Chapter 5

# System Design

## 5.1 SYSTEM/SOFTWARE DESIGN

Design is a meaningful representation of something that is to be built. It can be traced to a customer's requirements and at the same time assessed for quality against a set of predefined criteria for "good" design.

A set of design concepts has evolved over the years. According to M.A. Jackson, *"The beginning of wisdom for a software engineer is to recognize the difference between getting a program to work and getting it right."* The various design concepts discussed in this chapter provide the necessary framework for "getting it right."

### 5.1.1 Definition of Software Design

The definitions of software design are as diverse as design methods. Some important software design definitions are outlined below.

**According to Coad and Yourdon.** *Software Design is the practice of taking a specification of externally observable behavior and adding details needed for actual computer system implementation, including human interaction, task management, and data management details.*

**According to Webster.** *In a sense, design is representation of an object being created. A design information base that describes aspects of this object, and the design process can be viewed as successive elaboration of representations, such as adding more information or even backtracking and exploring alternatives.*

**According to Stevens.** *Software Design is the process of inventing and selecting programs that meet the objectives for software systems.*

Input includes an understanding of the following:

- Requirements
- Environmental constraints
- Design criteria

The output of the design effort is composed of the following:

- Architecture design which shows how pieces are interrelated
- Specifications for any new pieces
- Definitions for any new data

### 5.1.2 Design Objectives/Properties

The various desirable properties or objectives of software design are:

1. ***Correctness.*** The design of a system is correct if the system built precisely according to the design satisfies the requirements of that system. Clearly, the goal during the design phase is to produce correct designs. However, correctness is not the sole criterion during the design phase, as there can be many correct designs. The goal of the design process is not simply to produce a design for the system. Instead, the goal is to find the best possible design within the limitations imposed by the requirements and the physical and social environment in which the system will operate.
2. ***Verifiability.*** Design should be correct and it should be verified for correctness. Verifiability is concerned with how easily the correctness of the design can be checked. Various verification techniques should be easily applied to design.
3. ***Completeness.*** Completeness requires that all the different components of the design should be verified, i.e., all the relevant data structures, modules, external interfaces, and module interconnections are specified.
4. ***Traceability.*** Traceability is an important property that can get design verification. It requires that the entire design element be traceable to the requirements.
5. ***Efficiency.*** Efficiency of any system is concerned with the proper use of scarce resources by the system. The need for efficiency arises due to cost

considerations. If some resources are scarce and expensive, it is desirable that those resources be used efficiently. In computer systems, the resources that are most often considered for efficiency are processor time and memory. An efficient system consumes less processor time and memory.

6. ***Simplicity.*** Simplicity is perhaps the most important quality criteria for software systems. Maintenance of a software system is usually quite expensive. The design of the system is one of the most important factors affecting the maintainability of the system.

### 5.1.3 Design Principles

The three design principles are as follows:

- Problem partitioning
- Abstraction
- Top-down and Bottom-up design

1. ***Problem Partitioning.*** When solving a small problem, the entire problem can be tackled at once. For solving larger problems, the basic principle is the time-tested principle of "divide and conquer." This principle suggests dividing into smaller pieces, so that each piece can be conquered separately.

   For software design, therefore, the goal is to divide the problem into manageably small pieces that can be solved separately. The basic rationale behind this strategy is the belief that if the pieces of a problem are solvable separately, the cost of solving the entire problem is more than the sum of the cost of solving all the pieces.

   However, the different pieces cannot be entirely independent of each other as they together form the system. The different pieces have to cooperate and communicate to solve the larger problem. This communication adds complexity, which arises due to partitioning and may not have existed in the original problem. As the number of components increases, the cost of partitioning, together with the cost of this added complexity, may become more than the savings achieved by partitioning. It is at this point that no further partitioning needs to be done. The designer has to make the judgment about when to stop partitioning.

   Problem partitioning can be divided into two categories:

   (*i*) Horizontal partitioning

   (*ii*) Vertical partitioning

   (*i*) *Horizontal Partitioning.* Horizontal partitioning defines separate branches of modular hierarchy for each major program function. The simplest

approach to horizontal partitioning defines three partitions: input, data transformation (often called processing), and output. Partitioning their architecture horizontally provides a number of distinct benefits:

- Software that is easier to test.
- Software that is easier to maintain.
- Software that is easier to extend.
- Propagation of fewer side effects.

Conversely, horizontal partitioning often causes more data to be passed across module interfaces and can complicate the overall control of program flow.

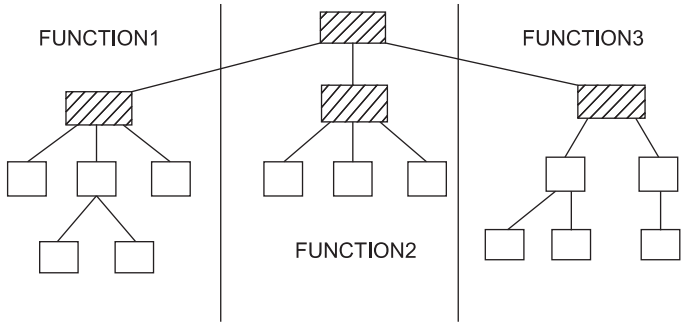



**FIGURE 5.1** Horizontal Partitioning

(*ii*) *Vertical Partitioning.* Vertical partitioning, often called factoring, suggests that control and work should be distributed from top-down in the program structure. Top-level modules should perform control functions and do actual processing work. Modules that reside low in the structure should be the workers, performing all input, compilation, and output tasks.

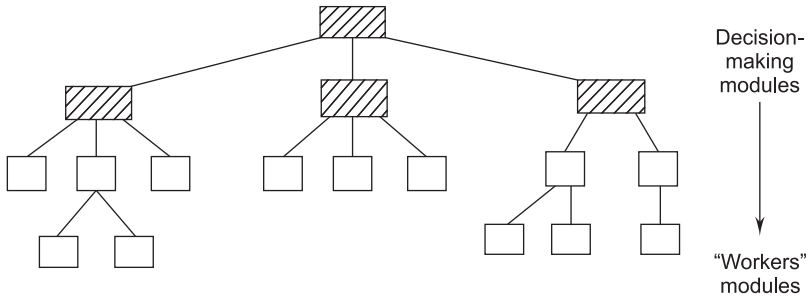



**FIGURE 5.2** Vertical Partitioning

2. ***Abstraction.*** An abstraction of a component describes the external behavior of that component without bothering with the internal details that produce the behavior.

   Abstraction is an indispensable part of the design process and it is essential for problem partitioning. Partitioning essentially is the exercise in determining the components of a system. However, these components are not isolated from each other, but interact with other components. In order to allow the designer to concentrate on one component at a time, abstraction of other components is used.

   Abstraction is used for existing components as well as components that are being designed. Abstraction of existing components plays an important role in the maintenance phase.

   During the design process, abstractions are used in a reverse manner not in the process of understanding a system. During design, the components do not exist, and in the design the designer specifies only the abstract specifications of the different components. The basic goal of system design is to specify the modules in a system and their abstractions. Once the different modules are specified, during the detailed design the designer can concentrate on one module at a time. The task in detailed design and implementation is essentially to implement the modules so that the abstract specifications of each module are satisfied.

   There are two common abstraction mechanisms for software systems: Functional abstraction and data abstraction. In functional abstraction, a module is specified by the function it performs. For example, a module to sort an input array can be represented by the specification of sorting. Functional abstraction is the basis of partitioning in function-oriented approaches. That is, when the problem is being partitioned, the overall transformation function for the system is partitioned into smaller functions that comprise the system function.

   The second unit for abstraction is data abstraction. There are certain operations required from a data object, depending on the object and the environment in which it is used. Data abstraction supports this view. Data is not treated simply as objects, but is treated as objects with some predefined operations on them. The operations defined on a data object are the only operations that can be performed on those objects. From outside an object, the internals of the object are hidden; only the operations on the object are visible.

3. ***Top-down and Bottom-up Design.*** A system consists of components, which have components of their own; indeed a system is a hierarchy of components. The highest-level components correspond to the total system.

To design such hierarchies there are two possible approaches: top-down and bottom-up. The top-down approach starts from the highest-level component of the hierarchy and proceeds through to lower levels. By contrast, a bottom-up approach starts with the lowest-level component of the hierarchy and proceeds through progressively higher levels to the top-level component.

A top-down design approach starts by identifying the major components of the system, decomposing them into their lower-level components and iterating until the desired level of detail is achieved. Top-down design methods often result in some form of stepwise refinement. Starting from an abstract design, in each step the design is refined to a more concrete level, until we reach a level where no more refinement is needed and the design can be implemented directly. The top-down approach has been promulgated by many researchers and has been found to be extremely useful for design. Most design methodologies are based on the top-down approach.

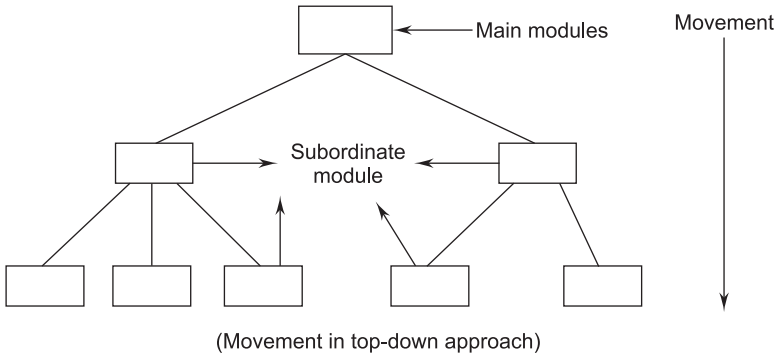



**FIGURE 5.3** Top-Down Approach

A bottom-up design approach starts with designing the most basic or primitive components and proceeds to higher-level components that use these lower-level components. Bottom-up methods work with layers of abstraction. Starting from the very bottom, operations that provide a layer of abstraction are implemented. The operations of this layer are then used to implement more powerful operations and still a higher layer of abstraction, until the stage is reached where the operations supported by the layer are those desired by the system.

A top-down approach is suitable only if the specifications of the system are clearly known and the system development is from scratch. However, if a system is to be built from an existing system, a bottom-up approach is more suitable, as it starts from some existing components. So, for example, if an iterative enhancement type of process is being followed, in later iterations, the bottom-up approach could be more suitable (in the first iteration a top-down approach can be used).

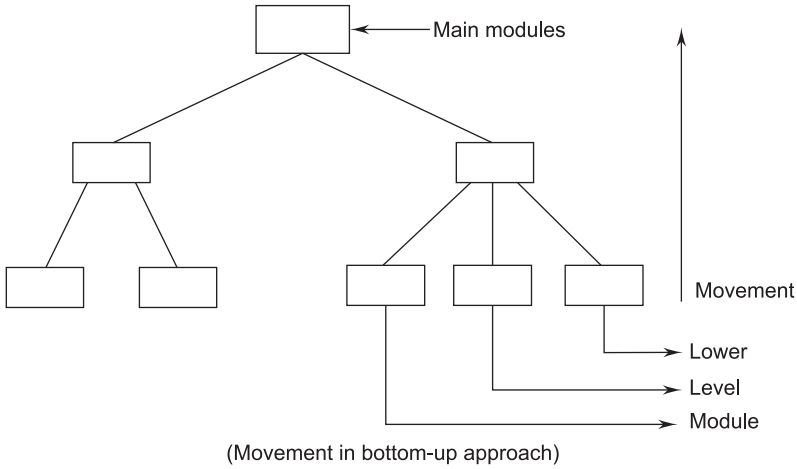



**FIGURE 5.4** Bottom-Up Approach

## 5.2 ARCHITECTURAL DESIGN

Large systems are always decomposed into subsystems that provide some related set of services. The initial design process of identifying these subsystems and establishing a framework for subsystem control and communication is called architectural design.

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.

Architectural design methods have a look into various architectural styles for designing a system. These are:

- Data-centric architecture
- Data-flow architecture
- Object-oriented architecture
- Layered architecture

Data-centric architecture involves the use of a central database operation of inserting and updating it in the form of a table. Data-flow architecture is central around the pipe and filter mechanism. This architecture is applied when input data takes the form of output after passing through various phases of transformations. These transformations can be via manipulations or various computations done

on the data. In object-oriented architecture the software design moves around the clauses and objects of the system. The class encapsulates the data and methods. Layered architecture defines a number of layers and each layer performs tasks. The outer-most layer handles the functionality of the user interface and the inner-most layer mainly handles interaction with the hardware.

### 5.2.1 Objectives of Architectural Design

The objective of architectural design is to develop a model of software architecture, which gives an overall organization of the program module in the software product. Software architecture encompasses two aspects of structures of the data and hierarchical structures of the software components. Let us see how a single problem can be translated to a collection of solution domains (see Figure 5.5).

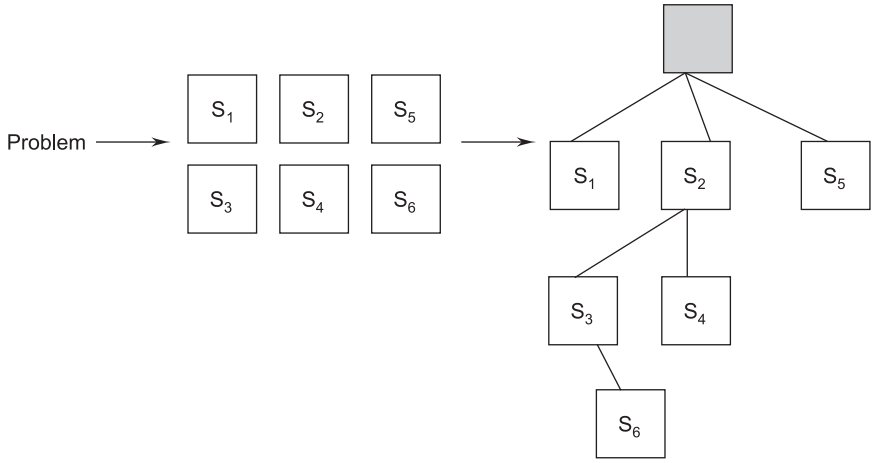



**FIGURE 5.5** Problems, Solutions, and Architecture

Architectural design defines the organization of program components. It does not provide the details of each component and its implementation. Figure 5.6 depicts the architecture of a financial accounting system.

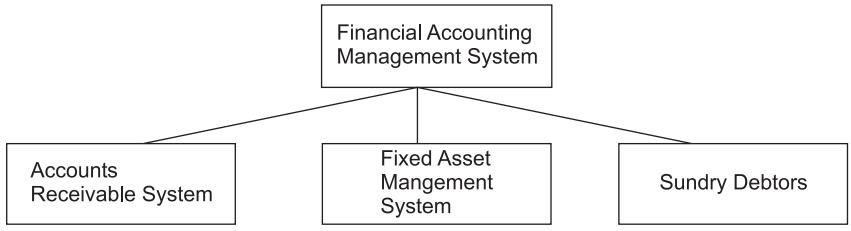



**FIGURE 5.6** Architecture of a Financial Accounting System

The objective of architectural design is also to control the relationship between modules. One module may control another module or may be controlled by another module. These characteristics are defined by the fan-in and fan-out of a particular module. The organization of a module can be represented by a tree-like structure.

The number of levels of a component in the structure is called depth and the number of components across the horizontal section is called width. The number of components, which controls the component, is called fan-in, i.e., the number of incoming edges to a component. The number of components that are controlled by the module is called fan-out, i.e., the number of outgoing edges.

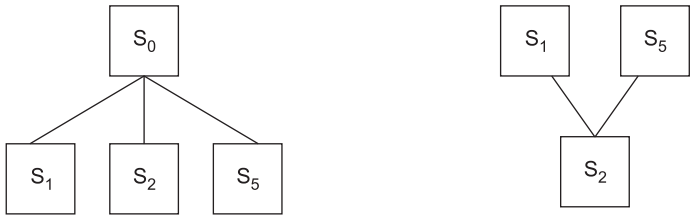



**FIGURE 5.7** Fan-in and Fan-out

$S_0$ controls three components, hence, the fan-out is 3. $S_2$ is controlled by two components, namely, $S_1$ and $S_2$, hence, the fan-in is 2 (see Figure 5.7).

## 5.3 LOW-LEVEL DESIGN

### 5.3.1 Modularization

A system is considered modular if it consists of discreet components so that each component can be implemented separately, and a change to one component has minimal impact on other components.

There are many definitions of the term module. They range from "a module is a FORTRAN subroutine" to "a module is an ADA package" to "a module is a work assignment for an individual programmer." All of these definitions are correct, in the sense that modular systems incorporate collections of abstractions in which each functional abstraction, each data abstraction, and each control abstraction handles a local aspect of the problem being solved. Modular system consists of well-defined, manageable units with well-defined interfaces among the units. Desirable properties of a modular system include:

- Each function in each abstraction has a single, well-defined purpose.
- Each function manipulates no more than one major data structure.

- Functions share global data selectively. It is easy to identify all routines that share a major data structure.
- Functions that manipulate instances of abstract data types are encapsulated with the data structure being manipulated.

Modularity enhances design clarity, which in turn eases implementation, debugging, testing, documenting, and maintenance of the software product.

Modules that may be created during program modularizations are:

- **Process support modules:** In these all the functions and data items that are required to support a particular business process are grouped together.
- **Data abstraction modules:** These are abstract types that are created by associating data with processing components.
- **Functional modules:** In these all the functions that carry out similar or closely related tasks are grouped together.
- **Hardware modules:** In these all the functions, which control particular hardware are grouped together.

### Classification of Modules

A module can be classified into three types depending on the activating mechanism.

- An incremental module is activated by an interruption and can be interrupted by another interrupt during the execution prior to completion.
- A sequential module is a module that is referenced by another module and without interruption of any external software.
- Parallel modules are executed in parallel with other modules.

The main purpose of modularity is that it allows the principle of separation of concerns to be applied in two phases: when dealing with the details of each module in isolation (and ignoring the details of other modules) and when dealing with the overall characteristics of all modules and their relationships in order to integrate them into a coherent system. If the two phases are temporally executed in the order mentioned, then we say that the system is designed bottom-up; the converse denotes top-down design.

### Advantages of Modular Systems

- Modular systems are easier to understand and explain because their parts are functionally independent.
- Modular systems are easier to document because each part can be documented as an independent unit.
- Programming individual modules is easier because the programmer can focus on just one small, simple problem rather than a large complex problem.

- Testing and debugging individual modules is easier because they can be dealt with in isolation from the rest of the program.
- Bugs are easier to isolate and understand, and they can be fixed without fear of introducing problems outside the module.
- Well-composed modules are more reusable because they are more likely to comprise part of a solution to many problems. Also, a good module should be easy to extract from one program and insert into another.

Modularity is an important property of most engineering processes and products. For example, in the automobile industry, the construction of cars proceeds by assembling building blocks that are designed and built separately. Furthermore, parts are often reused from model to model, perhaps after minor changes. Most industrial processes are essentially modular, made out of work packages that are combined in simple ways (sequentially or overlapping) to achieve the desired result.

### 5.3.2 Structure Charts

The structure chart is one of the most commonly used methods for system design. Structure charts are used during architectural design to document hierarchical structures, parameters, and interconnections in a system.

It partitions a system into black boxes. A black box means that functionality is known to the user without the knowledge of internal design. Inputs are given to a black box and appropriate outputs are generated by the black box. This concept reduces complexity because details are hidden from those who have no need or desire to know. Thus, systems are easy to construct and easy to maintain. Here, black boxes are arranged in hierarchical format as shown in Figures 5.8 (*a*) and (*b*).

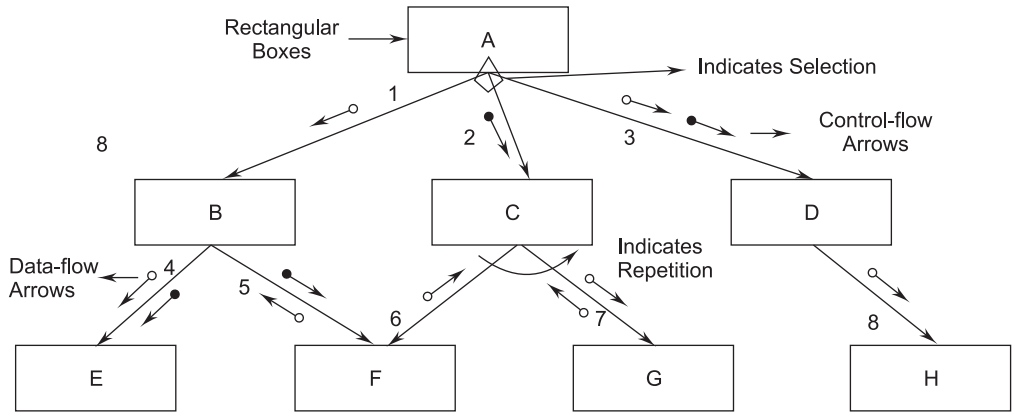



**FIGURE 5.8 (a)** Hierarchical Format of a Structure Chart

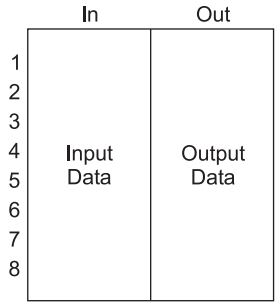


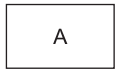

**FIGURE 5.8 (b)** Format of a Structure Chart

Modules at the top level call the modules at the lower level. The connections between modules are represented by lines between the rectangular boxes. The components are generally read from top to bottom, left to right. Modules are numbered in a hierarchical numbering scheme. In any structure chart there is one and only one module at the top called the root.

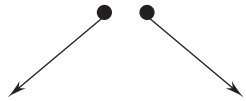
### Basic Building Blocks of a Structure Chart

The basic building blocks of a structure chart are the following:

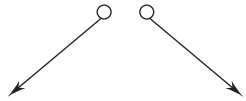
1. **Rectangular Boxes.** A rectangular box represents a module. Usually a rectangular box is annotated with the name of the module it represents.



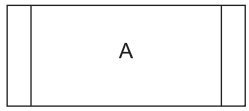

2. **Arrows.** An arrow connecting two modules implies that during program execution, control is passed from one module to the other in the direction of the connecting arrow.



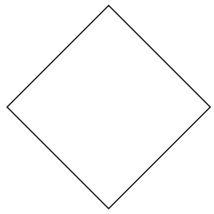
3. **Data-flow Arrows.** Data-flow arrows represent that the named data passes from one module to the other in the direction of the arrow.



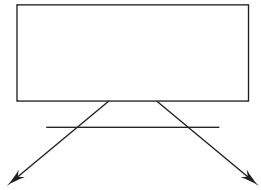
4. **Library Modules.** Library modules are the frequently called modules and are usually represented by a rectangle with double edges. Usually when a module is invoked by many other modules, it is made into a library module.




5. **Selection.** The diamond symbol represents that one module out of several modules connected with the diamond symbol are invoked depending on the condition satisfied, which is written in the diamond symbol.



6. **Repetitions.** A loop around the control-flow arrows denotes that the respective modules are invoked repeatedly.



**Example 5.1.** A software system called RMS calculating software reads three integral numbers from the user in the range between –1000 and +1000 and determines the root mean square (rms) of the three input numbers and then displays it.

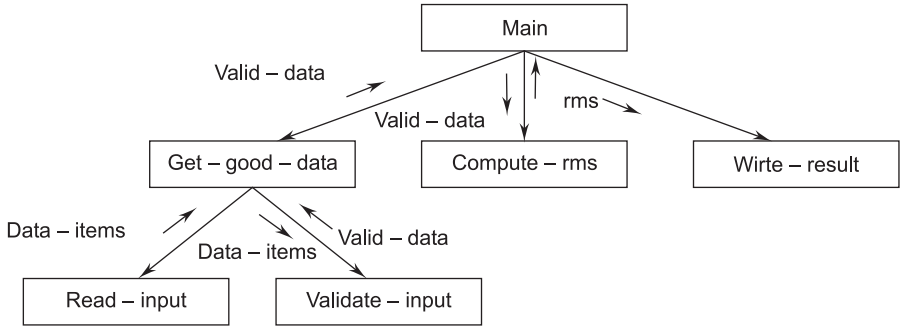



**FIGURE 5.9** Structure Chart For Example 5.1

### 5.3.3 Pseudo-Code

"Pseudo" means imitation or false and "code" refers to the instructions written in a programming language. Pseudo-code notation can be used in both the preliminary and detailed design phases. Using pseudo-code, the designer describes system characteristics using short, concise English language phrases that are structured by keywords, such as If-Then-Else, While-Do, and End. Keywords and indentation describe the flow of control, while the English phrases describe processing actions. Pseudo-code is also known as program-design language or structured English. A program-design language should have the following characteristics:

- A fixed syntax of keywords that provide for all structured constructs, data declarations, and modularity characteristics.
- A free syntax of natural language that describes a processing feature.
- A data-declaration facility.
- A subprogram definition and calling techniques.

#### Advantages of Pseudo-Code

The various advantages of pseudo-code are as follows:

- Converting a pseudo-code to a programming language is much easier compared to converting a flowchart or decision table.
- Compared to a flowchart, it is easier to modify the pseudo-code of program logic whenever program modifications are necessary.
- Writing of pseudo-code involves much less time and effort than the equivalent flowchart.
- Pseudo-code is easier to write than writing a program in a programming language because pseudo-code as a method has only a few rules to follow.

#### Disadvantages of Pseudo-Code

The various disadvantages of pseudo-code are as follows:

- In the case of pseudo-code, a graphic representation of program logic is not available as with flowcharts.
- There are no standard rules to follow in using pseudo-code. Different programmers use their own style of writing pseudo-code and hence communication problems occur due to lack of standardization.
- For a beginner, it is more difficult to follow the logic or write the pseudo-code as compared to flowcharting.

The use of pseudo-code for detailed design specification is illustrated in Figure 5.9 (*a*).

```
INITIALIZE tables and counters; OPEN files
READ the first text record
WHILE there are more text records DO
WHILE there are more words in the text record DO
EXTRACT the next word
SEARCH word_table for the extracted word
IF the extracted word is found THEN
INCREMENT the extracted word's occurrence count
ELSE
INSERT the extracted word into the word_table
ENDIF
INCREMENT the words_processed counter
ENDWHILE at the end of the text record
ENDWHILE when all text records have been processed
PRINT the word_table and the words_processed counter
CLOSE files
TERMINATE the program
```

**FIGURE 5.9 (a)** An Example of a Pseudo-Code Design Specification

Pseudo-code consists of English-like statements describing an algorithm. It is written using simple phrases and avoids cryptic symbols. It is independent of high-level languages and is a very good means of expressing an algorithm. It is written in a structured manner and indentation is used to increase clarity.

### 5.3.4 Flowcharts

A flowchart is a convenient technique to represent the flow of control in a program. A flowchart is a pictorial representation of an algorithm that uses symbols to show the operations and decisions to be followed by a computer in solving a problem. The actual instructions are written within symbols/boxes using clear statements. These boxes are connected by solid lines having arrow marks to indicate the flow of operation in a sequence.

In fact, flowcharts are the plan to be followed when the program is written. Expert programmers may write programs without drawing the flowcharts. But for a beginner it is recommended that a flowchart should be drawn before writing a program, which in turn will reduce the number of errors and omissions

in the program. Flowcharts also help during testing and modifications in the programs.

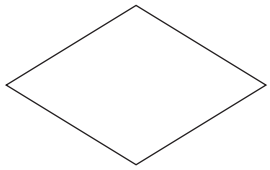
### Flowchart Symbols

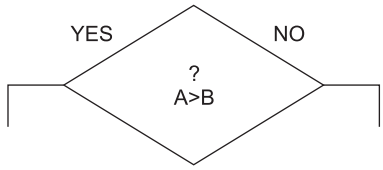
1. **Terminal Symbol.** Terminal symbols are used for two purposes: to define the starting (START or BEGIN) point of the flowchart and to define the ending point (END) of the flowchart.

2. **Input/Output Symbol.** Input/output symbols are used to indicate the logical positioning of input/output operations. The input operation is the entry of computer data and the output operation is the displayed output operation.

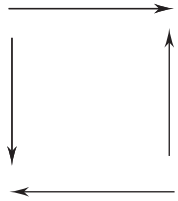
3. **Processing Symbol.** Processing symbols are used to indicate the arithmetic and data-movement instructions. Therefore, all arithmetic processing of adding, subtracting, multiplying, and dividing are represented with a processing symbol box.

4. **Decision Symbol.** Decision symbols have one entry point and there will be at least two exit points depending upon the decision taken inside the symbol. When a condition is tested, if the condition is true, the path for "yes" is followed. If the condition is false, the path for "no" is followed.

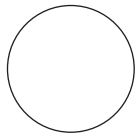





5. **Flow Lines.** Flow lines, which have arrowheads, are used to indicate the flow of program logic in a flowchart. These arrows are used to indicate the direction of the flow of control. This means these statements indicate the next statement to be executed.



6. **Connector Symbol.** If a flowchart is discontinued at some point and continued again in another place, the connector symbol is used. It is a circle with a number written inside it. If a flowchart is discontinued at some point, a circle is drawn pointing away from the chart. Another circle with the same number inside is placed where the flowchart is continued.



7. **Hexagon (Flat).** This is the preparation box. This box contains the loop-setting statement, i.e., some iterative statement.

### Flowchart Drawing Rules

Important rules and guidelines used for drawing flowcharts are:

- Only conventional flowchart symbols should be used.
- Arrows can be used to indicate the flow of control in the problem. However, flow lines should not cross each other.
- Processing logic should flow from top to bottom and from left to right.
- Words in the flowchart symbols should be common statements and easy to understand. These should be independent of programming languages.
- Be consistent in using names and variables in the flowchart.
- If the flowchart becomes large and complex then connector symbols should be used to avoid crossing of flow lines.
- Properly labeled connectors should be used to link the portions of the flowchart on different pages.
- Flowcharts should have start and stop points.

### Advantages of Flowcharts

The various advantages of flowcharts are as follows:

- **Synthesis.** Flowcharts are used as working models in designing new programs and software systems.
- **Documentation.** Program documentation consists of activities, such as collecting, organizing, storing, and maintaining all related records of a program.
- **Coding.** Flowcharts guide the programmer in writing the actual code in a high-level language, which is supposed to give an error-free program developed expeditiously.
- **Debugguing.** The errors in a program are detected only after its execution on a computer. These errors are called bugs and the process of removing these errors is called debugging. In the debugging process, a flowchart acts as an important tool in detecting, locating, and removing bugs from a program.
- **Communication.** A flowchart is a pictorial representation of a program. Therefore, it is an excellent communication technique to explain the logic of a program to other programmers/people.
- **Analysis.** Effective analysis of a logical problem can be easily done with the help of a related flowchart.
- **Testing.** A flowchart is an important tool in the hands of a programmer, which helps him in designing the test data for systematic testing of programs.

### Limitations of Flowcharts

The various limitations of flowcharts are as follows:

- The drawing of flowcharts is a very time-consuming process and laborious especially for large, complex problems.
- The redrawing of flowcharts is even more difficult and time consuming. It is very difficult to include any new step in the existing flowchart; redrawing of the flowchart is the only solution.
- There are no standards, which specify the detail that should be included in any flowchart.
- If an algorithm has complex branches and loops, flowcharts become very difficult to draw.
- Sometimes flowcharts are not as detailed as desired.

#### Example of a Flowchart

As an example, consider an algorithm to find the average of $n$ numbers. The flowchart is shown in Figure 5.10 followed by the algorithm. Here $n$ is the integer

variable denoting the number of values considered for computing the average. Count is another integer variable denoting the number of values that are processed at any instant. The number is an integer variable for storing the values.

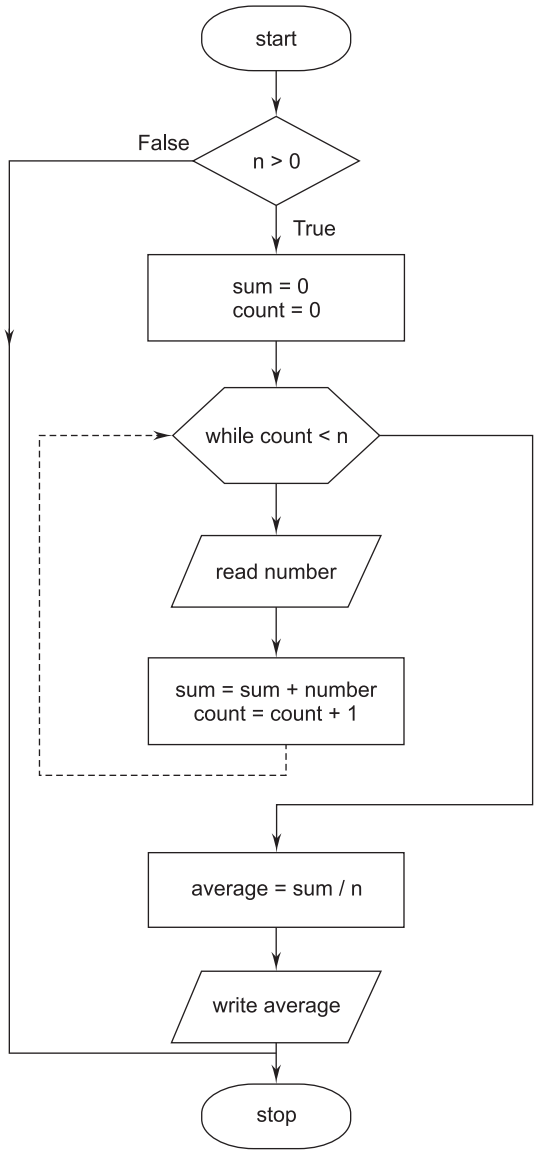



**FIGURE 5.10** Flowchart to Find the Average of *n* Numbers

### 5.3.5 Difference Between Flowcharts and Structure Charts

A structure chart differs from a flowchart in the following ways:

- It is usually difficult to identify different modules of the software from its flowchart representation.
- Data interchange among different modules is not represented in a flowchart.
- Sequential ordering of tasks inherent in a flowchart is suppressed in a structure chart.
- A structure chart has no decision boxes.

Unlike flowcharts, structure charts show how different modules within a program interact and the data that is passed between them.

## 5.4 COUPLING AND COHESION

### 5.4.1 Coupling

The coupling between two modules indicates the degree of interdependence between them. If two modules interchange a large amount of data, then they are highly interdependent. The degree of coupling between two modules depends on their interface complexity. The interface complexity is basically determined by the number of types of parameters that are interchanged while invoking the functions of the module.

**Highly Coupled:** When the modules are highly dependent on each other then they are called highly coupled.

**Loosely Coupled:** When the modules are dependent on each other but the interconnection among them is weak then they are called loosely coupled.

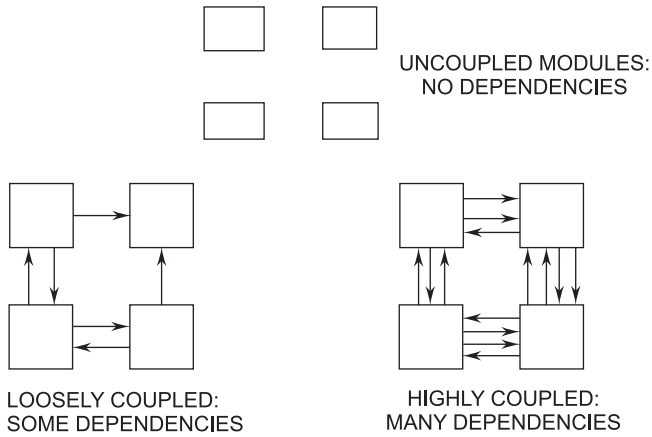



**FIGURE 5.11** Coupling

**Uncoupled:** When the different modules have no interconnection among them then they are called uncoupled modules.

### Factors Affecting Coupling Between Modules

The various factors which affect the coupling between modules are depicted in Table 5.1.

**TABLE 5.1 Factors Affecting Coupling**

| | Interface Complexity | Type of Connection | Type of Communication |
|---|---|---|---|
| Low | Simple Obvious | To module by name | Data |
| High | Complicated Obscure | To internal elements | ControlHybrid |

### Types of Couplings

Different types of couplings include content, common, external, control, stamp, and data. The strength of a coupling from the lowest coupling (best) to the highest coupling (worst) is given in Figure 5.12.

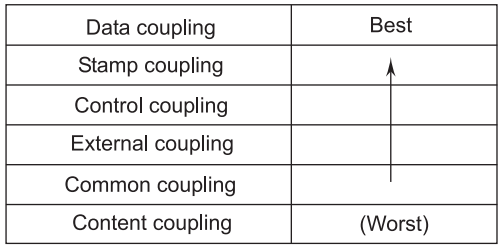



**FIGURE 5.12** The Types of Module Coupling

1. **Data Coupling.** Two modules are data coupled if they communicate using an elementary data item that is passed as a parameter between the two; for example, an integer, a float, a character, etc. This data item should be problem related and not used for a control purpose.

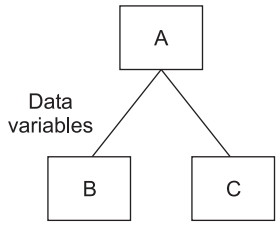



**FIGURE 5.13** Data Coupling

When a non-global variable is passed to a module, modules are called data coupled. It is the lowest form of a coupling. For example, passing the variable from one module in C and receiving the variable by value (i.e., call by value).

2. **Stamp Coupling.** Two modules are stamp coupled if they communicate using a composite data item, such as a record, structure, object, etc. When a module passes a non-global data structure or an entire structure to another module, they are said to be stamp coupled. For example, passing a record in PASCAL or a structure variable in C or an object in C++ language to a module.
3. **Control Coupling.** Control coupling exists between two modules if data from one module is used to direct the order of instruction execution in another. An example of control coupling is a flag set in one module that is tested in another module.

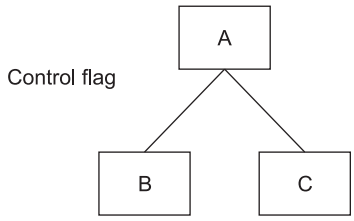



**FIGURE 5.14** Control Coupling

The sending module must know a great deal about the inner workings of the receiving module. A variable that controls decisions in subordinate module C is set in super-ordinate module A and then passed to C.

4. **External Coupling.** It occurs when modules are tied to an environment external to software. External coupling is essential but should be limited to a small number of modules with structures.
5. **Common Coupling.** Two modules are common coupled if they share some global data items (e.g., Global variables). 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 couplings and take special care to guard against them.
6. **Content Coupling.** Content coupling exists between two modules if their code is shared; for example, a branch from one module into another module. It is when one module directly refers to the inner workings of another module. Modules are highly interdependent on each other. It is the highest form of coupling. It is also the least desirable coupling as one component actually modifies another and thereby the modified component is completely dependent on the modifying one.

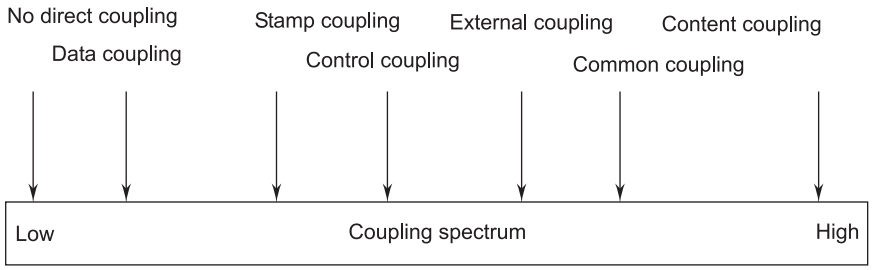



**FIGURE 5.15** Couplings

High coupling among modules not only makes a design difficult to understand and maintain, but it also increases development effort as the modules having high coupling cannot be developed independently by different team members. Modules having high coupling are difficult to implement and debug.

### 5.4.2 Cohesion

Cohesion is a measure of the relative functional strength of a module. The cohesion of a component is a measure of the closeness of the relationships between its components. A cohesive module performs a single task within a software procedure, requiring little interaction with procedures being performed in other parts of a program.

A strongly cohesive module implements functionality that is related to one feature of the solution and requires little or no interaction with other modules. This is shown in Figure 5.16. Cohesion may be viewed as the glue that keeps the module together. It is a measure of the mutual officity of the components of a module.

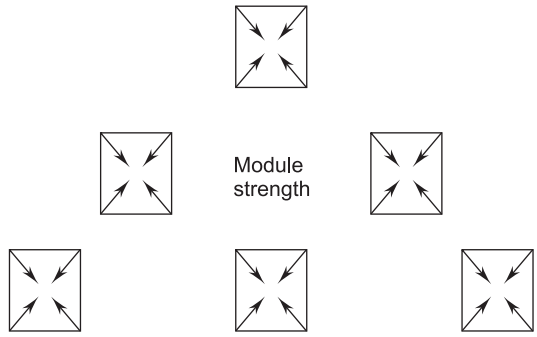



**FIGURE 5.16** Cohesion-strength of Relation within Modules

Thus, we want to maximize the interaction within a module. Hence, an important design objective is to maximize the module cohesion and minimize the module coupling.

## Types of Cohesion

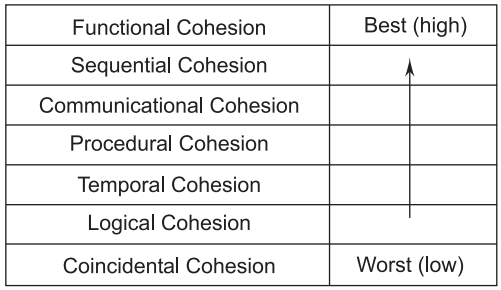



**FIGURE 5.17** The Types of Module Cohesion

There are seven levels of cohesion in decreasing order of desirability, which are as follows:

1. **Functional Cohesion.** Functional cohesion is said to exist if different elements of a module cooperate to achieve a single function (e.g., managing an employee's payroll). When a module displays functional cohesion, and if we are asked to describe what the module does, we can describe it using a single sentence.

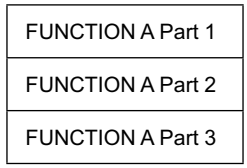



**FIGURE 5.18** Functional Cohesion: Sequential with Complete, Related Functions

2. **Sequential Cohesion.** A module is said to possess sequential cohesion if the elements of a module form the parts of a sequence, where the output from one element of the sequence is input to the next.

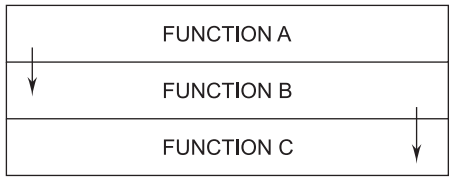



**FIGURE 5.19** Sequential Cohesion: Output of One Part is Input to Next

3. **Communicational Cohesion.** A module is said to have communicational cohesion if all the functions of the module refer to or update the same data

structure; for example, the set of functions defined on an array or a stack. All the modules in communicational cohesion are bound tightly because they operate on the same input or output data. For example, the set of functions defined on an array or a stack.

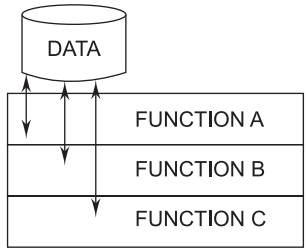



**FIGURE 5.20** Communicational Cohesion: Access Same Data

4. **Procedural Cohesion.** A module is said to possess procedural cohesion if the set of functions of the module are all part of a procedure (algorithm) in which a certain sequence of steps has to be carried out for achieving an objective; for example, the algorithm for decoding a message.

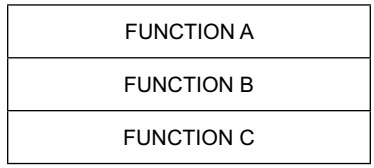



**FIGURE 5.21** Procedural Cohesion Related by Order of Function

5. **Temporal Cohesion.** When a module contains functions that are related by the fact that all the functions must be executed in the same time span, the module is said to exhibit temporal cohesion. The set of functions responsible for initialization, start-up, shutdown of some process, etc., exhibit temporal cohesion.

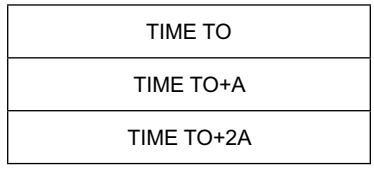



**FIGURE 5.22** Temporal Cohesion Related by Time

6. **Logical Cohesion.** A module is said to be logically cohesive if all elements of the module perform similar operations; for example, error handling, data input, data output, etc. An example of logical cohesion is the case where a

set of print functions generating different output reports are arranged into a single module.

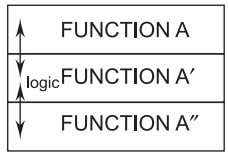



**FIGURE 5.23** Logical Cohesion Similar Functions

7. **Coincidental Cohesion.** A module is said to have coincidental cohesion if it performs a set of tasks that relate to each other very loosely. In this case, the module contains a random collection of functions. It means that the functions have been put in the module out of pure coincidence without any thought or design. It is the worst type of cohesion.

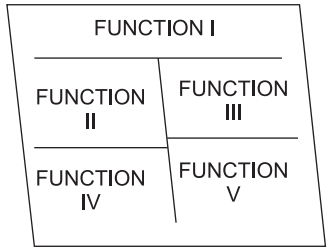



**FIGURE 5.24** Coincidental Cohesion Parts Unrelated

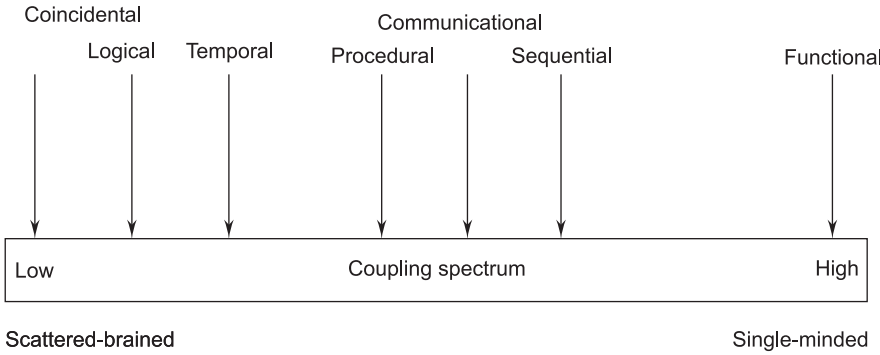



**FIGURE 5.25** Cohesion

### 5.4.3 Relationship Between Coupling and Cohesion

A software engineer must design the modules with the goal of high cohesion and low coupling.

A good example of a system that has high cohesion and low coupling is the 'plug and play' feature of a computer system. Various slots in the motherboard of the system simply facilitate to add or remove the various services/functionalities without affecting the entire system. This is because the add-on components provide the services in a highly cohesive manner. Figure 5.26 provides a graphical review of cohesion and coupling.

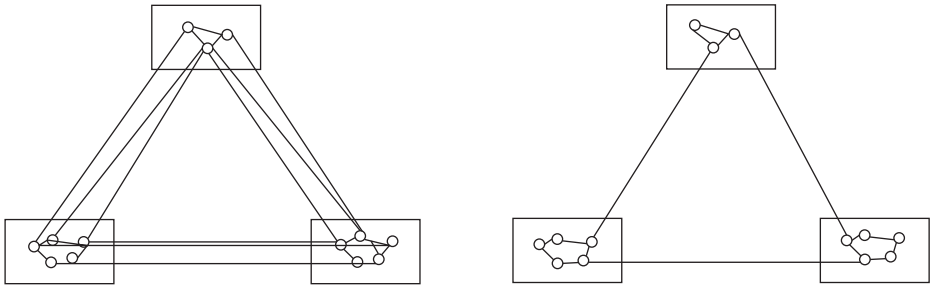



**FIGURE 5.26** View of Cohesion and Coupling

Module design with high cohesion and low coupling characterizes a module as a black box when the entire structure of the system is described. Each module can be dealt with separately when the module functionality is described.

## 5.5 FUNCTIONAL-ORIENTED VERSUS THE OBJECT-ORIENTED APPROACH

Some of the differences between the functional-oriented and the object-oriented approaches, which are very indispensable, are described in Table 5.2.

**TABLE 5.2**

| S. No. | Functional-oriented Approach | Object-oriented Approach |
|---|---|---|
| 1. | In the functional-oriented design approach, the basic abstractions, which are given to the user, are real-world functions, such as sort, merge, track, display, etc. | In the object-oriented design approach, the basic abstractions are not the real-world functions, but are the data abstraction where the real-world entities are represented, such as picture, machine, radar system, customer, student, employee, etc. |

| | | |
|---|---|---|
| 2. | In function-oriented design, functions are grouped together by which a higher-level function is obtained. An example of this technique is SA/SD. | In this design, the functions are grouped together on the basis of the data they operate on, such as in class person, function displays are made member functions to operate on its data members such as the person name, age, etc. |
| 3. | In this approach, the state information is often represented in a centralized shared memory. | In this approach, the state information is not represented in a centralized shared memory but is implemented/distributed among the objects of the system. |

## 5.6 DESIGN SPECIFICATIONS

Design specifications address different aspects of the design model and are completed as the designer refines his representation of the software. First, the overall scope of the design effort is described, which is derived from system specification and the analysis model (software requirements specification).

Then, data design is specified, which includes data structures, any external file structures, internal data structures, and a cross-reference that connects data objects to specific files.

Then architectural design indicates how the program architecture has been derived from the analysis model. Structure charts are used to represent the module hierarchy.

Interface design indicates the design of external and internal program interfaces along with a detailed design of the human/machine interface. A detailed prototype of a GUI may also be represented.

Procedural design specifies components—separately addressable elements of software—such as subroutines, functions, or procedures in the form of English-language processing narratives. This narrative explains the procedural function of a component (module).

Design specification contains a requirements cross-reference. The purpose of this cross-reference is:

- To establish that all requirements are satisfied by the software design.
- To indicate which components are critical to the implementation of specific requirements.

The final section of the design specification contains supplementary data, such as algorithm descriptions, alternative procedures, tabular data, excerpts from

other documents, and other relevant information presented as a special note or a separate appendix.

**TABLE 5.3**

| System objective | Human-machine interface |
|---|---|
| Major software requirements design | Specification and design |
| Constraints, limitations | External interface design |
| Data design | Interfaces to external/systems |
| Data objects and resultant data structures | Internal design rules<br>Processing narrative |
| File and database structures | Interface description |
| External file structures | Design language description |
| Logical structures | Modules used |
| Access method | Data structures used |
| Global data | Comments |
| File and data cross-reference | Requirements cross-reference |

## 5.7 VERIFICATION FOR DESIGN

The output of the system design phase, such as the output of other phases in the development process, should be verified before proceeding with the activities of the next phase. If the design is expressed in some formal notation for which analysis tools are available, then through tools it can be checked for internal consistency (e.g., those modules used by another are defined, the interface of a module is consistent with the way others use it, data usage is consistent with declaration, etc.). If the design is not specified in a formal, executable language, it cannot be processed through tools, and other means for verification have to be used.

There are two fundamental approaches to verification. The first consists of experimenting with the behavior of a product to see whether the product performs as expected (i.e., testing the product). The other consists of analyzing the product—or any design documentation related to it—to deduce its correct operation as a logical consequence of the design decisions. The two categories of verification techniques are also classified as dynamic or static, since the former requires—by definition—executing the system to be verified, while the latter does not. Not surprisingly, the two techniques turn out to be nicely complementary.

## 5.8 MONITORING AND CONTROL FOR DESIGN

Software project management is crucial to the success of a project. The basic task is to plan the detailed implementation of the development process to ensure that the cost and quality objectives are met. It specifies what is needed to meet the cost, quality, and schedule objectives. For this purpose we need monitoring and control. Monitoring obtains information from the development process and exerts the required control over it. Figure 5.27 shows where monitoring and control is carried out in project management:

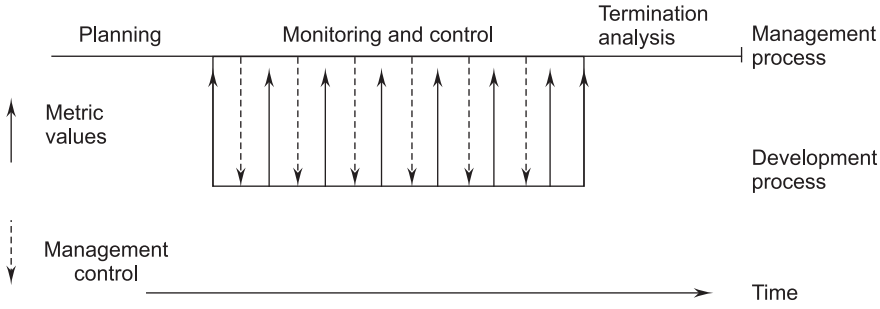



**FIGURE 5.27** Phases of Project Management

Monitoring and control systems are an important class of a real-time system. They check sensors and provide information about the system's environment and take actions depending on the sensor reading. Monitoring systems take action when some exceptional sensor value is detected. Control systems continuously control hardware actuators depending on the value of associated sensors.

Consider the following example: A burglar alarm system is to be implemented for a building. This uses several different types of sensors. These include movement detectors in individual rooms, window sensors on ground floor windows, which detect if a window has been broken, and door sensors, which detect a door opening on corridor doors. There are 50 window sensors, 30 door sensors, and 200 movement detectors in the system.

When a sensor detects the presence of an intruder, the system automatically calls the local police and, using a voice synthesizer, reports the location of the alarm. It switches on lights in the rooms around the active sensor and sets off an audible alarm. The alarm system is normally powered by the main power source but is equipped with a battery backup. Power loss is detected using a separate power circuit monitor that monitors the main voltage. It interrupts the alarm system when a voltage drop is detected.

## EXERCISES

1. What is system design?
2. Explain, in detail, the three design principles in system design.
3. What is abstraction? What are the verification metrics for system design?
4. Define:
   (*i*) Problem partitioning
   (*ii*) Abstraction
   (*iii*) Top-down and bottom-up design
5. Define architectural design.
6. What are the objectives of architectural design?
7. Explain the various design techniques that come under the category of low-level design.
8. Define:
   (*i*) Modularization (*ii*) Structure charts
   (*iii*) Pseudo-code (*iv*) Flowcharts
9. Give any two important differences between the function-oriented and object-oriented design approaches.
10. Discuss the major advantages of the object-oriented design approach over the function-oriented design approach.
11. What is a flowchart? Explain some of its symbols. Also give a suitable example.
12. Give the hierarchical format of a structure chart. Also, give the basic building blocks of a structure chart.
13. Explain the term design specification.
14. Discuss the term verification in reference to system design.
15. Enumerate the term monitoring and control in system design.
16. Discuss some methods of monitoring and control of a software-development process.
17. What is meant by the term coupling in software design? Is it true that in a good design, the modules should have low coupling? Why?
18. Explain the different types of coupling that two modules might exhibit.
19. Explain the different types of cohesion that a module might exhibit.
20. What is coupling and cohesion in reference to software design? How are these concepts useful in arriving at a good design of a system?
21. Is it true that whenever we increase the cohesion of different modules in our design, coupling between these modules automatically decreases? Justify your answer with the help of an appropriate example.
22. What is a flowchart? How is the flow-charting technique useful for software development?
23. Discuss the major advantages of the object-oriented design (OOD) methodology over the data flow-oriented design methodologies.