? Are there generic guidelines that will lead to a good design?

1. A design should exhibit an architectural structure that (1) has been created using recognizable design patterns, (2) is composed of components that exhibit good design characteristics (these are discussed later in this chapter), and (3) can be implemented in an evolutionary fashion, thereby facilitating implementation and testing.

Quote:

"There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

C. A. R. Hoare

2. A design should be modular; that is, the software should be logically partitioned into elements that perform specific functions and subfunctions.
3. A design should contain distinct representations of data, architecture, interfaces, and components (modules).
4. A design should lead to data structures that are appropriate for the objects to be implemented and are drawn from recognizable data patterns.
5. A design should lead to components that exhibit independent functional characteristics.
6. A design should lead to interfaces that reduce the complexity of connections between modules and with the external environment.
7. A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis.

These criteria are not achieved by chance. The software design process encourages good design through the application of fundamental design principles, systematic methodology, and thorough review.

13.2.2 The Evolution of Software Design

The evolution of software design is a continuing process that has spanned the past four decades. Early design work concentrated on criteria for the development of modular programs [DEN73] and methods for refining software structures in a top-down manner [WIR71]. Procedural aspects of design definition evolved into a philosophy called *structured programming* [DAH72], [MIL72]. Later work proposed methods for the translation of data flow [STE74] or data structure [JAC75], [WAR74] into a design definition. Newer design approaches (e.g., [JAC92], [GAM95]) proposed an object-oriented approach to design derivation. Today, the emphasis in software design has been on software architecture [SHA96], [BAS98] and the design patterns that can be used to implement software architectures [GAM95], [BUS96], [BRO98].

? What characteristics are common to all design methods?

Many design methods, growing out of the work just noted, are being applied throughout the industry. Like the analysis methods presented in Chapter 12, each software design method introduces unique heuristics and notation, as well as a somewhat parochial view of what characterizes design quality. Yet, all of these methods have a number of common characteristics: (1) a mechanism for the translation of analysis model into a design representation, (2) a notation for representing functional components and their interfaces, (3) heuristics for refinement and partitioning, and (4) guidelines for quality assessment.

Regardless of the design method that is used, a software engineer should apply a set of fundamental principles and basic concepts to data, architectural, interface, and component-level design. These principles and concepts are considered in the sections that follow.

13.3 DESIGN PRINCIPLES

Software design is both a process and a model. The design *process* is a sequence of steps that enable the designer to describe all aspects of the software to be built. It is important to note, however, that the design process is not simply a cookbook. Creative skill, past experience, a sense of what makes "good" software, and an overall commitment to quality are critical success factors for a competent design.

The design *model* is the equivalent of an architect's plans for a house. It begins by representing the totality of the thing to be built (e.g., a three-dimensional rendering of the house) and slowly refines the thing to provide guidance for constructing each detail (e.g., the plumbing layout). Similarly, the design model that is created for software provides a variety of different views of the computer software.

Basic design principles enable the software engineer to navigate the design process. Davis [DAV95] suggests a set[1] of principles for software design, which have been adapted and extended in the following list:

- **The design process should not suffer from "tunnel vision."** A good designer should consider alternative approaches, judging each based on the requirements of the problem, the resources available to do the job, and the design concepts presented in Section 13.4.
- **The design should be traceable to the analysis model.** Because a single element of the design model often traces to multiple requirements, it is necessary to have a means for tracking how requirements have been satisfied by the design model.
- **The design should not reinvent the wheel.** Systems are constructed using a set of design patterns, many of which have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Time is short and resources are limited! Design time should be invested in representing truly new ideas and integrating those patterns that already exist.

Design consistency and uniformity are crucial when large systems are to be built. A set of design rules should be established for the software team before work begins.

- **The design should "minimize the intellectual distance" [DAV95] between the software and the problem as it exists in the real world.** That is, the structure of the software design should (whenever possible) mimic the structure of the problem domain.
- **The design should exhibit uniformity and integration.** A design is uniform if it appears that one person developed the entire thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components.

---

1 Only a small subset of Davis's design principles are noted here. For more information, see [DAV95].

- **The design should be structured to accommodate change.** The design concepts discussed in the next section enable a design to achieve this principle.
- **The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.** Well-designed software should never "bomb." It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.
- **Design is not coding, coding is not design.** Even when detailed procedural designs are created for program components, the level of abstraction of the design model is higher than source code. The only design decisions made at the coding level address the small implementation details that enable the procedural design to be coded.
- **The design should be assessed for quality as it is being created, not after the fact.** A variety of design concepts (Section 13.4) and design measures (Chapters 19 and 24) are available to assist the designer in assessing quality.

XRef

Guidelines for conducting effective design reviews are presented in Chapter 8.

- **The design should be reviewed to minimize conceptual (semantic) errors.** There is sometimes a tendency to focus on minutiae when the design is reviewed, missing the forest for the trees. A design team should ensure that major conceptual elements of the design (omissions, ambiguity, inconsistency) have been addressed before worrying about the syntax of the design model.

When these design principles are properly applied, the software engineer creates a design that exhibits both external and internal quality factors [MEY88]. *External quality factors* are those properties of the software that can be readily observed by users (e.g., speed, reliability, correctness, usability).[2] *Internal quality factors* are of importance to software engineers. They lead to a high-quality design from the technical perspective. To achieve internal quality factors, the designer must understand basic design concepts.

13.4 DESIGN CONCEPTS

A set of fundamental software design concepts has evolved over the past four decades. Although the degree of interest in each concept has varied over the years, each has stood the test of time. Each provides the software designer with a foundation from which more sophisticated design methods can be applied. Each helps the software engineer to answer the following questions:

- What criteria can be used to partition software into individual components?
- How is function or data structure detail separated from a conceptual representation of the software?
- What uniform criteria define the technical quality of a software design?

2 A more detailed discussion of quality factors is presented in Chapter 19.

M. A. Jackson once said: "The beginning of wisdom for a [software engineer] is to recognize the difference between getting a program to work, and getting it right" [JAC75]. Fundamental software design concepts provide the necessary framework for "getting it right."

13.4.1 Abstraction

Quote:

"Abstraction is one of the fundamental ways that we as humans cope with complexity."

Grady Booch

When we consider a modular solution to any problem, many *levels of abstraction* can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more procedural orientation is taken. Problem-oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented. Wasserman [WAS83] provides a useful definition:

> [T]he psychological notion of "abstraction" permits one to concentrate on a problem at some level of generalization without regard to irrelevant low level details; use of abstraction also permits one to work with concepts and terms that are familiar in the problem environment without having to transform them to an unfamiliar structure . . .

Each step in the software process is a refinement in the level of abstraction of the software solution. During system engineering, software is allocated as an element of a computer-based system. During software requirements analysis, the software solution is stated in terms "that are familiar in the problem environment." As we move through the design process, the level of abstraction is reduced. Finally, the lowest level of abstraction is reached when source code is generated.

As we move through different levels of abstraction, we work to create procedural and data abstractions. A *procedural abstraction* is a named sequence of instructions that has a specific and limited function. An example of a procedural abstraction would be the word *open* for a door. *Open* implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).

ADVICE

As a designer, work hard to derive both procedural and data abstractions that serve the problem at hand, but that also can be reused in other situations.

A *data abstraction* is a named collection of data that describes a data object (Chapter 12). In the context of the procedural abstraction *open*, we can define a data abstraction called **door**. Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction *open* would make use of information contained in the attributes of the data abstraction **door**.

Many modern programming languages provide mechanisms for creating abstract data types. For example, the Ada package is a programming language mechanism that provides support for both data and procedural abstraction. The original abstract data type is used as a template or generic data structure from which other data structures can be instantiated.

Control abstraction is the third form of abstraction used in software design. Like procedural and data abstraction, control abstraction implies a program control mechanism without specifying internal details. An example of a control abstraction is the *synchronization semaphore* [KAI83] used to coordinate activities in an operating system. The concept of the control abstraction is discussed briefly in Chapter 14.

13.4.2 Refinement

Stepwise refinement is a top-down design strategy originally proposed by Niklaus Wirth [WIR71]. A program is developed by successively refining levels of procedural detail. A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached. An overview of the concept is provided by Wirth [WIR71]:

> In each step (of the refinement), one or several instructions of the given program are decomposed into more detailed instructions. This successive decomposition or refinement of specifications terminates when all instructions are expressed in terms of any underlying computer or programming language . . . As tasks are refined, so the data may have to be refined, decomposed, or structured, and it is natural to refine the program and the data specifications in parallel.
>
> Every refinement step implies some design decisions. It is important that . . . the programmer be aware of the underlying criteria (for design decisions) and of the existence of alternative solutions . . .

There is a tendency to move immediately to full detail, skipping the refinement steps. This leads to errors and omissions and makes the design much more difficult to review. Perform stepwise refinement.

The process of program refinement proposed by Wirth is analogous to the process of refinement and partitioning that is used during requirements analysis. The difference is in the level of implementation detail that is considered, not the approach.

Refinement is actually a process of *elaboration.* We begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information. Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction and refinement are complementary concepts. Abstraction enables a designer to specify procedure and data and yet suppress low-level details. Refinement helps the designer to reveal low-level details as design progresses. Both concepts aid the designer in creating a complete design model as the design evolves.

13.4.3 Modularity

The concept of modularity in computer software has been espoused for almost five decades. Software architecture (described in Section 13.4.4) embodies modularity; that is, software is divided into separately named and addressable components, often called *modules,* that are integrated to satisfy problem requirements.

It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable" [MYE78]. Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a reader. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. To illustrate this point, consider the following argument based on observations of human problem solving.

Let *C(x)* be a function that defines the perceived complexity of a problem *x*, and *E(x)* be a function that defines the effort (in time) required to solve a problem *x.* For two problems, p_1 and p_2, if

$$C(p_1) > C(p_2) \tag{13-1a}$$

it follows that

$$E(p_1) > E(p_2) \tag{13-1b}$$

As a general case, this result is intuitively obvious. It does take more time to solve a difficult problem.

Quote:

"There's always an easy solution to every human problem—neat, plausible, and wrong."

H. L. Mencken

Another interesting characteristic has been uncovered through experimentation in human problem solving. That is,

$$C(p_1 + p_2) > C(p_1) + C(p_2) \tag{13-2}$$

Expression (13-2) implies that the perceived complexity of a problem that combines p_1 and p_2 is greater than the perceived complexity when each problem is considered separately. Considering Expression (13-2) and the condition implied by Expressions (13-1), it follows that

$$E(p_1 + p_2) > E(p_1) + E(p_2) \tag{13-3}$$

This leads to a "divide and conquer" conclusion—it's easier to solve a complex problem when you break it into manageable pieces. The result expressed in Expression (13-3) has important implications with regard to modularity and software. It is, in fact, an argument for modularity.

ADVICE

Don't overmodularize. The simplicity of each module will be overshadowed by the complexity of integration.

It is possible to conclude from Expression (13-3) that, if we subdivide software indefinitely, the effort required to develop it will become negligibly small! Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. Referring to Figure 13.2, the effort (cost) to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows. These characteristics lead to a total cost or effort curve shown in the figure. There is a number, *M,* of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict *M* with assurance.

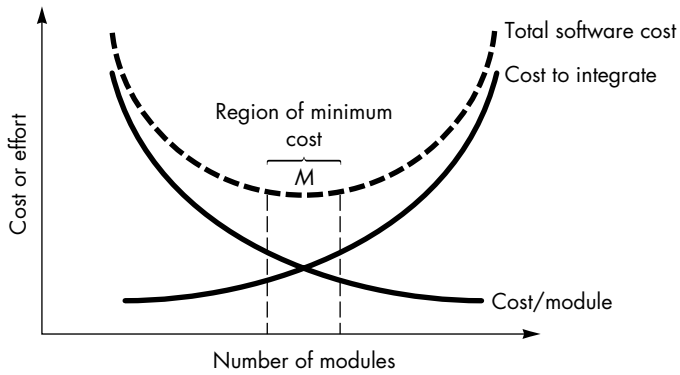

FIGURE 13.2 Modularity and software cost

The curves shown in Figure 13.2 do provide useful guidance when modularity is considered. We should modularize, but care should be taken to stay in the vicinity of *M.* Undermodularity or overmodularity should be avoided. But how do we know "the vicinity of M"? How modular should we make software? The answers to these questions require an understanding of other design concepts considered later in this chapter.

XRef

Design methods are discussed in Chapters 14, 15, 16, and 22.

Another important question arises when modularity is considered. How do we define an appropriate module of a given size? The answer lies in the method(s) used to define modules within a system. Meyer [MEY88] defines five criteria that enable us to evaluate a design method with respect to its ability to define an effective modular system:

? How can we evaluate a design method to determine if it will lead to effective modularity?

Modular decomposability. If a design method provides a systematic mechanism for decomposing the problem into subproblems, it will reduce the complexity of the overall problem, thereby achieving an effective modular solution.

Modular composability. If a design method enables existing (reusable) design components to be assembled into a new system, it will yield a modular solution that does not reinvent the wheel.

Modular understandability. If a module can be understood as a stand-alone unit (without reference to other modules), it will be easier to build and easier to change.

Modular continuity. If small changes to the system requirements result in changes to individual modules, rather than systemwide changes, the impact of change-induced side effects will be minimized.

Modular protection. If an aberrant condition occurs within a module and its effects are constrained within that module, the impact of error-induced side effects will be minimized.

Finally, it is important to note that a system may be designed modularly, even if its implementation must be "monolithic." There are situations (e.g., real-time software,

embedded software) in which relatively minimal speed and memory overhead introduced by subprograms (i.e., subroutines, procedures) is unacceptable. In such situations, software can and should be designed with modularity as an overriding philosophy. Code may be developed "in-line." Although the program source code may not look modular at first glance, the philosophy has been maintained and the program will provide the benefits of a modular system.

13.4.4 Software Architecture

WebRef

The STARS *Software Architecture Technology Guide* provides in-depth information and resources at **www-ast.tds-gn.lmco.com/arch/guide.html**

Software architecture alludes to "the overall structure of the software and the ways in which that structure provides conceptual integrity for a system" [SHA95a]. In its simplest form, architecture is the hierarchical structure of program components (modules), the manner in which these components interact and the structure of data that are used by the components. In a broader sense, however, *components* can be generalized to represent major system elements and their interactions.[3]

One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enable a software engineer to reuse design-level concepts.

Quote:

"A software architecture is the development work product that gives the highest return on investment with respect to quality, schedule and cost."

Len Bass et al.

Shaw and Garlan [SHA95a] describe a set of properties that should be specified as part of an architectural design:

Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects, filters) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.

Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

KEY POINT

Five different types of models are used to represent the architectural design.

Given the specification of these properties, the architectural design can be represented using one or more of a number of different models [GAR95]. *Structural models* represent architecture as an organized collection of program components. *Framework models* increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (patterns) that are encountered in similar types of applications. *Dynamic models* address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events. *Process models* focus on the design of the business

3 For example, the architectural components of a client/server system are represented at a different level of abstraction. See Chapter 28 for details.

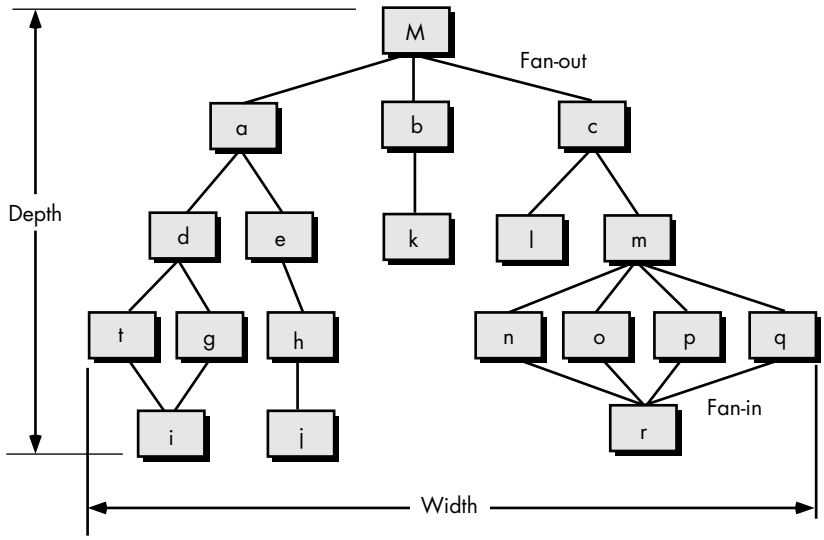

FIGURE 13.3 Structure terminology for a call and return architectural style

or technical process that the system must accommodate. Finally, *functional models* can be used to represent the functional hierarchy of a system.

A number of different *architectural description languages* (ADLs) have been developed to represent these models [SHA95b]. Although many different ADLs have been proposed, the majority provide mechanisms for describing system components and the manner in which they are connected to one another.

13.4.5 Control Hierarchy

XRef

A detailed discussion of architectural styles and patterns is presented in Chapter 14.

Control hierarchy, also called *program structure,* represents the organization of program components (modules) and implies a hierarchy of control. It does not represent procedural aspects of software such as sequence of processes, occurrence or order of decisions, or repetition of operations; nor is it necessarily applicable to all architectural styles.

Different notations are used to represent control hierarchy for those architectural styles that are amenable to this representation. The most common is the treelike diagram (Figure 13.3) that represents hierarchical control for call and return architectures.[4] However, other notations, such as Warnier-Orr [ORR77] and Jackson diagrams [JAC83] may also be used with equal effectiveness. In order to facilitate later discussions of structure, we define a few simple measures and terms. Referring to Figure 13.3, *depth* and *width* provide an indication of the number of levels of control and overall *span of control*, respectively. *Fan-out* is a measure of the number of modules that are directly controlled by another module. *Fan-in* indicates how many modules directly control a given module.

ADVICE

If you develop object-oriented software, the structural measures noted here do not apply. However, others (considered in Part Four) are applicable.

4 A call and return architecture (Chapter 14) is a classic program structure that decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components.

The control relationship among modules is expressed in the following way: A module that controls another module is said to be *superordinate* to it, and conversely, a module controlled by another is said to be *subordinate* to the controller [YOU79]. For example, referring to Figure 13.3, module *M* is superordinate to modules *a, b,* and *c.* Module *h* is subordinate to module *e* and is ultimately subordinate to module *M*. Width-oriented relationships (e.g., between modules *d* and *e*) although possible to express in practice, need not be defined with explicit terminology.

The control hierarchy also represents two subtly different characteristics of the software architecture: visibility and connectivity. *Visibility* indicates the set of program components that may be invoked or used as data by a given component, even when this is accomplished indirectly. For example, a module in an object-oriented system may have access to a wide array of data objects that it has inherited, but makes use of only a small number of these data objects. All of the objects are visible to the module. *Connectivity* indicates the set of components that are directly invoked or used as data by a given component. For example, a module that directly causes another module to begin execution is connected to it.[5]

13.4.6 Structural Partitioning

If the architectural style of a system is hierarchical, the program structure can be partitioned both horizontally and vertically. Referring to Figure 13.4a, *horizontal partitioning* defines separate branches of the modular hierarchy for each major program function. *Control modules,* represented in a darker shade are used to coordinate communication between and execution of the functions. The simplest approach to horizontal partitioning defines three partitions—input, data transformation (often called *processing*) and output. Partitioning the architecture horizontally provides a number of distinct benefits:

? What are the benefits of horizontal partitioning?

- software that is easier to test
- software that is easier to maintain
- propagation of fewer side effects
- software that is easier to extend

Because major functions are decoupled from one another, change tends to be less complex and extensions to the system (a common occurrence) tend to be easier to accomplish without side effects. On the negative side, horizontal partitioning often causes more data to be passed across module interfaces and can complicate the overall control of program flow (if processing requires rapid movement from one function to another).

---

5 In Chapter 20, we explore the concept of inheritance for object-oriented software. A program component can inherit control logic and/or data from another component without explicit reference in the source code. Components of this sort would be visible but not directly connected. A structure chart (Chapter 14) indicates connectivity.

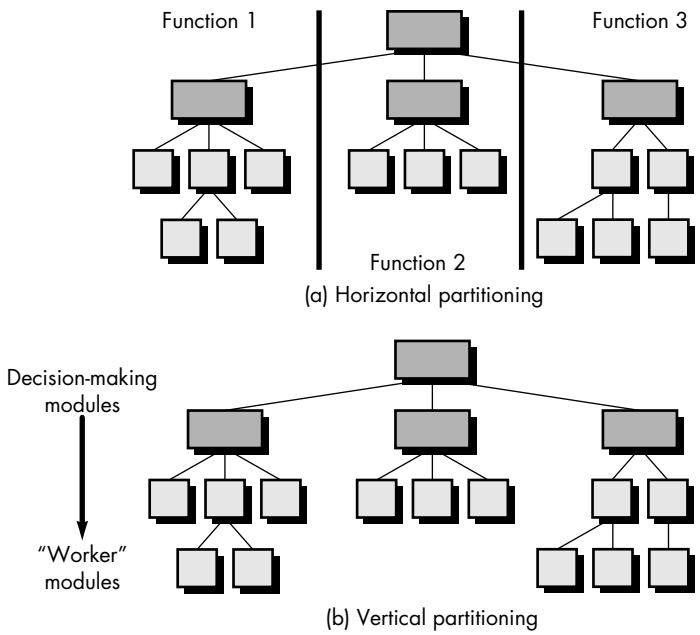

FIGURE 13.4 Structural partitioning

Vertical partitioning (Figure 13.4b), often called *factoring,* suggests that control (decision making) and work should be distributed top-down in the program structure. Top-level modules should perform control functions and do little actual processing work. Modules that reside low in the structure should be the workers, performing all input, computation, and output tasks.

KEY POINT

"Worker" modules tend to change more frequently than control modules. By placing the workers low in the structure, side effects (due to change) are reduced.

The nature of change in program structures justifies the need for vertical partitioning. Referring to Figure 13.4b, it can be seen that a change in a control module (high in the structure) will have a higher probability of propagating side effects to modules that are subordinate to it. A change to a worker module, given its low level in the structure, is less likely to cause the propagation of side effects. In general, changes to computer programs revolve around changes to input, computation or transformation, and output. The overall control structure of the program (i.e., its basic behavior is far less likely to change). For this reason vertically partitioned structures are less likely to be susceptible to side effects when changes are made and will therefore be more maintainable—a key quality factor.

13.4.7 Data Structure

Data structure is a representation of the logical relationship among individual elements of data. Because the structure of information will invariably affect the final procedural design, data structure is as important as program structure to the representation of software architecture.

Data structure dictates the organization, methods of access, degree of associativity, and processing alternatives for information. Entire texts (e.g., [AHO83], [KRU84], [GAN89]) have been dedicated to these topics, and a complete discussion is beyond the scope of this book. However, it is important to understand the classic methods available for organizing information and the concepts that underlie information hierarchies.

Quote:

"The order and connection of ideas is the same as the order and connection of things."

Baruch Spinoza

The organization and complexity of a data structure are limited only by the ingenuity of the designer. There are, however, a limited number of classic data structures that form the building blocks for more sophisticated structures.

A *scalar item* is the simplest of all data structures. As its name implies, a scalar item represents a single element of information that may be addressed by an identifier; that is, access may be achieved by specifying a single address in memory. The size and format of a scalar item may vary within bounds that are dictated by a programming language. For example, a scalar item may be a logical entity one bit long, an integer or floating point number that is 8 to 64 bits long, or a character string that is hundreds or thousands of bytes long.

When scalar items are organized as a list or contiguous group, a *sequential vector* is formed. Vectors are the most common of all data structures and open the door to variable indexing of information.

When the sequential vector is extended to two, three, and ultimately, an arbitrary number of dimensions, an *n-dimensional space* is created. The most common *n*-dimensional space is the two-dimensional matrix. In many programming languages, an *n*-dimensional space is called an *array*.

Items, vectors, and spaces may be organized in a variety of formats. A *linked list* is a data structure that organizes noncontiguous scalar items, vectors, or spaces in a manner (called *nodes*) that enables them to be processed as a list. Each node contains the appropriate data organization (e.g., a vector) and one or more pointers that indicate the address in storage of the next node in the list. Nodes may be added at any point in the list by redefining pointers to accommodate the new list entry.

ADVICE

Spend at least as much time designing data structures as you intend to spend designing the algorithms to manipulate them. If you do, you'll save time in the long run.

Other data structures incorporate or are constructed using the fundamental data structures just described. For example, a *hierarchical data structure* is implemented using multilinked lists that contain scalar items, vectors, and possibly, *n*-dimensional spaces. A hierarchical structure is commonly encountered in applications that require information categorization and associativity.

It is important to note that data structures, like program structure, can be represented at different levels of abstraction. For example, a stack is a conceptual model of a data structure that can be implemented as a vector or a linked list. Depending on the level of design detail, the internal workings of a **stack** may or may not be specified.

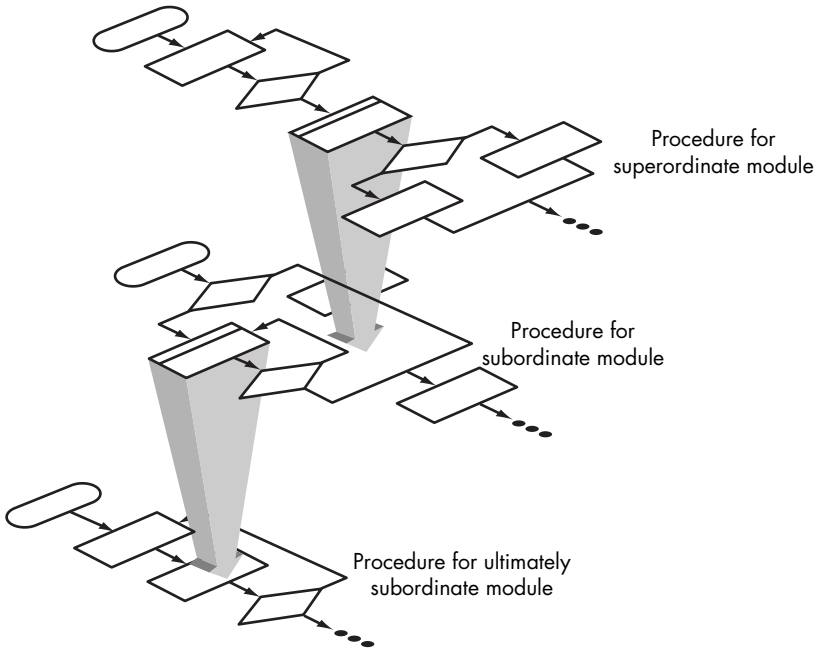

FIGURE 13.5
Procedure is layered

13.4.8 Software Procedure

Program structure defines control hierarchy without regard to the sequence of processing and decisions. Software procedure focuses on the processing details of each module individually. Procedure must provide a precise specification of processing, including sequence of events, exact decision points, repetitive operations, and even data organization and structure.

There is, of course, a relationship between structure and procedure. The processing indicated for each module must include a reference to all modules subordinate to the module being described. That is, a procedural representation of software is layered as illustrated in Figure 13.5.[6]

13.4.9 Information Hiding

The concept of modularity leads every software designer to a fundamental question: "How do we decompose a software solution to obtain the best set of modules?" The principle of *information hiding* [PAR72] suggests that modules be

6 This is not true for all architectural styles. For example, hierarchical layering of procedure is not encountered in object-oriented architectures.

"characterized by design decisions that (each) hides from all others." In other words, modules should be specified and designed so that information (procedure and data) contained within a module is inaccessible to other modules that have no need for such information.

Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module [ROS75].

The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later, during software maintenance. Because most data and procedure are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.

13.5 EFFECTIVE MODULAR DESIGN

All the fundamental design concepts described in the preceding section serve to precipitate modular designs. In fact, modularity has become an accepted approach in all engineering disciplines. A modular design reduces complexity (see Section 13.4.3), facilitates change (a critical aspect of software maintainability), and results in easier implementation by encouraging parallel development of different parts of a system.

13.5.1 Functional Independence

The concept of *functional independence* is a direct outgrowth of modularity and the concepts of abstraction and information hiding. In landmark papers on software design Parnas [PAR72] and Wirth [WIR71] allude to refinement techniques that enhance module independence. Later work by Stevens, Myers, and Constantine [STE74] solidified the concept.

Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules. Stated another way, we want to design software so that each module addresses a specific subfunction of requirements and has a simple interface when viewed from other parts of the program structure. It is fair to ask why independence is important. Software with effective modularity, that is, independent modules, is easier to develop because function may be compartmentalized and interfaces are simplified (consider the ramifications when development is conducted by a team). Independent modules are easier to maintain (and test) because secondary effects caused by design or code modification are limited, error propagation is reduced, and reusable modules are possible. To summarize, functional independence is a key to good design, and design is the key to software quality.

A module is "single minded" if you can describe it with a simple sentence—subject, predicate, object.

Independence is measured using two qualitative criteria: cohesion and coupling. *Cohesion* is a measure of the relative functional strength of a module. *Coupling* is a measure of the relative interdependence among modules.

13.5.2 Cohesion

Cohesion is a natural extension of the information hiding concept described in Section 13.4.9. A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.

Cohesion is a qualitative indication of the degree to which a module focuses on just one thing.

Cohesion may be represented as a "spectrum." We always strive for high cohesion, although the mid-range of the spectrum is often acceptable. The scale for cohesion is nonlinear. That is, low-end cohesiveness is much "worse" than middle range, which is nearly as "good" as high-end cohesion. In practice, a designer need not be concerned with categorizing cohesion in a specific module. Rather, the overall concept should be understood and low levels of cohesion should be avoided when modules are designed.

At the low (undesirable) end of the spectrum, we encounter a module that performs a set of tasks that relate to each other loosely, if at all. Such modules are termed *coincidentally cohesive.* A module that performs tasks that are related logically (e.g., a module that produces all output regardless of type) is *logically cohesive.* When a module contains tasks that are related by the fact that all must be executed with the same span of time, the module exhibits *temporal cohesion.*

As an example of low cohesion, consider a module that performs error processing for an engineering analysis package. The module is called when computed data exceed prespecified bounds. It performs the following tasks: (1) computes supplementary data based on original computed data, (2) produces an error report (with graphical content) on the user's workstation, (3) performs follow-up calculations requested by the user, (4) updates a database, and (5) enables menu selection for subsequent processing. Although the preceding tasks are loosely related, each is an independent functional entity that might best be performed as a separate module. Combining the functions into a single module can serve only to increase the likelihood of error propagation when a modification is made to one of its processing tasks.

If you concentrate on only one thing during component-level design, make it cohesion.

Moderate levels of cohesion are relatively close to one another in the degree of module independence. When processing elements of a module are related and must be executed in a specific order, *procedural cohesion* exists. When all processing elements concentrate on one area of a data structure, *communicational cohesion* is present. High cohesion is characterized by a module that performs one distinct procedural task.

As we have already noted, it is unnecessary to determine the precise level of cohesion. Rather it is important to strive for high cohesion and recognize low cohesion so that software design can be modified to achieve greater functional independence.

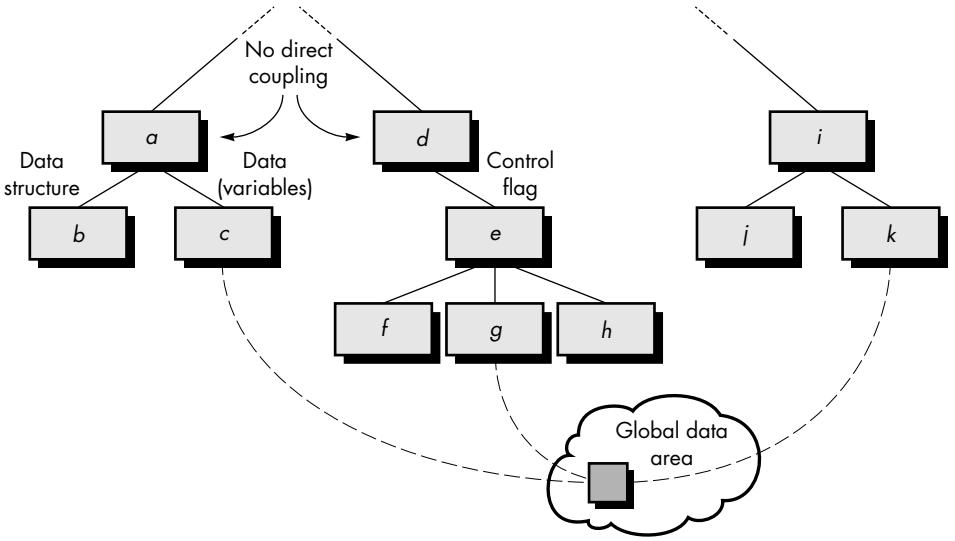

FIGURE 13.6 Types of coupling

13.5.3 Coupling

Coupling is a measure of interconnection among modules in a software structure. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

KEY POINT

Coupling is a qualitative indication of the degree to which a module is connected to other modules and to the outside world.

In software design, we strive for lowest possible coupling. Simple connectivity among modules results in software that is easier to understand and less prone to a "ripple effect" [STE74], caused when errors occur at one location and propagate through a system.

Figure 13.6 provides examples of different types of module coupling. Modules *a* and *d* are subordinate to different modules. Each is unrelated and therefore no direct coupling occurs. Module *c* is subordinate to module *a* and is accessed via a conventional argument list, through which data are passed. As long as a simple argument list is present (i.e., simple data are passed; a one-to-one correspondence of items exists), low coupling (called *data coupling*) is exhibited in this portion of structure. A variation of data coupling, called *stamp coupling,* is found when a portion of a data structure (rather than simple arguments) is passed via a module interface. This occurs between modules *b* and *a*.

ADVICE

Highly coupled systems lead to debugging nightmares. Avoid them.

At moderate levels, coupling is characterized by passage of control between modules. *Control coupling* is very common in most software designs and is shown in Figure 13.6 where a "control flag" (a variable that controls decisions in a subordinate or superordinate module) is passed between modules *d* and *e*.

Relatively high levels of coupling occur when modules are tied to an environment external to software. For example, I/O couples a module to specific devices, formats, and communication protocols. *External coupling* is essential, but should be limited to

a small number of modules with a structure. High coupling also occurs when a number of modules reference a global data area. *Common coupling,* as this mode is called, is shown in Figure 13.6. Modules *c, g,* and *k* each access a data item in a global data area (e.g., a disk file or a globally accessible memory area). Module *c* initializes the item. Later module *g* recomputes and updates the item. Let's assume that an error occurs and *g* updates the item incorrectly. Much later in processing module, *k* reads the item, attempts to process it, and fails, causing the software to abort. The apparent cause of abort is module *k*; the actual cause, module *g.* Diagnosing problems in structures with considerable common coupling is time consuming and difficult. However, this does not mean that the use of global data is necessarily "bad." It does mean that a software designer must be aware of potential consequences of common coupling and take special care to guard against them.

The highest degree of coupling, *content coupling,* occurs when one module makes use of data or control information maintained within the boundary of another module. Secondarily, content coupling occurs when branches are made into the middle of a module. This mode of coupling can and should be avoided.

The coupling modes just discussed occur because of design decisions made when structure was developed. Variants of external coupling, however, may be introduced during coding. For example, *compiler coupling* ties source code to specific (and often nonstandard) attributes of a compiler; *operating system* (OS) *coupling* ties design and resultant code to operating system "hooks" that can create havoc when OS changes occur.

13.6 DESIGN HEURISTICS FOR EFFECTIVE MODULARITY

Once program structure has been developed, effective modularity can be achieved by applying the design concepts introduced earlier in this chapter. The program structure can be manipulated according to the following set of heuristics:

Quote:

"The notion that good [design] techniques restrict creativity is like saying that an artist can paint without learning the details of form or a musician does not need knowledge of music theory."

Marvin Zelkowitz et al.

1. *Evaluate the "first iteration" of the program structure to reduce coupling and improve cohesion.* Once the program structure has been developed, modules may be exploded or imploded with an eye toward improving module independence. An *exploded module* becomes two or more modules in the final program structure. An *imploded module* is the result of combining the processing implied by two or more modules.

   An exploded module often results when common processing exists in two or more modules and can be redefined as a separate cohesive module. When high coupling is expected, modules can sometimes be imploded to reduce passage of control, reference to global data, and interface complexity.

2. *Attempt to minimize structures with high fan-out; strive for fan-in as depth increases.* The structure shown inside the cloud in Figure 13.7 does not make effective use of factoring. All modules are "pancaked" below a single control

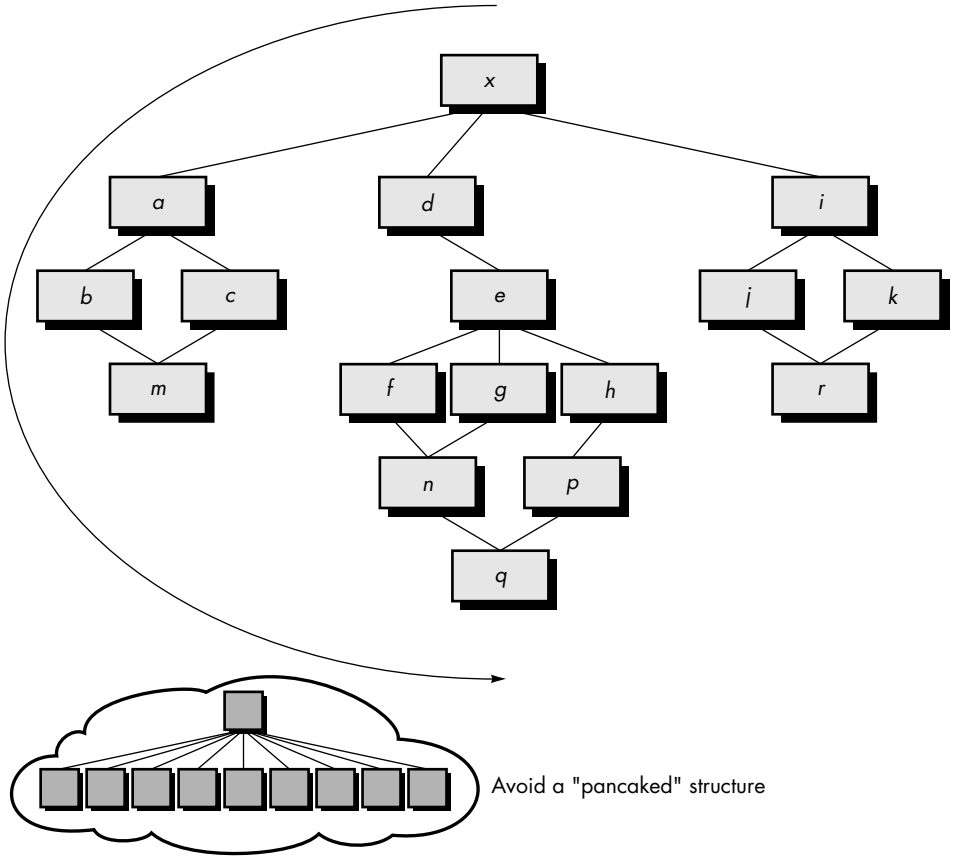

FIGURE 13.7
Program structures

module. In general, a more reasonable distribution of control is shown in the upper structure. The structure takes an oval shape, indicating a number of layers of control and highly utilitarian modules at lower levels.

3. *Keep the scope of effect of a module within the scope of control of that module.* The *scope of effect* of module *e* is defined as all other modules that are affected by a decision made in module *e.* The *scope of control* of module *e* is all modules that are subordinate and ultimately subordinate to module *e.* Referring to Figure 13.7, if module *e* makes a decision that affects module *r,* we have a violation of this heuristic, because module *r* lies outside the scope of control of module *e.*

4. *Evaluate module interfaces to reduce complexity and redundancy and improve consistency.* Module interface complexity is a prime cause of software errors. Interfaces should be designed to pass information simply and should be consistent with the function of a module. Interface inconsistency (i.e., seemingly

unrelated data passed via an argument list or other technique) is an indication of low cohesion. The module in question should be reevaluated.

5. *Define modules whose function is predictable, but avoid modules that are overly restrictive.* A module is predictable when it can be treated as a black box; that is, the same external data will be produced regardless of internal processing details.[7] Modules that have internal "memory" can be unpredictable unless care is taken in their use.

   A module that restricts processing to a single subfunction exhibits high cohesion and is viewed with favor by a designer. However, a module that arbitrarily restricts the size of a local data structure, options within control flow, or modes of external interface will invariably require maintenance to remove such restrictions.

6. *Strive for "controlled entry" modules* by *avoiding "pathological connections."* This design heuristic warns against content coupling. Software is easier to understand and therefore easier to maintain when module interfaces are constrained and controlled. *Pathological connection* refers to branches or references into the middle of a module.

A detailed report on software design methods including a discussion of all design concepts and principles found in this chapter can be obtained at **www.dacs.dtic.mil/techs/design/Design.ToC.html**

13.7 THE DESIGN MODEL

The design principles and concepts discussed in this chapter establish a foundation for the creation of the design model that encompasses representations of data, architecture, interfaces, and components. Like the analysis model before it, each of these design representations is tied to the others, and all can be traced back to software requirements.

In Figure 13.1, the design model was represented as a pyramid. The symbolism of this shape is important. A pyramid is an extremely stable object with a wide base and a low center of gravity. Like the pyramid, we want to create a software design that is stable. By establishing a broad foundation using data design, a stable mid-region with architectural and interface design, and a sharp point by applying component-level design, we create a design model that is not easily "tipped over" by the winds of change.

It is interesting to note that some programmers continue to design implicitly, conducting component-level design as they code. This is akin to taking the design pyramid and standing it on its point—an extremely unstable design results. The smallest change may cause the pyramid (and the program) to topple.

The methods that lead to the creation of the design model are presented in Chapters 14, 15, 16, and 22 (for object-oriented systems). Each method enables the designer

7 A "black box" module is a procedural abstraction.

to create a stable design that conforms to fundamental concepts that lead to high-quality software.

13.8 DESIGN DOCUMENTATION

The *Design Specification* addresses different aspects of the design model and is completed as the designer refines his representation of the software. First, the overall scope of the design effort is described. Much of the information presented here is derived from the *System Specification* and the analysis model (*Software Requirements Specification*).

Next, the data design is specified. Database structure, any external file structures, internal data structures, and a cross reference that connects data objects to specific files are all defined.

Software Design Specification

The architectural design indicates how the program architecture has been derived from the analysis model. In addition, structure charts are used to represent the module hierarchy (if applicable).

The design of external and internal program interfaces is represented and a detailed design of the human/machine interface is described. In some cases, a detailed prototype of a GUI may be represented.

Components—separately addressable elements of software such as subroutines, functions, or procedures—are initially described with an English-language processing narrative. The processing narrative explains the procedural function of a component (module). Later, a procedural design tool is used to translate the narrative into a structured description.

The *Design Specification* contains a requirements cross reference. The purpose of this cross reference (usually represented as a simple matrix) is (1) to establish that all requirements are satisfied by the software design and (2) to indicate which components are critical to the implementation of specific requirements.

The first stage in the development of test documentation is also contained in the design document. Once program structure and interfaces have been established, we can develop guidelines for testing of individual modules and integration of the entire package. In some cases, a detailed specification of test procedures occurs in parallel with design. In such cases, this section may be deleted from the *Design Specification.*

Design constraints, such as physical memory limitations or the necessity for a specialized external interface, may dictate special requirements for assembling or packaging of software. Special considerations caused by the necessity for program overlay, virtual memory management, high-speed processing, or other factors may cause modification in design derived from information flow or structure. In addition, this section describes the approach that will be used to transfer software to a customer site.

The final section of the *Design Specification* contains supplementary data. Algorithm descriptions, alternative procedures, tabular data, excerpts from other docu-

ments, and other relevant information are presented as a special note or as a separate appendix. It may be advisable to develop a *Preliminary Operations/Installation Manual* and include it as an appendix to the design document.

13.8 SUMMARY

Design is the technical kernel of software engineering. During design, progressive refinements of data structure, architecture, interfaces, and procedural detail of software components are developed, reviewed, and documented. Design results in representations of software that can be assessed for quality.

A number of fundamental software design principles and concepts have been proposed over the past four decades. Design principles guide the software engineer as the design process proceeds. Design concepts provide basic criteria for design quality.

Modularity (in both program and data) and the concept of abstraction enable the designer to simplify and reuse software components. Refinement provides a mechanism for representing successive layers of functional detail. Program and data structure contribute to an overall view of software architecture, while procedure provides the detail necessary for algorithm implementation. Information hiding and functional independence provide heuristics for achieving effective modularity.

We conclude our discussion of design fundamentals with the words of Glenford Myers [MYE78]:

> We try to solve the problem by rushing through the design process so that enough time will be left at the end of the project to uncover errors that were made because we rushed through the design process . . .

The moral is this: Don't rush through it! Design is worth the effort.

We have not concluded our discussion of design. In the chapters that follow, design methods are discussed. These methods, combined with the fundamentals in this chapter, form the basis for a complete view of software design.

REFERENCES

[AHO83] Aho, A.V., J. Hopcroft, and J. Ullmann, *Data Structures and Algorithms,* Addison-Wesley, 1983.

[BAS98] Bass, L., P. Clements, and R. Kazman, *Software Architecture in Practice,* Addison-Wesley, 1998.

[BEL81] Belady, L., Foreword to *Software Design: Methods and Techniques* (L.J. Peters, author), Yourdon Press, 1981.

[BRO98] Brown, W.J., et al., *Anti-Patterns,* Wiley, 1998.

[BUS96] Buschmann, F. et al., *Pattern-Oriented Software Architecture,* Wiley, 1996.

[DAH72] Dahl, O., E. Dijkstra, and C. Hoare, *Structured Programming,* Academic Press, 1972.

[DAV95] Davis, A., *201 Principles of Software Development,* McGraw-Hill, 1995.
[DEN73] Dennis, J., "Modularity," in *Advanced Course on Software Engineering* (F.L. Bauer, ed.), Springer-Verlag, New York, 1973, pp. 128–182.
[GAM95] Gamma, E. et al., *Design Patterns,* Addison-Wesley, 1995.
[GAN89] Gonnet, G., *Handbook of Algorithms and Data Structures,* 2nd ed., Addison-Wesley, 1989.
[GAR95] Garlan, D. and M. Shaw, "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering,* vol. I (V. Ambriola and G. Tortora, eds.), World Scientific Publishing Company, 1995.
[JAC75] Jackson, M.A., *Principles of Program Design,* Academic Press, 1975.
[JAC83] Jackson, M.A., *System Development,* Prentice-Hall, 1983.
[JAC92] Jacobson, I., *Object-Oriented Software Engineering,* Addison-Wesley, 1992.
[KAI83] Kaiser, S.H., *The Design of Operating Systems for Small Computer Systems,* Wiley-Interscience, 1983, pp. 594 ff.
[KRU84] Kruse, R.L., *Data Structures and Program Design,* Prentice-Hall, 1984.
[MCG91] McGlaughlin, R., "Some Notes on Program Design," *Software Engineering Notes,* vol. 16, no. 4, October 1991, pp. 53–54.
[MEY88] Meyer, B., *Object-Oriented Software Construction,* Prentice-Hall, 1988.
[MIL72] Mills, H.D., "Mathematical Foundations for Structured Programming," Technical Report FSC 71-6012, IBM Corp., Federal Systems Division, Gaithersburg, Maryland, 1972.
[MYE78] Myers, G., *Composite Structured Design,* Van Nostrand,1978.
[ORR77] Orr, K.T., *Structured Systems Development,* Yourdon Press, 1977.
[PAR72] Parnas, D.L., "On Criteria to Be Used in Decomposing Systems into Modules," *CACM,* vol. 14, no. 1, April 1972, pp. 221–227.
[ROS75] Ross, D., J. Goodenough, and C. Irvine, "Software Engineering: Process, Principles and Goals," *IEEE Computer,* vol. 8, no. 5, May 1975.
[SHA95a] Shaw, M. and D. Garlan, "Formulations and Formalisms in Software Architecture," *Volume 1000—Lecture Notes in Computer Science,* Springer-Verlag, 1995.
[SHA95b] Shaw, M. et al., "Abstractions for Software Architecture and Tools to Support Them," *IEEE Trans. Software Engineering,* vol. SE-21, no. 4, April 1995, pp. 314–335.
[SHA96] Shaw, M. and D. Garlan, *Software Architecture,* Prentice-Hall, 1996.
[SOM89] Sommerville, I., *Software Engineering,* 3rd ed., Addison-Wesley, 1989.
[STE74] Stevens, W., G. Myers, and L. Constantine, "Structured Design," *IBM Systems Journal,* vol. 13, no. 2, 1974, pp. 115–139.
[WAR74] Warnier, J., *Logical Construction of Programs,* Van Nostrand-Reinhold, 1974.
[WAS83] Wasserman, A., "Information System Design Methodology," in *Software Design Techniques* (P. Freeman and A. Wasserman, eds.), 4th ed., IEEE Computer Society Press, 1983, p. 43.
[WIR71] Wirth, N., "Program Development by Stepwise Refinement," *CACM,* vol. 14, no. 4, 1971, pp. 221–227.
[YOU79] Yourdon, E., and L. Constantine, *Structured Design,* Prentice-Hall, 1979.

PROBLEMS AND POINTS TO PONDER

13.1. Do you design software when you "write" a program? What makes software design different from coding?

13.2. Develop three additional design principles to add to those noted in Section 13.3.

13.3. Provide examples of three data abstractions and the procedural abstractions that can be used to manipulate them.

13.4. Apply a "stepwise refinement approach" to develop three different levels of procedural abstraction for one or more of the following programs:

a. Develop a check writer that, given a numeric dollar amount, will print the amount in words normally required on a check.
b. Iteratively solve for the roots of a transcendental equation.
c. Develop a simple round-robin scheduling algorithm for an operating system.

13.5. Is there a case when Expression (13-2) may not be true? How might such a case affect the argument for modularity?

13.6. When should a modular design be implemented as monolithic software? How can this be accomplished? Is performance the only justification for implementation of monolithic software?

13.7. Develop at least five levels of abstraction for one of the following software problems:

a. A video game of your choosing.
b. A 3D transformation package for computer graphics applications.
c. A programming language interpreter.
d. A two degree of freedom robot controller.
e. Any problem mutually agreeable to you and your instructor.

As the level of abstraction decreases, your focus may narrow so that at the last level (source code) only a single task need be described.

13.8. Obtain the original Parnas paper [PAR72] and summarize the software example that he uses to illustrate decomposition of a system into modules. How is information hiding used to achieve the decomposition?

13.9. Discuss the relationship between the concept of information hiding as an attribute of effective modularity and the concept of module independence.

13.10. Review some of your recent software development efforts and grade each module (on a scale of 1—low to 7—high). Bring in samples of your best and worst work.

13.11. A number of high-level programming languages support the internal procedure as a modular construct. How does this construct affect coupling? information hiding?

13.12. How are the concepts of coupling and software portability related? Provide examples to support your discussion.

13.13. Discuss how structural partitioning can help to make software more maintainable.

13.14. What is the purpose of developing a program structure that is factored?

13.15. Describe the concept of information hiding in your own words.

13.16. Why is it a good idea to keep the scope of effect of a module within its scope of control?

FURTHER READINGS AND INFORMATION SOURCES

Donald Norman has written two books (*The Design of Everyday Things,* Doubleday, 1990, and *The Psychology of Everyday Things,* HarperCollins, 1988) that have become classics in the design literature and "must" reading for anyone who designs anything that humans use. Adams (*Conceptual Blockbusting,* 3rd ed., Addison-Wesley, 1986) has written a book that is essential reading for designers who want to broaden their way of thinking. Finally, a classic text by Polya (*How to Solve It,* 2nd ed., Princeton University Press, 1988) provides a generic problem-solving process that can help software designers when they are faced with complex problems.

Following in the same tradition, Winograd et al. (*Bringing Design to Software,* Addison-Wesley, 1996) discusses software designs that work, those that don't, and why. A fascinating book edited by Wixon and Ramsey (*Field Methods Casebook for Software Design,* Wiley, 1996) suggests field research methods (much like those used by anthropologists) to understand how end-users do the work they do and then design software that meets their needs. Beyer and Holtzblatt (*Contextual Design: A Customer-Centered Approach to Systems Designs,* Academic Press, 1997) offer another view of software design that integrates the customer/user into every aspect of the software design process.

McConnell (*Code Complete,* Microsoft Press, 1993) presents an excellent discussion of the practical aspects of designing high-quality computer software. Robertson (*Simple Program Design*, 3rd ed., Boyd and Fraser Publishing, 1999) presents an introductory discussion of software design that is useful for those beginning their study of the subject.

An excellent historical survey of important papers on software design is contained in an anthology edited by Freeman and Wasserman (*Software Design Techniques,* 4th ed., IEEE, 1983). This tutorial reprints many of the classic papers that have formed the basis for current trends in software design. Good discussions of software design fundamentals can be found in books by Myers [MYE78], Peters (*Software Design: Methods and Techniques,* Yourdon Press, 1981), Macro (*Software Engineering: Concepts and*

Management, Prentice-Hall, 1990), and Sommerville (*Software Engineering,* Addison-Wesley, 5th ed., 1996).

Mathematically rigorous treatments of computer software and design fundamentals may be found in books by Jones (*Software Development: A Rigorous Approach,* Prentice-Hall, 1980), Wulf (*Fundamental Structures of Computer Science,* Addison-Wesley, 1981), and Brassard and Bratley (*Fundamental of Algorithmics,* Prentice-Hall, 1995). Each of these texts helps to supply a necessary theoretical foundation for our understanding of computer software.

Kruse (*Data Structures and Program Design,* Prentice-Hall, 1994) and Tucker et al. *(Fundamentals of Computing II: Abstraction, Data Structures, and Large Software Systems,* McGraw-Hill, 1995) present worthwhile information on data structures. Measures of design quality, presented from both the technical and management perspectives, are considered by Card and Glass (*Measuring Software Design Quality,* Prentice-Hall, 1990).

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/design-principles.mhtml

CHAPTER

14 ARCHITECTURAL DESIGN

KEY CONCEPTS

Design has been described as a multistep process in which representations of data and program structure, interface characteristics, and procedural detail are synthesized from information requirements. This description is extended by Freeman [FRE80]:

> [D]esign is an activity concerned with making major decisions, often of a structural nature. It shares with programming a concern for abstracting information representation and processing sequences, but the level of detail is quite different at the extremes. Design builds coherent, well planned representations of programs that concentrate on the interrelationships of parts at the higher level and the logical operations involved at the lower levels . . .

As we have noted in the preceding chapter, design is information driven. Software design methods are derived from consideration of each of the three domains of the analysis model. The data, functional, and behavioral domains serve as a guide for the creation of the software design.

Methods required to create "coherent, well planned representations" of the data and architectural layers of the design model are presented in this chapter. The objective is to provide a systematic approach for the derivation of the architectural design—the preliminary blueprint from which software is constructed.

QUICK LOOK

What is it? Architectural design represents the structure of data and program components that are required to build a computer-based system. It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system.

Who does it? Although a software engineer can design both data and architecture, the job is often allocated to specialists when large, complex systems are to be built. A database or data warehouse designer creates the data architecture for a system. The "system architect" selects an appropriate architectural style for the requirements derived during system engineering and software requirements analysis.

Why is it important? In the *Quick Look* for the last chapter, we asked: "You wouldn't attempt to build a house without a blueprint, would you?" You also wouldn't begin drawing blueprints by sketching the plumbing layout for the house. You'd need to look at the big picture—the house itself—before you worry about details. That's what architectural design does—it provides you with the big picture and ensures that you've got it right.

What are the steps? Architectural design begins with data design and then proceeds to the derivation of one or more representations of the

QUICK LOOK

architectural structure of the system. Alternative architectural styles or patterns are analyzed to derive the structure that is best suited to customer requirements and quality attributes. Once an alternative has been selected, the architecture is elaborated using an architectural design method.

What is the work product? An architecture model encompassing data architecture and program structure is created during architectural design. In addition, component properties and relationships (interactions) are described.

How do I ensure that I've done it right? At each stage, software design work products are reviewed for clarity, correctness, completeness, and consistency with requirements and with one another.

14.1 SOFTWARE ARCHITECTURE

In their landmark book on the subject, Shaw and Garlan [SHA96] discuss software architecture in the following manner:

> Ever since the first program was divided into modules, software systems have had architectures, and programmers have been responsible for the interactions among the modules and the global properties of the assemblage. Historically, architectures have been implicit—accidents of implementation, or legacy systems of the past. Good software developers have often adopted one or several architectural patterns as strategies for system organization, but they use these patterns informally and have no means to make them explicit in the resulting system.

Today, effective software architecture and its explicit representation and design have become dominant themes in software engineering.

14.1.1 What Is Architecture?

When we discuss the architecture of a building, many different attributes come to mind. At the most simplistic level, we consider the overall shape of the physical structure. But in reality, architecture is much more. It is the manner in which the various components of the building are integrated to form a cohesive whole. It is the way in which the building fits into its environment and meshes with other buildings in its vicinity. It is the degree to which the building meets its stated purpose and satisfies the needs of its owner. It is the aesthetic feel of the structure—the visual impact of the building—and the way textures, colors, and materials are combined to create the external facade and the internal "living environment." It is small details—the design of lighting fixtures, the type of flooring, the placement of wall hangings, the list is almost endless. And finally, it is art.

A useful list of software architecture resources can be found at **www2.umassd.edu/SECenter/SAResources.html**

But what about software architecture? Bass, Clements, and Kazman [BAS98] define this elusive term in the following way:

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

The architecture is not the operational software. Rather, it is a representation that enables a software engineer to (1) analyze the effectiveness of the design in meeting its stated requirements, (2) consider architectural alternatives at a stage when making design changes is still relatively easy, and (3) reducing the risks associated with the construction of the software.

Quote:

"The architecture of a system is a comprehensive framework that describes its form and structure—its components and how they fit together."

Jerrold Grochow

This definition emphasizes the role of "software components" in any architectural representation. In the context of architectural design, a software component can be something as simple as a program module, but it can also be extended to include databases and "middleware" that enable the configuration of a network of clients and servers. The properties of components are those characteristics that are necessary to an understanding of how the components interact with other components. At the architectural level, internal properties (e.g., details of an algorithm) are not specified. The relationships between components can be as simple as a procedure call from one module to another or as complex as a database access protocol.

In this book the design of software architecture considers two levels of the design pyramid (Figure 13.1)—data design and architectural design. In the context of the preceding discussion, data design enables us to represent the data component of the architecture. Architectural design focuses on the representation of the structure of software components, their properties, and interactions.

14.1.2 Why Is Architecture Important?

In a book dedicated to software architecture, Bass and his colleagues {BAS98] identify three key reasons that software architecture is important:

KEY POINT

The architectural model provides a Gestalt view of a system, allowing the software engineer to examine it as a whole.

- Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.
- The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.
- Architecture "constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together" [BAS98].

The architectural design model and the architectural patterns contained within it are transferrable. That is, architecture styles and patterns (Section 14.3.1) can be applied to the design of other systems and represent a set of abstractions that enable software engineers to describe architecture in predictable ways.

14.2 DATA DESIGN

Like other software engineering activities, *data design* (sometimes referred to as *data architecting*) creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data). This data model is then refined into progressively more implementation-specific representations that can be processed by the computer-based system. In many software applications, the architecture of the data will have a profound influence on the architecture of the software that must process it.

The structure of data has always been an important part of software design. At the program component level, the design of data structures and the associated algorithms required to manipulate them is essential to the creation of high-quality applications. At the application level, the translation of a data model (derived as part of requirements engineering) into a database is pivotal to achieving the business objectives of a system. At the business level, the collection of information stored in disparate databases and reorganized into a "data warehouse" enables data mining or knowledge discovery that can have an impact on the success of the business itself. In every case, data design plays an important role.

14.2.1 Data Modeling, Data Structures, Databases, and the Data Warehouse

Quote:

"Data quality is the difference between a data warehouse and a data garbage dump."

Jarrett Rosenberg

The data objects defined during software requirements analysis are modeled using entity/relationship diagrams and the data dictionary (Chapter 12). The data design activity translates these elements of the requirements model into data structures at the software component level and, when necessary, a database architecture at the application level.

In years past, data architecture was generally limited to data structures at the program level and databases at the application level. But today, businesses large and small are awash in data. It is not unusual for even a moderately sized business to have dozens of databases serving many applications encompassing hundreds of gigabytes of data. The challenge for a business has been to extract useful information from this data environment, particularly when the information desired is cross-functional (e.g., information that can be obtained only if specific marketing data are cross-correlated with product engineering data).

WebRef

Up-to-date information on data warehouse technologies can be obtained at **www.data warehouse.com**

To solve this challenge, the business IT community has developed *data mining* techniques, also called *knowledge discovery in databases* (KDD), that navigate through existing databases in an attempt to extract appropriate business-level information. However, the existence of multiple databases, their different structures, the degree of detail contained with the databases, and many other factors make data mining difficult within an existing database environment. An alternative solution, called a *data warehouse,* adds an additional layer to the data architecture.

A data warehouse is a separate data environment that is not directly integrated with day-to-day applications but encompasses all data used by a business [MAT96]. In a sense, a data warehouse is a large, independent database that encompasses some, but not all, of the data that are stored in databases that serve the set of applications required by a business. But many characteristics differentiate a data warehouse from the typical database [INM95]:

Subject orientation. A data warehouse is organized by major business subjects, rather than by business process or function. This leads to the exclusion of data that may be necessary for a particular business function but is generally not necessary for data mining.

Integration. Regardless of the source, the data exhibit consistent naming conventions, units and measures, encoding structures, and physical attributes, even when inconsistency exists across different application-oriented databases.

Time variancy. For a transaction-oriented application environment, data are accurate at the moment of access and for a relatively short time span (typically 60 to 90 days) before access. For a data warehouse, however, data can be accessed at a specific moment in time (e.g., customers contacted on the date that a new product was announced to the trade press). The typical time horizon for a data warehouse is five to ten years.

Nonvolatility. Unlike typical business application databases that undergo a continuing stream of changes (inserts, deletes, updates), data are loaded into the warehouse, but after the original transfer, the data do not change.

A data warehouse encompasses all data that are used by a business. The intent is to enable access to "knowledge" that might not be otherwise available.

These characteristics present a unique set of design challenges for a data architect.

A detailed discussion of the design of data structures, databases, and the data warehouse is best left to books dedicated to these subjects (e.g., [PRE98], [DAT95], [KIM98]). The interested reader should see the Further Readings and Information Sources section of this chapter for additional references.

14.2.2 Data Design at the Component Level

Data design at the component level focuses on the representation of data structures that are directly accessed by one or more software components. Wasserman [WAS80] has proposed a set of principles that may be used to specify and design such data structures. In actuality, the design of data begins during the creation of the analysis model. Recalling that requirements analysis and design often overlap, we consider the following set of principles [WAS80] for data specification:

What principles are applicable to data design?

1. *The systematic analysis principles applied to function and behavior should also be applied to data.* We spend much time and effort deriving, reviewing, and specifying functional requirements and preliminary design. Representations of data flow and content should also be developed and reviewed, data

objects should be identified, alternative data organizations should be considered, and the impact of data modeling on software design should be evaluated. For example, specification of a multiringed linked list may nicely satisfy data requirements but lead to an unwieldy software design. An alternative data organization may lead to better results.

2. *All data structures and the operations to be performed on each should be identified.* The design of an efficient data structure must take the operations to be performed on the data structure into account (e.g., see [AHO83]). For example, consider a data structure made up of a set of diverse data elements. The data structure is to be manipulated in a number of major software functions. Upon evaluation of the operations performed on the data structure, an abstract data type is defined for use in subsequent software design. Specification of the abstract data type may simplify software design considerably.
3. *A data dictionary should be established and used to define both data and program design.* The concept of a data dictionary has been introduced in Chapter 12. A data dictionary explicitly represents the relationships among data objects and the constraints on the elements of a data structure. Algorithms that must take advantage of specific relationships can be more easily defined if a dictionarylike data specification exists.
4. *Low-level data design decisions should be deferred until late in the design process.* A process of stepwise refinement may be used for the design of data. That is, overall data organization may be defined during requirements analysis, refined during data design work, and specified in detail during component-level design. The top-down approach to data design provides benefits that are analogous to a top-down approach to software design—major structural attributes are designed and evaluated first so that the architecture of the data may be established.
5. *The representation of data structure should be known only to those modules that must make direct use of the data contained within the structure.* The concept of information hiding and the related concept of coupling (Chapter 13) provide important insight into the quality of a software design. This principle alludes to the importance of these concepts as well as "the importance of separating the logical view of a data object from its physical view" [WAS80].
6. *A library of useful data structures and the operations that may be applied to them should be developed.* Data structures and operations should be viewed as a resource for software design. Data structures can be designed for reusability. A library of data structure templates (abstract data types) can reduce both specification and design effort for data.
7. *A software design and programming language should support the specification and realization of abstract data types.* The implementation of a sophisticated

data structure can be made exceedingly difficult if no means for direct specification of the structure exists in the programming language chosen for implementation.

These principles form a basis for a component-level data design approach that can be integrated into both the analysis and design activities.

14.3 ARCHITECTURAL STYLES

WebRef

The ABLE project at CMU provides useful papers and examples of architectural styles: **tom.cs.cmu.edu/able/**

When a builder uses the phrase "center hall colonial" to describe a house, most people familiar with houses in the United States will be able to conjure a general image of what the house will look like and what the floor plan is likely to be. The builder has used an *architectural style* as a descriptive mechanism to differentiate the house from other styles (e.g., A-frame, raised ranch, Cape Cod). But more important, the architectural style is also a pattern for construction. Further details of the house must be defined, its final dimensions must be specified, customized features may be added, building materials are to be be determined, but the pattern—a "center hall colonial"—guides the builder in his work.

? What is an architectural style?

The software that is built for computer-based systems also exhibits one of many architectural styles.[1] Each style describes a system category that encompasses (1) a set of *components* (e.g., a database, computational modules) that perform a function required by a system; (2) a set of *connectors* that enable "communication, coordinations and cooperation" among components; (3) *constraints* that define how components can be integrated to form the system; and (4) *semantic models* that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts [BAS98]. In the section that follows, we consider commonly used architectural patterns for software.

14.3.1 A Brief Taxonomy of Styles and Patterns

Although millions of computer-based systems have been created over the past 50 years, the vast majority can be categorized {see [SHA96], {BAS98], BUS96]) into one of a relatively small number of architectural styles:

Quote:

"The use of patterns and styles of design is pervasive in engineering disciplines."

Mary Shaw and David Garlan

Data-centered architectures. A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figure 14.1 illustrates a typical data-centered style. Client software accesses a central repository. In some cases the data repository is *passive.* That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a "blackboard" that sends notifications to client software when data of interest to the client change.

1 The terms *styles* and *patterns* are used interchangeably in this discussion.

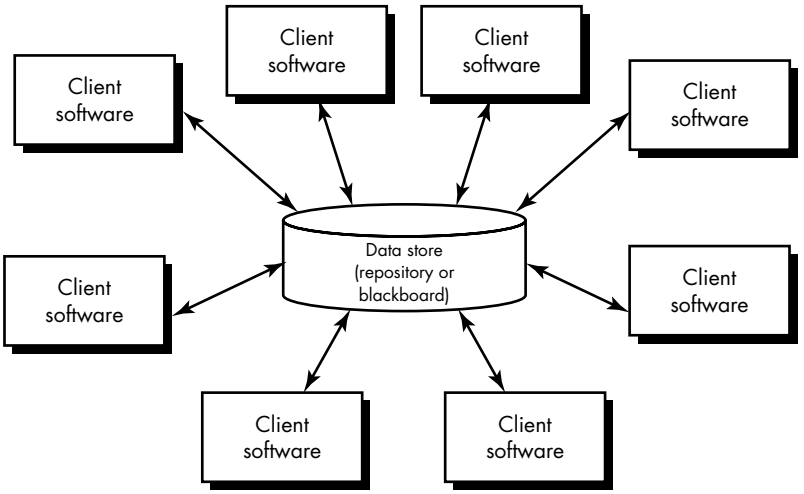

FIGURE 14.1 Data-centered architecture

Data-centered architectures promote *integrability* [BAS98]. That is, existing components can be changed and new client components can be added to the architecture without concern about other clients (because the client components operate independently). In addition, data can be passed among clients using the blackboard mechanism (i.e., the blackboard component serves to coordinate the transfer of information between clients). Client components independently execute processes.

Quote:

"A good architect is the principal keeper of the user's vision of the end product."

Norman Simenson

Data-flow architectures. This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A *pipe and filter pattern* (Figure 14.2a) has a set of components, called *filters,* connected by pipes that transmit data from one component to the next. Each filter works independently of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the working of its neighboring filters.

If the data flow degenerates into a single line of transforms, it is termed *batch sequential.* This pattern (Figure 14.2b) accepts a batch of data and then applies a series of sequential components (filters) to transform it.

Call and return architectures. This architectural style enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale. A number of substyles [BAS98] exist within this category:

- *Main program/subprogram architectures.* This classic program structure decomposes function into a control hierarchy where a "main" program invokes a number of program components, which in turn may invoke still other components. Figure 13.3 illustrates an architecture of this type.

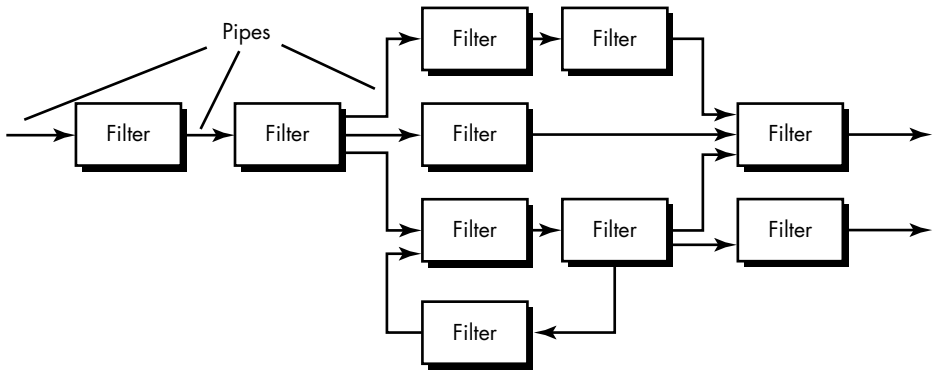

FIGURE 14.2 Data flow architectures

(a) Pipes and filters

(b) Batch sequential

- *Remote procedure call architectures*. The components of a main program/subprogram architecture are distributed across multiple computers on a network

XRef

A detailed discussion of object-oriented architectures is presented in Part Four.

Object-oriented architectures. The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components is accomplished via message passing.

Layered architectures. The basic structure of a layered architecture is illustrated in Figure 14.3. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set. At the outer layer, components service user interface operations. At the inner layer, components perform operating system interfacing. Intermediate layers provide utility services and application software functions.

These architectural styles are only a small subset of those available to the software designer.[2] Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural pattern (style) or combination of patterns (styles) that best fits those characteristics and constraints can be chosen. In

2 See [SHA96], [SHA97], [BAS98], and [BUS96] for a detailed discussion of architectural styles and patterns.

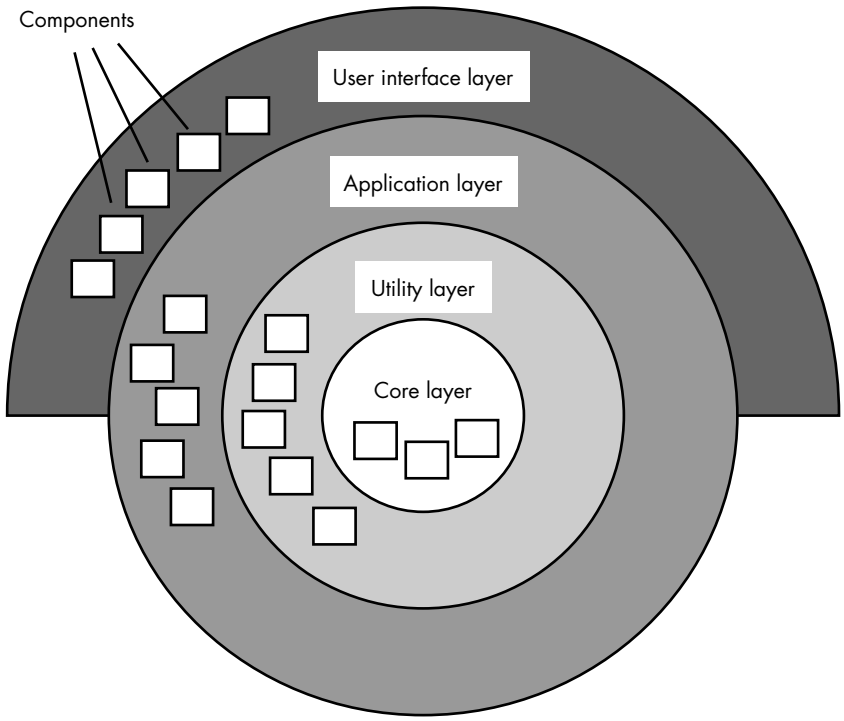

FIGURE 14.3 Layered architecture

many cases, more than one pattern might be appropriate and alternative architectural styles might be designed and evaluated.

14.3.2 Organization and Refinement

Because the design process often leaves a software engineer with a number of architectural alternatives, it is important to establish a set of design criteria that can be used to assess an architectural design that is derived. The following questions [BAS98] provide insight into the architectural style that has been derived:

How do I assess an architectural style that has been derived?

> **Control.** How is control managed within the architecture? Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy? How do components transfer control within the system? How is control shared among components? What is the control topology (i.e., the geometric form[3] that the control takes)? Is control synchronized or do components operate asynchronously?

3 A hierarchy is one geometric form, but others such as a hub and spoke control mechanism in a client/server system are also encountered.

Data. How are data communicated between components? Is the flow of data continuous, or are data objects passed to the system sporadically? What is the mode of data transfer (i.e., are data passed from one component to another or are data available globally to be shared among system components)? Do data components (e.g., a blackboard or repository) exist, and if so, what is their role? How do functional components interact with data components? Are data components *passive* or *active* (i.e., does the data component actively interact with other components in the system)? How do data and control interact within the system?

These questions provide the designer with an early assessment of design quality and lay the foundation for more-detailed analysis of the architecture.

14.4 ANALYZING ALTERNATIVE ARCHITECTURAL DESIGNS

Quote:

"Maybe it's in the basement, let me go up stairs and check."

M. C. Escher

The questions posed in the preceding section provide a preliminary assessment of the architectural style chosen for a given system. However, a more complete method for evaluating the quality of an architecture is essential if design is to be accomplished effectively. In the sections that follow, we consider two different approaches for the analysis of alternative architectural designs. The first method uses an iterative method to assess design trade-offs. The second approach applies a pseudo-quantitative technique for assessing design quality.

14.4.1 An Architecture Trade-off Analysis Method

The Software Engineering Institute (SEI) has developed an *architecture trade-off analysis method* (ATAM) [KAZ98] that establishes an iterative evaluation process for software architectures. The design analysis activities that follow are performed iteratively:

WebRef

Detailed discussion of software architectural trade-off analysis can be found at **www.sei.cmu.edu/ata/ata_method.html**

1. *Collect scenarios.* A set of use-cases (Chapter 11) is developed to represent the system from the user's point of view.
2. *Elicit requirements, constraints, and environment description.* This information is required as part of requirements engineering and is used to be certain that all customer, user, and stakeholder concerns have been addressed.
3. *Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements.* The style(s) should be described using architectural views such as
   - *Module view* for analysis of work assignments with components and the degree to which information hiding has been achieved.
   - *Process view* for analysis of system performance.
   - *Data flow view* for analysis of the degree to which the architecture meets functional requirements.

XRef
A detailed discussion of quality attributes is presented in Chapters 8 and 19.

4. *Evaluate quality attributes by considering each attribute in isolation.* The number of quality attributes chosen for analysis is a function of the time available for review and the degree to which quality attributes are relevant to the system at hand. Quality attributes for architectural design assessment include reliability, performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability.
5. *Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.* This can be accomplished by making small changes in the architecture and determining how sensitive a quality attribute, say performance, is to the change. Any attributes that are significantly affected by variation in the architecture are termed *sensitivity points.*
6. *Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5.* The SEI describes this approach in the following manner [KAZ98]:

   > Once the architectural sensitivity points have been determined, finding trade-off points is simply the identification of architectural elements to which multiple attributes are sensitive. For example, the performance of a client-server architecture might be highly sensitive to the number of servers (performance increases, within some range, by increasing the number of servers). The availability of that architecture might also vary directly with the number of servers. However, the security of the system might vary inversely with the number of servers (because the system contains more potential points of attack). The number of servers, then, is a trade-off point with respect to this architecture. It is an element, potentially one of many, where architectural trade-offs will be made, consciously or unconsciously.

These six steps represent the first ATAM iteration. Based on the results of steps 5 and 6, some architecture alternatives may be eliminated, one or more of the remaining architectures may be modified and represented in more detail, and then the ATAM steps are reapplied.

14.4.2 Quantitative Guidance for Architectural Design

One of the many problems faced by software engineers during the design process is a general lack of quantitative methods for assessing the quality of proposed designs. The ATAM approach discussed in Section 14.4.1 is representative of a useful but undeniably qualitative approach to design analysis.

Work in the area of quantitative analysis of architectural design is still in its formative stages. Asada and his colleagues [ASA96] suggest a number of pseudo-quantitative techniques that can be used to complement the ATAM approach as a method for the analysis of architectural design quality.

Asada proposes a number of simple models that assist a designer in determining the degree to which a particular architecture meets predefined "goodness" criteria.

These criteria, sometimes called *design dimensions,* often encompass the quality attributes defined in the last section: reliability, performance, security, maintainability, flexibility, testability, portability, reusability, and interoperability, among others.

The first model, called *spectrum analysis,* assesses an architectural design on a "goodness" spectrum from the best to worst possible designs. Once the software architecture has been proposed, it is assessed by assigning a "score" to each of its design dimensions. These dimension scores are summed to determine the total score, *S,* of the design as a whole. Worst-case scores[4] are then assigned to a hypothetical design, and a total score, S_w, for the worst case architecture is computed. A best-case score, S_b, is computed for an optimal design.[5] We then calculate a *spectrum index,* I_s, using the equation

Quote:

"A doctor can bury his mistakes, but an architect can only advise his clients to plant vines."

Frank Lloyd Wright

$$I_s = [(S - S_w)/(S_b - S_w)] \times 100$$

The spectrum index indicates the degree to which a proposed architecture approaches an optimal system within the spectrum of reasonable choices for a design.

If modifications are made to the proposed design or if an entirely new design is proposed, the spectrum indices for both may be compared and an *improvement index,* I_{mp}, may be computed:

$$I_{mp} = I_{s1} - I_{s2}$$

This provides a designer with a relative indication of the improvement associated with architectural changes or a new proposed architecture. If I_{mp} is positive, then we can conclude that system 1 has been improved relative to system 2.

Design selection analysis is another model that requires a set of design dimensions to be defined. The proposed architecture is then assessed to determine the number of design dimensions that it achieves when compared to an ideal (best-case) system. For example, if a proposed architecture would achieve excellent component reuse, and this dimension is required for an idea system, the reusability dimension has been achieved. If the proposed architecture has weak security and strong security is required, that design dimension has not been achieved.

We calculate a *design selection index, d,* as

$$d = (N_s/N_a) \times 100$$

where N_s is the number of design dimensions achieved by a proposed architecture and N_a is the total number of dimensions in the design space. The higher the design selection index, the more closely the proposed architecture approaches an ideal system.

Contribution analysis "identifies the reasons that one set of design choices gets a lower score than another" [ASA96]. Recalling our discussion of quality function deployment (QFD) in Chapter 11, value analysis is conducted to determine the

4 The design must still be applicable to the problem at hand, even if it is not a particularly good solution.

5 The design might be optimal, but constraints, costs, or other factors will not allow it to be built.

relative priority of requirements determined during function deployment, information deployment, and task deployment. A set of "realization mechanisms" (features of the architecture) are identified. All customer requirements (determined using QFD) are listed and a cross-reference matrix is created. The cells of the matrix indicate the relative strength of the relationship (on a numeric scale of 1 to 10) between a realization mechanism and a requirement for each alternative architecture. This is sometimes called a *quantified design space* (QDS). The QDS is relatively easy to implement as a spreadsheet model and can be used to isolate why one set of design choices gets a lower score than another.

14.4.3 Architectural Complexity

A useful technique for assessing the overall complexity of a proposed architecture is to consider dependencies between components within the architecture. These dependencies are driven by information/control flow within the system.

Zhao [ZHA98] suggests three types of dependencies:

Sharing dependencies represent dependence relationships among consumers who use the same resource or producers who produce for the same consumers. For example, for two components *u* and *v*, if *u* and *v* refer to the same global data, then there exists a shared dependence relationship between *u* and *v*.

Flow dependencies represent dependence relationships between producers and consumers of resources. For example, for two components *u* and *v*, if *u* must complete before control flows into *v* (prerequisite), or if *u* communicates with *v* by parameters, then there exists a flow dependence relationship between *u* and *v*.

Constrained dependencies represent constraints on the relative flow of control among a set of activities. For example, for two components *u* and *v*, *u* and *v* cannot execute at the same time (mutual exclusion), then there exists a constrained dependence relationship between *u* and *v*.

The sharing and flow dependencies noted by Zhao are similar in some ways to the concept of coupling discussed in Chapter 13. Simple metrics for evaluating these dependencies are discussed in Chapter 19.

14.5 MAPPING REQUIREMENTS INTO A SOFTWARE ARCHITECTURE

In Chapter 13 we noted that software requirements can be mapped into various representations of the design model. The architectural styles discussed in Section 14.3.1 represent radically different architectures, so it should come as no surprise that a comprehensive mapping that accomplishes the transition from the requirements model to a variety of architectural styles does not exist. In fact, there is no practical mapping for some architectural styles, and the designer must approach the translation of requirements to design for these styles in an ad hoc fashion.

To illustrate one approach to architectural mapping, we consider the call and return architecture—an extremely common structure for many types of systems.[6] The mapping technique to be presented enables a designer to derive reasonably complex call and return architectures from data flow diagrams within the requirements model. The technique, sometimes called *structured design,* has its origins in earlier design concepts that stressed modularity [DEN73], top-down design [WIR71], and structured programming [DAH72], [LIN79]. Stevens, Myers, and Constantine [STE74] were early proponents of software design based on the flow of data through a system. Early work was refined and presented in books by Myers [MYE78] and Yourdon and Constantine [YOU79].

? What steps should we follow to map DFDs into a call and return architecture?

Structured design is often characterized as a data flow-oriented design method because it provides a convenient transition from a data flow diagram to software architecture.[7] The transition from information flow (represented as a DFD) to program structure is accomplished as part of a six-step process: (1) the type of information flow is established; (2) flow boundaries are indicated; (3) the DFD is mapped into program structure; (4) control hierarchy is defined; (5) resultant structure is refined using design measures and heuristics; and (6) the architectural description is refined and elaborated.

The type of information flow is the driver for the mapping approach required in step 3. In the following sections we examine two flow types.

14.5.1 Transform Flow

XRef

Data flow diagrams are discussed in detail in Chapter 12.

Recalling the fundamental system model (level 0 data flow diagram), information must enter and exit software in an "external world" form. For example, data typed on a keyboard, tones on a telephone line, and video images in a multimedia application are all forms of external world information. Such externalized data must be converted into an internal form for processing. Information enters the system along paths that transform external data into an internal form. These paths are identified as *incoming flow.* At the kernel of the software, a transition occurs. Incoming data are passed through a *transform center* and begin to move along paths that now lead "out" of the software. Data moving along these paths are called *outgoing flow.* The overall flow of data occurs in a sequential manner and follows one, or only a few, "straight line" paths.[8] When a segment of a data flow diagram exhibits these characteristics, *transform flow* is present.

---

6 It is also important to note that the call and return architecture can reside within other more sophisticated architectures discussed earlier in this chapter. For example, the architecture of one or more components of a client/server architecture might be call and return.

7 It should be noted that other elements of the analysis model (e.g., the data dictionary, PSPECs, CSPECs) are also used during the mapping method.

8 An obvious mapping for this type of information flow is the data flow architecture described in Section 14.3.1. There are many cases, however, where the data flow architecture may not be the best choice for a complex system. Examples include systems that will undergo substantial change over time or systems in which the processing associated with the data flow is not necessarily sequential.

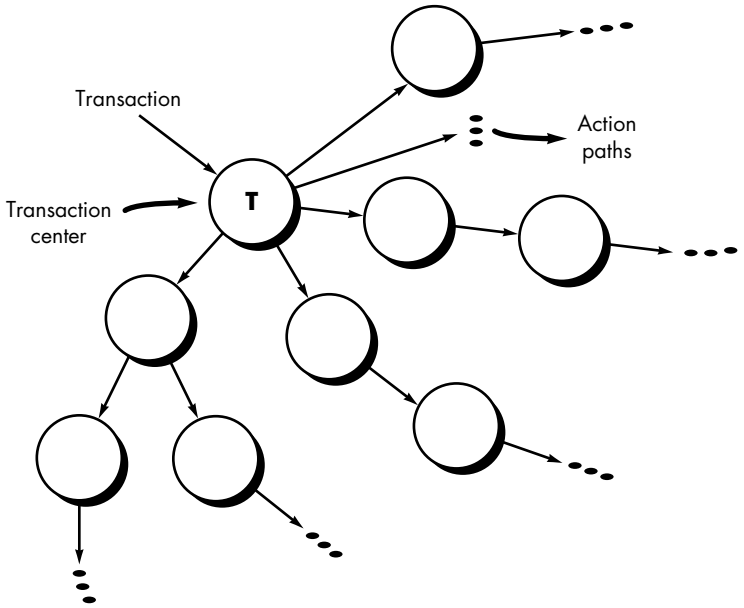

FIGURE 14.4
Transaction flow

14.5.2 Transaction Flow

The fundamental system model implies transform flow; therefore, it is possible to characterize all data flow in this category. However, information flow is often characterized by a single data item, called a *transaction,* that triggers other data flow along one of many paths. When a DFD takes the form shown in Figure 14.4, *transaction flow* is present.

Transaction flow is characterized by data moving along an incoming path that converts external world information into a transaction. The transaction is evaluated and, based on its value, flow along one of many *action paths* is initiated. The hub of information flow from which many action paths emanate is called a *transaction center.*

It should be noted that, within a DFD for a large system, both transform and transaction flow may be present. For example, in a transaction-oriented flow, information flow along an action path may have transform flow characteristics.

14.6 TRANSFORM MAPPING

Transform mapping is a set of design steps that allows a DFD with transform flow characteristics to be mapped into a specific architectural style. In this section transform mapping is described by applying design steps to an example system—a portion of the *SafeHome* security software presented in earlier chapters.

14.6.1 An Example

The *SafeHome* security system, introduced earlier in this book, is representative of many computer-based products and systems in use today. The product monitors the real world and reacts to changes that it encounters. It also interacts with a user through

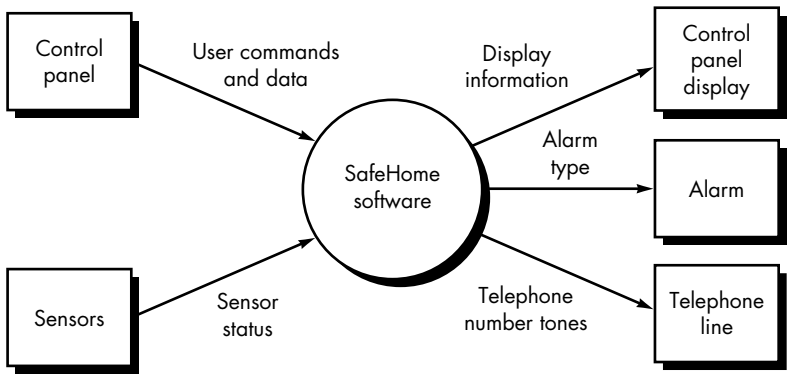

FIGURE 14.5
Context level DFD for SafeHome

a series of typed inputs and alphanumeric displays. The level 0 data flow diagram for *SafeHome,* reproduced from Chapter 12, is shown in Figure 14.5.

During requirements analysis, more detailed flow models would be created for *SafeHome.* In addition, control and process specifications, a data dictionary, and various behavioral models would also be created.

14.6.2 Design Steps

The preceding example will be used to illustrate each step in transform mapping. The steps begin with a re-evaluation of work done during requirements analysis and then move to the design of the software architecture.

Step 1. Review the fundamental system model. The fundamental system model encompasses the level 0 DFD and supporting information. In actuality, the design step begins with an evaluation of both the *System Specification* and the *Software Requirements Specification.* Both documents describe information flow and structure at the software interface. Figures 14.5 and 14.6 depict level 0 and level 1 data flow for the *SafeHome* software.

Step 2. Review and refine data flow diagrams for the software. Information obtained from analysis models contained in the *Software Requirements Specification* is refined to produce greater detail. For example, the level 2 DFD for *monitor sensors* (Figure 14.7) is examined, and a level 3 data flow diagram is derived as shown in Figure 14.8. At level 3, each transform in the data flow diagram exhibits relatively high cohesion (Chapter 13). That is, the process implied by a transform performs a single, distinct function that can be implemented as a module[9] in the *SafeHome* software. Therefore, the DFD in Figure 14.8 contains sufficient detail for a "first cut" at the design of architecture for the *monitor sensors* subsystem, and we proceed without further refinement.

If the DFD is refined further at this time, strive to derive bubbles that exhibit high cohesion.

9 The use of the term *module* in this chapter is equivalent to *component* as it was used in earlier discussions of software architecture.

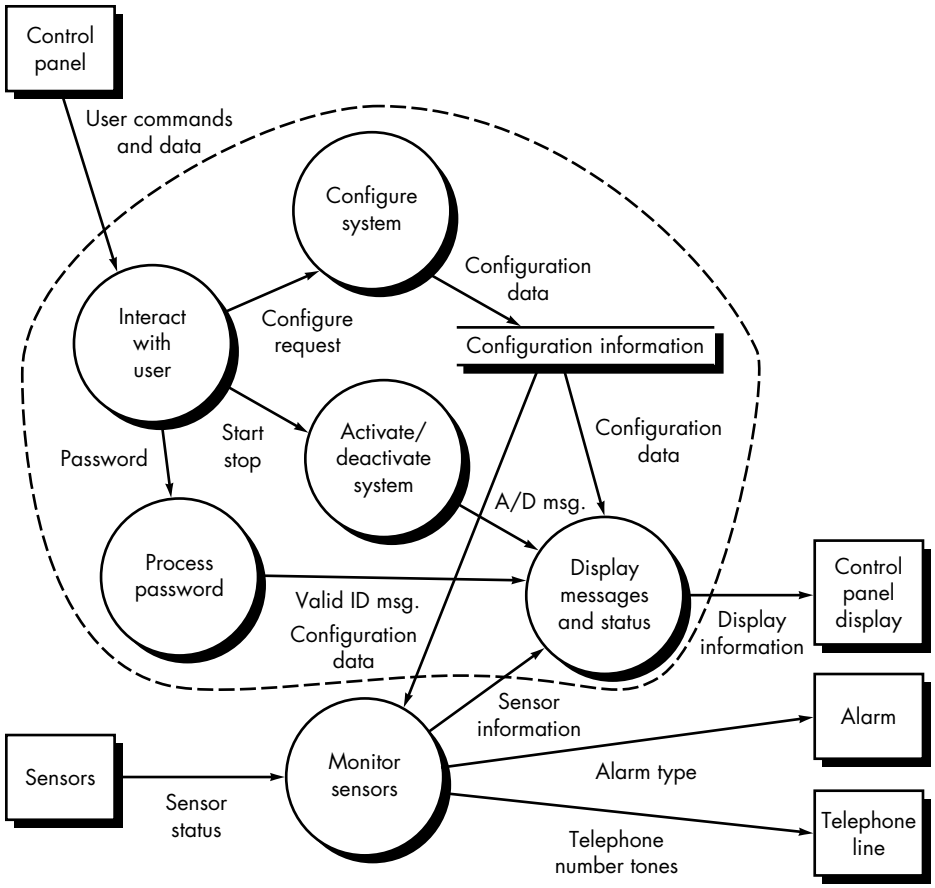

FIGURE 14.6 Level 1 DFD for SafeHome

KEY POINT

You will often encounter both types of data flow within the same analysis model. The flows are partitioned and program structure is derived using the appropriate mapping.

Step 3. Determine whether the DFD has transform or transaction flow characteristics. In general, information flow within a system can always be represented as transform. However, when an obvious transaction characteristic (Figure 14.4) is encountered, a different design mapping is recommended. In this step, the designer selects global (softwarewide) flow characteristics based on the prevailing nature of the DFD. In addition, local regions of transform or transaction flow are isolated. These *subflows* can be used to refine program architecture derived from a global characteristic described previously. For now, we focus our attention only on the *monitor sensors* subsystem data flow depicted in Figure 14.8.

Evaluating the DFD (Figure 14.8), we see data entering the software along one incoming path and exiting along three outgoing paths. No distinct transaction center is implied (although the transform establishes alarm conditions that could be perceived as such). Therefore, an overall transform characteristic will be assumed for information flow.

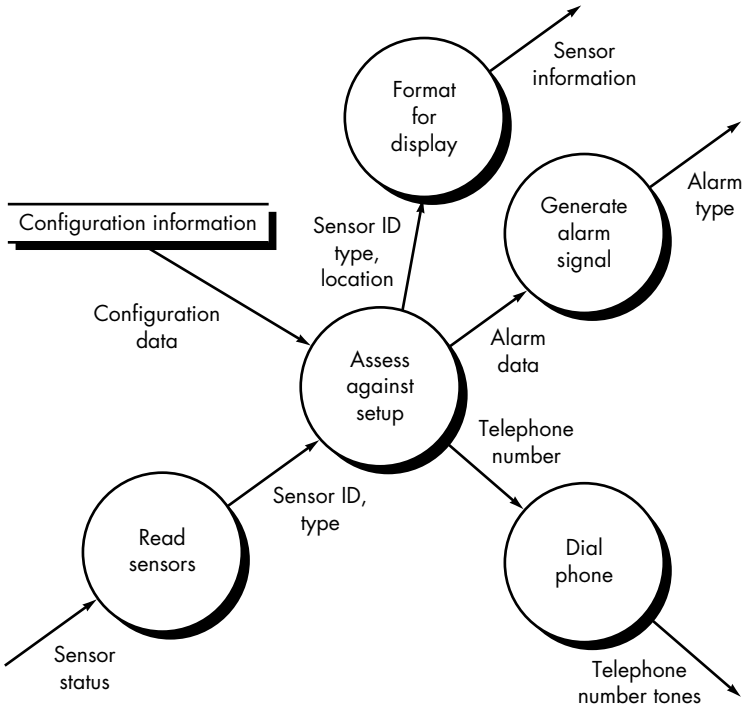

FIGURE 14.7
Level 2 DFD that refines the monitor sensors process

Vary the location of flow boundaries in an effort to explore alternative program structures. This takes very little time and can provide important insight.

Step 4. Isolate the transform center by specifying incoming and outgoing flow boundaries. In the preceding section incoming flow was described as a path in which information is converted from external to internal form; outgoing flow converts from internal to external form. Incoming and outgoing flow boundaries are open to interpretation. That is, different designers may select slightly different points in the flow as boundary locations. In fact, alternative design solutions can be derived by varying the placement of flow boundaries. Although care should be taken when boundaries are selected, a variance of one bubble along a flow path will generally have little impact on the final program structure.

Flow boundaries for the example are illustrated as shaded curves running vertically through the flow in Figure 14.8. The transforms (bubbles) that constitute the transform center lie within the two shaded boundaries that run from top to bottom in the figure. An argument can be made to readjust a boundary (e.g, an incoming flow boundary separating *read sensors* and *acquire response info* could be proposed). The emphasis in this design step should be on selecting reasonable boundaries, rather than lengthy iteration on placement of divisions.

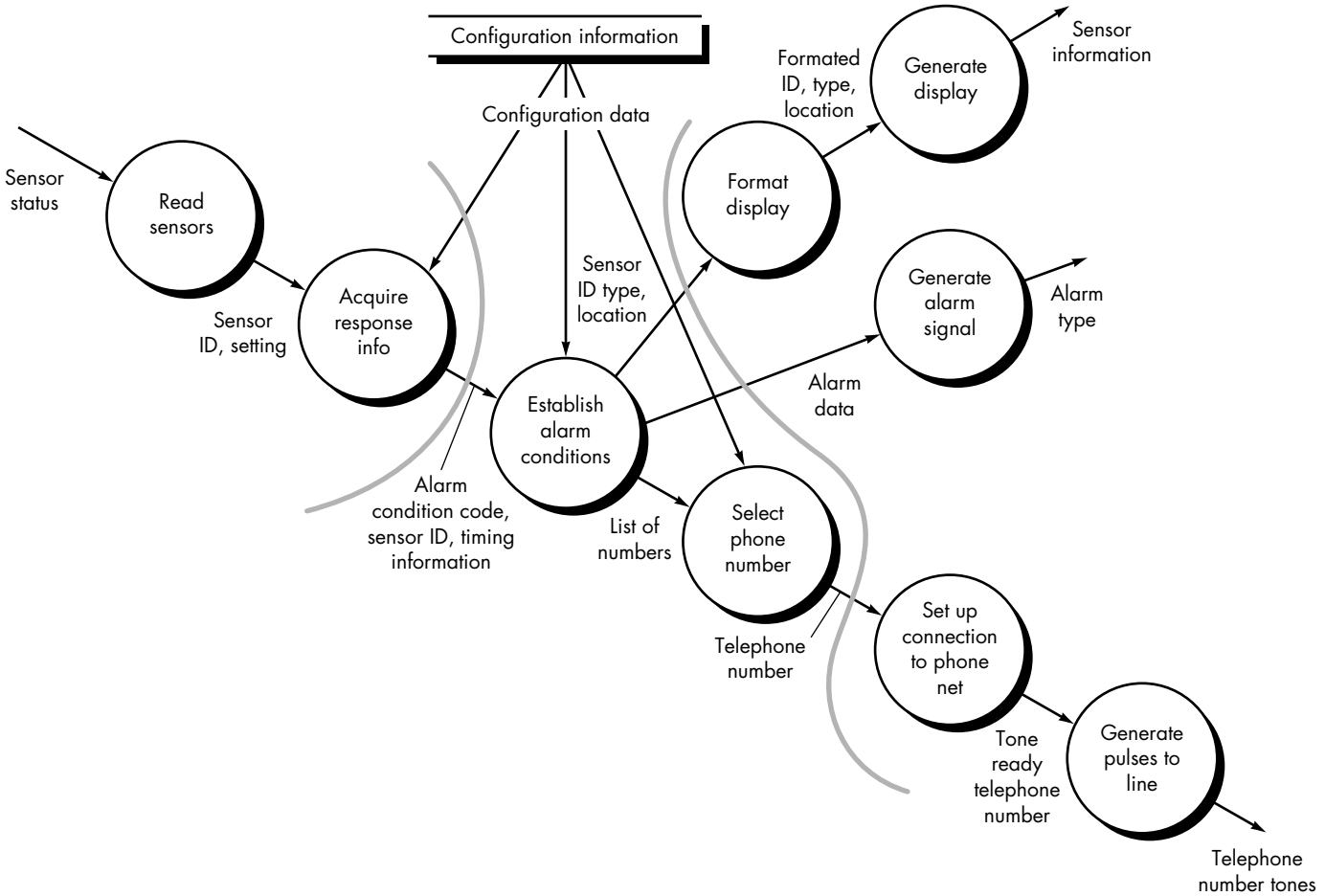

FIGURE 14.8 Level 3 DFD for monitor sensors with flow boundaries

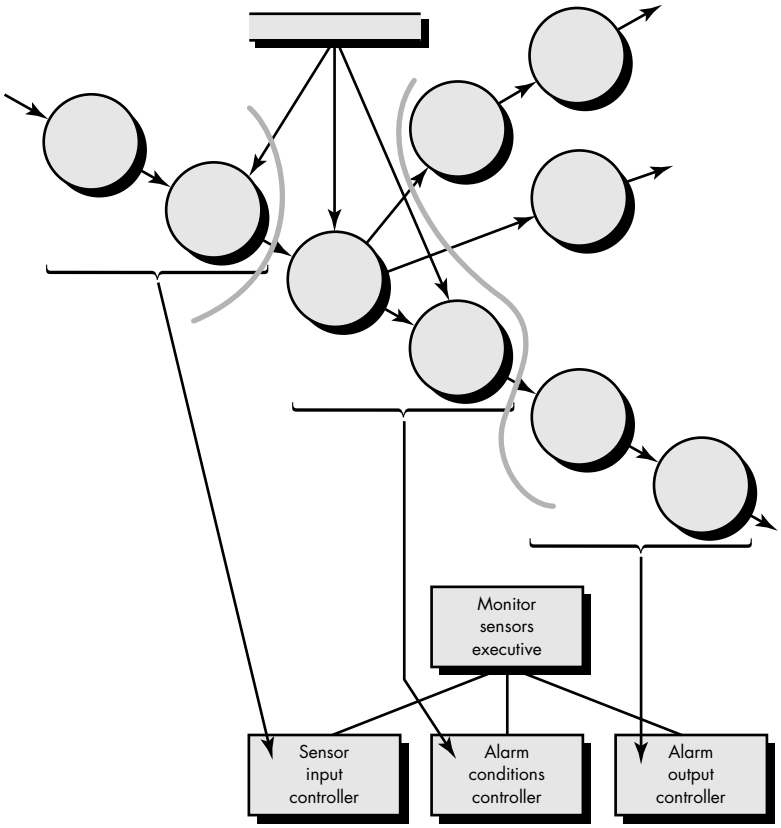

FIGURE 14.9
First-level factoring for monitor sensors

Don't become dogmatic at this stage. It may be necessary to establish two or more controllers for input processing or computation, based on the complexity of the system to be built. If common sense dictates this approach, do it!

Step 5. Perform "first-level factoring." Program structure represents a top-down distribution of control. Factoring results in a program structure in which top-level modules perform decision making and low-level modules perform most input, computation, and output work. Middle-level modules perform some control and do moderate amounts of work.

When transform flow is encountered, a DFD is mapped to a specific structure (a call and return architecture) that provides control for incoming, transform, and outgoing information processing. This first-level factoring for the *monitor sensors* subsystem is illustrated in Figure 14.9. A main controller (called *monitor sensors executive*) resides at the top of the program structure and coordinates the following subordinate control functions:

- An incoming information processing controller, called *sensor input controller,* coordinates receipt of all incoming data.
- A transform flow controller, called *alarm conditions controller,* supervises all operations on data in internalized form (e.g., a module that invokes various data transformation procedures).

- An outgoing information processing controller, called *alarm output controller,* coordinates production of output information.

Although a three-pronged structure is implied by Figure 14.9, complex flows in large systems may dictate two or more control modules for each of the generic control functions described previously. The number of modules at the first level should be limited to the minimum that can accomplish control functions and still maintain good coupling and cohesion characteristics.

Step 6. Perform "second-level factoring." Second-level factoring is accomplished by mapping individual transforms (bubbles) of a DFD into appropriate modules within the architecture. Beginning at the transform center boundary and moving outward along incoming and then outgoing paths, transforms are mapped into subordinate levels of the software structure. The general approach to second-level factoring for the *SafeHome* data flow is illustrated in Figure 14.10.

Keep "worker" modules low in the program structure. This will lead to an architecture that is easier to modify.

Although Figure 14.10 illustrates a one-to-one mapping between DFD transforms and software modules, different mappings frequently occur. Two or even three bubbles can be combined and represented as one module (recalling potential problems with cohesion) or a single bubble may be expanded to two or more modules. Practical considerations and measures of design quality dictate the outcome of second-level factoring. Review and refinement may lead to changes in this structure, but it can serve as a "first-iteration" design.

Second-level factoring for incoming flow follows in the same manner. Factoring is again accomplished by moving outward from the transform center boundary on the incoming flow side. The transform center of *monitor sensors* subsystem software is mapped somewhat differently. Each of the data conversion or calculation transforms of the transform portion of the DFD is mapped into a module subordinate to the transform controller. A completed first-iteration architecture is shown in Figure 14.11.

Eliminate redundant control modules. That is, if a control module does nothing except control one other module, its control function should be imploded at a higher level.

The modules mapped in the preceding manner and shown in Figure 14.11 represent an initial design of software architecture. Although modules are named in a manner that implies function, a brief processing narrative (adapted from the PSPEC created during analysis modeling) should be written for each. The narrative describes

- Information that passes into and out of the module (an interface description).
- Information that is retained by a module, such as data stored in a local data structure.
- A procedural narrative that indicates major decision points and tasks.
- A brief discussion of restrictions and special features (e.g., file I/O, hardware-dependent characteristics, special timing requirements).

The narrative serves as a first-generation *Design Specification.* However, further refinement and additions occur regularly during this period of design.

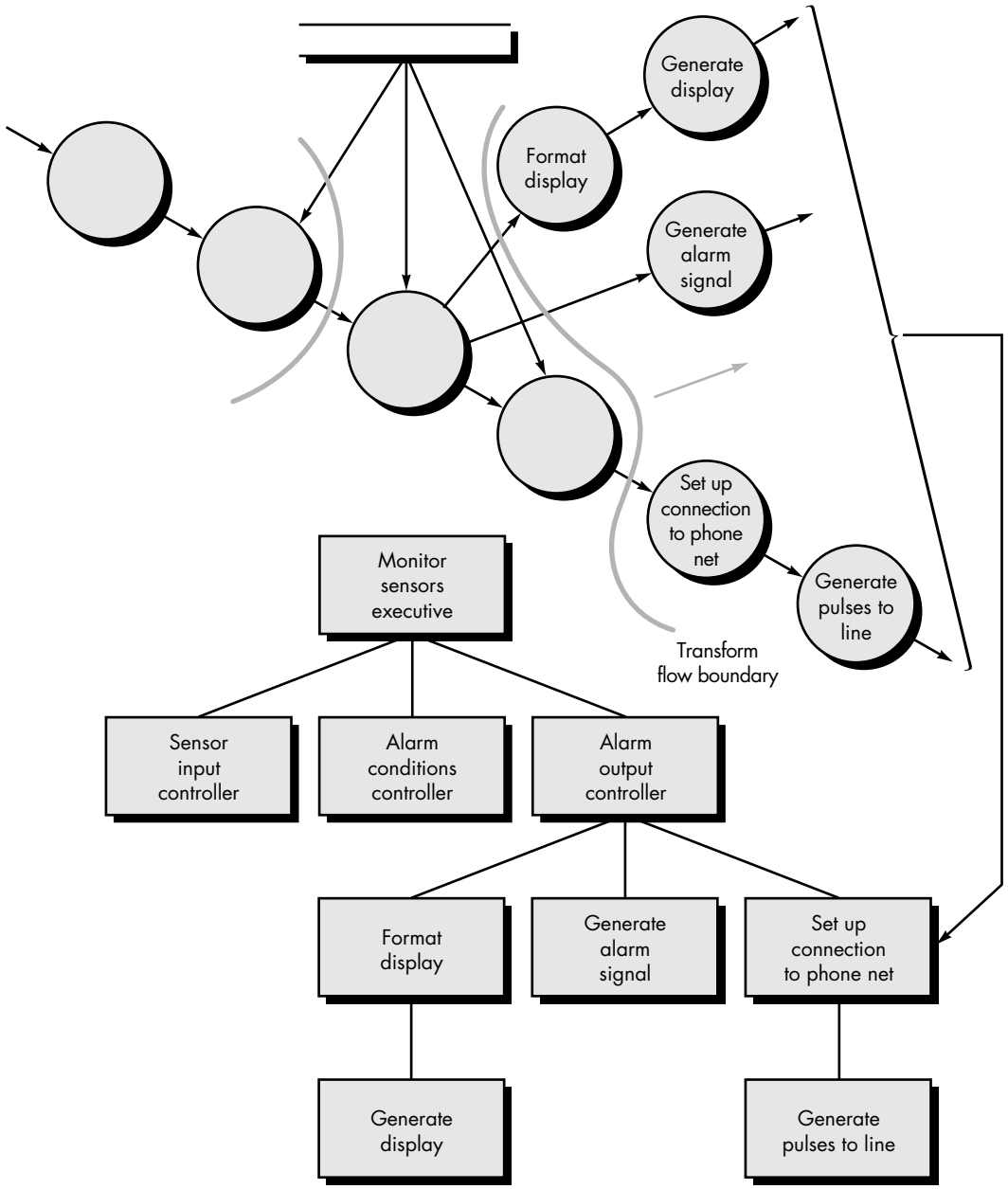

FIGURE 14.10 Second-level factoring for monitor sensors

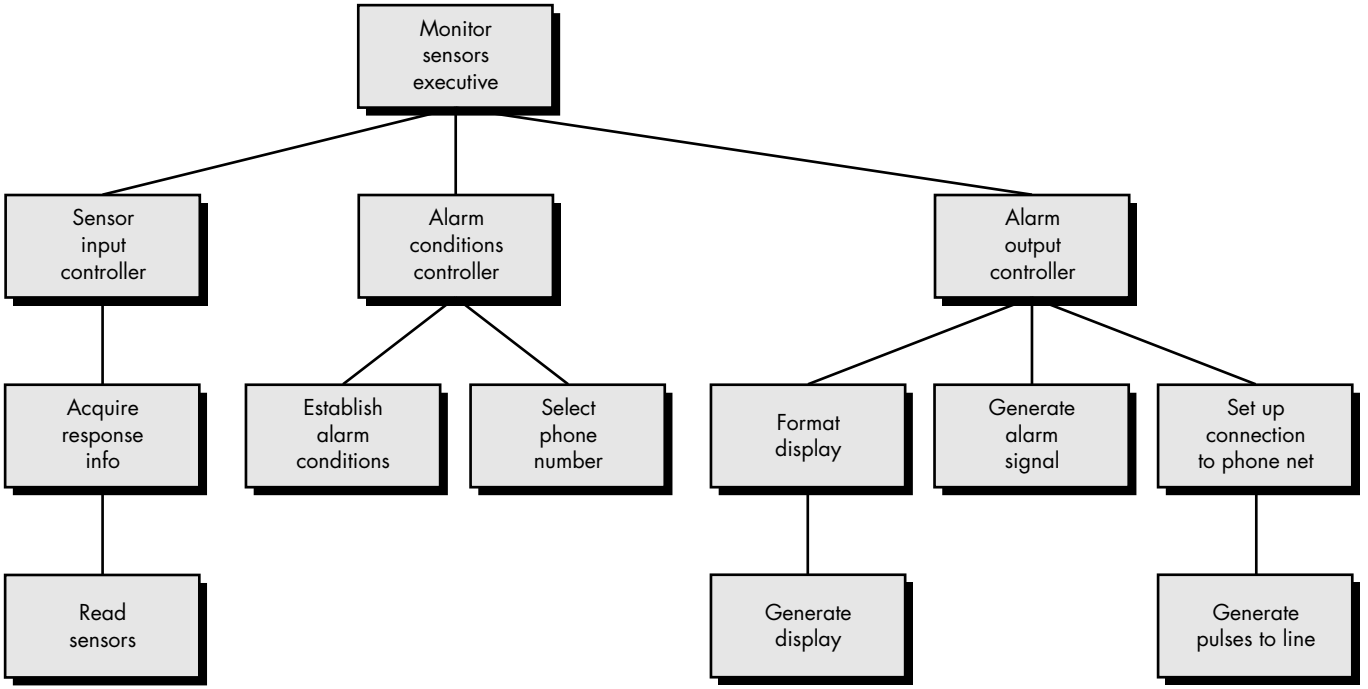

FIGURE 14.11 "First-iteration" program structure for monitor sensors

Step 7. Refine the first-iteration architecture using design heuristics for improved software quality. A first-iteration architecture can always be refined by applying concepts of module independence (Chapter 13). Modules are exploded or imploded to produce sensible factoring, good cohesion, minimal coupling, and most important, a structure that can be implemented without difficulty, tested without confusion, and maintained without grief.

Focus on the functional independence of the modules you've derived. High cohesion and low coupling should be your goal.

Refinements are dictated by the analysis and assessment methods described briefly in Section 14.4, as well as practical considerations and common sense. There are times, for example, when the controller for incoming data flow is totally unnecessary, when some input processing is required in a module that is subordinate to the transform controller, when high coupling due to global data cannot be avoided, or when optimal structural characteristics (see Section 13.6) cannot be achieved. Software requirements coupled with human judgment is the final arbiter.

Many modifications can be made to the first iteration architecture developed for the *SafeHome monitor sensors* subsystem. Among many possibilities,

1. The incoming controller can be removed because it is unnecessary when a single incoming flow path is to be managed.
2. The substructure generated from the transform flow can be imploded into the module *establish alarm conditions* (which will now include the processing implied by *select phone number*). The transform controller will not be needed and the small decrease in cohesion is tolerable.
3. The modules *format display* and *generate display* can be imploded (we assume that display formatting is quite simple) into a new module called *produce display*.

The refined software structure for the monitor sensors subsystem is shown in Figure 14.12.

The objective of the preceding seven steps is to develop an architectural representation of software. That is, once structure is defined, we can evaluate and refine software architecture by viewing it as a whole. Modifications made at this time require little additional work, yet can have a profound impact on software quality.

The reader should pause for a moment and consider the difference between the design approach described and the process of "writing programs." If code is the only representation of software, the developer will have great difficulty evaluating or refining at a global or holistic level and will, in fact, have difficulty "seeing the forest for the trees."

14.7 TRANSACTION MAPPING

In many software applications, a single data item triggers one or a number of information flows that effect a function implied by the triggering data item. The data item,

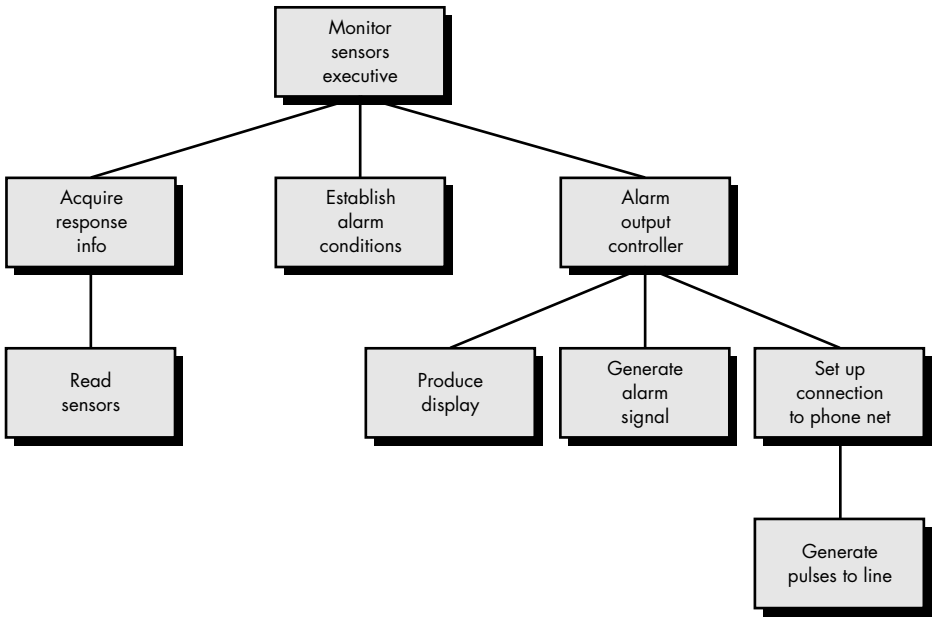

FIGURE 14.12 Refined program structure for monitor sensors

called a *transaction,* and its corresponding flow characteristics are discussed in Section 14.5.2. In this section we consider design steps used to treat transaction flow.

14.7.1 An Example

Transaction mapping will be illustrated by considering the *user interaction* subsystem of the *SafeHome* software. Level 1 data flow for this subsystem is shown as part of Figure 14.6. Refining the flow, a level 2 data flow diagram (a corresponding data dictionary, CSPEC, and PSPECs would also be created) is developed and shown in Figure 14.13.

As shown in the figure, **user commands** flows into the system and results in additional information flow along one of three action paths. A single data item, **command type,** causes the data flow to fan outward from a hub. Therefore, the overall data flow characteristic is transaction oriented.

It should be noted that information flow along two of the three action paths accommodate additional incoming flow (e.g., system parameters and data are input on the "configure" action path). Each action path flows into a single transform, *display messages and status.*

14.7.2 Design Steps

The design steps for transaction mapping are similar and in some cases identical to steps for transform mapping (Section 14.6). A major difference lies in the mapping of DFD to software structure.

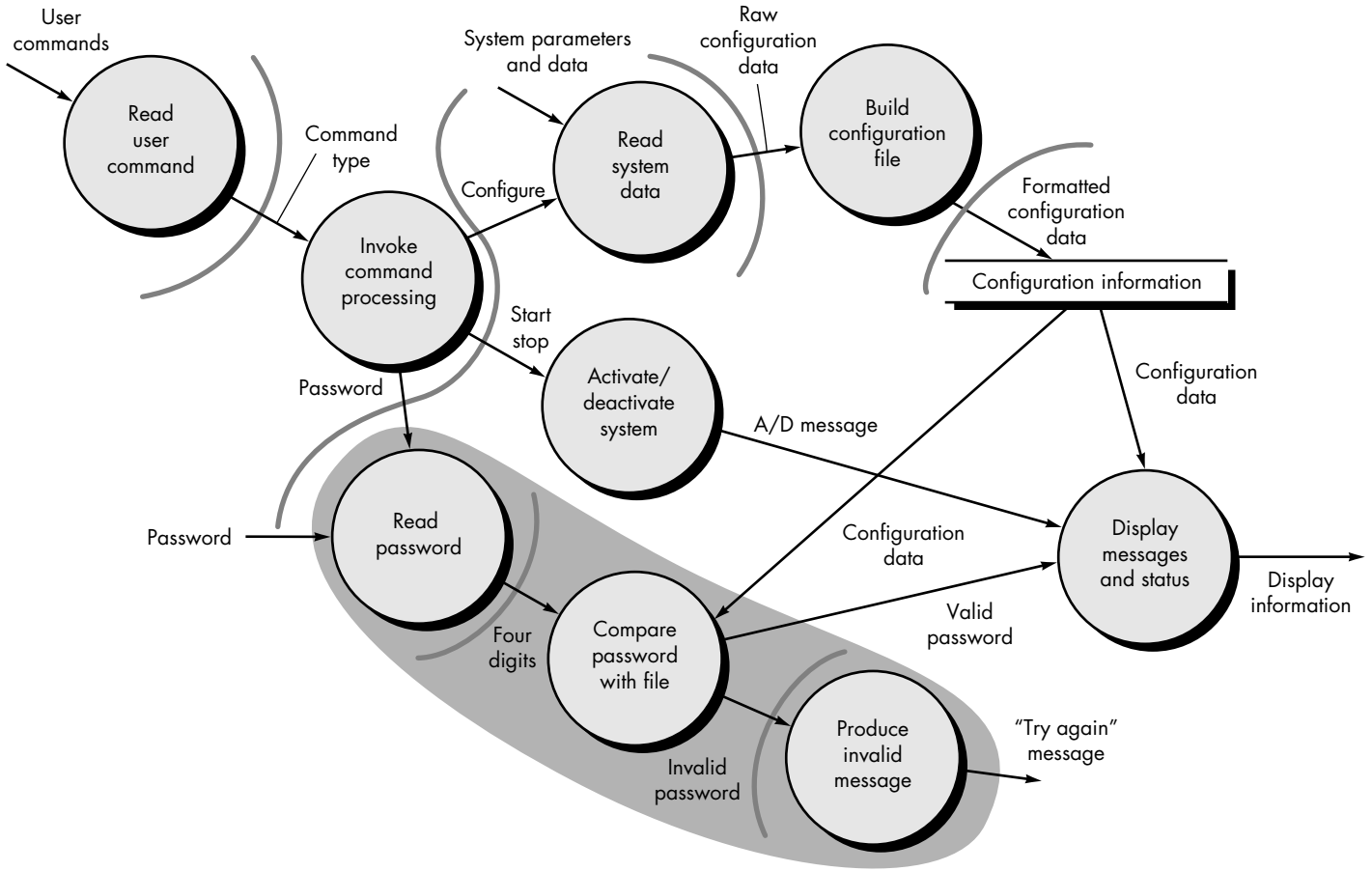

FIGURE 14.13 Level 2 DFD for user interaction subsystem with flow boundaries

Step 1. Review the fundamental system model.

Step 2. Review and refine data flow diagrams for the software.

Step 3. Determine whether the DFD has transform or transaction flow characteristics. Steps 1, 2, and 3 are identical to corresponding steps in transform mapping. The DFD shown in Figure 14.13 has a classic transaction flow characteristic. However, flow along two of the action paths emanating from the *invoke command processing* bubble appears to have transform flow characteristics. Therefore, flow boundaries must be established for both flow types.

Step 4. Identify the transaction center and the flow characteristics along each of the action paths. The location of the transaction center can be immediately discerned from the DFD. The transaction center lies at the origin of a number of actions paths that flow radially from it. For the flow shown in Figure 14.13, the *invoke command processing* bubble is the transaction center.

The incoming path (i.e., the flow path along which a transaction is received) and all action paths must also be isolated. Boundaries that define a reception path and action paths are also shown in the figure. Each action path must be evaluated for its individual flow characteristic. For example, the "password" path (shown enclosed by a shaded area in Figure 14.13) has transform characteristics. Incoming, transform, and outgoing flow are indicated with boundaries.

Step 5. Map the DFD in a program structure amenable to transaction processing. Transaction flow is mapped into an architecture that contains an incoming branch and a dispatch branch. The structure of the incoming branch is developed in much the same way as transform mapping. Starting at the transaction center, bubbles along the incoming path are mapped into modules. The structure of the dispatch branch contains a dispatcher module that controls all subordinate action modules. Each action flow path of the DFD is mapped to a structure that corresponds to its specific flow characteristics. This process is illustrated schematically in Figure 14.14.

First-level factoring results in the derivation of the control hierarchy for the software. Second-level factoring distributes "worker" modules under the appropriate controller.

Considering the *user interaction* subsystem data flow, first-level factoring for step 5 is shown in Figure 14.15. The bubbles *read user command* and *activate/deactivate system* map directly into the architecture without the need for intermediate control modules. The transaction center, *invoke command processing*, maps directly into a dispatcher module of the same name. Controllers for system configuration and password processing are created as illustrated in Figure 14.14.

Step 6. Factor and refine the transaction structure and the structure of each action path. Each action path of the data flow diagram has its own information flow characteristics. We have already noted that transform or transaction flow may be encountered. The action path-related "substructure" is developed using the design steps discussed in this and the preceding section.

As an example, consider the password processing information flow shown (inside shaded area) in Figure 14.13. The flow exhibits classic transform characteristics. A

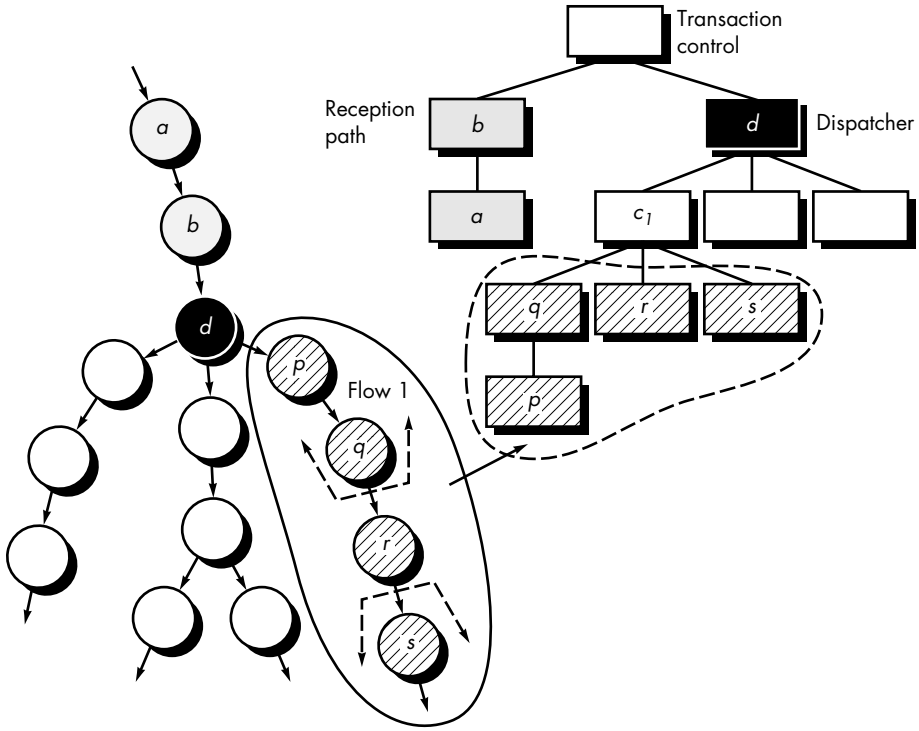

FIGURE 14.14
Transaction mapping

password is input (incoming flow) and transmitted to a transform center where it is compared against stored passwords. An alarm and warning message (outgoing flow) are produced (if a match is not obtained). The "configure" path is drawn similarly using the transform mapping. The resultant software architecture is shown in Figure 14.16.

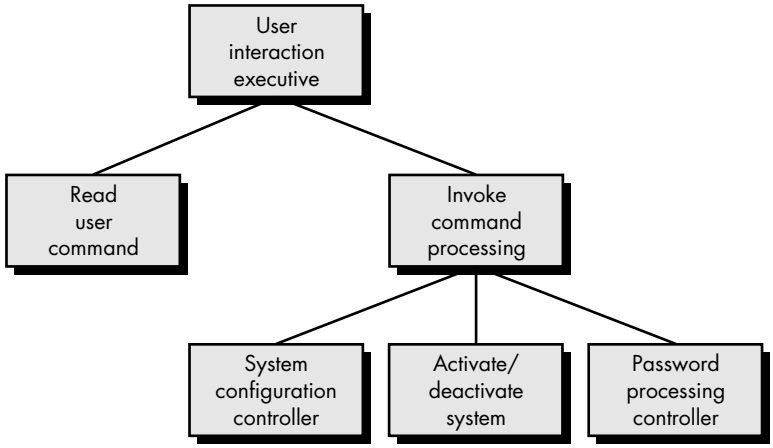

FIGURE 14.15
First-level factoring for user interaction subsystem

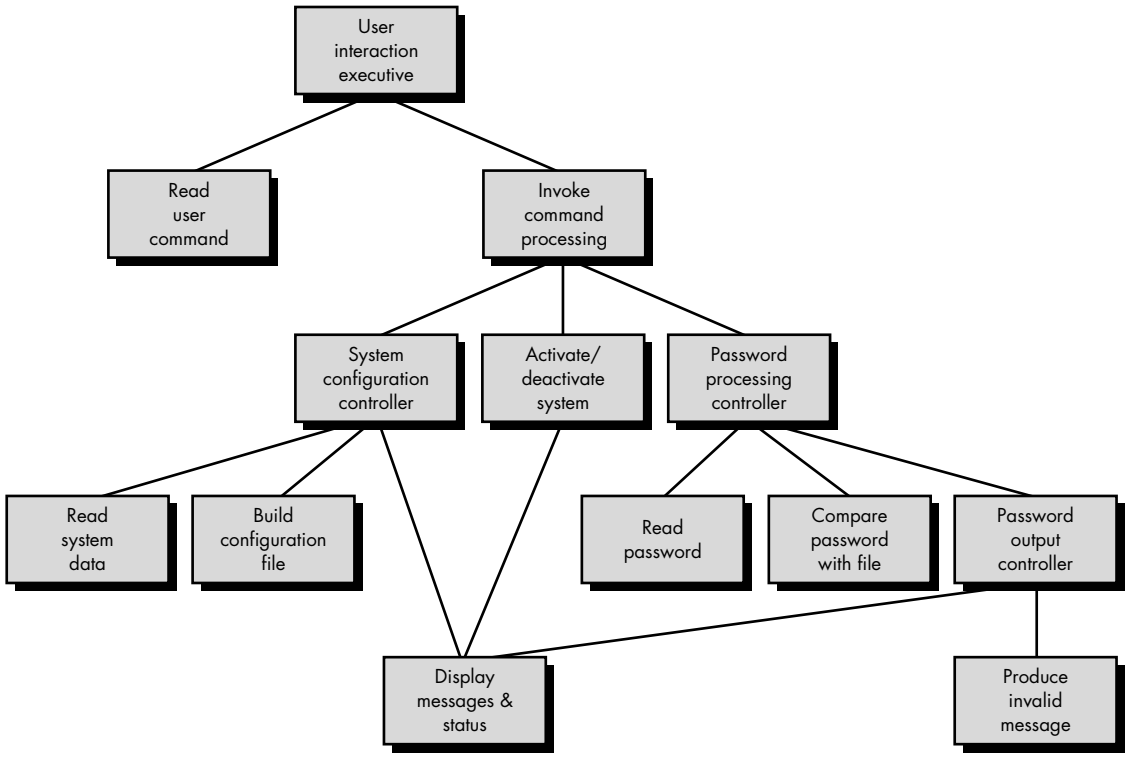

FIGURE 14.16 First-iteration architecture for user interaction subsystem

Step 7. Refine the first-iteration architecture using design heuristics for improved software quality. This step for transaction mapping is identical to the corresponding step for transform mapping. In both design approaches, criteria such as module independence, practicality (efficacy of implementation and test), and maintainability must be carefully considered as structural modifications are proposed.

14.8 REFINING THE ARCHITECTURAL DESIGN

Successful application of transform or transaction mapping is supplemented by additional documentation that is required as part of architectural design. After the program structure has been developed and refined, the following tasks must be completed:

? What happens after the architecture has been created?

- A processing narrative must be developed for each module.
- An interface description is provided for each module.
- Local and global data structures are defined.
- All design restrictions and limitations are noted.

- A set of design reviews are conducted.
- Refinement is considered (if required and justified).

A processing narrative is (ideally) an unambiguous, bounded description of processing that occurs within a module. The narrative describes processing tasks, decisions, and I/O. The interface description describes the design of internal module interfaces, external system interfaces, and the human/computer interface {Chapter 15). The design of data structures can have a profound impact on architecture and the procedural details for each software component. Restrictions and/or limitations for each module are also documented. Typical topics for discussion include restriction on data type or format, memory or timing limitations; bounding values or quantities of data structures; special cases not considered; specific characteristics of an individual module. The purpose of a restrictions and limitations section is to reduce the number of errors introduced because of assumed functional characteristics.

Software Design Specification

Once design documentation has been developed for all modules, one or more design reviews is conducted (see Chapter 8 for review guidelines). The review emphasizes traceability to software requirements, quality of the software architecture, interface descriptions, data structure descriptions, implementation and test practicality, and maintainability.

Any discussion of design refinement should be prefaced with the following comment: "Remember that an 'optimal design' that doesn't work has questionable merit." The software designer should be concerned with developing a representation of software that will meet all functional and performance requirements and merit acceptance based on design measures and heuristics.

Refinement of software architecture during early stages of design is to be encouraged. As we discussed earlier in this chapter, alternative architectural styles may be derived, refined, and evaluated for the "best" approach. This approach to optimization is one of the true benefits derived by developing a representation of software architecture.

It is important to note that structural simplicity often reflects both elegance and efficiency. Design refinement should strive for the smallest number of modules that is consistent with effective modularity and the least complex data structure that adequately serves information requirements.

14.9 SUMMARY

Software architecture provides a holistic view of the system to be built. It depicts the structure and organization of software components, their properties, and the connections between them. Software components include program modules and the various data representations that are manipulated by the program. Therefore, data design is an integral part of the derivation of the software architecture. Architecture

highlights early design decisions and provides a mechanism for considering the benefits of alternative system structures.

Data design translates the data objects defined in the analysis model into data structures that reside within the software. The attributes that describe the object, the relationships between data objects and their use within the program all influence the choice of data structures. At a higher level of abstraction, data design may lead to the definition of an architecture for a database or a data warehouse.

A number of different architectural styles and patterns are available to the software engineer. Each style describes a system category that encompasses a set of components that perform a function required by a system, a set of connectors that enable communication, coordination and cooperation among components, constraints that define how components can be integrated to form the system, and semantic models that enable a designer to understand the overall properties of a system.

Once one or more architectural styles have been proposed for a system, an architecture trade-off analysis method may be used to assess the efficacy of each proposed architecture. This is accomplished by determining the sensitivity of selected quality attributes (also called design dimensions) to various realization mechanisms that reflect properties of the architecture.

The architectural design method presented in this chapter uses data flow characteristics described in the analysis model to derive a commonly used architectural style. A data flow diagram is mapped into program structure using one of two mapping approaches—transform mapping or transaction mapping. Transform mapping is applied to an information flow that exhibits distinct boundaries between incoming and outgoing data. The DFD is mapped into a structure that allocates control to input, processing, and output along three separately factored module hierarchies. Transaction mapping is applied when a single information item causes flow to branch along one of many paths. The DFD is mapped into a structure that allocates control to a substructure that acquires and evaluates a transaction. Another substructure controls all potential processing actions based on a transaction.

Once an architecture has been derived, it is elaborated and then analyzed against quality criteria.

Architectural design encompasses the initial set of design activities that lead to a complete design model of the software. In the chapters that follow, the design focus shifts to interfaces and components.

REFERENCES

[AHO83] Aho, A.V., J. Hopcroft, and J. Ullmann, *Data Structures and Algorithms,* Addison-Wesley, 1983.

[ASA96] Asada, T., et al., "The Quantified Design Space," in *Software Architecture* (Shaw, M. and D. Garlan), Prentice-Hall, 1996, pp. 116–127.

[BAS98] Bass, L., P. Clements, and R. Kazman, *Software Architecture in Practice,* Addison-Wesley, 1998.

[BUS96] Buschmann, F., *Pattern-Oriented Software Architecture,* Wiley, 1996.
[DAH72] Dah., O., E. Dijkstra, and C. Hoare, *Structured Programming,* Academic Press, 1972.
[DAT95] Date, C.J., *An Introduction to Database Systems,* 6th ed., Addison-Wesley, 1995.
[DEN73] Dennis, J.B., "Modularity," in *Advanced Course on Software Engineering* (F.L. Bauer, ed.), Springer-Verlag, 1973, pp. 128–182.
[FRE80] Freeman, P., "The Context of Design," in *Software Design Techniques,* 3rd ed. (P. Freeman and A. Wasserman, eds.), IEEE Computer Society Press, 1980, pp. 2–4.
[INM95] Inmon, W.H., "What Is a Data Warehouse?" Prism Solutions, 1995, presented at http://www.cait.wustl.edu/cait/papers/prism/vol1_no1.
[KAZ98] Kazman, R. et al., *The Architectural Tradeoff Analysis Method,* Software Engineering Institute, CMU/SEI-98-TR-008, July 1998.
[KIM98] Kimball, R., L. Reeves, M. Ross, and W. Thornthwaite, *The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing, and Deploying Data Warehouses,* Wiley, 1998.
[LIN79] Linger, R.C., H.D. Mills, and B.I. Witt, *Structured Programming,* Addison-Wesley, 1979.
[MAT96] Mattison, R., *Data Warehousing: Strategies, Technologies and Techniques,* McGraw-Hill, 1996.
[MYE78] Myers, G., *Composite Structured Design,* Van Nostrand, 1978.
[PRE98] Preiss, B.R., *Data Structures and Algorithms: With Object-Oriented Design Patterns in C++*, Wiley, 1998.
[SHA96] Shaw, M. and D. Garlan, *Software Architecture,* Prentice-Hall, 1996.
[SHA97] Shaw, M. and P. Clements, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems," *Proc. COMPSAC,* Washington, DC, August 1997.
[STE74] Stevens, W., G. Myers, and L. Constantine, "Structured Design," *IBM System Journal,* vol.13, no. 2, 1974, pp.115–139.
[WAS80] Wasserman, A., "Principles of Systematic Data Design and Implementation," in *Software Design Tehcniques* (P. Freeman and A. Wasserman, ed.), 3rd ed., IEEE Computer Society Press, 1980, pp. 287–293.
[WIR71] Wirth, N., "Program Development by Stepwise Refinement," *CACM,* vol. 14, no. 4, 1971, pp. 221–227.
[YOU79] Yourdon, E. and L. Constantine, *Structured Design,* Prentice-Hall, 1979.
[ZHA98] Zhao, J, "On Assessing the Complexity of Software Architectures," *Proc. Intl. Software Architecture Workshop,* ACM, Orlando, FL, 1998, p. 163–167.

PROBLEMS AND POINTS TO PONDER

14.1. Using the architecture of a house or building as a metaphor, draw comparisons with software architecture. How are the disciplines of classical architecture and the software architecture similar? How do they differ?

14.2. Write a three- to five-page paper that presents guidelines for selecting data structures based on the nature of problem. Begin by delineating the classical data structures encountered in software work and then describe criteria for selecting from these for particular types of problems.

14.3. Explain the difference between a database that services one or more conventional business applications and a data warehouse.

14.4. Write a three- to five-page paper that describes how data mining techniques are used in a business context and the current state of KDD techniques.

14.5. Present two or three examples of applications for each of the architectural styles noted in Section 14.3.1.

14.6. Some of the architectural styles noted in Section 14.3.1 are hierarchical in nature and others are not. Make a list of each type. How would the architectural styles that are not hierarchical be implemented?

14.7. Select an application with which you are familiar. Answer each of the questions posed for control and data in Section 14.3.2.

14.8. Research the ATAM (using [KAZ98]) and present a detailed discussion of the six steps presented in Section 14.4.1.

14.9. Select an application with which you are familiar. Using best guesses where required, identify a set of design dimensions and then perform spectrum analysis and design selection analysis.

14.10. Research the QDS (using [ASA96]) and develop a quantified design space for an application with which you are familiar.

14.11. Some designers contend that all data flow may be treated as transform oriented. Discuss how this contention will affect the software architecture that is derived when a transaction-oriented flow is treated as transform. Use an example flow to illustrate important points.

14.12. If you haven't done so, complete problem 12.13. Use the design methods described in this chapter to develop a software architecture for the PHTRS.

14.13. Using a data flow diagram and a processing narrative, describe a computer-based system that has distinct transform flow characteristics. Define flow boundaries and map the DFD into a software structure using the technique described in Section 14.6.

14.14. Using a data flow diagram and a processing narrative, describe a computer-based system that has distinct transaction flow characteristics. Define flow boundaries and map the DFD into a software structure using the technique described in Section 14.7.

14.15. Using requirements that are derived from a classroom discussion, complete the DFDs and architectural design for the *SafeHome* example presented in Sections

14.6 and 14.7. Assess the functional independence of all modules. Document your design.

14.16. Discuss the relative merits and difficulties of applying data flow-oriented design in the following areas: (a) embedded microprocessor applications, (b) engineering/scientific analysis, (c) computer graphics, (d) operating system design, (e) business applications, (f) database management system design, (g) communications software design, (h) compiler design, (i) process control applications, and (j) artificial intelligence applications.

14.17. Given a set of requirements provided by your instructor (or a set of requirements for a problem on which you are currently working) develop a complete architectural design. Conduct a design review (Chapter 8) to assess the quality of your design. This problem may be assigned to a team, rather than an individual.

FURTHER READINGS AND INFORMATION SOURCES

The literature on software architecture has exploded over the past decade. Books by Shaw and Garlan [SHA96], Bass, Clements, and Kazman [BAS98] and Buschmann et al. [BUS96] provide in-depth treatment of the subject. Earlier work by Garlan (*An Introduction to Software Architecture,* Software Engineering Institute, CMU/SEI-94-TR-021, 1994) provides an excellent introduction.

Implementation specific books on architecture address architectural design within a specific development environment or technology. Mowbray (*CORBA Design Patterns,* Wiley, 1997) and Mark et al. (*Object Management Architecture Guide,* Wiley, 1996) provide detailed design guidelines for the CORBA distributed application support framework. Shanley (*Protected Mode Software Architecture,* Addison-Wesley, 1996) provides architectural design guidance for anyone designing PC-based real-time operating systems, multi-task operating systems, or device drivers.

Current software architecture research is documented yearly in the *Proceedings of the International Workshop on Software Architecture,* sponsored by the ACM and other computing organizations, and the *Proceedings of the International Conference on Software Engineering*.

Data modeling is a prerequisite to good data design. Books by Teory (*Database Modeling and Design,* Academic Press, 1998); Schmidt (*Data Modeling for Information Professionals,* Prentice-Hall, 1998); Bobak (*Data Modeling and Design for Today's Architectures,* Artech House, 1997); Silverston, Graziano, and Inmon (*The Data Model Resource Book,* Wiley, 1997); Date [DAT95], Reingruber and Gregory (*The Data Modeling Handbook: A Best-Practice Approach to Building Quality Data Models,* Wiley, 1994); and Hay (*Data Model Patterns: Conventions of Thought,* Dorset House, 1994) contain detailed presentations of data modeling notation, heuristics, and database design approaches. The design of data warehouses has become increasingly important in

recent years. Books by Humphreys, Hawkins, and Dy (*Data Warehousing: Architecture and Implementation,* Prentice-Hall, 1999); Kimball et al. [KIM98]; and Inmon [INM95] cover the topic in considerable detail.

Dozens of current books address data design and data structures, usually in the context of a specific programming language. Typical examples are

Horowitz, E. and S. Sahni, *Fundamentals of Data Structures in Pascal,* 4th ed., W.H. Freeman and Co., 1999.

Kingston, J.H., *Algorithms and Data Structures: Design, Correctness, Analysis,* 2nd ed., Addison-Wesley, 1997.

Main, M., *Data Structures and Other Objects Using Java,* Addison-Wesley, 1998.

Preiss, B.R., *Data Structures and Algorithms: With Object-Oriented Design Patterns in C++,* Wiley, 1998.

Sedgewick, R., *Algorithms in C++: Fundamentals, Data Structures, Sorting, Searching,* Addison-Wesley, 1999.

Standish, T.A., *Data Structures in Java,* Addison-Wesley, 1997.

Standish, T.A., *Data Structures, Algorithms, and Software Principles in C,* Addison-Wesley, 1995.

General treatment of software design with discussion of architectural and data design issues can be found in most books dedicated to software engineering. Books by Pfleeger (*Software Engineering: Theory and Practice,* Prentice-Hall, 1998) and Sommerville *(Software Engineering,* 5th ed., Addison-Wesley,1996) are representative of those that cover design issues in some detail.

More rigorous treatments of the subject can be found in Feijs (*Formalization of Design Methods,* Prentice-Hall, 1993), Witt et al. (*Software Architecture and Design Principles,* Thomson Publishing, 1994), and Budgen (*Software Design,* Addison-Wesley, 1994).

Complete presentations of data flow-oriented design may be found in Myers [MYE78], Yourdon and Constantine [YOU79], Buhr (*System Design with Ada,* Prentice-Hall, 1984), and Page-Jones (*The Practical Guide to Structured Systems Design,* 2nd ed., Prentice-Hall, 1988). These books are dedicated to design alone and provide comprehensive tutorials in the data flow approach.

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/arch-design.mhtml

CHAPTER

15 USER INTERFACE DESIGN

KEY CONCEPTS

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.

QUICK LOOK

What is it? User interface design creates an effective communication medium between a human and a computer. Following a set of interface design principles, design identifies interface objects and actions and then creates a screen layout that forms the basis for a user interface prototype.

Who does it? A software engineer designs the user interface by applying an iterative process that draws on predefined 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. Because it molds a user's perception of the software, the interface has to be right.

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 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. Tools are used to prototype and ultimately implement the design model, and the result is evaluated for quality.

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.

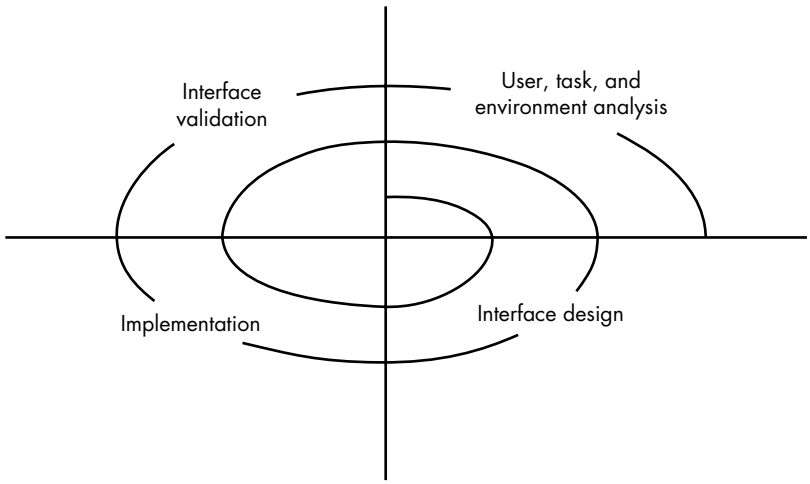

FIGURE 15.1
The user interface design process

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

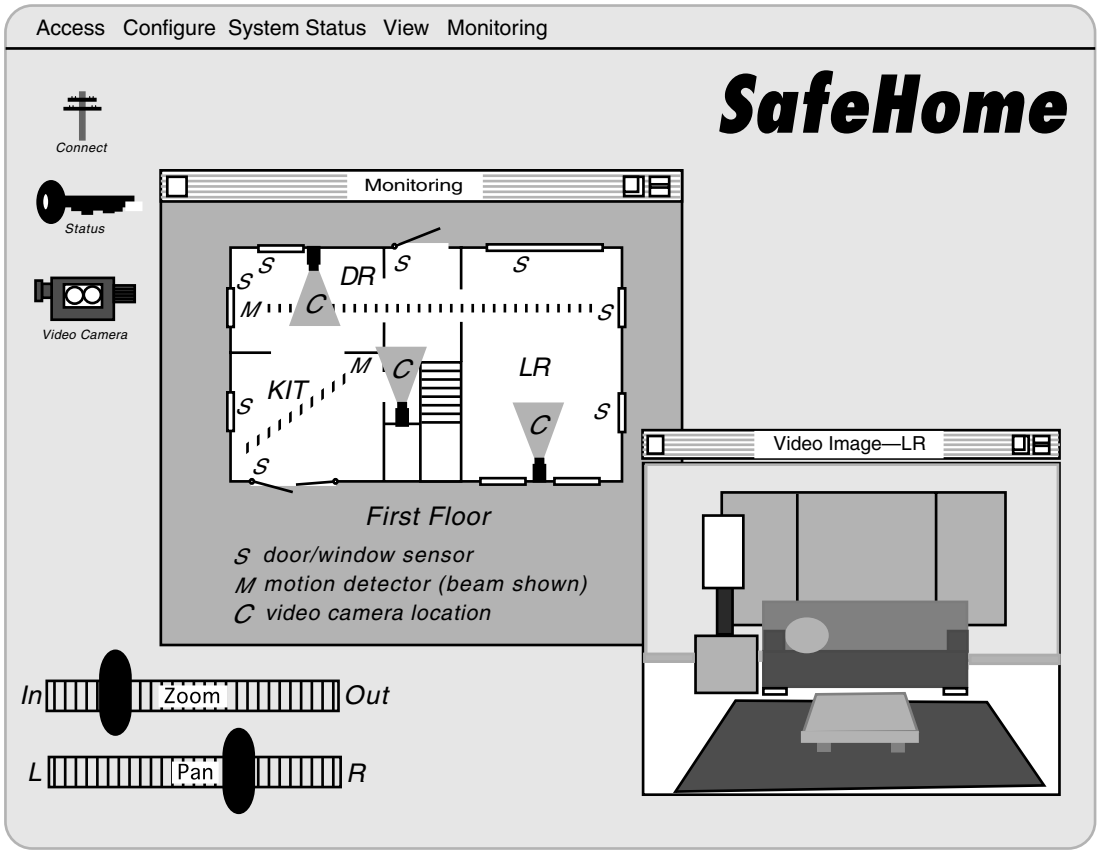

FIGURE 5.2 Preliminary screen layout

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

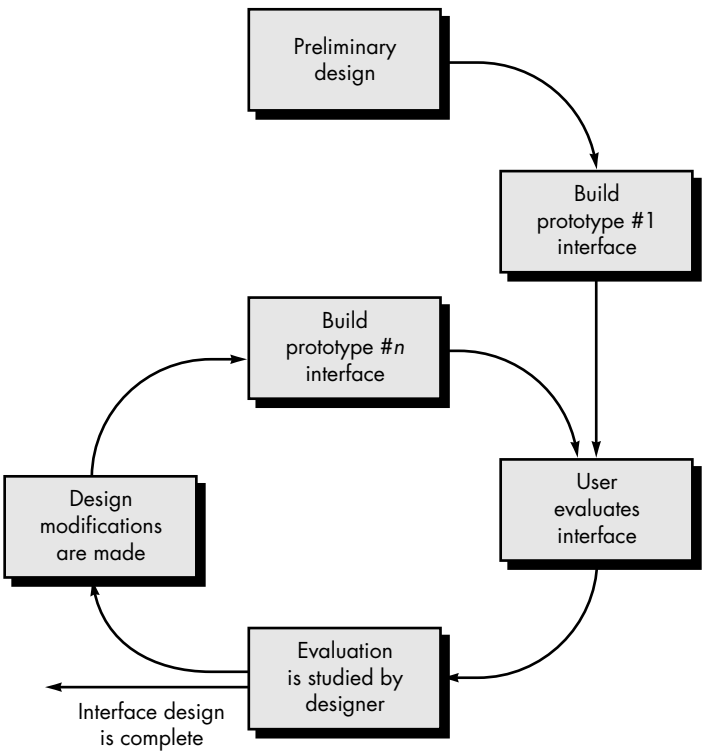

FIGURE 15.3 The interface design evaluation cycle

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

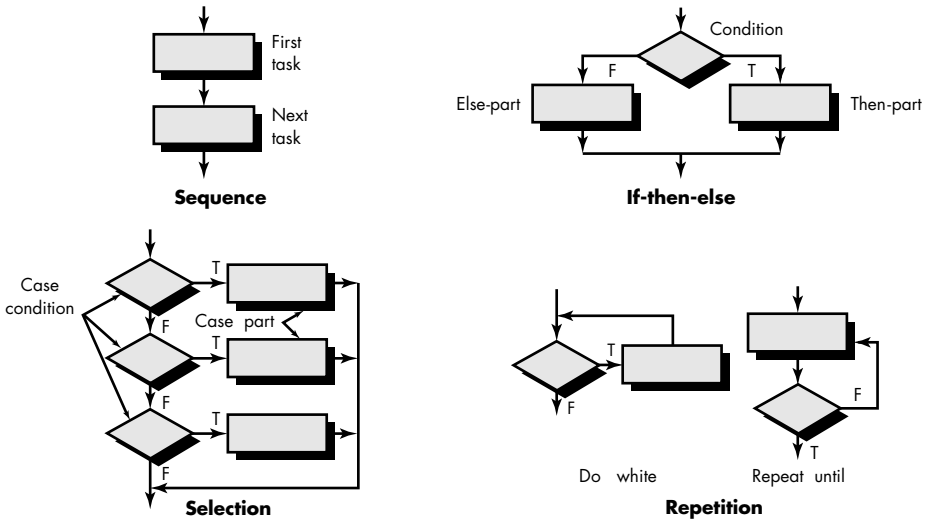

FIGURE 16.1
Flowchart constructs

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

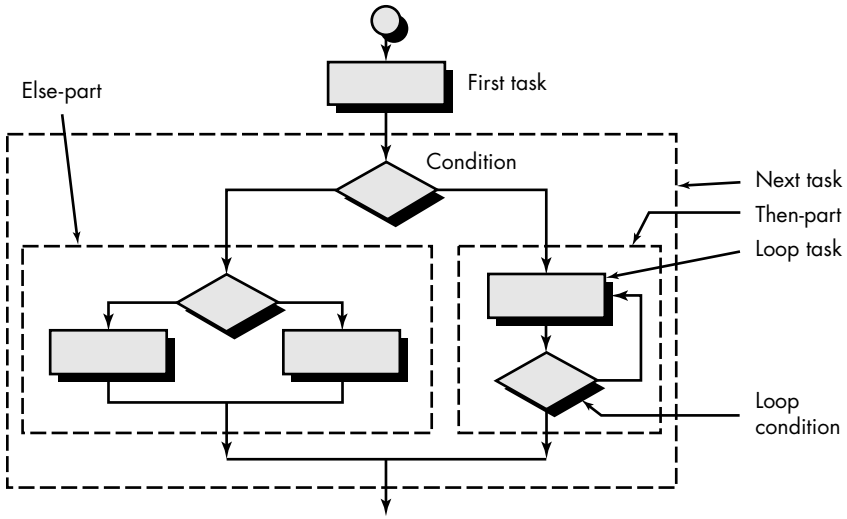

FIGURE 16.2 Nesting constructs

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

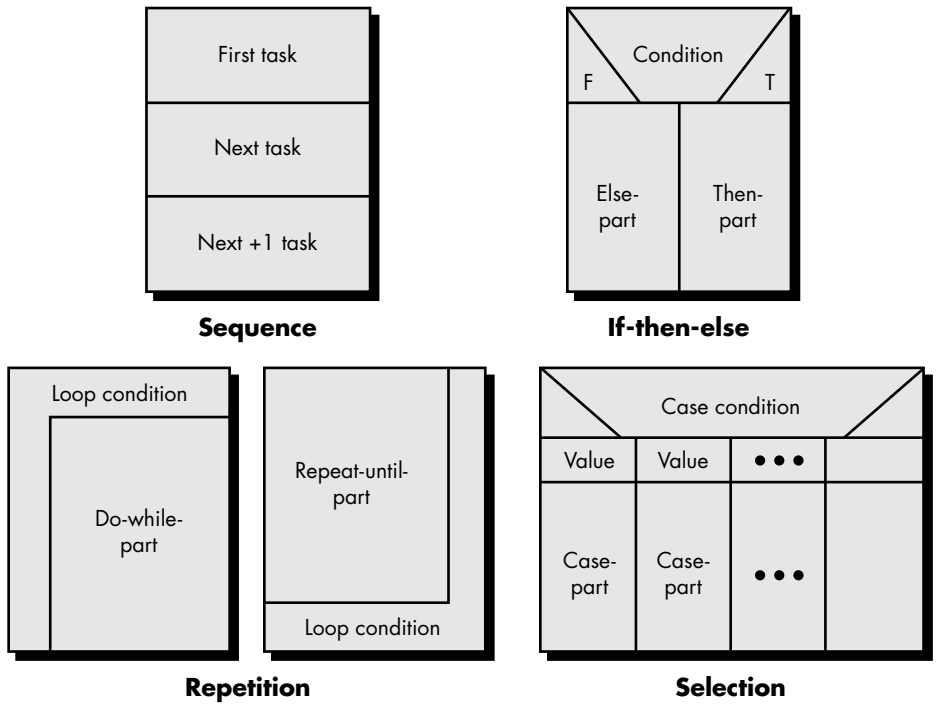

FIGURE 16.3
Box diagram constructs

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**