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

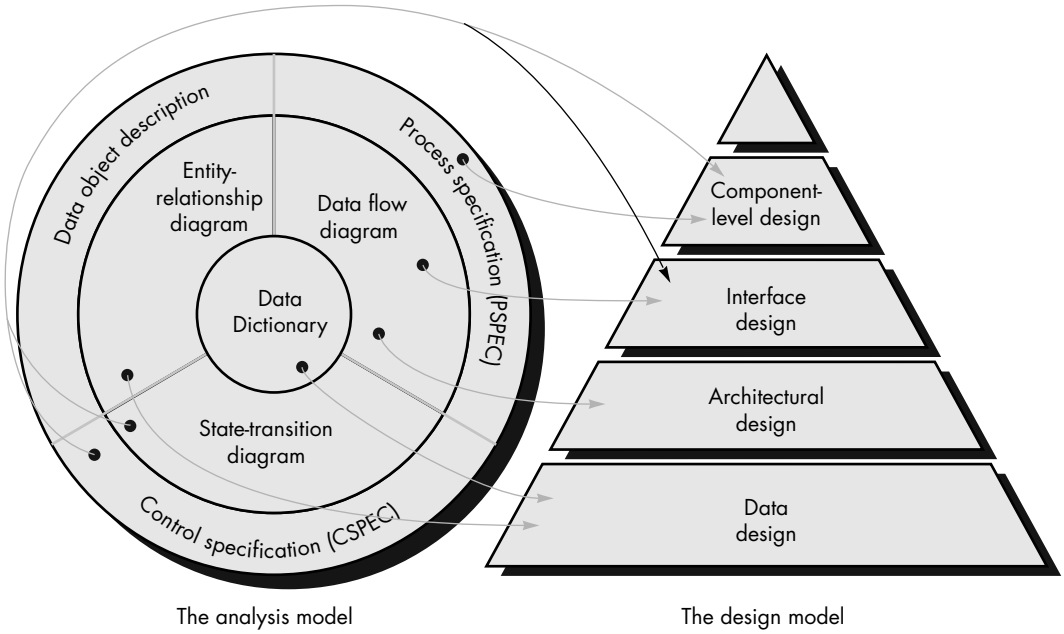

FIGURE 13.1 Translating the analysis model into a software design

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.

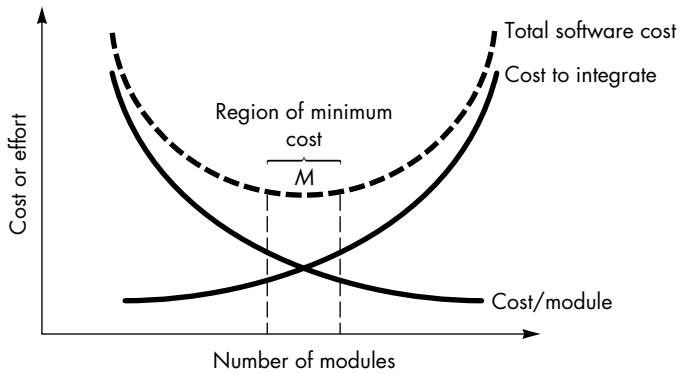

FIGURE 13.2 Modularity and software cost

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.

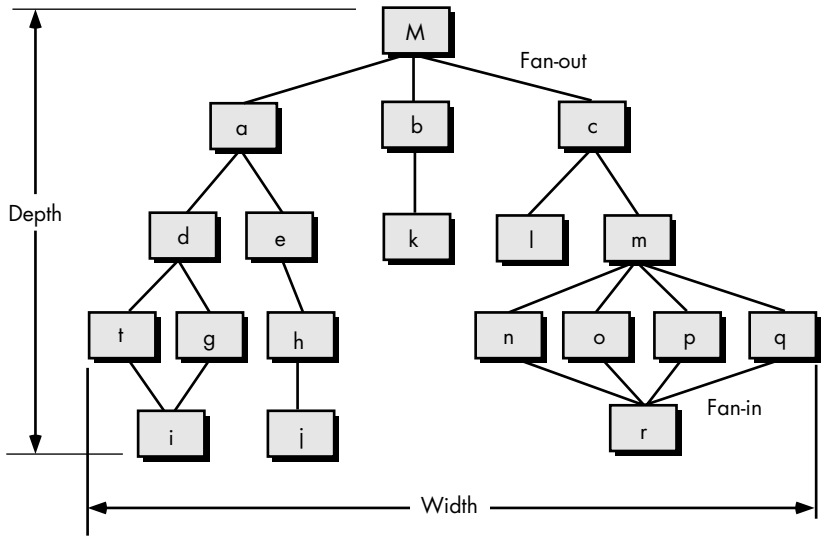

FIGURE 13.3 Structure terminology for a call and return architectural style

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.

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.

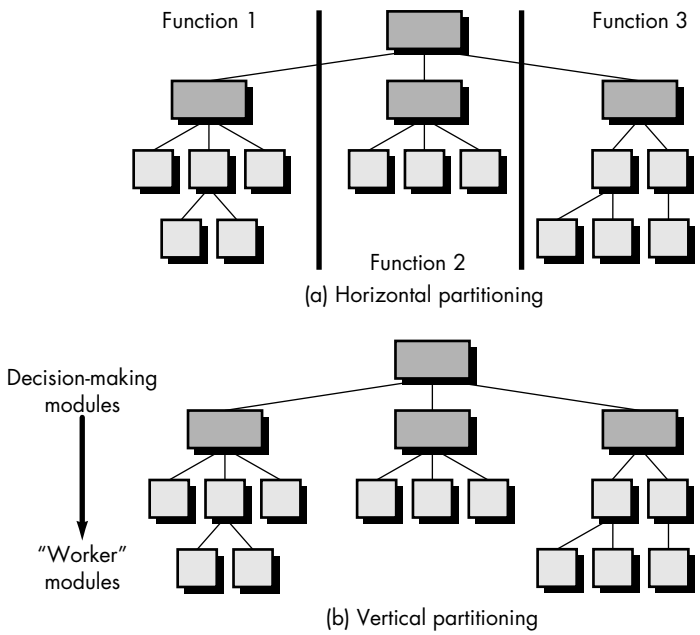

FIGURE 13.4 Structural partitioning

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.

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.

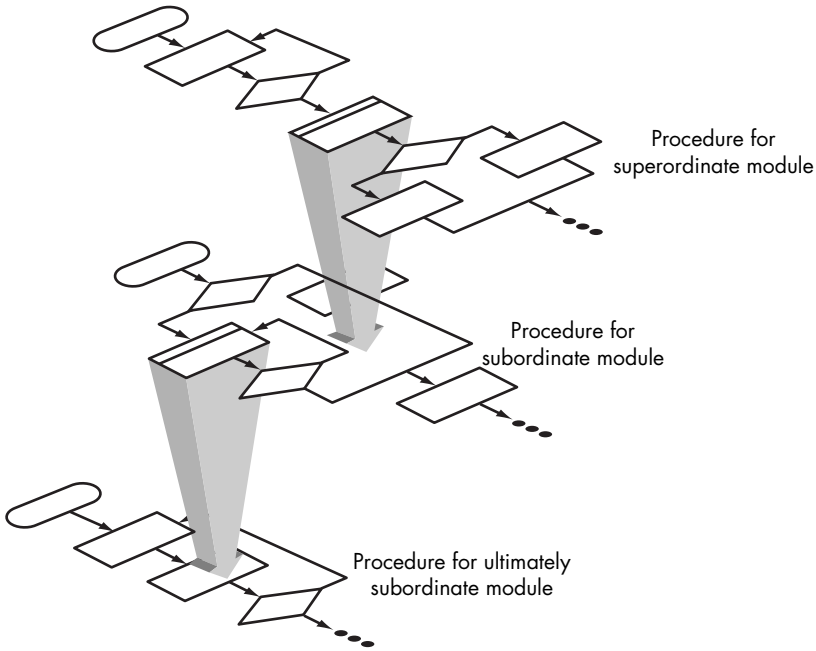

FIGURE 13.5
Procedure is layered

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.

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

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.

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.

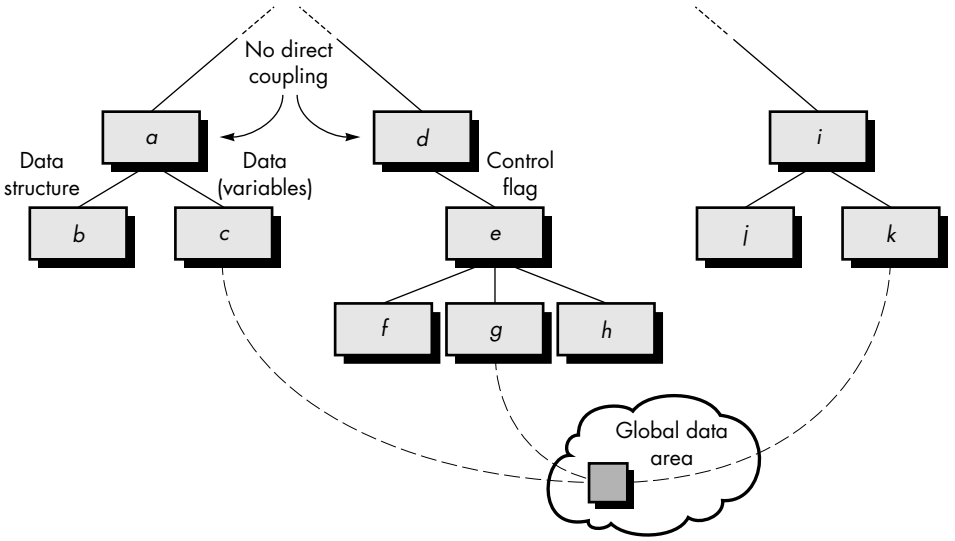

FIGURE 13.6 Types of coupling

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

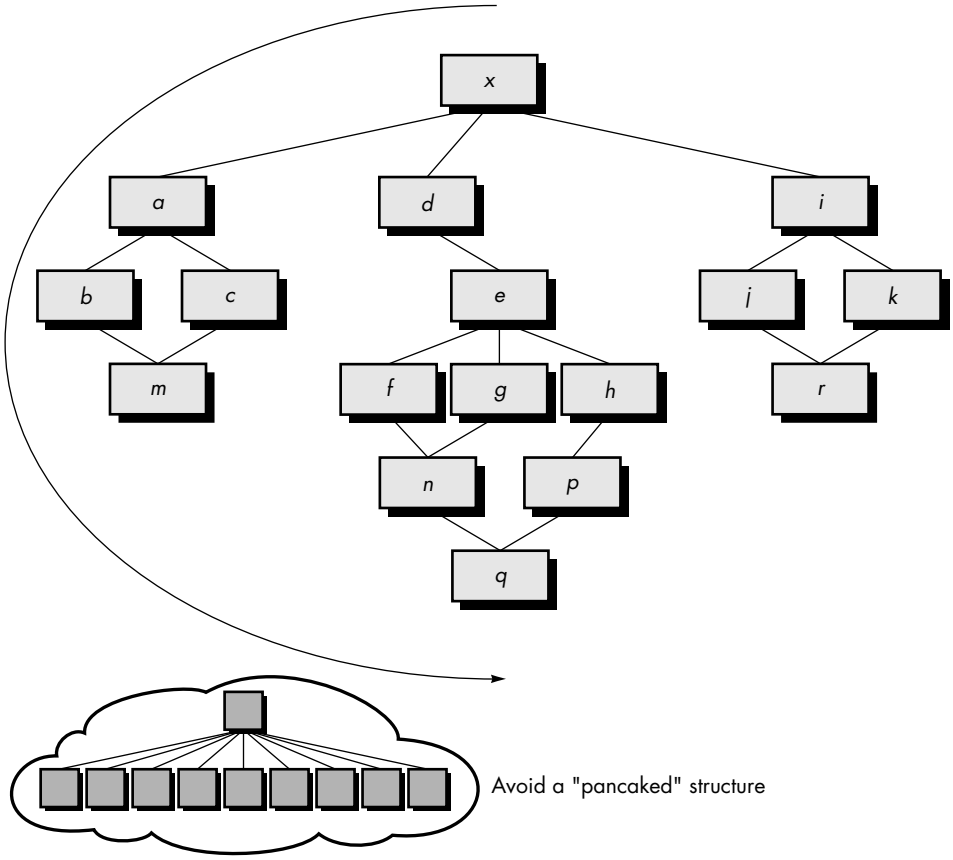

FIGURE 13.7 Program structures

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