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.

WebRef
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.datawarehouse.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]:

KEY POINT

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.

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.

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

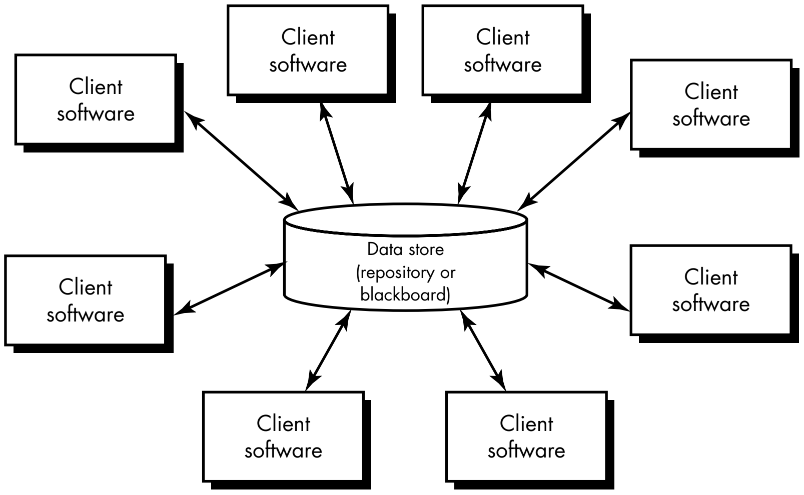
Although millions of computer-based systems have been created over the past 50 years, the vast majority can be categorized {see [SHA96], {BAS98], BUS96]) into one of a relatively small number of architectural styles:

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

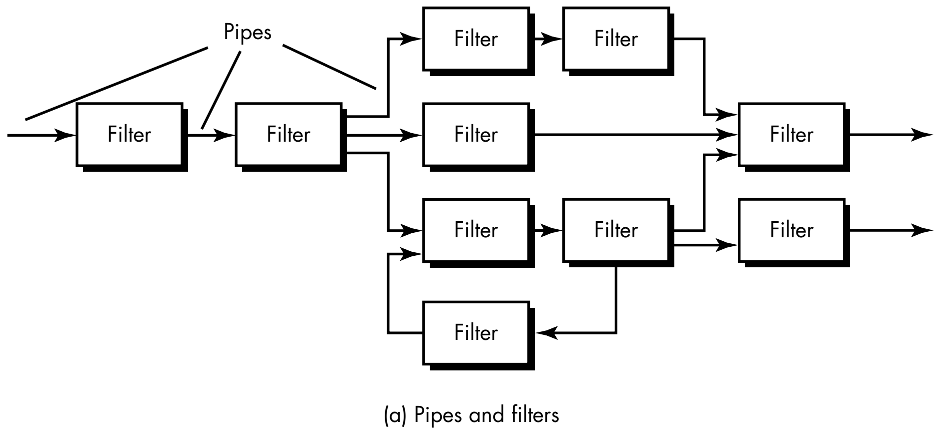
Data-flow architectures. This architecture is applied when input data are to be transformed through a series of computational or manipulative components into output data. A *pipe and filter pattern* (Figure 14.2a) has a set of components, called *filters,* connected by pipes that transmit data from one 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.

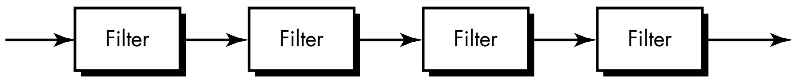
Call and return architectures. This architectural style enables a software designer (system architect) to achieve a program structure that is relatively easy to modify and scale. A number of substyles [BAS98] exist within this category:

- *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.

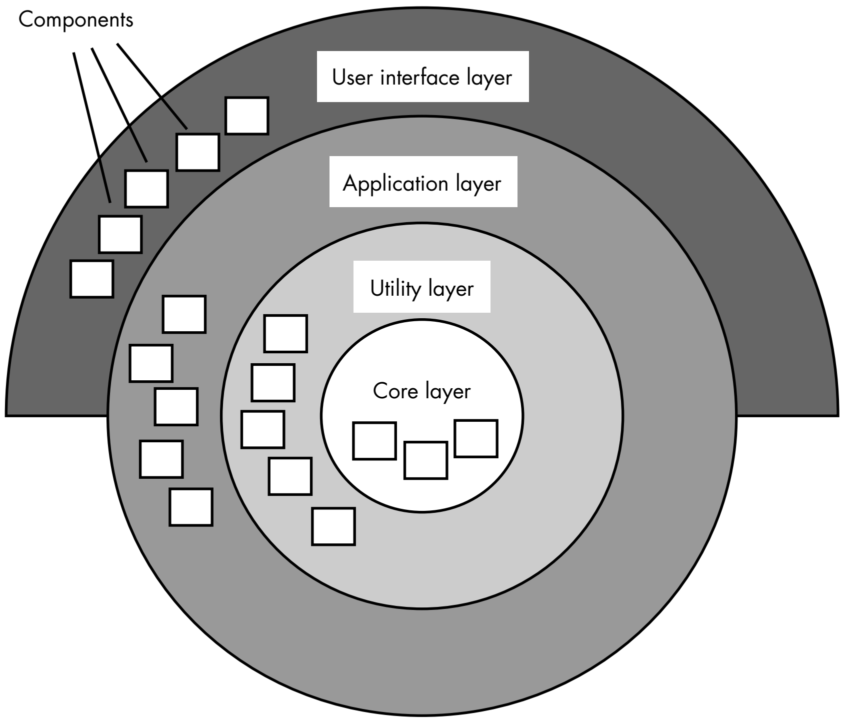
Object-oriented architectures. The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components is accomplished via message passing.

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.

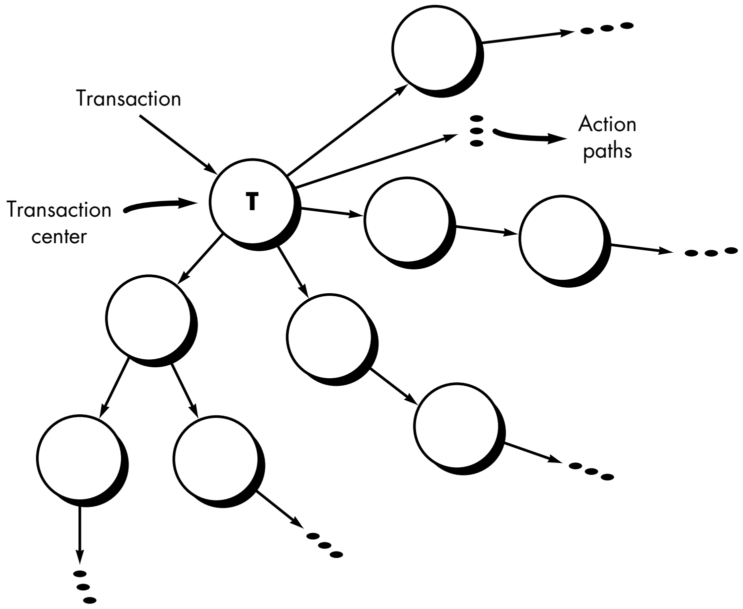


FIGURE 14.4 Transaction flow

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.

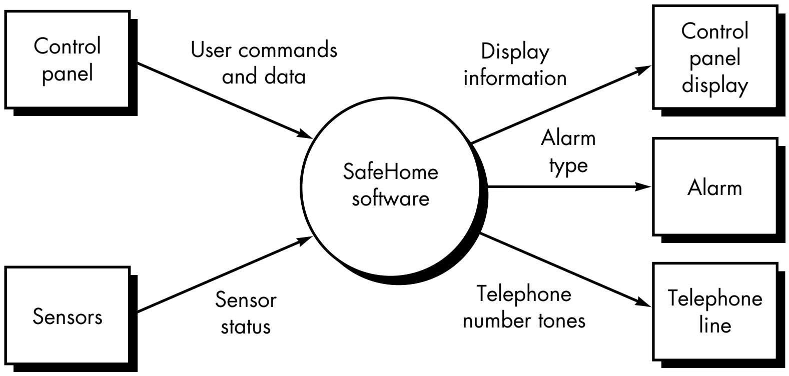
14.6 TRANSFORM MAPPING

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.

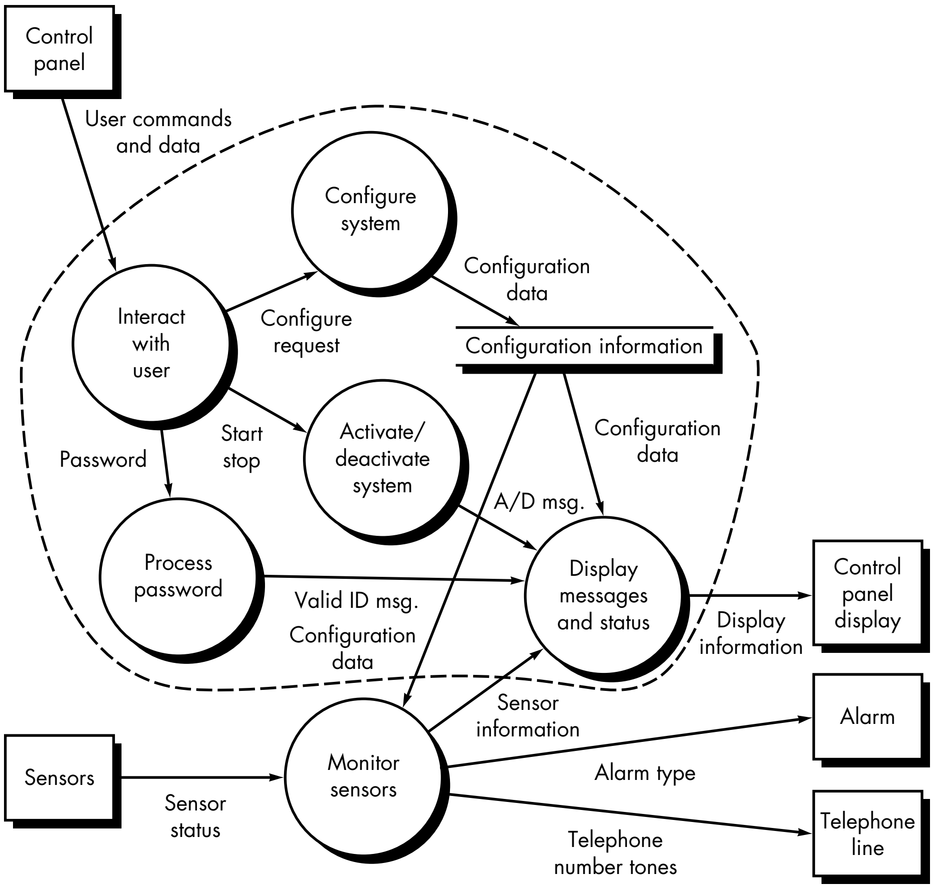
ADVICE

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

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.

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.

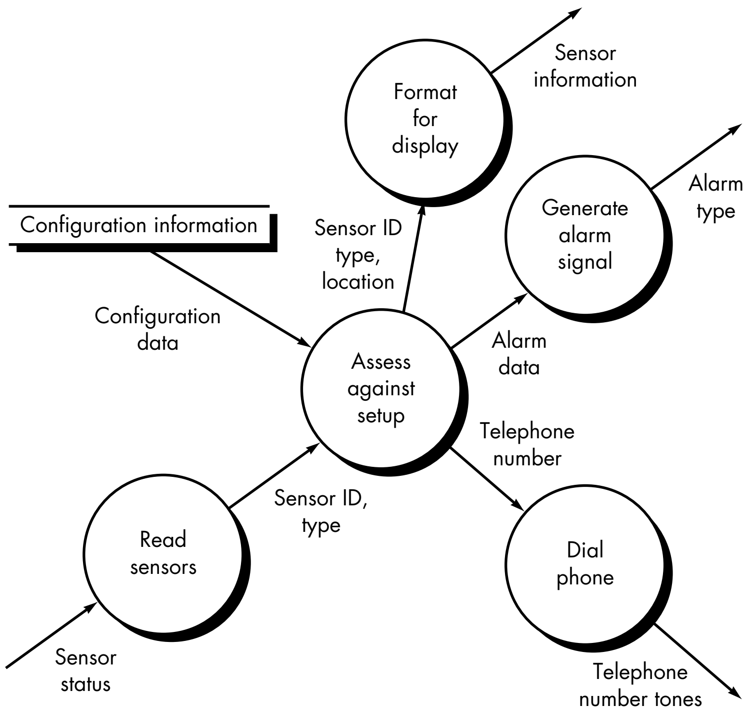


FIGURE 14.7
Level 2 DFD that refines the monitor sensors process

ADVICE

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.

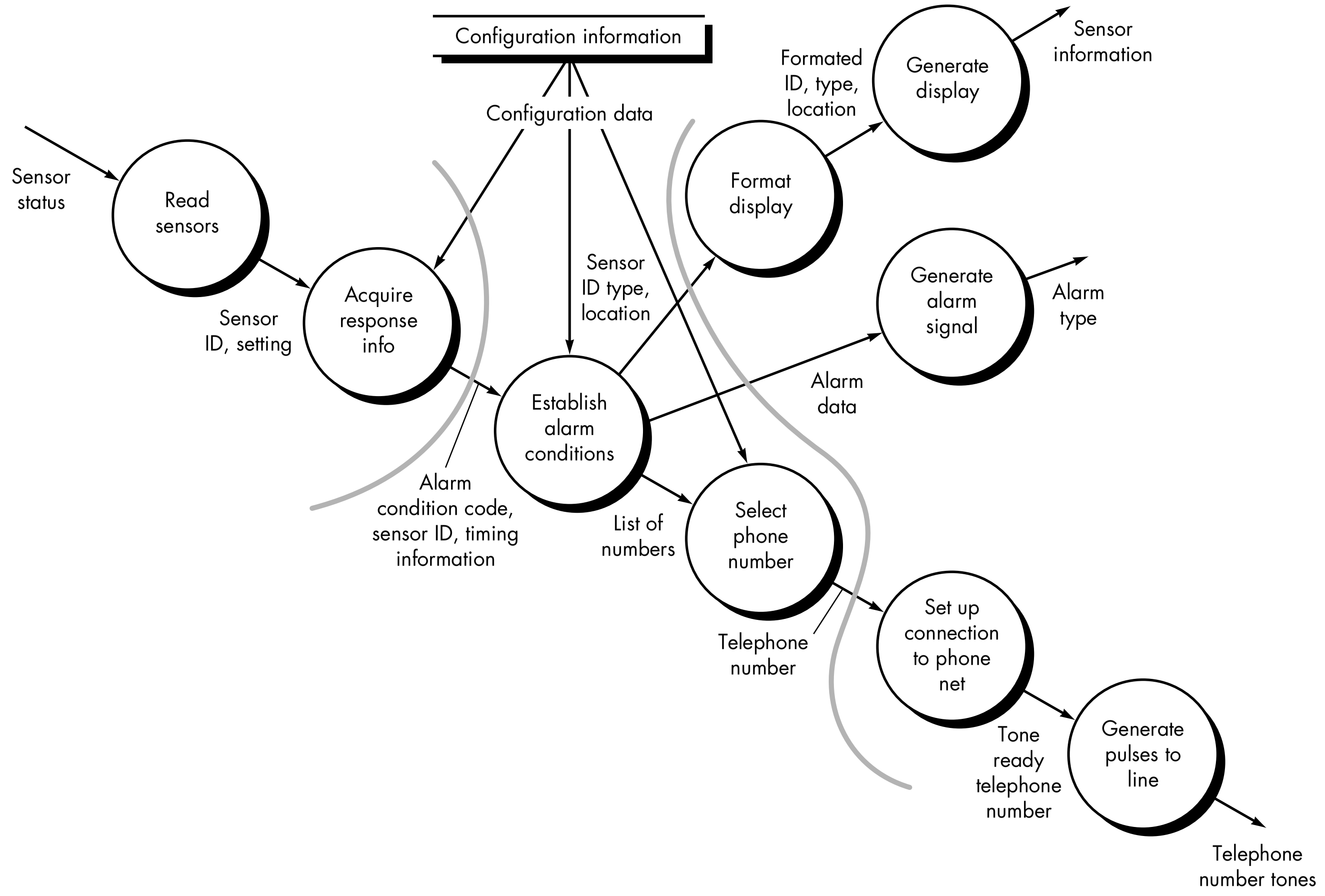
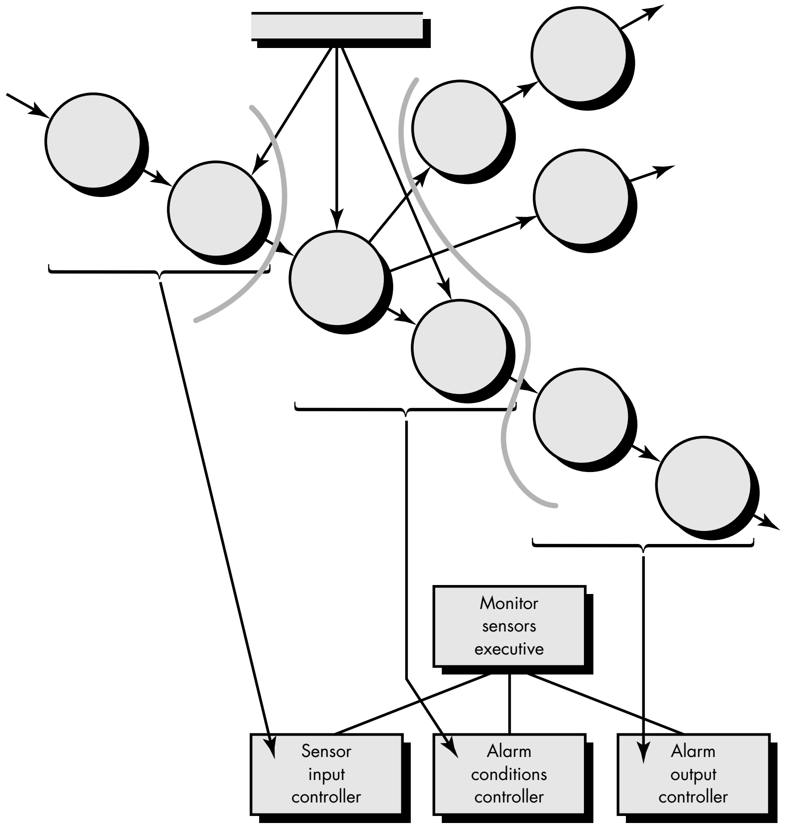


FIGURE 14.8 Level 3 DFD for monitor sensors with flow boundaries

FIGURE 14.9
First-level factoring for monitor sensors



ADVICE

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.

ADVICE

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.

ADVICE

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.

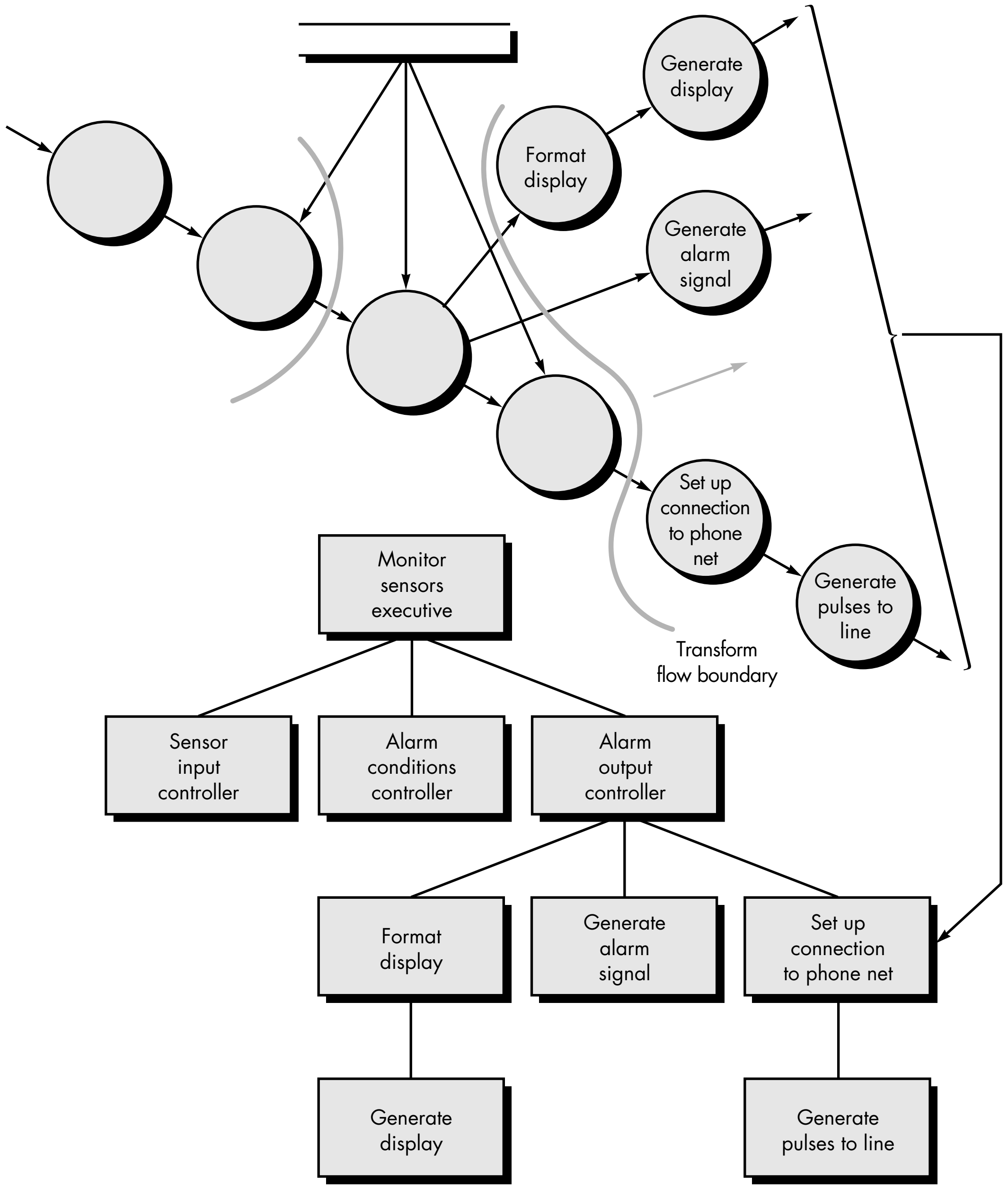


FIGURE 14.10 Second-level factoring for monitor sensors

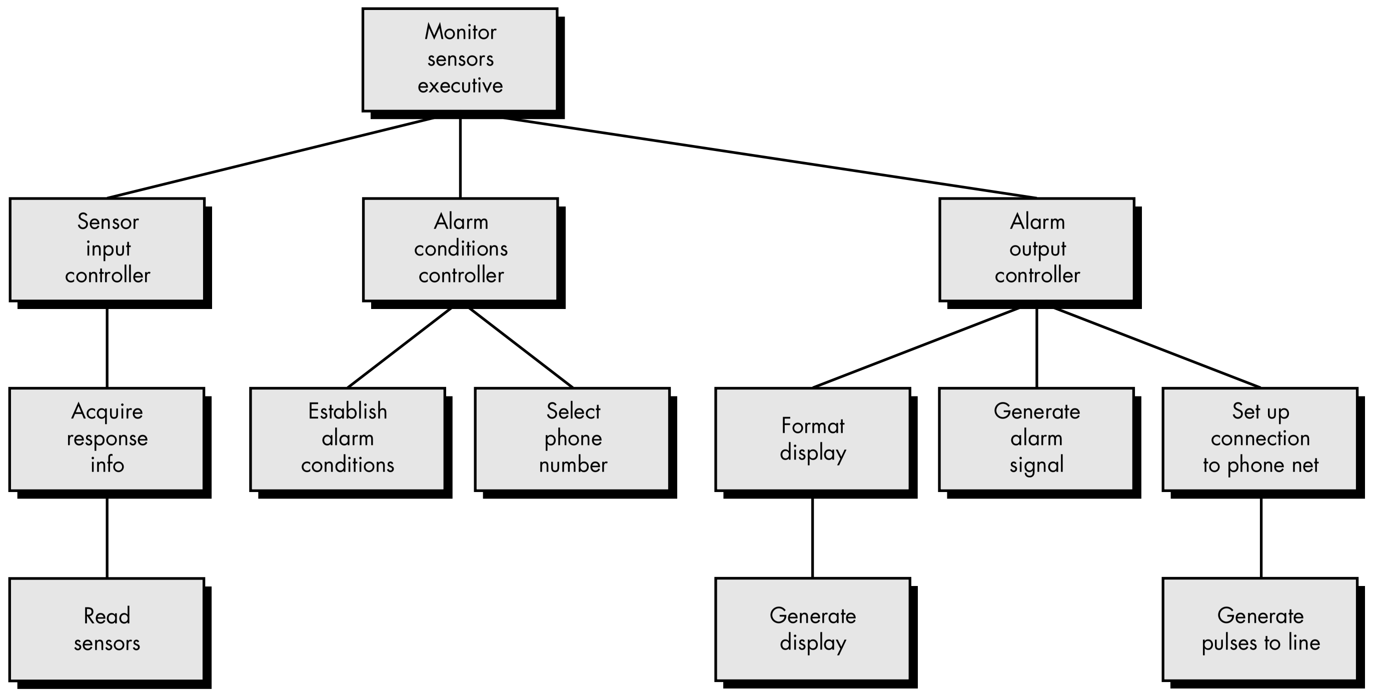


FIGURE 14.11 "First-iteration" program structure for monitor sensors

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.

ADVICE

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.

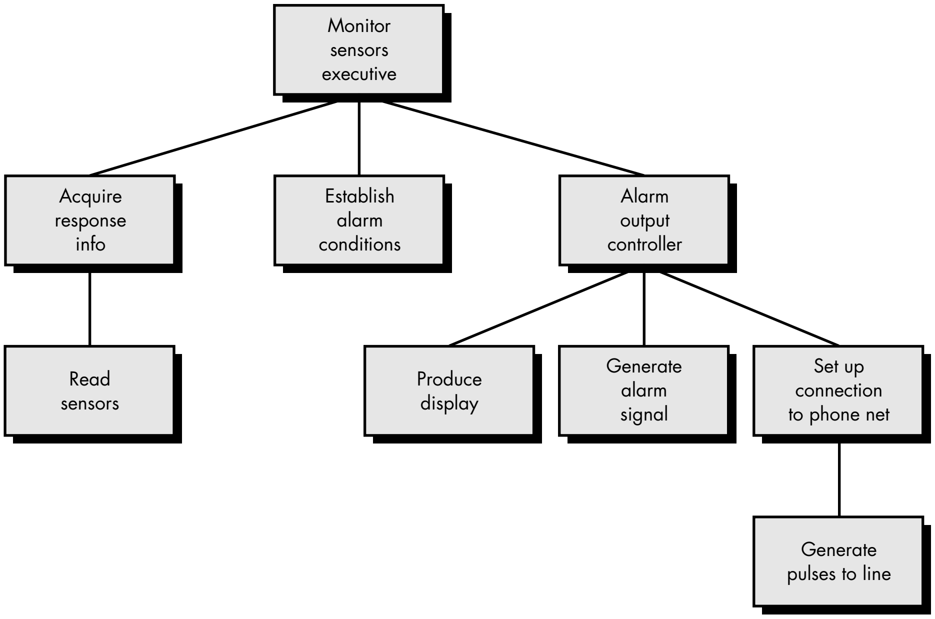
The objective of the preceding seven steps is to develop an architectural representation of software. That is, once structure is defined, we can evaluate and refine software architecture by viewing it as a whole. Modifications made at this time require little additional work, yet can have a profound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

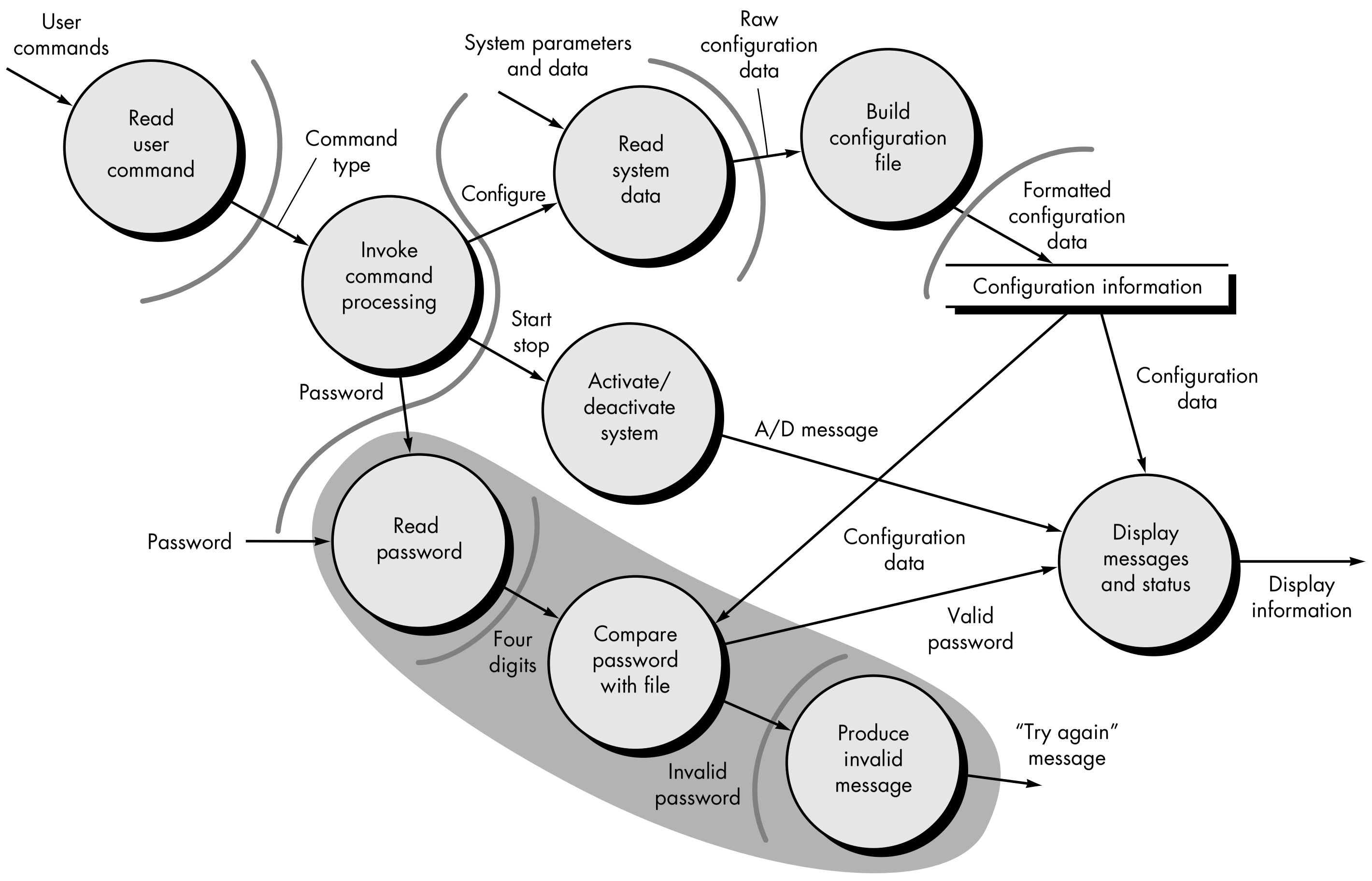


FIGURE 14.13 Level 2 DFD for user interaction subsystem with flow boundaries

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.

KEY POINT

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

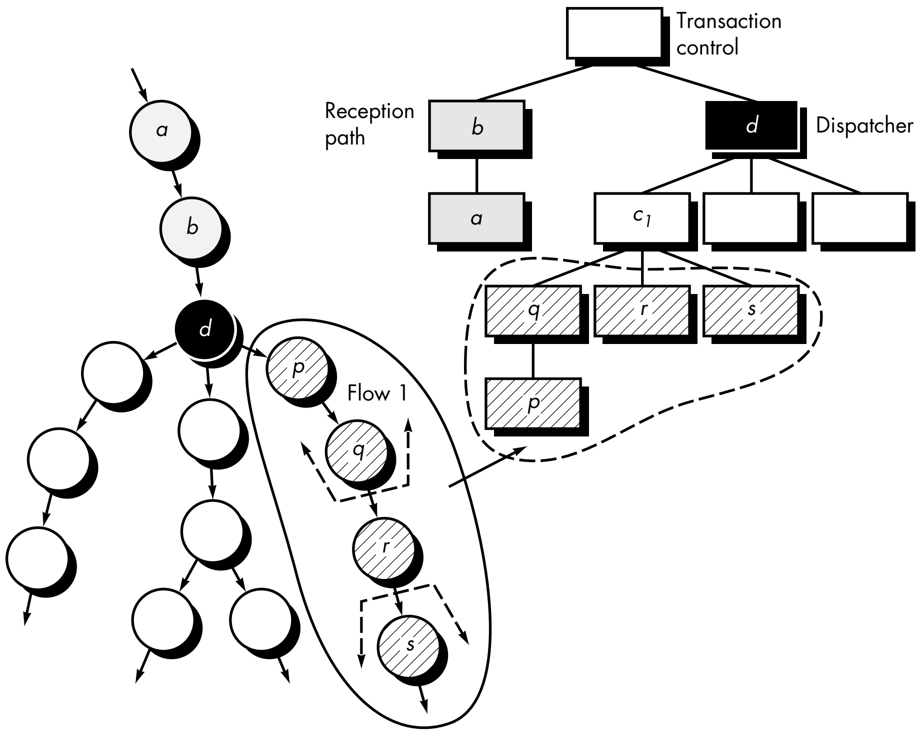


FIGURE 14.14
Transaction mapping

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.

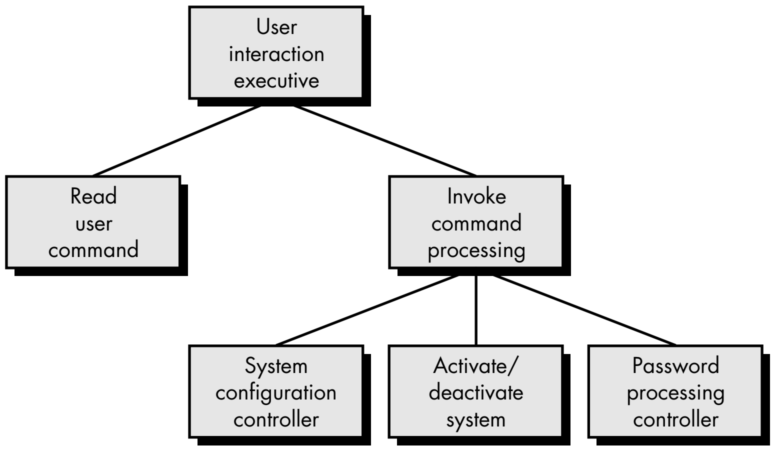

User interaction executive
Read user command
Invoke command processing
System configuration controller
Activate/ deactivate system
Password processing controller

FIGURE 14.15
First-level factoring for user interaction subsystem

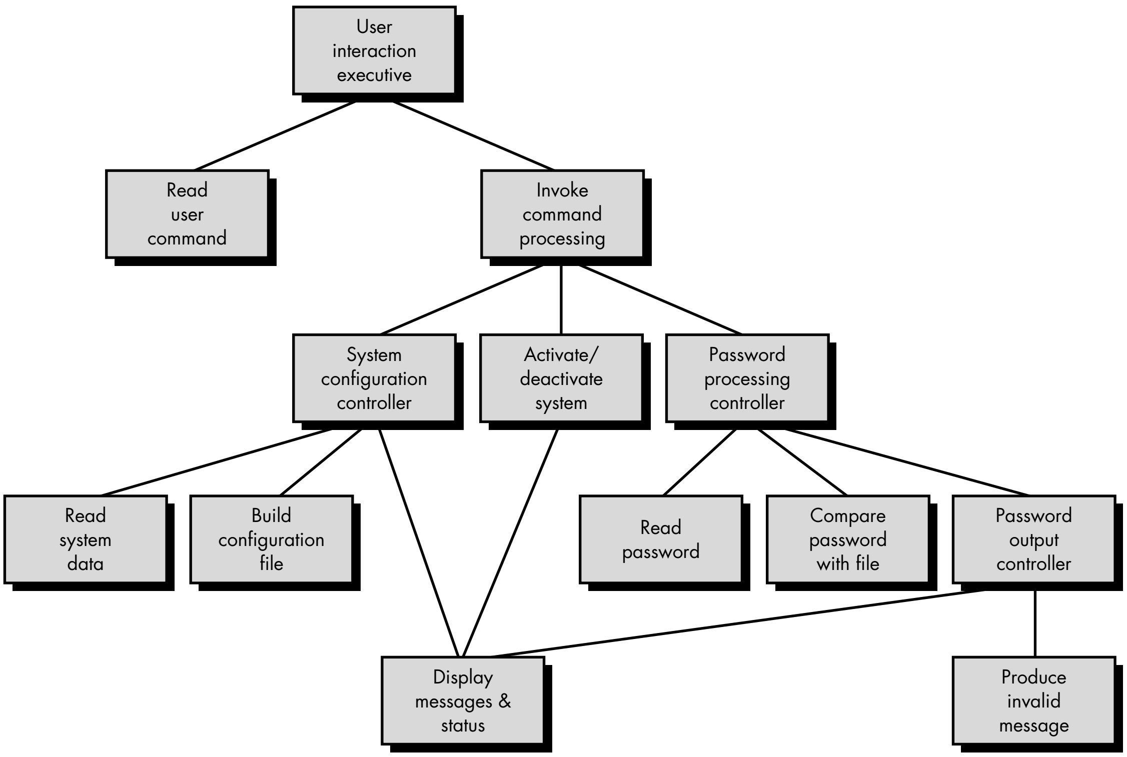

User interaction executive
Read user command
Invoke command processing
System configuration controller
Activate/ deactivate system
Password processing controller
Read system data
Build configuration file
Read password
Compare password with file
Password output controller
Display messages & status
Produce invalid message

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

actions	410
design models	405
error processing	414
golden rules	402
help facility	413
design evaluation	406
design process	407
interface objects	410
memory load	404
response time	413
user scenario	411
user types	406
variability	413

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.