**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**

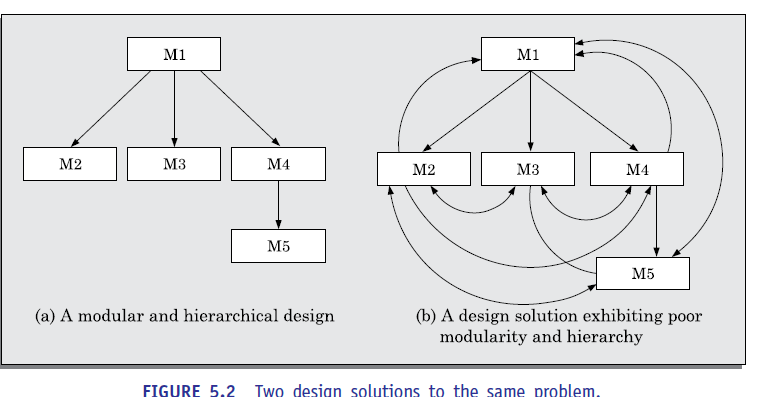
* While performing the design of a certain problem, assume that we have arrived at a large number of design solutions and need to choose the best one. Obviously, all incorrect designs have to be discarded first. Out of the correct design solutions, how can we identify the best one?
* A good design should 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.



**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**

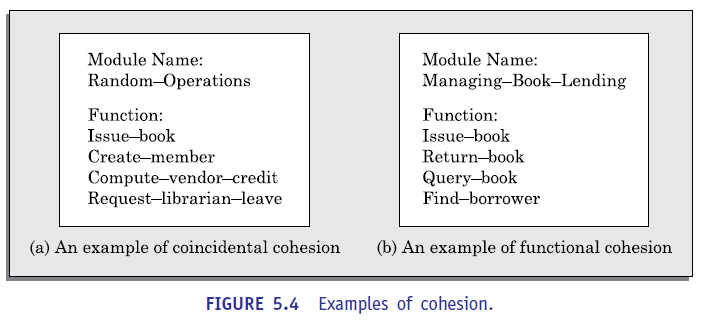
* We have so far discussed that effective problem decomposition is an important characteristic of a good design. Good module decomposition is indicated through *high cohesion* of the individual modules and *low coupling* of the modules with each other. Let us now define what is meant by cohesion and 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.



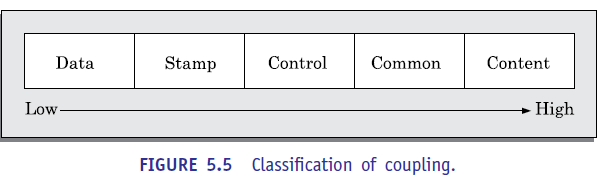
* **Coincidental cohesion:** A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all. In this case, we can say that the module contains a random collection of functions.
* It is likely that the functions have been placed in the module 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.



* **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**

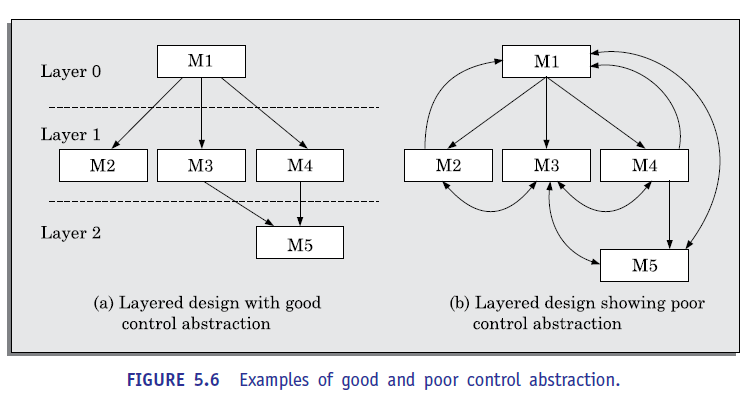
* The coupling between two modules indicates the degree interdependence between them. Intuitively, if two modules interchange large amounts of data, then they are highly interdependent or coupled. We can alternately state this concept as follows.
* The interface complexity is determined based on the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

****

* **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 differenTrat 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.



**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.