UNIT-III

CHAPTER-I

SOFTWARE DESIGN

Overview of the Design Process

The design process essentially transforms the SRS document into a design document.

Outcome of the Design Process

- **Different modules required:** The different modules in the solution should be identified. Each module is a collection of functions and the data shared by these functions. Each module should accomplish some well-defined tasks out of the overall responsibility of the software. Each module should be named according to the task it performs.
- For example, in an academic automation software, the module consisting of the functions and data necessary to accomplish the task of registration of the students should be named handle student registration.
- The activities carried out during the design phase (called as *design process*) transform the SRS document into the design document.
- **Control relationships among modules:** A control relationship between two modules essentially arises due to *function calls* across the two modules. The control relationships existing among various modules should be identified in the design document.
- **Interfaces among different modules:** The interfaces between two modules identifies the exact data items that are exchanged between the two modules when one module invokes a function of the other module.
- **Data structures of the individual modules:** Each module normally stores some data that the functions of the module need to share to accomplish the overall responsibility of the module. Suitable data structures for storing and managing the data of a module need to be properly designed and documented.
- **Algorithms required to implement the individual modules:** Each function in a module usually performs some processing activity. The algorithms required to accomplish the processing activities of various modules need to be carefully designed and documented with due considerations given to the accuracy of the results, space and time complexities. Starting with the SRS document the design documents are produced through iterations over a series of steps. The design documents are reviewed by the members of the development team to ensure that the design solution conforms to the requirements specification.

Classification of Design Activities

- A good software design is seldom realized by using a single step procedure, rather it requires iterating over a series of steps called the *design activities*. Let us first classify the design activities before discussing them in detail. Depending on the order in which various design activities are performed, we can broadly classify them into two important stages.
- Preliminary (or high-level) design, and
- Detailed design.
- The meaning and scope of these two stages can vary considerably from one design methodology to another. However, for the traditional function-oriented design approach, it is possible to define the objectives of the high-level design as follows:
- The outcome of high-level design is called the *program structure* or the *software architecture.* High-level design is a crucial step in the overall design of a software. When the high-level design is complete, the problem should have been decomposed into many small functionally independent modules that are cohesive, have low coupling among themselves, and are arranged in a hierarchy.
- Many different types of notations have been used to represent a high-level design. A notation that is widely being used for procedural development is a tree-like diagram called the *structure chart*. Another popular design representation techniques called UML that is being used to document object-oriented design, involves developing several types of diagrams to document the object-oriented design of a systems.
- Once the high-level design is complete, detailed design is undertaken.
- The outcome of the detailed design stage is usually documented in the form of a *module specification* (MSPEC) document. After the high-level design is complete, the problem would have been decomposed into small modules, and the data structures and algorithms to be used described using MSPEC and can be easily grasped by programmers for initiating coding. In this text, we do not discuss MSPECs and confine our attention to high-level design only.
- Through high-level design, a problem is decomposed into a set of modules. The control relationships among the modules are identified, and also, the interfaces among various modules are identified.
- During detailed design each module is examined carefully to design its data structures and the algorithms.

Classification of Design Methodologies

- The design activities vary considerably based on the specific design methodology being used. A large number of software design methodologies are available. We can roughly classify these methodologies into procedural and object-oriented approaches. These two approaches are two fundamentally different design paradigms.

Analysis *versus* design

- Analysis and design activities differ in goal and scope. The analysis results are generic and does not consider implementation or the issues associated with specific platforms. The analysis model is usually documented using some graphical formalism. In case of the function-oriented approach that we are going to discuss, the analysis model would be documented using *data flow diagrams* (DFDs),
- whereas the design would be documented using structure chart. On the other hand, for object-oriented approach, both the design model and the analysis model will be documented using *unified modelling language* (UML). The analysis model would normally be very difficult to implement using a programming language.

HOW TO CHARACTERISE A GOOD SOFTWARE DESIGN?

Coming up with an accurate characterization of a good software design that would hold across diverse problem domains is certainly not easy. In fact, the definition of a "good" software design can vary depending on the exact application being designed. For example, "memory size used up by a program" may be an important way to characterize a good solution for embedded software development—since embedded applications are often required to work under severely limited memory sizes due to cost, space, orpower consumption considerations.

- **Correctness:** A good design should first of all be correct. That is, it should correctly implement all the functionalities of the system.
- **Understandability:** A good design should be easily understandable. Unless a design solution is easily understandable, it would be difficult to implement and maintain it.
- **Efficiency:** A good design solution should adequately address resource, time, and cost optimisation issues.
- **Maintainability:** A good design should be easy to change. This is an important requirement, since change requests usually keep coming from the customer even after product release.

Understandability of a Design: A Major Concern

- While performing the design of a certain problem, assume that we have arrived at a large number of design solutions and need to choose the best one. Obviously, all incorrect designs have to be discarded first. Out of the correct design solutions, how can we identify the best one?
- A good design should help overcome the human cognitive limitations that arise due to limited short-term memory. A large problem overwhelms design would make the matter worse. Unless a design solution is easily understandable, it could lead to an implementation having a large number of defects and at the same time tremendously pushing up the development costs. Therefore, a good design solution should be simple and easily understandable. A design that is easy to understand is also easy to develop and maintain.

- A complex design would lead to severely increased life cycle costs. Unless a design is easily understandable, it would require tremendous effort to implement, test, debug, and maintain it. that about 60 per cent of the total effort in the life cycle of a typical product is spent on maintenance.
- If the software is not easy to understand, not only would it lead to increased development costs, the effort required to maintain the product would also increase manifold. Besides, a design solution that is difficult to understand would lead to a program that is full of bugs and is unreliable that understandability of a design solution can be enhanced through clever applications of the principles of abstraction and decomposition. the human mind, and a poor.

An understandable design is modular and layered

- How can the understandability of two different designs be compared, so that we can pick the better one? To be able to compare the understandability of two design solutions, we should at least have an understanding of the general features that an easily understandable design should possess. A design solution should have the following characteristics to beeasily understandable:
- It should assign consistent and meaningful names to various design components.
- It should make use of the principles of decomposition and abstraction in good measures to simplify the design.

Modularity

- A modular design is an effective decomposition of a problem. It is a basic characteristic of any good design solution. A *modular design*, in simple words, implies that the problem has been decomposed into a set of modules that have only limited interactions with each other.
- Decomposition of a problem into modules facilitates taking advantage of the *divide and conquer* principle. If different modules have either no interactions or little interactions with each other, then each module can be understood separately. This reduces the perceived complexity of the design solution greatly. To understand why this is so, remember that it may be very difficult to break a bunch of sticks which have been tied together, but very easy to break the sticks individually.
- It is not difficult to argue that modularity is an important characteristic of a good design solution. But, even with this, how can we compare the modularity of two alternate design solutions? From an inspection of the module structure, it is at least possible to intuitively form an idea as to which design is more modular.
- For example, consider two alternate design solutions to a problem that are represented in Figure 5.2, in which the modules *M*1*, M*2*,* etc. have been drawn as rectangles. The invocation of a module by another module has been shown as an arrow. It can easily be seen that the design solution of Figure 5.2(a) would be easier to understand since the interactions among the different modules is low. But, can we quantitatively measure the modularity of a design solution? Unless we are able to quantitatively measure the

modularity of a design solution, it will be hard to say which design solution is more modular than another.

- Unfortunately, there are no quantitative metrics available yet to directly measure the modularity of a design. However, we can quantitatively characterize the modularity of a design solution based on the cohesion and coupling existing in the design.
- A software design with high cohesion and low coupling among modules is the effective problem decomposition. Such a design would lead to increased productivity during program development by bringing down the perceived problem complexity.

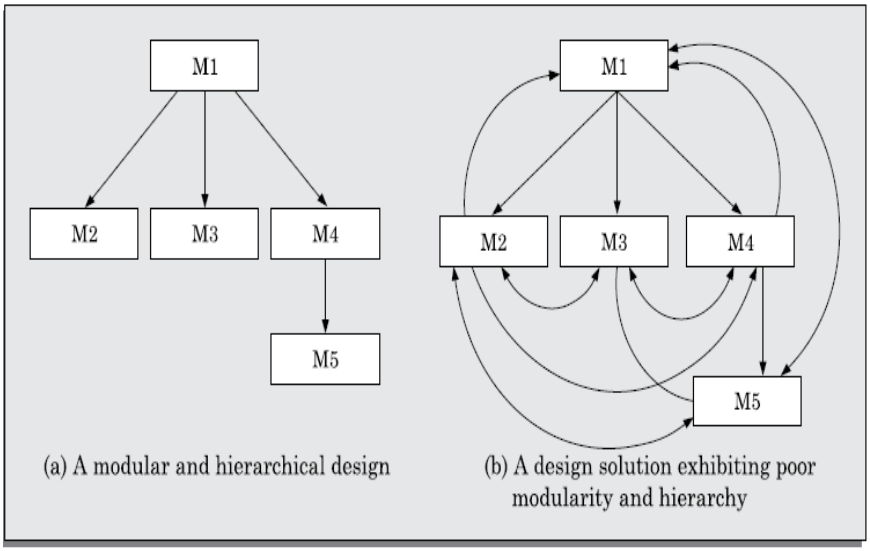

FIGURE 5.2 Two design solutions to the same problem.

Layered design

- A layered design is one in which when the call relations among different modules are represented graphically, it would result in a tree-like diagram with clear layering. In a layered design solution, the modules are arranged in a hierarchy of layers. A module can only invoke functions of the modules in the layer immediately below it. The higher layer modules can be considered to be similar to managers that invoke (order) the lower layer modules to get certain tasks done.
- A layered design can be considered to be implementing *control abstraction*, since a module at a lower layer is unaware of (about how to call) the higher layer modules. When a failure is detected while executing a module, it is obvious that the modules below it can possibly be the source of the error.
- This greatly simplifies debugging since one would need to concentrate only on a few modules to detect the error. We shall elaborate these concepts governing layered design of modules

COHESION AND COUPLING

- We have so far discussed that effective problem decomposition is an important characteristic of a good design. Good module decomposition is indicated through *high cohesion* of the individual modules and *low coupling* of the modules with each other. Let us now define what is meant by cohesion and coupling.
- In this section, we first elaborate the concepts of cohesion and coupling. Subsequently, we discuss the classification of cohesion and coupling.
- **Coupling:** Intuitively, we can think of coupling as follows. Two modules are said to be highly coupled, if either of the following two situations arise:
- If the function calls between two modules involve passing large chunks of shared data, the modules are tightly coupled.
- If the interactions occur through some shared data, then also we say that they are highly coupled.
- If two modules either do not interact with each other at all or at best interact by passing no data or only a few primitive data items, they are said to have low coupling.
- **Cohesion:** To understand cohesion, let us first understand an analogy. Suppose you listened to a talk by some speaker. You would call the speech to be cohesive, if all the sentences of the speech played some role in giving the talk a single and focused theme. Now, we can extend this to a module in a design solution.
- When the functions of the module co-operate with each other for performing a single objective, then the module has good cohesion. If the functions of the module do very different things and do not co-operate with each other to perform a single piece of work, then the module has very poor cohesion
- **Functional independence**
- By the term *functional independence,* we mean that a module performs a single task and needs very little interaction with other modules. Functional independence is a key to any good design primarily due to the following
- **Error isolation:** Whenever an error exists in a module, functional independence reduces the chances of the error propagating to the other modules. The reason behind this is that if a module is functionally independent, its interaction with other modules is low. Therefore, an error existing in the module is very unlikely to affect the functioning of other modules advantages it offers:
- Further, once a failure is detected, error isolation makes it very easy to locate the error. On the other hand, when a module is not functionally independent, once a failure is detected in a functionality provided by the module, the error can be potentially in any of the large number of modules and propagated to the functioning of the module.
- **Scope of reuse:** Reuse of a module for the development of other applications becomes easier. The reasons for this is as follows. A functionally independent module performs some well-defined and precise task and the interfaces of the module with other modules

are very few and simple. A functionally independent module can therefore be easily taken out and reused in a different program. On the other hand, if a module interacts with several other modules or the functions of a module perform very different tasks, then it would be difficult to reuse it. This is especially so, if the module accesses the data (or code) internal to other modules.

- **Understandability:** When modules are functionally independent, complexity of the design is greatly reduced. This is because of the fact that different modules can be understood in isolation, since the modules are independent of each other that understandability is a major advantage of a modular design. Besides the three we have listed here, there are many other advantages of a modular design as well. We shall not list those here, and leave it as an assignment to the reader to identify them.

Classification of Cohesiveness

- Cohesiveness of a module is the degree to which the different functions of the module co-operate to work towards a single objective. The different modules of a design can possess different degrees of freedom. However, the different classes of cohesion that modules can possess
- The cohesiveness increases from coincidental to functional cohesion. That is, coincidental is the worst type of cohesion and functional is the best cohesion possible. These different classes of cohesion are elaborated below.

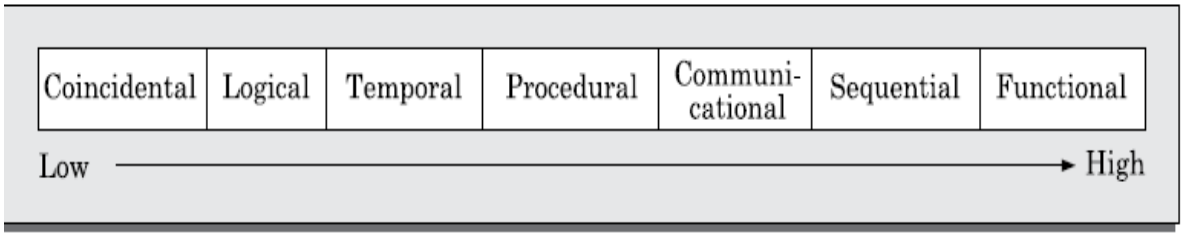

FIGURE 5.3 Classification of cohesion.

- **Coincidental cohesion:** A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all. In this case, we can say that the module contains a random collection of functions.
- It is likely that the functions have been placed in the module out of pure coincidence rather than through some thought or design. The designs made by novice programmers often possess this category of cohesion, since they often bundle functions to modules rather arbitrarily.
- An example of a module with coincidental cohesion has been shown in Figure 5.4(a). Observe that the different functions of the module carry out very different and unrelated activities starting from issuing of library books to creating library member records on one hand, and handling librarian leave request on the other.

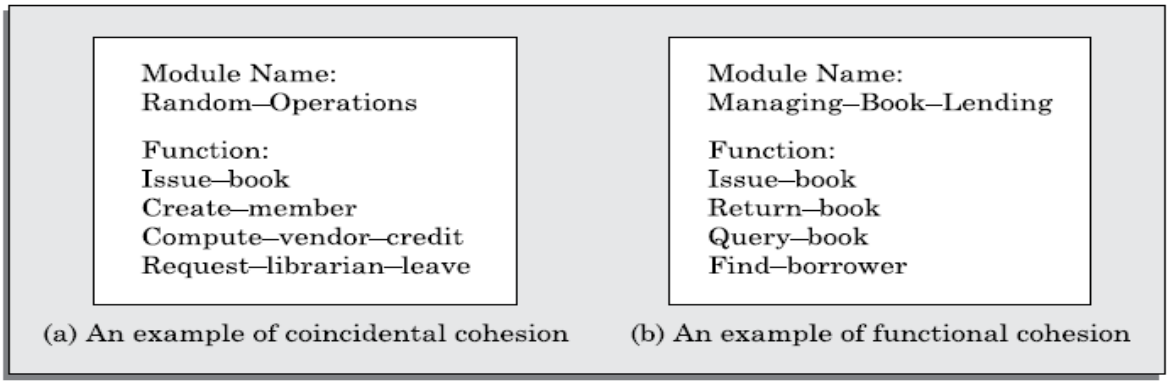

FIGURE 5.4 Examples of cohesion.

- **Logical cohesion:** A module is said to be logically cohesive, if all elements of the module perform similar operations, such as error handling, data input, data output, etc.
- As an example of logical cohesion, consider a module that contains a set of print functions to generate various types of output reports such as grade sheets, salary slips, annual reports, etc.
- **Temporal cohesion:** When a module contains functions that are related by the fact that these functions are executed in the same time span, then the module is said to possess temporal cohesion.
- As an example, consider the following situation. When a computer is booted, several functions need to be performed. These include initialization of memory and devices, loading the operating system, etc. When a single module performs all these tasks, then the module can be said to exhibit temporal cohesion.
- Other examples of modules having temporal cohesion are the following. Similarly, a module would exhibit temporal cohesion, if it comprises functions for performing initialization, or start-up, or shut-down of some process.
- **Procedural cohesion:** A module is said to possess procedural cohesion, if the set of functions of the module are executed one after the other, though these functions may work towards entirely different purposes and operate on very different data.
- Consider the activities associated with order processing in a trading house. The functions login (), place-order (), check-order (), print-bill (), place-order-on-vendor (), update inventory(), and logout() all do different thing and operate on different data. However, they are normally executed one after the other during typical order processing by a sales clerk.
- **Communicational cohesion:** A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure. As an example of procedural cohesion, consider a module named student in which the different functions in the module such as admit Student, enter Marks, print Grade Sheet, etc. access and manipulate data stored in an array named student Records defined within the module.
- **Sequential cohesion:** A module is said to possess sequential cohesion, if the different functions of the module execute in a sequence, and the output from one function is input

to the next in the sequence. As an example, consider the following situation. In an on-line store consider that after a customer request for some item, it is first determined if the item is in stock. In this case, if the functions create-order (), check-item-availability (), place-order-on-vendor () are placed in a single module, then the module would exhibit sequential cohesion. Observe that the function create-order () creates an order that is processed by the function check-item-availability () (whether the items are available in the required quantities in the inventory) is input to place-order-on-vendor ().

- **Functional cohesion:** A module is said to possess functional cohesion, if different functions of the module co-operate to complete a single task. For example, a module containing all the functions required to manage employees' pay-roll displays functional cohesion. In this case, all the functions of the module (e.g., compute Overtime(), compute Work Hours(), compute Deductions(), etc.) work together to generate the pay slips of the employees.

- Another example of a module possessing functional cohesion. In this example, the functions issue-book(), return-book(), query-book(), and find-borrower(), together manage all activities concerned with book lending. When a module possesses functional cohesion, then we should be able to describe what the module does using only one simple sentence. For we can describe the overall responsibility of the module by saying "It manages the book V lendingprocedure of the library."

- A simple way to determine the cohesiveness of any given module is as follows. First examine what do the functions of the module perform. Then, try to write down a sentence to describe the overall work performed by the module. If you need a compound sentence to describe the functionality of the module, then it has sequential or communicational cohesion. If you need words such as "first", "next", "after", "then", etc., then it possesses sequential or temporal cohesion. If it needs words such as "initialize", "setup", "shut down", etc., to define its functionality, then it has temporal cohesion. We can now make the following observation. A cohesive module is one in which the functions interact among themselves heavily to achieve a single goal. As a result, if any of these functions is removed to a different module, the coupling would increase as the functions would now interact across two different modules.

Classification of Coupling

- The coupling between two modules indicates the degree interdependence between them. Intuitively, if two modules interchange large amounts of data, then they are highly interdependent or coupled. We can alternately state this concept as follows.

- The interface complexity is determined based on the number of parameters and the complexity of the parameters that are interchanged while one module invokes the functions of the other module.

- Let us now classify the different types of coupling that can exist between two modules. Between any two interacting modules, any of the following five different types of coupling can exist. These different types of coupling, in increasing order of their severities

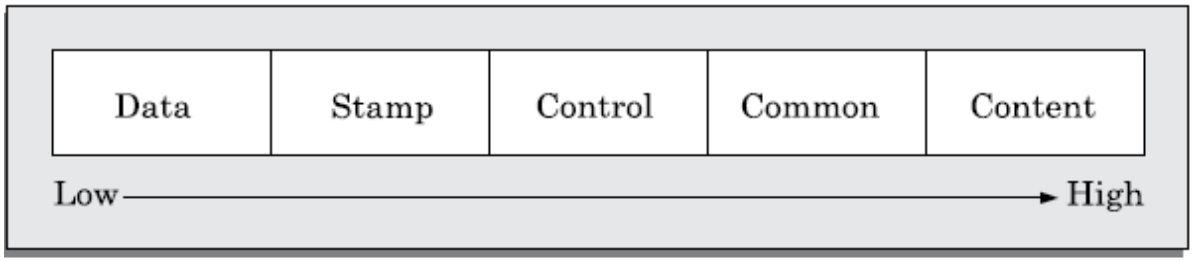

FIGURE 5.5 Classification of coupling.

- **Data coupling:** Two modules are data coupled, if they communicate using an elementary data item that is passed as a parameter between the two, e.g. an integer, a float, a character, etc. This data item should be problem related and not used for control purposes.
- **Stamp coupling:** Two modules are stamp coupled, if they communicate using a composite data item such as a record in PASCAL or a structure in C.
- **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 and tested in another module.
- **Common coupling:** Two modules are common coupled, if they share some global data items.
- **Content coupling:** Content coupling exists between two modules, if they share code. That is, a jump from one module into the code of another module can occur. Modern high-level programming languages such as C do not support such jumps across modules.
- The degree of coupling increases from data coupling to content coupling. High coupling among modules not only makes a design solution difficult to understand and maintain, but it also increases development effort and also makes it very difficult to get these modules developed independently by different team members.

LAYERED ARRANGEMENT OF MODULES

- The *control hierarchy* represents the organization of program components in terms of them call relationships. Thus, we can say that the control hierarchy of a design is determined by the order in which different modules call each other. Many different types of notations have been used to represent the control hierarchy. The most common notation is a treelike diagram known as a *structure chart*
- However, other notations such as Warnier-Orr [1977, 1981] or Jackson diagrams [1975] may also be used. Since, Warnier-Orr and Jackson's notations are not widely used nowadays, In a layered design solution, the modules are arranged into several layers based on their call relationships. A module is allowed to call only the modules that are at a lower layer.

- That is, a module should not call a module that is either at a higher layer or even in the same layer. Figure 5.6(a) shows a layered design, whereas Figure 5.6(b) shows a design that is not layered.
- Figure 5.6(b), is actually not layered since all the modules can be considered to be in the same layer. In the following, we state the significance of a layered design and subsequently we explain it.
- In a layered design, the top-most module in the hierarchy can be considered as a manager that only invokes the services of the lower-level module to discharge its responsibility. The modules at the intermediate layers offer services to their higher layer by invoking the services of the lower layer modules and also by doing some work themselves to a limited extent. The modules at the lowest layer are the worker modules. These do not invoke services of any module and entirely carry out their responsibilities by themselves.
- Understanding a layered design is easier since to understand one module, one would have to at best consider the modules at the lower layers (that is, the modules whose services it invokes). Besides, in a layered design errors are isolated, since an error in one module can affect only the higher layer modules. As a result, in case of any failure of a module, only the modules at the lower levels need to be investigated for the possible error. Thus, debugging time reduces significantly in a layered design.
- On the other hand, if the different modules call each other arbitrarily, then this situation would correspond to modules arranged in a single layer. Locating an error would be both difficult and time consuming. This is because, once a failure is observed, the cause of failure (i.e. error) can potentially be in any module, and all modules would have to be investigated for the error.
- **Superordinate and subordinate modules:** In a control hierarchy, a module that controls another module is said to be *superordinate* to it. Conversely, a module controlled by another module is said to be *subordinate* to the controller.
- **Visibility:** A module B is said to be visible to another module A, if A directly calls B. Thus, only the immediately lower layer modules are said to be visible to a module.
- **Control abstraction:** In a layered design, a module should only invoke the functions of the modules that are in the layer immediately below it. In other words, the modules at the higher layers, should not be visible (that is, abstracted out) to the modules at the lower layers. This is referred to as *control abstraction.*
- **Depth and width:** Depth and width of a control hierarchy provide an indication of the number of layers and the overall span of control respectively. For the design of Figure 5.6(a), the depth is 3 and width is also 3.
- **Fan-out:** Fan-out is a measure of the number of modules that are directly controlled by a given module. In Figure 5.6(a), the fan-out of the module M1 is 3. A design in which the modules have very high fan-out numbers is not a good design. The reason for this is that a very high fan-out is an indication that the module lacks cohesion. A module having a large fan-out (greater than 7) is likely to implement several different functions and not just a single cohesive function.

- **Fan-in:** Fan-in indicates the number of modules that directly invoke a given module. High fan-in represents code reuse and is in general, desirable in a good design. In Figure 5.6(a), the fan-in of the module M1 is 0, that of M2 is 1, and that of M5 is 2.

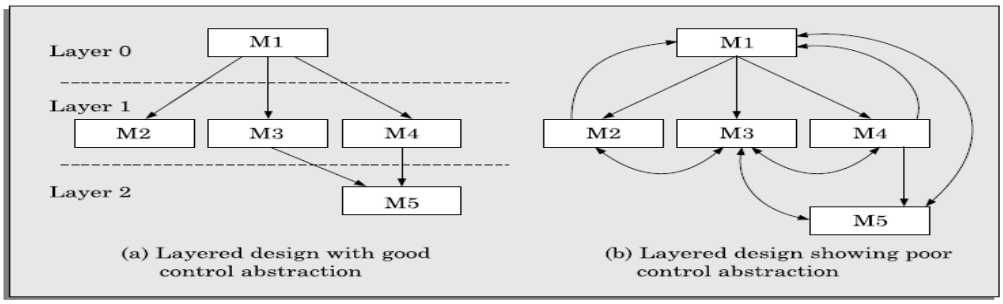

FIGURE 5.6 Examples of good and poor control abstraction.

APPROACHES TO SOFTWARE DESIGN

- There are two fundamentally different approaches to software design that are in use today—function-oriented design, and object-oriented design. Though these two design approaches are radically different; they are complementary rather than competing techniques. The object-oriented approach is a relatively newer technology and is still evolving. For development of large programs, the object-oriented approach is becoming increasingly popular due to certain advantages that it offers. On the other hand, function-oriented designing is a mature technology and has a large following.
- Function-oriented Design
- The following are the salient features of the function-oriented design approach:
- Top-down decomposition: A system, to start with, is viewed as a black box that provides certain services (also known as high-level functions) to the users of the system.
- For example, consider a function create-new-library member which essentially creates the record for a new member, assigns a unique membership number to him, and prints a bill towards his membership charge. This high-level function may be refined into the following subfunctions:
- assign-membership-number
- create-member-record
- print-bill
- Each of these subfunctions may be split into more detailed subfunctions and so on.

Centralized system state:

The system state can be defined as the values of certain data items that determine the response of the system to a user action or external event. For example, the set of books (i.e. whether borrowed by different users or available for issue) determines the state of a library automation system. Such data in procedural programs usually have global scope and are shared by many modules.

- For example, in the library management system, several functions such as the following share data such as member-records for reference and updating:
- create-new-member
- delete-member
- update-member-record
- A large number of function-oriented design approaches have been proposed in the past. A few of the well-established function-oriented design approaches are as following:
- Structured design by Constantine and Yourdon [1979]
- Jackson's structured design by Jackson [1975]
- Warnier-Orr methodology [1977, 1981]
- Step-wise refinement by Wirth [1971]
- Hatley and Pirbhai's Methodology [1987]

Object-oriented Design

- In the *object-oriented design* (OOD) approach, a system is viewed as being made up of a collection of objects (i.e., entities). Each object is associated with a set of functions that are called its *methods*. Each object contains its own data and is responsible for managing it. The data internal to an object cannot be accessed directly by other objects and only through invocation of the methods of the object. The system state is decentralized since there is no globally shared data in the system and data is stored in each object.
- For example, in a library automation software, each library member may be a separate object with its own data and functions to operate on the stored data. The methods defined for one object cannot directly refer to or change the data of other objects.
- The object-oriented design paradigm makes extensive use of the principles of abstraction and decomposition as explained below. Objects decompose a system into functionally independent modules. Objects can also be considered as instances of *abstract data types* (ADTs). The ADT concept did not originate from the object-oriented approach. In fact, ADT concept was extensively used in the ADA programming language introduced in the 1970s.
- ADT is an important concept that forms an important pillar of object-orientation.
- **Data abstraction:** The principle of data abstraction implies that how data is exactly stored is abstracted away. This means that any entity external to the object (that is, an instance of an ADT) would have no knowledge about how data is exactly stored, organized, and manipulated inside the object.
- The entities external to the object can access the data internal to an object only by calling certain well-defined methods supported by the object. Consider an ADT such as a stack. The data of a stack object may internally be stored in an array, a linearly linked list, or a bidirectional linked list. The external entities have no knowledge of this and

can access data of a stack object only through the supported operations such as push and pop.

- **Data structure:** A data structure is constructed from a collection of primitive data items. Just as a civil engineer builds a large civil engineering structure using primitive building materials such as bricks, iron rods, and cement; a programmer can construct a data structure as an organized collection of primitive data items such as integer, floating point numbers, characters, etc.
- **Data type:** A type is a programming language terminology that refers to anything that can be instantiated. For example, int, float, char, etc., are the basic data types supported by C programming language. Thus, we can say that ADTs are user defined data types.
- In object-orientation, classes are ADTs. Thus, what is the advantage of developing an application using ADTs? Let us examine the three main advantages of using ADTs in programs:
- The data of objects are *encapsulated* within the methods. The encapsulation principle is also known as *data hiding*. The encapsulation principle requires that data can be accessed and manipulated only through the methods supported by the object and not directly. This localizes the errors. The reason for this is as follows. No program element is allowed to change a data, except through invocation of one of the methods. So, any error can easily be traced to the code segment changing the value. That is, the method that changes a data item, making it erroneous can be easily identified.
- An ADT-based design displays high cohesion and low coupling. Therefore, object-oriented designs are highly modular.
- Since the principle of abstraction is used, it makes the design solution easily understandable and helps to manage complexity.
- Similar objects constitute a class. In other words, each object is a member of some class. Classes may inherit features from a super class. Conceptually, objects communicate by message passing. Objects have their own internal data. Thus, an object may exist in different states depending the values of the internal data. In different states, an object may behave differently.

UNIT-III
CHAPTER-II
FUNCTION-ORIENTED SOFTWARE DESIGN

Overview of SA/SD Methodology

As the name itself implies, SA/SD methodology involves carrying out two distinct activities:

- Structured analysis (SA)
- Structured design (SD)

The roles of structured analysis (SA) and structured design (SD) have been shown schematically in Figure 6.1. Observe the following from the figure:

- During structured analysis, the SRS document is transformed into a *data flow diagram* (DFD) model.
- During structured design, the DFD model is transformed into a structure chart.

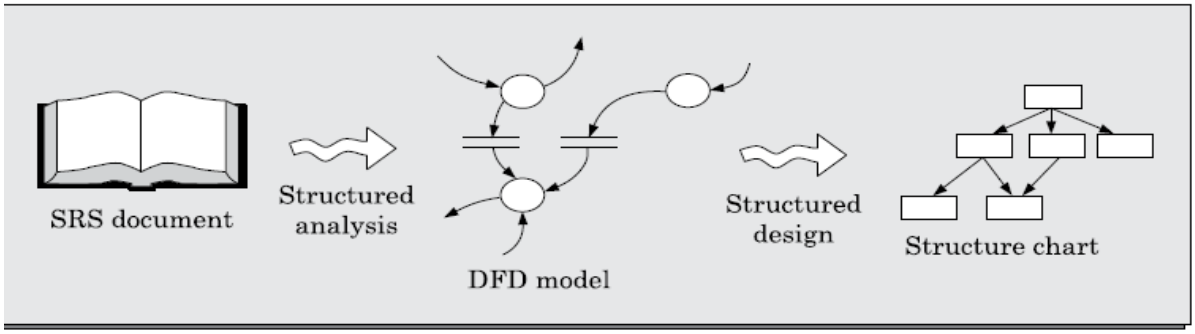

FIGURE 6.1 Structured analysis and structured design methodology.

As shown in Figure 6.1, the structured analysis activity transforms the SRS document into a graphic model called the DFD model. During structured analysis, functional decomposition of the system is achieved. That is, each function that the system needs to perform is analyzed and hierarchically decomposed into more detailed functions.

On the other hand, during structured design, all functions identified during structured analysis are mapped to a module structure. This module structure is also called the *high-level design* or the *software architecture* for the given problem. This is represented using a structure chart.

- The high-level design stage is normally followed by a detailed design stage. During the detailed design stage, the algorithms and data structures for the individual modules are designed. The detailed design can directly be implemented as a working system using a conventional programming language.
- The results of structured analysis can therefore, be easily understood by the user. In fact, the different functions and data in structured analysis are named using the user's terminology. The user can therefore even review the results of the structured analysis to ensure that it captures all his requirements.

- In the following section, we first discuss how to carry out structured analysis to construct the DFD model. Subsequently, we discuss how the DFD model can be transformed into structured design.

STRUCTURED ANALYSIS

We have already mentioned that during structured analysis, the major processing tasks (high-level functions) of the system are analyzed, and the data flow among these processing tasks are represented graphically. Significant contributions to the development of the structured analysis techniques have been made by Gane and Sarson [1979], and DeMarco and Yourdon [1978]. The structured analysis technique is based on the following underlying principles:

- Top-down decomposition approach.
- Application of divide and conquer principle. Through this each high-level function is independently decomposed into detailed functions.
- Graphical representation of the analysis results using *data flow diagrams* (DFDs).
- DFD representation of a problem, as we shall see shortly, is very easy to construct. Though extremely simple, it is a very powerful tool to tackle the complexity of industry standard problems.
- Please note that a DFD model only represents the data flow aspects and does not show the sequence of execution of the different functions and the conditions based on which a function may or may not be executed. In fact, it completely ignores aspects such as control flow, the specific algorithms used by the functions, etc. In the DFD terminology, each function is called a *process* or a *bubble*. It is useful to consider each function as a processing station (or process) that consumes some input data and produces some output data.
- DFD is an elegant modelling technique that can be used not only to represent the results of structured analysis of a software problem, but also useful for several other applications such as showing the flow of documents or items in an organization. how a DFD can be used to represent the processing activities and flow of material in an automated car assembling plant. We now elaborate how a DFD model can be constructed.

Data Flow Diagrams (DFDs)

- The DFD (also known as the *bubble chart*) is a simple graphical formalism that can be used to represent a system in terms of the input data to the system, various processing carried out on those data, and the output data generated by the system.
- The main reason why the DFD technique is so popular is probably because of the fact that DFD is a very simple formalism—it is simple to understand and use. A DFD model uses a very limited number of primitive symbols (shown in Figure 6.2) to represent the functions performed by a system and the data flow among these functions.
- Starting with a set of high-level functions that a system performs, a DFD model represents the subfunctions performed by the functions using a hierarchy of diagrams. We had pointed out while discussing the principle of abstraction in Section 1.3.2 that

any hierarchical representation is an effective means to tackle complexity. Human mind is such that it can easily understand any hierarchical model of a system—because in a hierarchical model, starting with a very abstract model of a system, various details of the system are slowly introduced through different levels of the hierarchy. The DFD technique is also based on a very simple set of intuitive concepts and rules. We now elaborate the different concepts associated with building a DFD model of a system.

Primitive symbols used for constructing DFDs

- There are essentially five different types of symbols used for constructing DFDs. These primitive symbols are depicted in Figure 6.2. The meaning of these symbols are explained as follows:
- **Function symbol:** A function is represented using a circle. This symbol is called a *process* or a *bubble*. Bubbles are annotated with the names of the corresponding functions (see Figure 6.3).
- **External entity symbol:** An external entity such as a librarian, a library member, etc. is represented by a rectangle. The external entities are essentially those physical entities external to the software system which interact with the system by inputting data to the system or by consuming the data produced by the system. In addition to the human users, the external entity symbols can be used to represent external hardware and software such as another application software that would interact with the software being modelled.
- **Data flow symbol:** A directed arc (or an arrow) is used as a data flow symbol. A data flow symbol represents the data flow occurring between two processes or between an external entity and a process in the direction of the data flow arrow. Data flow symbols are usually annotated with the corresponding data names. For example, the DFD in Figure 6.3(a) shows three data flows—the data item number flowing from the process read number to validate-number, data-item flowing into read-number, and valid-numberflowing out of validate-number.
- **Data store symbol:** A data store is represented using two parallel lines. It represents ab logical file. That is, a data store symbol can represent either a data structure or a physical file on disk. Each data store is connected to a process by means of a data flow symbol. The direction of the data flow arrow shows whether data is being read from or written into a data store. An arrow flowing in or out of a data store implicitly represents the entire with the name of the corresponding data items. As an example of a data store, number is a data store
- **Output symbol:** The output symbol is as shown in Figure 6.2. The output symbol is used when a hard copy is produced. The notations that we are following in this text are closer to the Yourdon's notations than to the other notations. You may sometimes find notations in other books that are slightly different than those discussed here. For example, the data store may look like a box with one end open. That is because, they may be following notations such as those of Gane and Sarson [1979].

Important concepts associated with constructing DFD models

Before we discuss how to construct the DFD model of a system, let us discuss some important concepts associated with DFDs:

Synchronous and asynchronous operations

- If two bubbles are directly connected by a data flow arrow, then they are synchronous. This means that they operate at the same speed. An example of such an arrangement is shown in Figure 6.3(a). Here, the validate-number bubble can start processing only after the read-number bubble has supplied data to it; and the read-number bubble has to wait until the validate-number bubble has consumed its data.
- However, if two bubbles are connected through a data store, as in Figure 6.3(b) then the speed of operation of the bubbles are independent. This statement can be explained using the following reasoning. The data produced by a producer bubble gets stored in the data store. It is therefore possible that the producer bubble stores several pieces of data items, even before the consumer bubble consumes any of them.

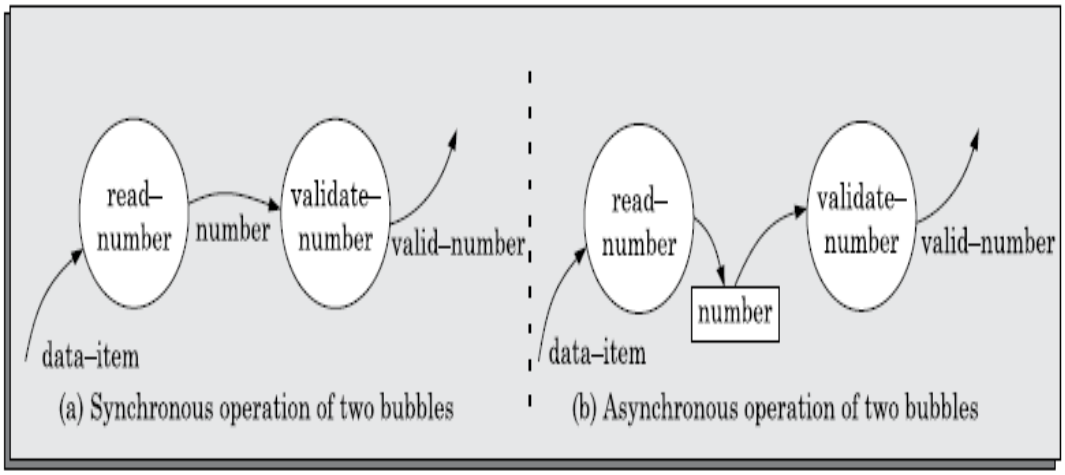

FIGURE 6.3 Synchronous and asynchronous data flow.

Data dictionary

- Every DFD model of a system must be accompanied by a data dictionary. A data dictionary lists all data items that appear in a DFD model. The data items listed include all data flows and the contents of all data stores appearing on all the DFDs in a DFD model. Please remember that the DFD model of a system typically consists of several DFDs, viz., level 0 DFD, level 1
- DFD, level 2 DFDs, etc., as shown in Figure 6.4 discussed in new subsection. However, a single data dictionary should capture all the data appearing in all the DFDs constituting the DFD model of a system.
- For example, a data dictionary entry may represent that the data *grossPay* consists of the components *regularPay* and *overtimePay*. *grossPay* = *regularPay* + *overtimePay* For the smallest units of data items, the data dictionary simply lists their name and their

type. Composite data items are expressed in terms of the component data items using certain operators. The operators using which a composite data item can be expressed in terms of its component data items are discussed subsequently.

- The dictionary plays a very important role in any software development process, especially for the following reasons:
- A data dictionary provides a standard terminology for all relevant data for use by the developers working in a project. A consistent vocabulary for data items is very important, since in large projects different developers of the project have a tendency to use different terms to refer to the same data, which unnecessarily causes confusion.
- The data dictionary helps the developers to determine the definition of different data structures in terms of their component elements while implementing the design.
- The data dictionary helps to perform impact analysis. That is, it is possible to determine the effect of some data on various processing activities and *vice versa*. Such impact analysis is especially useful when one wants to check the impact of changing an input value type, or a bug in some functionality, etc.

Data definition

- Composite data items can be defined in terms of primitive data items using the following data definition operators.
- + : denotes composition of two data items, e.g. a+b represents data a and b.
- [,,] : represents selection, i.e. any one of the data items listed inside the square bracket can occur. For example, [a,b] represents either a occurs or b occurs.
- () : the contents inside the brac a+(b) represents either a or a+ b occurs.
- {} : represents iterative data definition, e.g. {name}5 represents five name data. {name}* represents zero or more instances of name data.
- = : represents equivalence, e.g. a=b+c means that a is a composite data item comprising of both b and c.
- /**/ : Anything appearing within /* and */ is considered as comment. Ket represents optional data which may or may not appear.

DEVELOPING THE DFD MODEL OF A SYSTEM

- A DFD model of a system graphically represents how each input data is transformed to its corresponding output data through a hierarchy of DFDs.
- The DFD model of a system is constructed by using a hierarchy of DFDs (see Figure 6.4). The top level DFD is called the level 0 DFD or the context diagram. This is the most abstract (simplest) representation of the system (highest level). It is the easiest to draw and understand.
- At each successive lower level DFDs, more and more details are gradually introduced. To develop a higher-level DFD model, processes are decomposed into their subprocesses and the data flow among these subprocesses are identified.

- To develop the data flow model of a system, first the most abstract representation (highest level) of the problem is to be worked out. Subsequently, the lower level DFDs are developed. Level 0 and Level 1 consist of only one DFD each. Level 2 may contain up to 7 separate DFDs, and level 3 up to 49 DFDs, and so on. However, there is only a single data dictionary for the entire DFD model.
- All the data names appearing in all DFDs are populated in the data dictionary and the data dictionary contains the definitions of all the data items.

Context Diagram

- The context diagram is the most abstract (highest level) data flow representation of a system. It represents the entire system as a single bubble. The bubble in the context diagram is annotated with the name of the software system being developed (usually a noun). This is the only bubble in a DFD model, where a noun is used for naming the bubble.
- The bubbles at all other levels are annotated with verbs according to the main function performed by the bubble. This is expected since the purpose of the context diagram is to capture the context of the system rather than its functionality.
- As an example of a context diagram, consider the context diagram a software developed to automate the book keeping activities of a supermarket (see Figure 6.10). The context diagram has been labelled as 'Supermarket software'.
- The name context diagram of the level 0 DFD is justified because it represents the context in which the system would exist; that is, the external entities who would interact with the system and the specific data items that they would be supplying the system and the data items they would be receiving from the system. The various external entities with which the system interacts and the data flow occurring between the system and the external entities are represented. The data input to the system and the data output from the system are represented as incoming and outgoing arrows. These data flow arrows should be annotated with the corresponding data names.
- To develop the context diagram of the system, we have to analyze the SRS document to identify the different types of users who would be using the system and the kinds of data they would be inputting to the system and the data they would be receiving from the system. Here, the term users of the system also include any external systems which supply data to or receive data from the system.

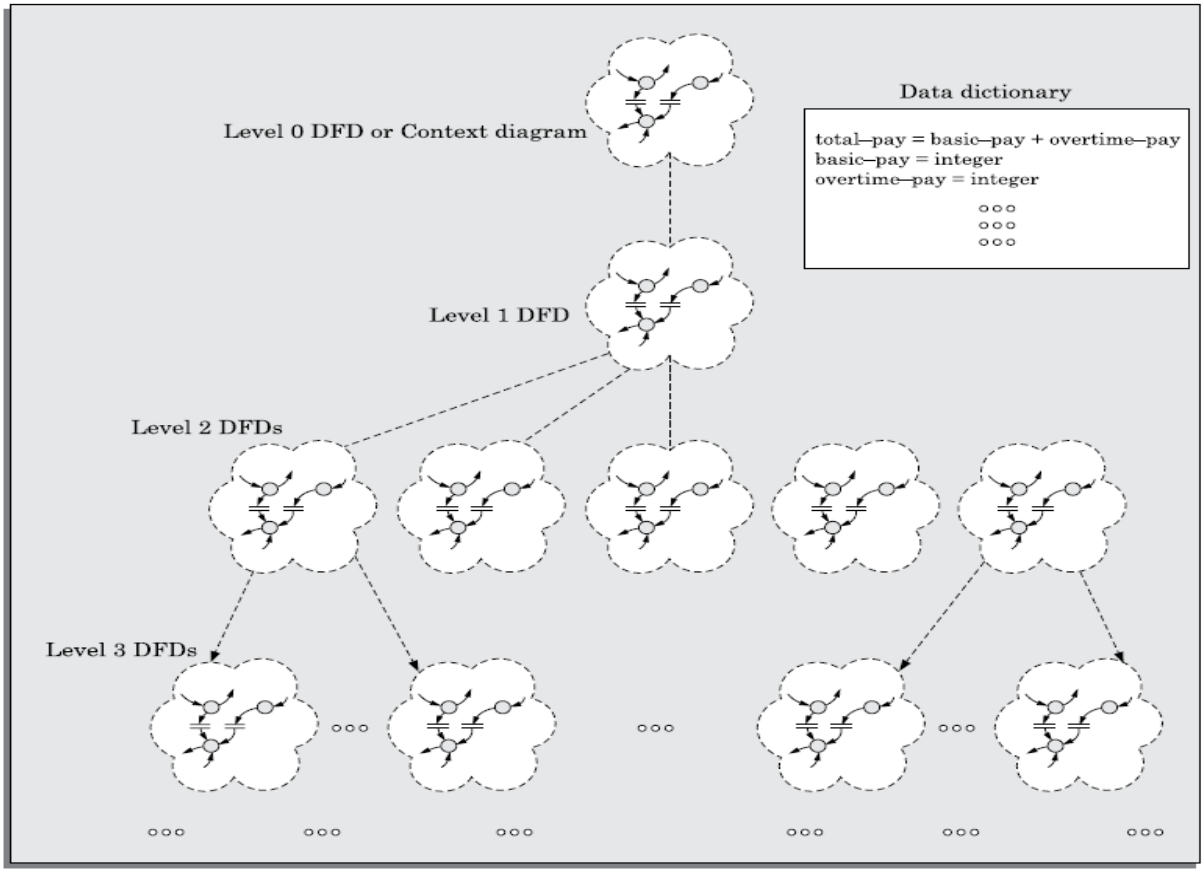

FIGURE 6.4 DFD model of a system consists of a hierarchy of DFDs and a single data

Level 1 DFD

- The level 1 DFD usually contains three to seven bubbles. That is, the system is represented as performing three to seven important functions. To develop the level 1 DFD, examine the high-level functional requirements in the SRS document. If there are three to seven high level functional requirements, then each of these can be directly represented as a bubble in the level 1 DFD. Next, examine the input data to these functions and the data output by these functions as documented in the SRS document and represent them appropriately in the diagram.

Decomposition

- Each bubble in the DFD represents a function performed by the system. The bubbles are decomposed into subfunctions at the successive levels of the DFD model. Decomposition of a bubble is also known as *factoring* or *exploding* a bubble. Each bubble at any level of DFD is usually decomposed to anything three to seven bubbles. A few bubbles at any level make that level superfluous. For example, if a bubble is decomposed to just one bubble or two bubbles, then this decomposition becomes trivial and redundant. On the other hand, too many bubbles at any level of a DFD makes the DFD model hard to understand. Decomposition of a bubble should be carried on until a level is reached at which the function of the bubble can be described using a simple algorithm. We can now describe how to go about developing the DFD model of a system more systematically.

1. **Construction of context diagram:** Examine the SRS document to determine:

- Different high-level functions that the system needs to perform.
- Data input to every high-level function.
- Data output from every high-level function.
- Interactions (data flow) among the identified high-level functions.
- Represent these aspects of the high-level functions in a diagrammatic form. This would form the top-level *data flow diagram* (DFD), usually called the DFD 0.

2. **Construction of level 1 diagram:** Examine the high-level functions described in the SRS document. If there are three to seven high-level requirements in the SRS document, then represent each of the high-level function in the form of a bubble. If there are more than seven bubbles, then some of them have to be combined. If there are less than three bubbles, then some of these have to be split.

3. **Construction of lower-level diagrams:** Decompose each high-level function into its constituent subfunctions through the following set of activities:

- Identify the different subfunctions of the high-level function.
- Identify the data input to each of these subfunctions.
- identify the data output from each of these subfunctions.
- Identify the interactions (data flow) among these subfunctions.
- Represent these aspects in a diagrammatic form using a DFD.
- Recursively repeat Step 3 for each subfunction until a subfunction can be represented by using a simple algorithm.

Numbering of bubbles

It is necessary to number the different bubbles occurring in the DFD. These numbers helping uniquely identifying any bubble in the DFD from its bubble number. The bubble at the context level is usually assigned the number 0 to indicate that it is the 0 level DFD. Bubble sat level 1 are numbered, 0.1, 0.2, 0.3, etc. When a bubble numbered x is decomposed, its children bubble is numbered x.1, x.2, x.3, etc. In this numbering scheme, by looking at the number of a bubble we can unambiguously determine its level, its ancestors, and itssuccessors.

Balancing DFDs

- The DFD model of a system usually consists of many DFDs that are organized in a hierarchy. In this context, a DFD is required to be balanced with respect to the corresponding bubble of the parent DFD. We illustrate the concept of balancing a DFD in Figure 6.5. In the level 1 DFD, data items d1 and d3 flow out of the bubble 0.1 and the data item d2 flows into the bubble 0.1 (shown by the dotted circle). In the next level, bubble 0.1 is decomposed into three DFDs (0.1.1, 0.1.2, 0.1.3). The decomposition is balanced, as d1 and d3 flow out of the level 2 diagram and d2 flows in. Please note that dangling arrows (d1, d2, d3) represent the data flows into or out of a diagram.

How far to decompose?

- A bubble should not be decomposed any further once a bubble is found to represent a simple set of instructions. For simple problems, decomposition up to level 1 should suffice. However, large industry standard problems may need decomposition up to level 3 or level 4. Rarely, if ever, decomposition beyond level 4 is needed.

Commonly made errors while constructing a DFD model

- Although DFDs are simple to understand and draw, students and practitioners alike encounter similar types of problems while modelling software problems using DFDs. While learning from experience is a powerful thing, it is an expensive pedagogical technique in the business world. It is therefore useful to understand the different types of mistakes that beginners usually make while constructing the DFD model of systems, so that you can consciously try to avoid them. The errors are as follows:
- Many beginners commit the mistake of drawing more than one bubble in the context diagram. Context diagram should depict the system as a single bubble.
- Many beginners create DFD models in which external entities appearing at all levels of DFDs. All external entities interacting with the system should be represented only in the context diagram. The external entities should not appear in the DFDs at any other level.
- It is a common oversight to have either too few or too many bubbles in a DFD. Only three to seven bubbles per diagram should be allowed. This also means that each bubble in a DFD should be decomposed three to seven bubbles in the next level.
- Many beginners leave the DFDs at the different levels of a DFD model unbalanced.
- A common mistake committed by many beginners while developing a DFD model is attempting to represent control information in a DFD.

Shortcomings of the DFD model

- DFD models suffer from several shortcomings. The important shortcomings of DFD models are the following:
- Imprecise DFDs leave ample scope to be imprecise. In the DFD model, we judge the function performed by a bubble from its label. However, a short label may not capture the entire functionality of a bubble. For example, a bubble named find book- position has only intuitive meaning and does not specify several things, e.g. what happens when some input information is missing or is incorrect. Further, the find-book-position bubble may not convey anything regarding what happens when the required book is missing.
- Not-well defined control aspects are not defined by a DFD. For instance, the order in which inputs are consumed and outputs are produced by a bubble is not specified.

A DFD model does not specify the order in which the different bubbles are executed Representation of such aspects is very important for modelling real-time systems.

- Decomposition: The method of carrying out decomposition to arrive at the successive levels and the ultimate level to which decomposition is carried out are highly subjective and depend on the choice and judgment of the analyst. Due to this reason, even for the same problem, several alternative DFD representations are possible. Further, many times it is not possible to say which DFD representation is superior or preferable to another one.
- Improper data flow diagram: The data flow diagramming technique does not provide any specific guidance as to how exactly to decompose a given function into its subfunctions and we have to use subjective judgment to carry out decomposition

STRUCTURED DESIGN

The aim of structured design is to transform the results of the structured analysis (that is, the DFD model) into a structure chart. A structure chart represents the software architecture. The various modules making up the system, the module dependency (i.e., which module calls which other modules), and the parameters that are passed among the different modules. The structure chart representation can be easily implemented using some programming language. Since the main focus in a structure chart representation is on module structure of a software and the interaction among the different modules, the procedural aspects (e.g., how a particular functionality is achieved) are not represented.

The basic building blocks using which structure charts are designed are as following:

- **Rectangular boxes:** A rectangular box represents a module. Usually, every rectangular box is annotated with the name of the module it represents.
- **Module invocation 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. However, just by looking at the structure chart, we cannot say whether a module calls another module just once or many times. Also, just by looking at the structure chart, we cannot tell the order in which the different modules are invoked.
- **Data flow arrows:** These are small arrows appearing alongside the module invocation arrows. The data flow arrows are annotated with the corresponding data name. Data flow arrows represent the fact that the named data passes from one module to the other in the direction of the arrow.
- **Library modules:** A library module is usually represented by a rectangle with double edges. Libraries comprise the frequently called *modules*. Usually, when a module is invoked by many other modules, it is made into a library module.
- **Selection:** The diamond symbol represents the fact that one module of several modules connected with the diamond symbol is invoked depending on the outcome of the condition attached with the diamond symbol.
- **Repetition:** A loop around the control flow arrows denotes that the respective modules are invoked repeatedly.
- In any structure chart, there should be one and only one module at the top, called the *root*. There should be at most one control relationship between any two modules in the

structure chart. This means that if module A invokes module B, module B cannot invoke module A. The main reason behind this restriction is that we can consider the different modules of a structure chart to be arranged in layers or levels. The principle of abstraction does not allow lower-level modules to be aware of the existence of the high-level modules However, it is possible for two higher-level modules to invoke the same lower-level module. An example of a properly layered design and another of a poorly layered design

Flow chart *versus* structure chart

- We are all familiar with the flow chart representation of a program. Flow chart is a convenient technique to represent the flow of control in a program. A structure chart differs from a flow chart in three principal ways:
- It is usually difficult to identify the different modules of a program from its flow chart representation.
- Data interchange among different modules is not represented in a flow chart.
- Sequential ordering of tasks that is inherent to a flow chart is suppressed in a structure chart.

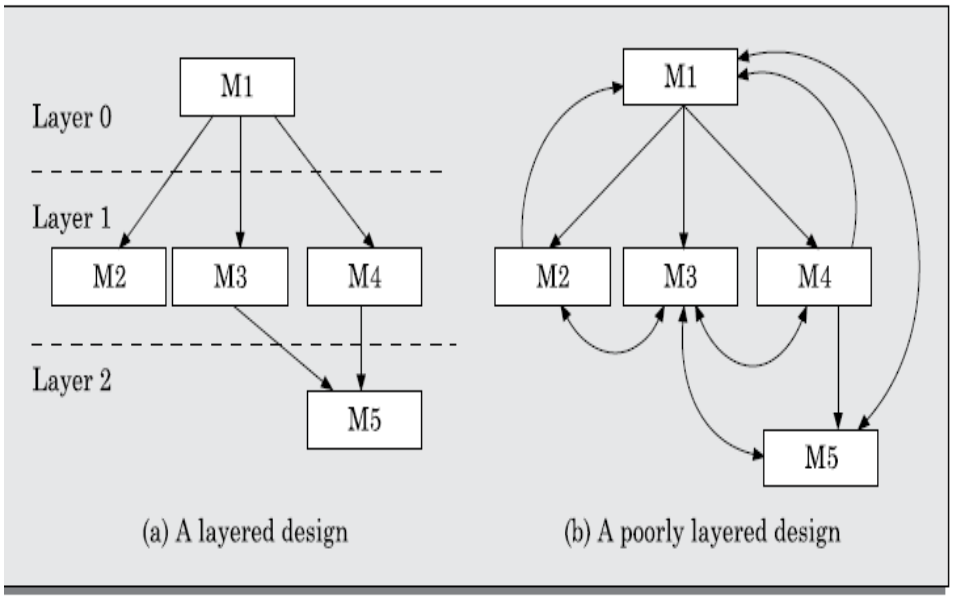

FIGURE 6.18 Examples of properly and poorly layered designs.

Transformation of a DFD Model into Structure Chart

- Systematic techniques are available to transform the DFD representation of a problem into a module structure represented by as a structure chart. Structured design provides two strategies to guide transformation of a DFD into a structure chart:
- Transform analysis
- Transaction analysis

- At each level of transformation, it is important to first determine whether the transform or the transaction analysis is applicable to a particular DFD.

Whether to apply transform or transaction processing?

- Given a specific DFD of a model, how does one decide whether to apply transform analysis or transaction analysis? For this, one would have to examine the data input to the diagram. The data input to the diagram can be easily spotted because they are represented by dangling arrows. If all the data flow into the diagram are processed in similar ways (i.e., if all the input data flow arrows are incident on the same bubble in the DFD) then transform analysis is applicable.
- Otherwise, transaction analysis is applicable. Normally, transform analysis is applicable only to very simple processing. Please recollect that the bubbles are decomposed until it represents a very simple processing that can be implemented using only a few lines of code. Therefore, transform analysis is normally applicable at the lower levels of a DFD model. Each different way in which data is processed corresponds to a separate transaction. Each transaction corresponds to a functionality that lets a user perform a meaningful piece of work using the software.

Transform analysis

- Transform analysis identifies the primary functional components (modules) and the input and output data for these components. The first step in transform analysis is to divide the DFD into three types of parts:
- Input.
- Processing.
- Output.

The input portion in the DFD includes processes that transform input data from physical (e.g., character from terminal) to logical form (e.g., internal tables, lists, etc.). Each input portion is called an *afferent branch.*

The output portion of a DFD transforms output data from logical form to physical form. Each output portion is called an *efferent branch.* The remaining portion of a DFD is called *central transform*.

The output portion of a DFD transforms output data from logical form to physical form. Each output portion is called an *efferent branch.* The remaining portion of a DFD is called *central transform*.

- In the next step of transform analysis, the structure chart is derived by drawing one functional component each for the central transform, the afferent and efferent branches. These are drawn below a root module, which would invoke these modules. Identifying the input and output parts requires experience and skill.
- One possible approach is to trace the input data until a bubble is found whose output data cannot be deduced from its inputs alone. Processes which validate input are not central transforms.

- Processes which sort input or filter data from it are central transforms. The first level of structure chart is produced by representing each input and output unit as a box and each central transform as a single box.
- In the third step of transform analysis, the structure chart is refined by adding subfunctions required by each of the high-level functional components. Many levels of functional components may be added. This process of breaking functional components into subcomponents is called *factoring*. Factoring includes adding read and write modules, error handling modules, initialization and termination process, identifying consumer modules etc. The factoring process is continued until all bubbles in the DFD are represented in the structure chart.

Transaction analysis

- Transaction analysis is an alternative to transform analysis and is useful while designing transaction processing programs. A transaction allows the user to perform some specific type of work by using the software. For example, 'issue book', 'return book', 'query book', etc., are transactions.
- As in transform analysis, first all data entering into the DFD need to be identified. In a transaction-driven system, different data items may pass through different computation paths through the DFD. This is in contrast to a transform centered system where each data item entering the DFD goes through the same processing steps.
- Each different way in which input data is processed is a transaction. A simple way to identify a transaction is the following. Check the input data. The number of bubbles on which the input data to the DFD are incident defines the number of transactions. However, some transactions may not require any input data. These transactions can be identified based on the experience gained from solving a large number of examples.
- For each identified transaction, trace the input data to the output. All the traversed bubbles belong to the transaction. These bubbles should be mapped to the same module on the structure chart. In the structure chart, draw a root module and below this module draw each identified transaction as a module. Every transaction carries a tag identifying its type. Transaction analysis uses this tag to divide the system into transaction modulesand a transaction-centre module.

DETAILED DESIGN

- During detailed design the pseudo code description of the processing and the different data structures are designed for the different modules of the structure chart. These are usually described in the form of module specifications (MSPEC). MSPEC is usually written using structured English. The MSPEC for the non-leaf modules describe the different conditions under which the responsibilities are delegated to the lower-level modules.
- The MSPEC for the leaf-level modules should describe in algorithmic form how the primitive processing steps are carried out. To develop the MSPEC of a module, it is usually necessary to refer to the DFD model and the SRS document to determine the functionality of the module.

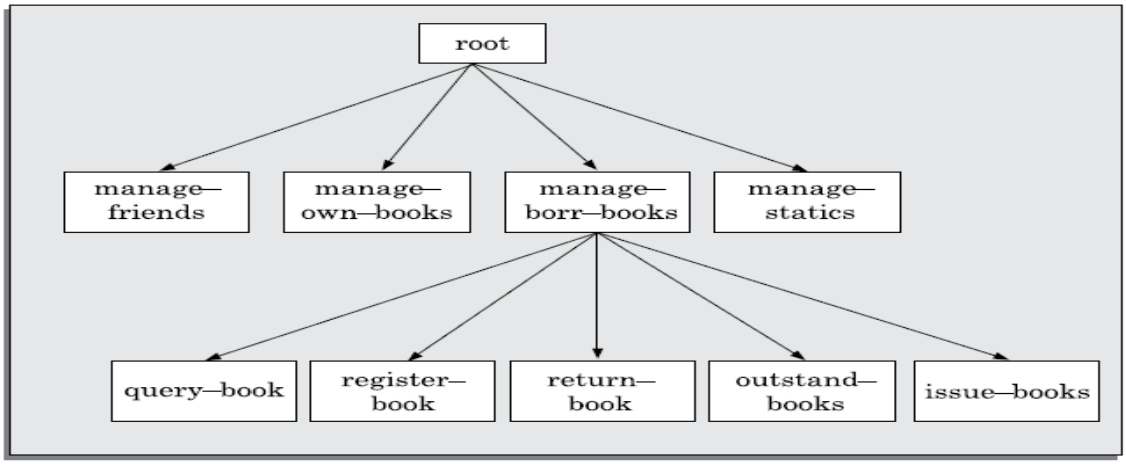

FIGURE 6.23 Structure chart for Problem 6.5.

DESIGN REVIEW

- After a design is complete, the design is required to be reviewed. The review team usually consists of members with design, implementation, testing, and maintenance perspectives, who may or may not be the members of the development team. Normally, members of the team who would code the design, and test the code, the analysts, and the maintainers attend the review meeting. The review team checks the design documents especially for the following aspects:
- **Traceability:** Whether each bubble of the DFD can be traced to some module in the structure chart and *vice versa*. They check whether each functional requirement in the SRS document can be traced to some bubble in the DFD model and *vice versa*.
- **Correctness:** Whether all the algorithms and data structures of the detailed design are correct.
- **Maintainability:** Whether the design can be easily maintained in future.
- **Implementation:** Whether the design can be easily and efficiently be implemented. After the points raised by the reviewers is addressed by the designers, the design document becomes ready for implementation.

UNIT-III
CHAPTER-III
USER INTERFACE DESIGN

The user interface portion of a software product is responsible for all interactions with the user. Almost every software product has a user interface (can you think of a software product that does not have any user interface?). In the early days of computer, no software product had any user interface.

CHARACTERISTICS OF A GOOD USER INTERFACE

How to develop user interfaces, it is important to identify the different characteristics that are usually desired of a good user interface. Unless we know what exactly is expected of a good user interface, we cannot possibly design one.

- **Speed of learning:** A good user interface should be easy to learn. Speed of learning is hampered by complex syntax and semantics of the command issue procedures. A good user interface should not require its users to memories commands. Neither should the user be asked to remember information from one screen to another while performing various tasks using the interface. Besides, the following three issues are crucial to enhance the speed of learning:

- **Use of metaphors**1 **and intuitive command names:** Speed of learning an interface is greatly facilitated if these are based on some day-to-day real-life examples or some physical objects with which the users are familiar with. The abstractions of real-life objects or concepts used in user interface design are called *metaphors*. If the user interface of a text editor uses concepts similar to the tools used by a writer for text editing such as cutting lines and paragraphs and pasting it at other places, users can immediately relate to it.

- Another popular metaphor is a shopping cart. Everyone knows how a shopping cart is used to make choices while purchasing items in a supermarket. If a user interface uses the shopping cart metaphor for designing the interaction style for a situation where similar types of choices have to be made, then the users can easily understand and learn to use the interface. Also, learning is facilitated by intuitive command names and symbolic command issue procedures.

- **Consistency:** Once, a user learns about a command, he should be able to use the similar commands in different circumstances for carrying out similar actions. This makes it easier to learn the interface since the user can extend his knowledge about one part of the interface to the other parts. Thus, the different commands supported by an interface should be consistent.

- **Component-based interface:** Users can learn an interface faster if the interaction style of the interface is very similar to the interface of other applications with which the user is already familiar with. This can be achieved if the interfaces of different applications are developed using some standard user interface components. This, in fact, is the theme of the component-based user interface

- **Speed of use:** Speed of use of a user interface is determined by the time and user effort necessary to initiate and execute different commands. This characteristic of the interface is sometimes referred to as *productivity support* of the interface. It indicates how fast the users can perform their intended tasks. The time and user effort necessary to initiate and execute different commands should be minimal.
- **Speed of recall:** Once users learn how to use an interface, the speed with which they can recall the command issue procedure should be maximized. This characteristic is very important for intermittent users. Speed of recall is improved if the interface is based on some metaphors, symbolic command issue procedures, and intuitive command names.
- **Error prevention:** A good user interface should minimize the scope of committing errors while initiating different commands. The error rate of an interface can be easily determined by monitoring the errors committed by an average user while using the interface. This monitoring can be automated by instrumenting the user interface code with monitoring code which can record the frequency and types of user error and later display the statistics of various kinds of errors committed by different users.
- **Aesthetic and attractive:** A good user interface should be attractive to use. An attractive user interface catches user attention and fancy. In this respect, graphics-based user interfaces have a definite advantage over text-based interfaces.
- **Consistency:** The commands supported by a user interface should be consistent. The basic purpose of consistency is to allow users to generalize the knowledge about aspects of the interface from one part to another. Thus, consistency facilitates speed of learning, speed of recall, and also helps in reduction of error rate.
- **Feedback:** A good user interface must provide feedback to various user actions. Especially, if any user request takes more than few seconds to process, the user should be informed about the state of the processing of his request. In the absence of any response from the computer for a long time, a novice user might even start recovery/shutdown procedures in panic. If required, the user should be periodically informed about the progress made in processing his command.
- **Support for multiple skill levels:** A good user interface should support multiple levels of sophistication of command issue procedure for different categories of users. This is necessary because users with different levels of experience in using an application prefer different types of user interfaces. Experienced users are more concerned about the efficiency of the command issue procedure, whereas novice users pay importance to usability aspects. Very cryptic and complex commands discourage a novice, whereas elaborate command sequences make the command issue procedure very slow and therefore put off experienced users
- **Error recovery (undo facility):** While issuing commands, even the expert users can commit errors. Therefore, a good user interface should allow a user to undo a mistake committed by him while using the interface. Users are inconvenienced if they cannot recover from the errors they commit while using a software. If the users cannot recover even from very simple types of errors, they feel irritated, helpless, and out of control.

- **User guidance and on-line help:** Users seek guidance and on-line help when they either forget a command or are unaware of some features of the software. Whenever users need guidance or seek help from the system, they should be provided with appropriate guidance and help.

BASIC CONCEPTS

Some basic concepts in user guidance and on-line help system. Next, we examine the concept of a mode-based and a modeless interface and the advantages of a graphical interface.

User Guidance and On-line Help

- Users may seek help about the operation of the software any time while using the software. This is provided by the on-line help system. This is different from the guidance and error messages which are flashed automatically without the user asking for them. The guidance messages prompt the user regarding the options he has regarding the next command, and the status of the last command, etc.
- **On-line help system:** Users expect the on-line help messages to be tailored to the context in which they invoke the "help system". Therefore, a good on-line help system should keep track of what a user is doing while invoking the help system and provide the output message in a context-dependent way. Also, the help messages should be tailored to the user's experience level.
- **Guidance messages:** The guidance messages should be carefully designed to prompt the user about the next actions he might pursue, the current status of the system, the progress so far made in processing his last command, etc. A good guidance system should have different levels of sophistication for different categories of users. For example, a user using a command language interface might need a different type of guidance compared to a user using a menu or iconic interface (These different types of interfaces are discussed later in this chapter). Also, users should have an option to turn off the detailed messages.
- **Error messages:** Error messages are generated by a system either when the user commits some error or when some errors encountered by the system during processing due to some exceptional conditions, such as out of memory, communication link broken, etc. Users do not like error messages that are either ambiguous or too general such as "invalid input or system error".
- Error messages should be polite. Error messages should not have associated noise which might embarrass the user. The message should suggest how a given error can be rectified. If appropriate, the user should be given the option of invoking the on-line help system to find out more about the error situation.

Mode-based *versus* Modeless Interface

- A *mode* is a state or collection of states in which only a subset of all user interaction tasks can be performed. In a modeless interface, the same set of commands can be invoked at any time during the running of the software. Thus, a modeless interface has

only a single mode and all the commands are available all the time during the operation of the software.

- On the other hand, in a mode-based interface, different sets of commands can be invoked depending on the mode in which the system is, i.e., the mode at any instant is determined by the sequence of commands already issued by the user.
- A mode-based interface can be represented using a state transition diagram, where each node of the state transition diagram would represent a mode. Each state of the state transition diagram can be annotated with the commands that are meaningful in that state.

Graphical User Interface (GUI) *versus* Text-based User Interface

Let us compare various characteristics of a GUI with those of a text-based user interface:

- In a GUI multiple windows with different information can simultaneously be displayed on the user screen. This is perhaps one of the biggest advantages of GUI over text-based interfaces since the user has the flexibility to simultaneously interact with several related items at any time and can have access to different system information displayed in different windows.
- Iconic information representation and symbolic information manipulation is possible in a GUI. Symbolic information manipulation such as dragging an icon representing a file to a trash for deleting is intuitively very appealing and the user can instantly remember it.
- A GUI usually supports command selection using an attractive and user-friendly menu selection system.
- In a GUI, a pointing device such as a mouse or a light pen can be used for issuing commands. The use of a pointing device increases the efficacy of command issue procedure.
- On the flip side, a GUI requires special terminals with graphics capabilities for running and also requires special input devices such a mouse. On the other hand, a text-based user interface can be implemented even on a cheap alphanumeric display terminal. Graphics terminals are usually much more expensive than alphanumeric terminals.

TYPES OF USER INTERFACES

Broadly speaking, user interfaces can be classified into the following three categories:

1. Command language-based interfaces
2. Menu-based interfaces
3. Direct manipulation interfaces

Each of these categories of interfaces has its own characteristic advantages and disadvantages. Therefore, most modern applications use a careful combination of all these three types of user interfaces for implementing the user command repertoire. It is very difficult to come up with a

simple set of guidelines as to which parts of the interface should be implemented using what type of interface.

This choice is to a large extent dependent on the experience and discretion of the designer of the interface. However, a study of the basic characteristics and the relative advantages of different types of interfaces would give a fair idea to the designer regarding which commands should be supported using what type of interface.

Command Language-based Interface

- A command language-based interface—as the name itself suggests, is based on designing a command language which the user can use to issue the commands. The user is expected to frame the appropriate commands in the language and type them appropriately whenever required. A simple command language-based interface might simply assign unique names to the different commands. However, a more sophisticated command language-based interface may allow users to compose complex commands by using a set of primitive commands. Such a facility to compose commands dramatically reduces the number of command names one would have to remember.
- Command language-based interfaces allow fast interaction with the computer and simplify the input of complex commands. Among the three categories of interfaces, the command language interface allows for most efficient command issue procedure requiring minimal typing. Further, a command language-based interface can be implemented even on cheap alphanumeric terminals. Also,
- A command language-based interface is easier to develop compared to a menu-based or a direct-manipulation interface because compiler writing techniques are well developed. One can systematically develop a command language interface by using the standard compiler writing tools Lex and Yacc.
- However, command language-based interfaces suffer from several drawbacks. Usually, command language-based interfaces are difficult to learn and require the user to memorize the set of primitive commands. Also, most users make errors while formulating commands in the command language and also while typing them

Issues in designing a command language-based interface

- Two overbearing command design issues are to reduce the number of primitive commands that a user has to remember and to minimize the total typing required. We elaborate these considerations in the following:
- The designer has to decide what mnemonics (command names) to use for the different commands. The designer should try to develop meaningful mnemonics and yet be concise to minimize the amount of typing required. For example, the shortest mnemonic should be assigned to the most frequently used commands.
- The designer has to decide whether the users will be allowed to redefine the command names to suit their own preferences. Letting a user define his own mnemonics for various commands is a useful feature, but it increases the complexity of user interface development.

- The designer has to decide whether it should be possible to compose primitive commands to form more complex commands. A sophisticated command composition facility would require the syntax and semantics of the various command composition
- options to be clearly and unambiguously specified. The ability to combine commands is a powerful facility in the hands of experienced users, but quite unnecessary for inexperienced users.

Menu-based Interface

- An important advantage of a menu-based interface over a command language-based interface is that a menu-based interface does not require the users to remember the exact syntax of the commands. A menu-based interface is based on recognition of the command names, rather than recollection. Humans are much better in recognizing something than recollecting it.
- Further, in a menu-based interface the typing effort is minimal as most interactions are carried out through menu selections using a pointing device. This factor is an important consideration for the occasional user who cannot type fast.
- In fact, a major challenge in the design of a menu-based interface is to structure large number of menu choices into manageable forms. In the following, we discuss some of the techniques available to structure a large number of menu items:
- **Scrolling menu:** Sometimes the full choice list is large and cannot be displayed within the menu area, scrolling of the menu items is required. This would enable the user to view and select the menu items that cannot be accommodated on the screen. However, in a scrolling menu all the commands should be highly correlated, so that the user can easily locate a command that he needs.
- This is important since the user cannot see all the commands at any one time. An example situation where a scrolling menu is frequently used is font size selection in a document processor (see Figure 9.1). Here, the user knows that the command list contains only the font sizes that are arranged in some order and he can scroll up or down to find the size he is looking for.

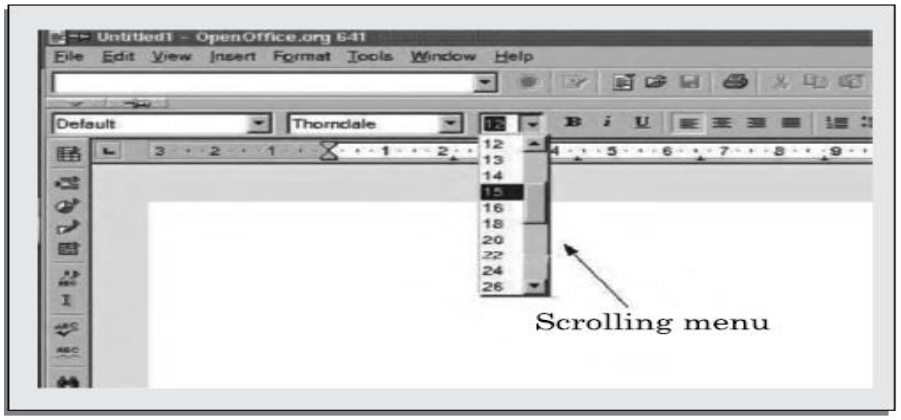

FIGURE 9.1 Font size selection using scrolling menu.

- **Walking menu:** Walking menu is very commonly used to structure a large collection of menu items. In this technique, when a menu item is selected, it causes further menu items to be displayed adjacent to it in a sub-menu.
- A walking menu can successfully be used to structure commands only if there are tens rather than hundreds of choices since each adjacently displayed menu does take up screen space and the total screen area is after all limited.
- **Hierarchical menu:** This type of menu is suitable for small screens with limited display area such as that in mobile phones. In a hierarchical menu, the menu items are organized in a hierarchy or tree structure. Selecting a menu item causes the current menu display to be replaced by an appropriate sub-menu. Thus, in this case, one can consider the menu and its various sub-menu to form a hierarchical tree-like structure.

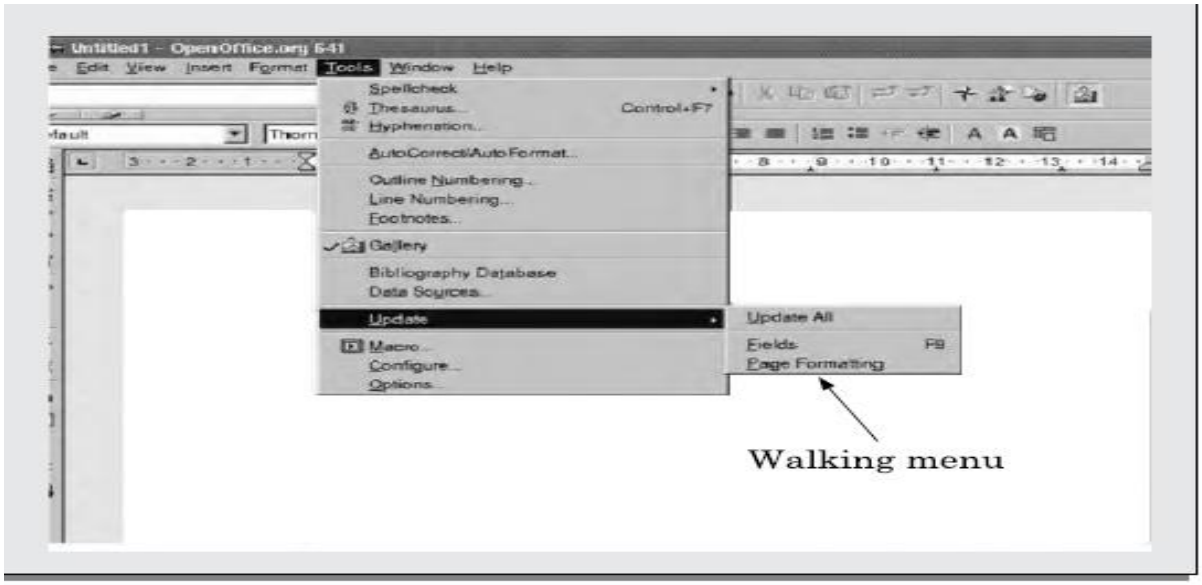

FIGURE 9.2 Example of walking menu.

Direct Manipulation Interfaces

Direct manipulation interfaces present the interface to the user in the form of visual models (i.e., icons2 or objects). For this reason, direct manipulation interfaces are sometimes called as *iconic interfaces.* In this type of interface, the user issues commands by performing actions

- on the visual representations of the objects, e.g., pull an icon representing a file into an icon representing a trash box, for deleting the file. Important advantages of iconic interfaces include the fact that the icons can be recognized by the users very easily, and that icons are language-independent. However, experienced users find direct manipulation interfaces very far too.
- Also, it is difficult to give complex commands using a direct manipulation interface. For example, if one has to drag an icon representing the file to a trash box icon for deleting a file, then in order to delete all the files in the directory one has to perform this operation individually for all files—which could be very easily done by issuing a command like delete *.*.

Performing User interface design: Golden rules.

1. Place the user in control.
2. Reduce the user's memory load.
3. Make the interface consistent.

These golden rules actually form the basis for a set of user interface design principles that guide this important aspect of software design

Place the User in Control

During a requirements-gathering session for a major new information system, a key user was asked about the attributes of the window-oriented graphical interface

Define interaction modes in a way that does not force a user into unnecessary or undesired actions. An interaction mode is the current state of the interface. For example, if spell check is selected in a word-processor menu, the software moves to a spell-checking mode.

Provide for flexible interaction. Because different users have different interaction preferences, choices should be provided.

- For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, a multi touch screen, or voice recognition commands.
- But every action is not amenable to every interaction mechanism

Allow user interaction to be interruptible and undoable

- Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else (without losing the work that had been done). The user should also be able to "undo" any action.

Streamline interaction as skill levels advance and allow the interaction to be customized. Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a "macro" mechanism that enables an advanced user to customize the interface to facilitate interaction.

Hide technical internals from the casual user. The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology.

Design for direct interaction with objects that appear on the screen. The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing. For example, an application interface that allows a user to "stretch" an object (scale it in size) is an implementation of direct manipulation.

Reduce the User's Memory Load

The more a user has to remember, the more error-prone the interaction with the system will be. It is for this reason that a well-designed user interface does not tax the user's memory.

- It defines design principles that enable an interface to reduce the user's memory load:

Reduce demand on short-term memory.

When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions, inputs, and results

Establish meaningful defaults.

The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences.

Define shortcuts that are intuitive.

When mnemonics are used to accomplish a system function (e.g., alt-P to invoke the print function), the mnemonic should be tied to the action in a way that is easy to remember (e.g., first letter of the task to be invoked).

The visual layout of the interface should be based on a real-world metaphor.

For example, a bill payment system should use a checkbook and check register metaphor to guide the user through the bill paying process. This enables the user to rely on well-understood visual cues, rather than memorizing an arcane interaction sequence.

Disclose information in a progressive fashion.

The interface should be organized hierarchically. That is, information about a task, an object, or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick

Make the Interface Consistent

The interface should present and acquire information in a consistent fashion. This implies that

(1) all visual information is organized according to design rules that are maintained throughout all screen displays,

(2) input mechanisms are constrained to a limited set that is used consistently throughout the application, and

(3) mechanisms for navigating from task to task are consistently defined and implemented.

Allow the user to put the current task into a meaningful context.

- Many interfaces implement complex layers of interactions with dozens of screen images. It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand.

Maintain consistency across a family of applications.

- A set of applications (or products) should all implement the same design rules so that consistency is maintained for all interaction

If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

- Once a particular interactive sequence has become a de facto standard (e.g., the use of alt-S to save a file), the user expects this in every application he encounters. A change (e.g., using alt-S to invoke scaling) will cause confusion.