**UNIT- III**

Software Design: Software Design Process, Characteristics of Good Software Design, Design Principles, Modular Design, Design Methodologies, Structured Design, Object-Oriented Design: Object oriented Analysis and Design Principles. UML Diagrams, Basic Behavioural Modelling: Interactions, Interaction diagrams. Case Study: The Unified Library application.

**SOFTWARE DESIGN PROCESS:**

Software design is a process to transform user requirements into some suitable form, which helps the programmer in software coding and implementation. For assessing user requirements,an SRS (Software Requirement Specification) document is created whereas for coding and implementation, there is a need of more specific and detailed requirements in software terms. The output of this process can directly be used into implementation in programming languages.

Software design is the first step in SDLC (Software Design Life Cycle), which moves the concentration from problem domain to solution domain. It tries to specify how to fulfill the requirements mentioned in SRS.

**Outcome of the Design Process:**

The following items are designed and documented during the design phase.

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 task 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.

**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.

**Classifification of Design Activities**

A good software design is seldom realised 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.

**Characteristics of Good Software Design:**

Good design relies on a combination of high-level systems thinking and low-level component knowledge. In modern software design, best practice revolves around creating modular components that you can call and deploy as needed. In doing this, you build software that is reusable, extensible, and easy to test. Characteristics of good design are :

**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 optimization 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.

**COHESION AND COUPLING:**

Good module decomposition is indicated through high cohesion of the individual modules and low coupling of the modules with each other.

**Coupling:** 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**: It is a measure of the functional strength of a module, whereas the coupling between two modules is a measure of the degree of interaction (or interdependence) between the two modules.

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**

A module that is highly cohesive and also has low coupling with other modules is said to be functionally independent of the other modules. Functional independence is a key to any good design primarily due to the following advantages it offers:

**Error isolation:** Whenever an error exists in a module, functional independence reduces the chances of the error propagating to the other modules.

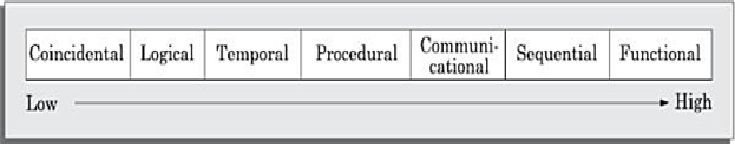
The reason behind this is that if a module is functionally independent, its interaction with other modules is low. Therefore, an error existing in the module is very unlikely to affect the functioning of other modules.

**Scope of reuse:** Reuse of a module for the development of other applications becomes easier. 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.

**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.

**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. The different classes of cohesion that modules can possess are depicted in following Figure. 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

**Logical cohesion:** A module is said to be logically cohesive, if all elements of the module similar operations, such as error handling, data input, data output, 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.

**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.

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

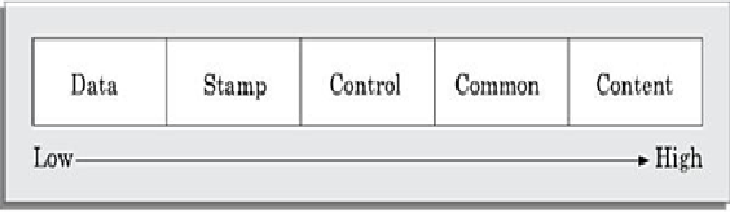
**Sequential cohesion:** A module is said to possess sequential cohesion, if the different functions of the module execute in a sequence, and the output from one function is input to the next in the sequence.

**Functional cohesion:** A module is said to possess functional cohesion, if different functions of the module co-operate to complete a single task.

**Classification of Coupling :**

The coupling between two modules indicates the degree of interdependence between them. Intuitively, if two modules interchange large amounts of data, then they are highly interdependent or coupled.

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. 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 have also been shown in Figure.



**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.

**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.

**Design Principles:**

Software design is both a process and a model. The design process is a sequence of steps that enable the designer to describe all aspects of the software to be built.

It is important to note, however, that the design process is not simply a cookbook. Creative skill, past experience, a sense of what makes “good” software, and an overall commitment to quality are critical success factors for a competent design. The design model is the equivalent of an architect’s plans for a house. It begins by representing the totality of the thing to be built (e.g., a three-dimensional rendering of the house) and slowly refines the thing to provide guidance for constructing each detail (e.g., the plumbing layout). Similarly, the design model that is created for software provides a variety of different views of the computer software. Basic design principles enable the software engineer to navigate the design process. Davis [DAV95] suggests a set of principles for software design, which have been adapted and extended in the following list:

**The design process should not suffer from “tunnel vision.”** A good designer should consider alternative approaches, judging each based on the requirements of the problem, the resources available to do the job.

**The design should be traceable to the analysis model.** Because a single element of the design model often traces to multiple requirements, it is necessary to have a means for tracking how requirements have been satisfied by the design model.

**The design should not reinvent(create) the wheel**. Systems are constructed using a set of design patterns, many of which have likely been encountered before. These patterns should always be chosen as an alternative to reinvention. Time is short and resources are limited! Design time should be invested in representing truly new ideas and integrating those patterns that already exist.

The design should “minimize the intellectual distance” [DAV95] between the software and the problem as it exists in the real world. That is, the structure of the software design should (whenever possible) mimic the structure of the problem domain.

T**he design should exhibit uniformity and integration**. A design is uniform if it appears that one person developed the entire thing. Rules of style and format should be defined for a design team before design work begins. A design is integrated if care is taken in defining interfaces between design components

**The design should be structured to accommodate change**. The design concepts discussed in the next section enable a design to achieve this principle.

**The design should be structured to degrade gently, even when aberrant data, events, or operating conditions are encountered.** Well designed software should never “bomb.” It should be designed to accommodate unusual circumstances, and if it must terminate processing, do so in a graceful manner.

**Design is not coding, coding is not design.** Even when detailed procedural designs are created for program components, the level of abstraction of the design model is higher than source code. The only design decisions made at the coding level address the small implementation details that enable the procedural design to be coded.

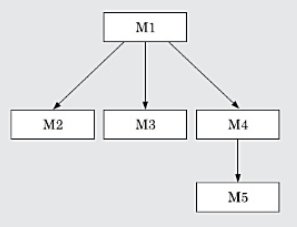
**The design should be assessed for quality as it is being created**, not after the fact. A variety of design concepts (Section 13.4) and design measures are available to assist the designer in assessing quality. •

**The design should be reviewed to minimize conceptual (semantic) errors.** There is sometimes a tendency to focus on minutiae when the design is reviewed, missing the forest for the trees. A design team should ensure that major conceptual elements of the design (omissions, ambiguity, inconsistency) have been addressed before worrying about the syntax of the design model.

**Modular Design:**

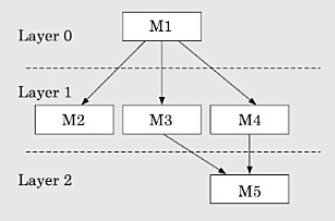
A modular design is an effective decomposition of a problem. It is a basic characteristic of any good design solution. A modular design, in simple words, implies that the problem has been decomposed into a set of modules that have only limited interactions with each other.

Decomposition of a problem into modules facilitates taking advantage of the divide and conquer principle.

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.

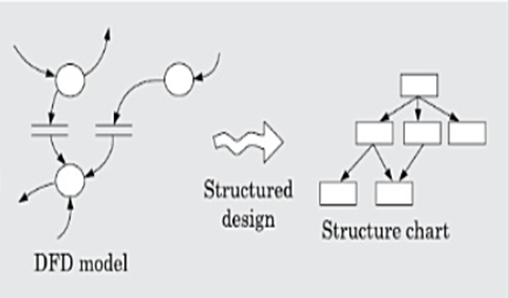
In a layered design solution, the modules are arranged in a hierarchy of layers.   


**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 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.



**Structured design methodologies**

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.

**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.

**Selection:** The diamond symbol represents the fact that one module of several odules

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.

**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:

**1) Transform analysis 2) Transaction analysis**

**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.

**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:

1) Input

2) Processing

3) 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.

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.

**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. 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.

**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 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).** ADT is an mportant concept that forms an important pillar of object-orientation. There are, in fact, three important concepts associated with an ADT

 data abstraction

 data structure & data type.

**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 onlyby calling certain well-defined methods supported by the object.

**Data structure:** A data structure is constructed from a collection of primitive data items. 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. The three main advantages of using ADTs in

programs are :

 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.

 An ADT-based design displays high cohesion and low coupling. Therefore, objectoriented designs are highly modular.

 Since the principle of abstraction is used, it makes the design solution easily

understandable and helps to manage complexity.

**Function-Oriented design :**

The following are the salient features of the function-oriented design approach:

Top-down decomposition :

In top-down decomposition, starting at a high-level view of the system, each high-level function is successively refined into more detailed functions.

This high-level function may be refined into the following subfunctions:

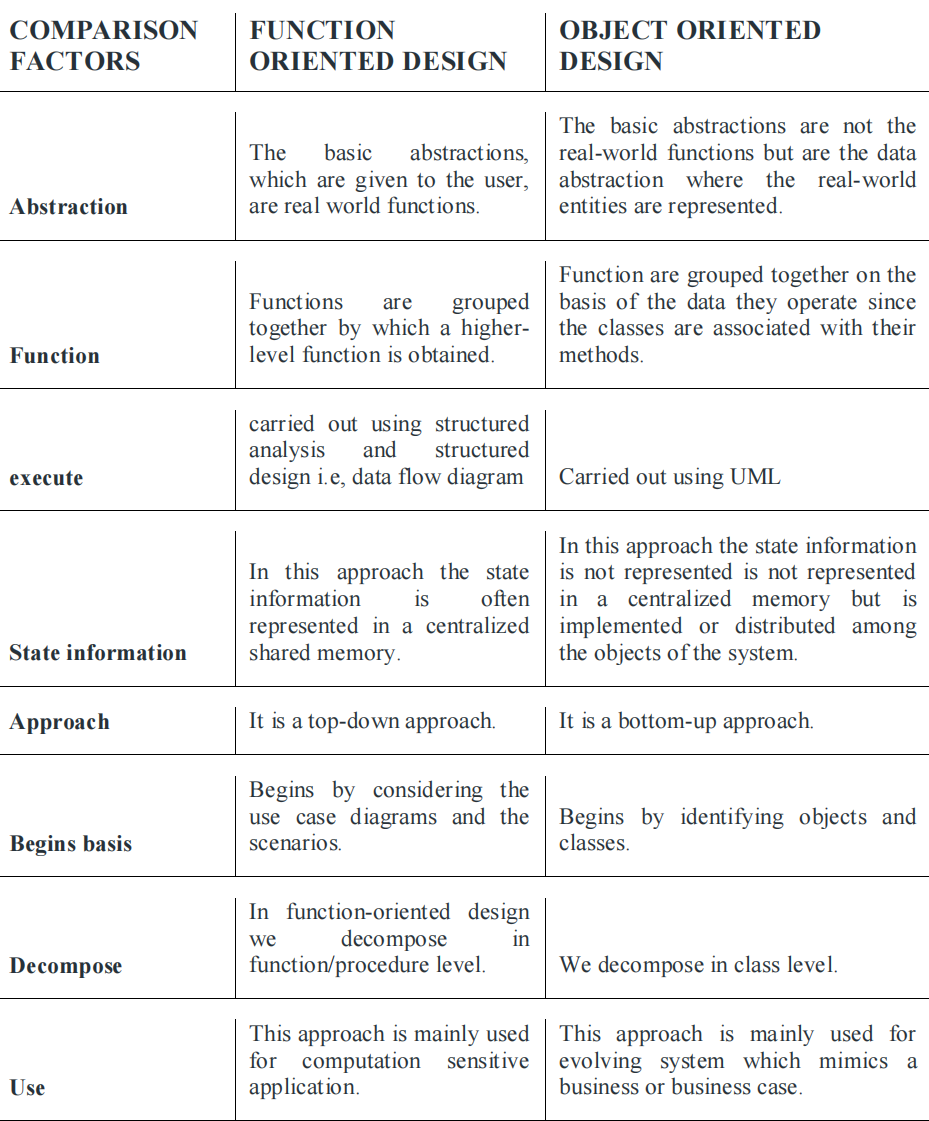
• assign-membership-number

• create-member-record  
 • print-bill

Each of these sub functions may be split into more detailed sub functions and so on.

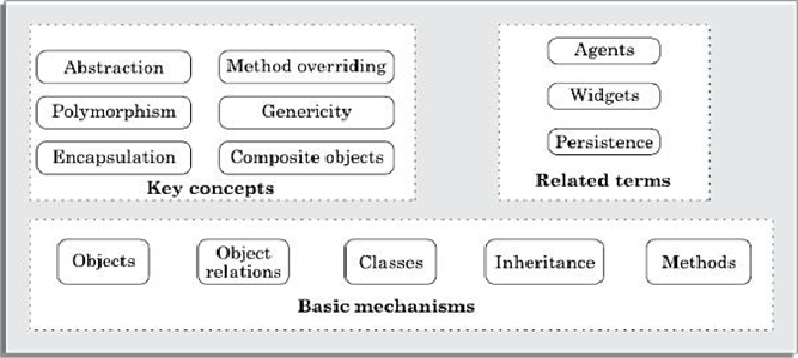
**Centralized system state :**

The system state is centralized and shared among different functions. 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. **Object – Oriented versus Function-Oriented design**



**Object oriented Analysis and Design Principles:**

The principles of object-orientation have been founded on a few simple concepts. These concepts are pictorially showed here



**Figure: Important concepts used in the object-oriented approach**

**Objects**

In the object-oriented approach, it is convenient to imagine the working of a software in terms of a set of interacting objects. Each object in an object-oriented program usually represents a tangible real-world entity such as a library member, a book, an issue register, etc.

**Class**

Similar objects constitute a class. That is, all the objects constituting a class possess similar attributes and methods. For example, the set of all library members would constitute the class Library Member in a library automation application.

**Abstract data:** The data of an object can be accessed only through its methods. In other words, the exact way data is stored internally (stack, array, queue, etc.) in the object is abstracted out (not known to the other objects).

**Data type:** In programming language terminology, a data type defines a collection of data values and a set of predefined operations on those values. Thus, a data type can be instantiated to create a variable of that type. A class is a data type.

**Methods**

The operations (such as create, issue, return, etc.) supported by an object are implemented in the form of methods.

**Class Relationships**

Classes in a programming solution can be related to each other in the ollowing **four** ways

Inheritance

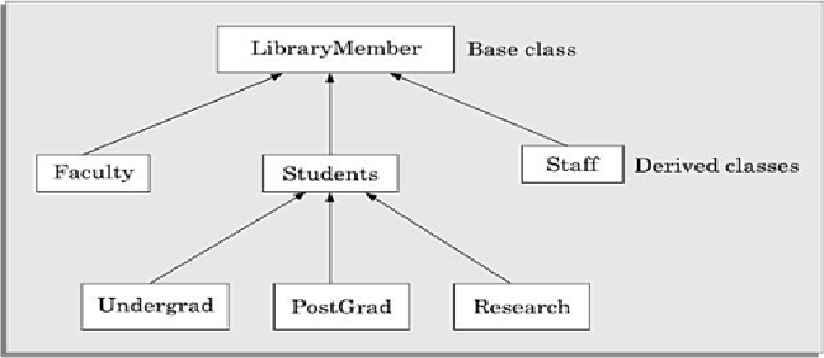
Association and link

Aggregation and composition

Dependency

**Inheritance**

The inheritance feature allows one to define a new class by incrementally extending the features of an existing class. The original class is called the base class (also called superclass or parent class ) and the new class obtained through inheritance is called the derived class (also called a subclass or a child class ). The derived class is said to inherit the features of the base class.



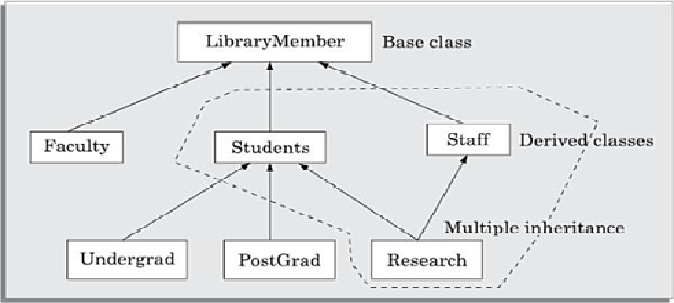
A base class is said to be a generalization of its derived classes. This means that the base class should contain only those properties (i.e., data and methods) that are common to all its derived classes. Each derived class can be considered as a specialization of its base class because it modifies or extends the basic properties of the base class in certain ways. Therefore, the inheritance relationship can be viewed as a generalization - specialization relationship.

Inheritance is a basic mechanism that every object-oriented programming language needs to support. If a language supports ADTs, but does not support inheritance, then it is called an object-based language and not object-oriented. An example of **an object-based programming language is Ada.**

Two important advantages of using the inheritance mechanism inprogramming include code reuse and simplicity of program design.

**Multiple inheritance :**

Multiple inheritance is a mechanism by which a subclass can inherit attributes and methods from more than one base class



**Association and link**

Association is a common type of relation among classes. When two classes are associated, they can take each other’s help (i.e. invoke each other’s methods) to serve user requests. More technically, we can say that if one class is associated with another bidirectionally, then the corresponding objects of the two classes know each other’s ids (identities). As a result, it becomes possible for the object of one class to invoke the methods of the corresponding object of the other class.

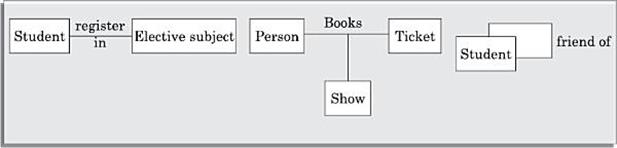
**n-ary association**

Binary association between classes is very commonly encountered in design problems. However, there can be situations where three or more different classes can be involved in an association.

As an example of a ternary association, consider the following—A person books a ticket for a certain show. Here, an association exists among the classes Person, Ticket, and Show. This example of ternary association relationship has pictorially been shown in following Figure.

A class can have an association relationship with itself. This is called recursive association or unary association. As an example, consider the following—two students may be friends. Here, an association named friendship exists among pairs of objects of the **Student** class.

When two classes are associated, the relationship between two objects of the corresponding classes is called a link.



**Composition and aggregation**

Composition and aggregation represent part/whole relationships among objects. Objects which contain other objects are called composite objects.

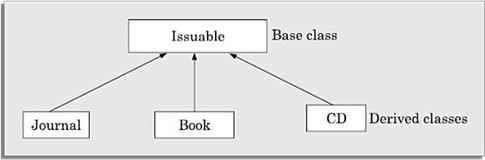
**Dependency**

A dependency relation between two classes shows that any change made to the independent class would require the corresponding change to be made to the dependent class.

**Abstract class**

Classes that are not intended to produce instances of themselves are called abstract classes. In other words, an abstract class cannot be instantiated. Definition of an abstract class helps to push reusable code up in the class hierarchy, thereby enhancing code reuse.

Abstract classes usually support generic methods. These methods help to standardise the method names and input and output parameters in the derived classes. The subclasses of the abstract classes are expected to provide the concrete implementations for these methods.



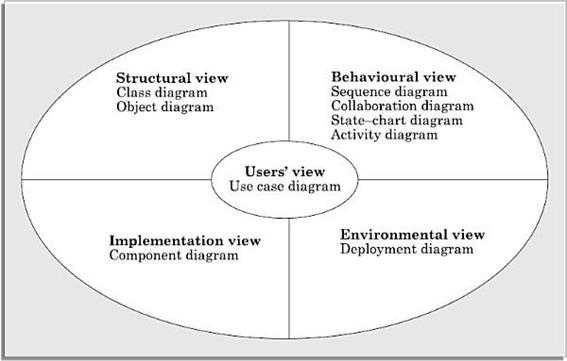
**UNIFIED MODELLING LANGUAGE (UML) DIAGRAMS**

UML was developed to standardize the large number of object-oriented modelling notations that existed in the early nineties. UML was adopted by object management group (OMG) as a defacto standard in 1997.

UML diagrams can capture the following views (models) of a system:

1.  User’s view
2.  Structural view
3.  Behaviourial view
4.  Implementation view
5.  Environmental view

The following figure shows the different views that the UML diagrams can document. Observe that the users’ view is shown as the central view. This is because based on the users’ view, all other views are developed and all views need to conform to the user’s view.



Different types of diagrams and views supported in UML

**Users’ view:**

The users’ view captures the view of the system in terms of the functionalities offered by the system to its users. The users’ view is a black-box view of the system where the internal structure, the dynamic behaviour of different system components, the implementation etc. are not captured.

The users’ view is very different from all other views in the sense that it is a functional model compared to all other views that are essentially object models. The users’ view can be considered as the central view and all other views are required to conform to this view.

**Structural view**

The structural view defines the structure of the problem (or the solution) in terms of the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It also captures the relationships among the classes (objects). The structural model is also called the **static model**, since the structure of a system does not change with time.

**Behaviourial view**

The behaviourial view captures how objects interact with each other in time to realise the system behaviour.

The system behaviour captures the time-dependent (dynamic) behaviour of the system. It therefore constitutes the dynamic model of the system.

**Implementation view**

This view captures the important components of the system and their inter dependencies. For example, the implementation view might show the GUI part, the middleware, and the database part as the different parts and also would capture their inter dependencies.

**Environmental view**

This view models how the different components are implemented on different pieces of hardware.