Unit – 3

Software Design

Software design deals with transforming the customer requirements, as described by the SRS document, into a form that is implementable using a programming language

For a design to be easily implementable in a conventional programming language, the following items must be designed during the design phase

- ❖ Different modules required to implement the design solution.
- ❖ Control relationships among the identified modules. The relationship is also known as the call relationship or invocation relationship among modules.
- ❖ Interface among different modules. The interface among different modules identifies the exact data items exchanged among the modules.
- ❖ Data structures of the individual modules.
- ❖ Algorithms required to implement the individual modules.

The objective of the design phase is to take the SRS document as the input and produce the above mentioned documents before completion of the design phase

We can broadly classified the design activities into two important parts:

1. Preliminary (or High level) Design

- ➢ High level design means identification of different modules, the control relationships and the definitions of the interfaces among them.
- ➢ The outcome of the high level design is called the program structure or software architecture.
- ➢ Many different types of notations have been used to represent a high level design. A popular way is to use a tree like diagram called the structure chart to represent the control hierarchy in a high level design.

2. Detailed Design

- ➢ During detailed design, the data structure and the algorithms of different modules are designed.
- ➢ The outcome of the detailed design stage is usually known as the module specification document.

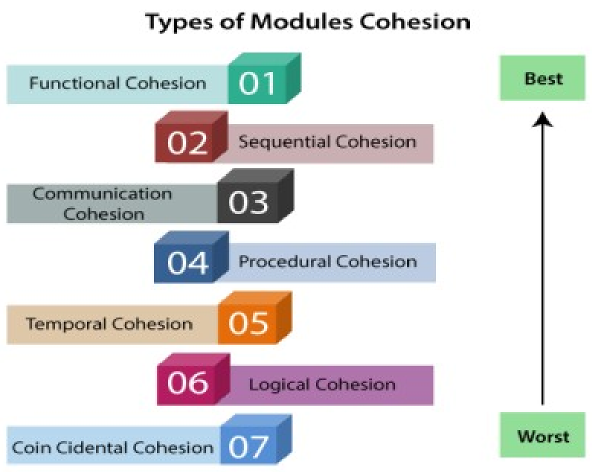
Difference between Good Design and a Bad Design

S.No	Characteristics	Good Design	Bad Design
1.	Change	If the design is good then it will not exhibit ripple effect i.e. change in one part of system will not affect other parts of the system.	A bad design will show ripple effect.
2.	Nature	It will be simple.	It will be complex
3.	Extension	System can be extended with changes in one place.	It can't add a new function without only breaking an existing function
4.	Link	The logic is near the data it operates on.	We can't remember where all the implicitly linked changes have to take place
5.	Cost	A good design costs less	A bad design has more cost.
6.	Logic	No need of logic duplication.	Logic has to be duplicated.

Cohesion & Coupling(modular Design)

Cohesion is the measure of strength of the association of elements within a module. Modules whose elements are strongly and genuinely related to each other are desired.
A module should be highly cohesive.

Types of Modules Cohesion



1. Functional Cohesion: Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.

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

3. Communicational Cohesion: A module is said to have communicational cohesion, if all tasks of the module refer to or update the same data structure, e.g., the set of functions defined on an array or a stack

4. Procedural Cohesion : A procedurally cohesive module is one whose elements are involved in different activities, but the activities are sequential. Procedural cohesion exists when processing elements of a module are related and must be executed in a specified order. For example, Do-while loops

5. Temporal Cohesion: When a module includes functions that are associated by the fact that all the methods must be executed in the same time, the module is said to exhibit temporal cohesion.

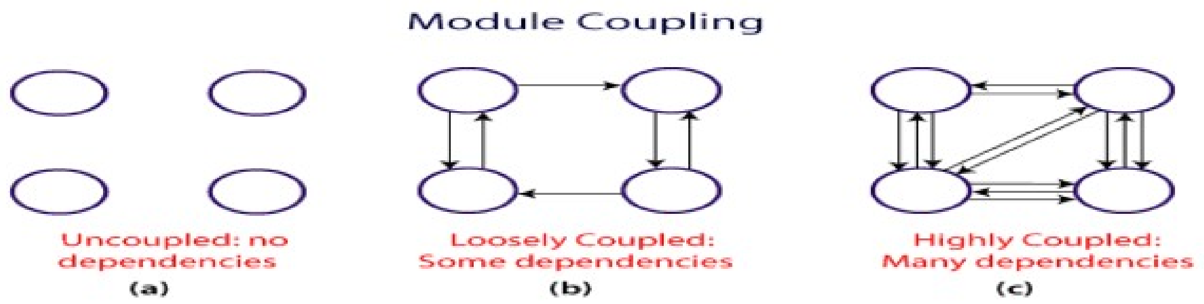
6. Coincidental Cohesion : A module has coincidental cohesion if its elements have no meaningful relationship to one another. It happens when a module is created by grouping unrelated instructions that appear repeatedly in other modules

7. Logical Cohesion: A module is said to be logically cohesive if all the elements of the module perform a similar operation. For example Error handling, data input and data output, etc.

Coupling

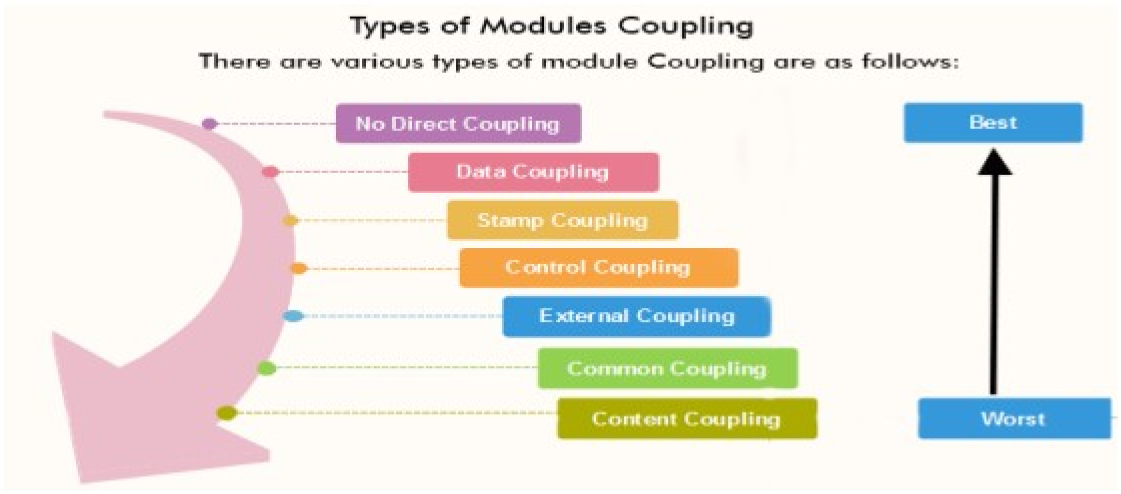
Coupling is the measure of the interdependence of one module to another. Modules should have low coupling. Low coupling minimizes the "ripple effect" where changes in one module cause errors in other modules.

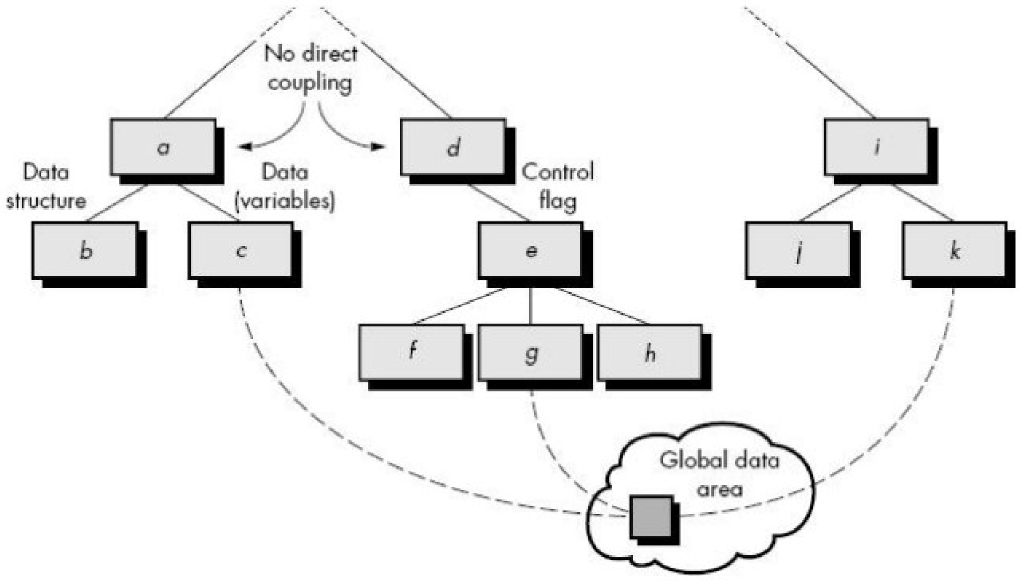
The various types of coupling techniques are shown in fig:


Module Coupling
Uncoupled: no dependencies
(a)
Loosely Coupled: Some dependencies
(b)
Highly Coupled: Many dependencies
(c)

A good design is the one that has low coupling. Coupling is measured by the number of relations between the modules. That is, the coupling increases as the number of calls between modules increase or the amount of shared data is large. Thus, it can be said that a design with high coupling will have more errors.

Types of Module Coupling





1. No Direct Coupling: These are independent modules and so are not really components of a single system. For e.g., this occurs between modules a and d.

2. Data Coupling: Two modules are data coupled if they communicate by passing parameters. This has been told to you as a"good design principle" since day one of your programming instruction. For e.g., this occurs between module a and c.

3. Stamp Coupling: Two modules are stamp coupled if they communicate via a passed data structure that contains more information than necessary for them to perform their functions. For e.g., this occurs between modules b and a

4. Control Coupling: Two modules are control coupled if they communicate using at least one "control flag". For e.g., this occurs between modules d and e

5. Common Coupling: Two modules are common coupled if they both share the same global data area. Another design principle you have been taught since day one: don't use global data. For e.g., this occurs between modules c, g and k.

6. Content Coupling: Two modules are content coupled if: i) . One module changes a statement in another (Lisp was famous for this ability). ii) . One module references or alters data contained inside another module. iii). One module branches into another module. For e.g., this occurs between modules b and f.

<u>**Control Hierarchy**</u>

Software engineering - Layered technology

- Software engineering is a fully layered technology.
- To develop a software, we need to go from one layer to another.
- All these layers are related to each other and each layer demands the fulfilment of the previous layer.

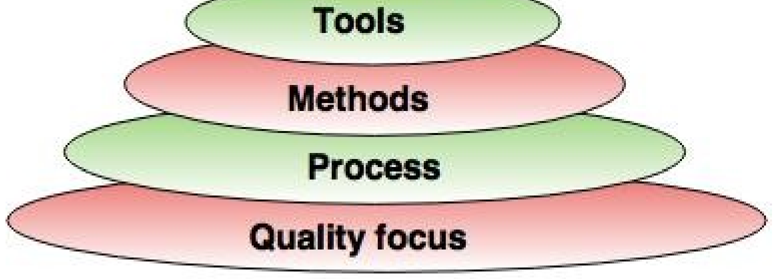


Fig. - Software Engineering Layers

The layered technology consists of:

1. **Quality focus**

   The characteristics of good quality software are:

   - Correctness of the functions required to be performed by the software.
   - Maintainability of the software
   - Integrity i.e. providing security so that the unauthorized user cannot access information or data.
   - Usability i.e. the efforts required to use or operate the software.

2. Process

- It is the base layer or foundation layer for the software engineering.
- The software process is the key to keep all levels together.
- It defines a framework that includes different activities and tasks.
- In short, it covers all activities, actions and tasks required to be carried out for software development.

3. Methods

- The method provides the answers of all 'how-to' that are asked during the process.
- It provides the technical way to implement the software.
- It includes collection of tasks starting from communication, requirement analysis, analysis and design modelling, program construction, testing and support.

4. Tools

- The software engineering tool is an automated support for the software development.
- The tools are integrated i.e the information created by one tool can be used by the other tool.

For example: The Microsoft publisher can be used as a web designing tool.

Control abstraction is the third form of abstraction used in software design. Like procedural and data abstraction, control abstraction implies a program control mechanism without specifying internal details. An example of a control abstraction is the synchronization semaphore used to coordinate activities in an operating system.

Control Hierarchy (Depth, width, fan-in, and fan-out)

a) It is also called as program structure

b) It represents the organization of program components (modules) and implies a hierarchy of control

c) It does not represent procedural aspects of software such as sequence of processes, occurrence or order of decisions or repetitions of operations nor is it necessarily applicable to all architectural styles

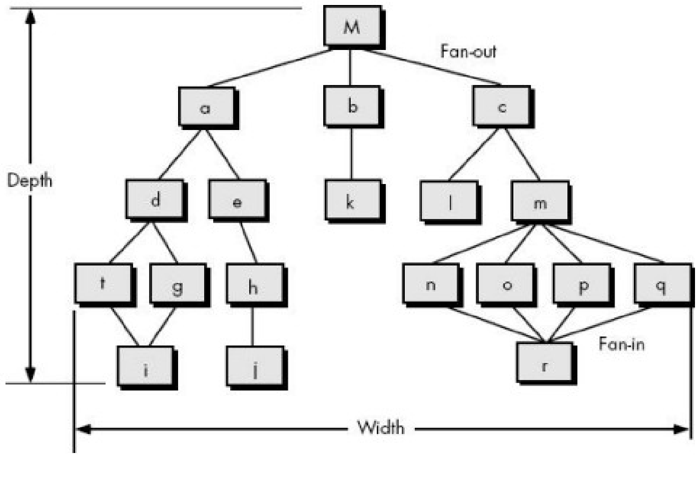
d) The most commonly used notation to represent control hierarchy is the tree–like diagram that represents hierarchical control for call and return architectures

e) 'Depth' and 'Width' provide an indication of the number of levels of control and overall span control respectively.

f) f) 'Fan-out' is a measure of the number of modules that are directly controlled by another module. For e.g. Fan-out of Mis3.

g) g) 'Fan-in' indicates how many modules directly control a given module. For e.g. Fan-in of r is 4

h) h) A module that controls another module is said to be super ordinate to it and conversely, a module controlled by another issaidtobe subordinate to the controller. For e.g. module M is super ordinate to modules a, b and c. Module h is subordinate to module and is ultimately subordinate to module M.



Function Oriented Design

In function-oriented design, the system is comprised of many smaller sub-systems known as functions. These functions are capable of performing significant task in the system. The system is considered as top view of all functions.

Function oriented design inherits some properties of structured design where divide and conquer methodology is used.

This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation.. These functional modules can share information among themselves by means of information passing and using information available globally.

Another characteristic of functions is that when a program calls a function, the function changes the state of the program, which sometimes is not acceptable by other modules. Function oriented design works well where the system state does not matter and program/functions work on input rather than on a state.

Design Process

- The whole system is seen as how data flows in the system by means of data flow diagram.
- DFD depicts how functions changes data and state of entire system.
- The entire system is logically broken down into smaller units known as functions on the basis of their operation in the system.
- Each function is then described at large.

Object Oriented Design

Object oriented design works around the entities and their characteristics instead of functions involved in the software system. This design strategies focuses on entities and its characteristics. The whole concept of software solution revolves around the engaged entities.

Let us see the important concepts of Object Oriented Design:

- **Objects -** All entities involved in the solution design are known as objects. For example, person, banks, company and customers are treated as objects. Every entity has some attributes associated to it and has some methods to perform on the attributes.
- **Classes -** A class is a generalized description of an object. An object is an instance of a class. Class defines all the attributes, which an object can have and methods, which defines the functionality of the object.In the solution design, attributes are stored as variables and functionalities are defined by means of methods or procedures.
- **Encapsulation -** In OOD, the attributes (data variables) and methods (operation on the data) are bundled together is called encapsulation. Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world. This is called information hiding.
- **Inheritance -** OOD allows similar classes to stack up in hierarchical manner where the lower or sub-classes can import, implement and re-use allowed variables and methods from their immediate super classes. This property of OOD is known as inheritance. This makes it easier to define specific class and to create generalized classes from specific ones.

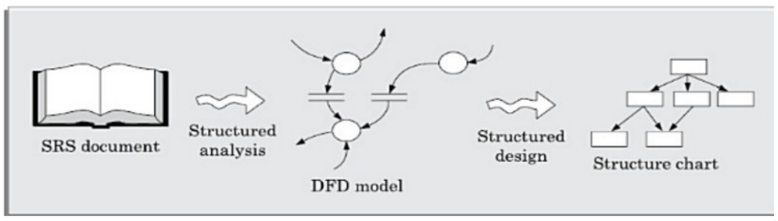
- **Polymorphism -** OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending upon how the function is invoked, respective portion of the code gets executed.

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 below figure.



- 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. The purpose of structured analysis is to capture the detailed structure of the system as perceived by the user

- 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 purpose of structured design is to define the structure of the solution that is suitable for implementation

- The high-level design stage is normally followed by a detailed design stage.

Structured Analysis

During structured analysis, the major processing tasks (high-level functions) of the system are analyzed, and t h e data flow among these processing tasks are represented graphically.
The structured analysis technique is based on the following underlying principles:

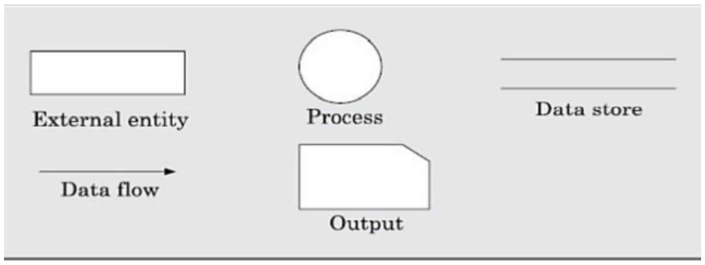
- Top-down decomposition approach.
- Application of divide and conquer principle
- Through this each high level function is independently decomposed into detailed functions. Graphical representation of the analysis results using data flow diagrams (DFDs)

What is DFD?

- A DFD is a hierarchical graphical model of a system that shows the different processing activities or functions that the system performs and the data interchange among those functions.

- Though extremely simple, it is a very powerful tool to tackle the complexity of industry standard problems.
- 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 the DFD terminology, each function is called a process or a bubble. each function as a processing station (or process) that consumes some input data and produces some output data.
- DFD is an elegant modeling technique not only to represent the results of structured analysis but also useful for several other applications.
- Starting with a set of high-level functions that a system performs, a DFD model represents the sub-functions performed by the functions using a hierarchy of diagrams.

Primitive symbols used for constructing DFDs
There are essentially five different types of symbols used for constructing DFDs.



- **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

- **External entity symbol**: 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.

- **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.

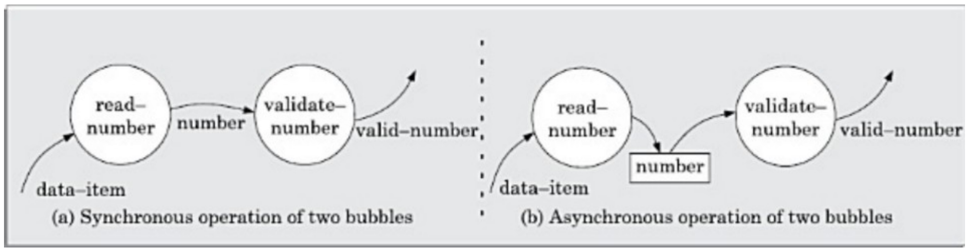
- **Data store symbol**: A data store is represented using two parallel lines. It represents a logical file. That is, a data store symbol can represent either a data structure or a physical file on disk. Connected to a process by means of a data flow symbol. The direction of the data flow arrow shows whether data is being read from or written into a data store.

- **Output symbol**: The output symbol is used when a hard copy is produced.

Important concepts associated with constructing DFD models

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.
- If two bubbles are connected through a data store, as in Figure (b) then the speed of operation of the bubbles are independent



(a) Synchronous operation of two bubbles (b) Asynchronous operation of two bubbles

<u>Data dictionary</u>

- 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.
- A data dictionary lists the purpose of all data items and the definition of all composite data items in terms of their component data items.
- It includes all data flows and the contents of all data stores appearing on all the DFDs in a DFD model.
- 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 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.
  - 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.
  - For large systems, the data dictionary can become extremely complex and voluminous.
- Computer-aided software engineering (CASE) tools come handy to overcome this problem.

- Most CASE tools usually capture the data items appearing in a DFD as the DFD is drawn, and automatically generate the data dictionary.

Object-Oriented Concepts

UML can be described as the successor of object-oriented (OO) analysis and design.

An object contains both data and methods that control the data. The data represents the state of the object. A class describes an object and they also form a hierarchy to model the real-world system. The hierarchy is represented as inheritance and the classes can also be associated in different ways as per the requirement.

Objects are the real-world entities that exist around us and the basic concepts such as abstraction, encapsulation, inheritance, and polymorphism all can be represented using UML.

UML is powerful enough to represent all the concepts that exist in object-oriented analysis and design. UML diagrams are representation of object-oriented concepts only. Thus, before learning UML, it becomes important to understand OO concept in detail.

Following are some fundamental concepts of the object-oriented world −

- **Objects** − Objects represent an entity and the basic building block.
- **Class** − Class is the blue print of an object.
- **Abstraction** − Abstraction represents the behavior of an real world entity.
- **Encapsulation** − Encapsulation is the mechanism of binding the data together and hiding them from the outside world.
- **Inheritance** − Inheritance is the mechanism of making new classes from existing ones.
- **Polymorphism** − It defines the mechanism to exists in different forms.

As UML describes the real-time systems, it is very important to make a conceptual model and then proceed gradually. The conceptual model of UML can be mastered by learning the following three major elements −

- UML building blocks
- Rules to connect the building blocks
- Common mechanisms of UML

The building blocks of UML can be defined as −

- Things
- Relationships
- Diagrams

Things:-
Things are the most important building blocks of UML. Things can be −

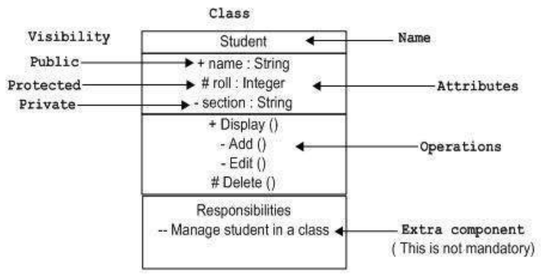
- Structural
- Behavioral
- Grouping
- Annotational

Structural Things:-

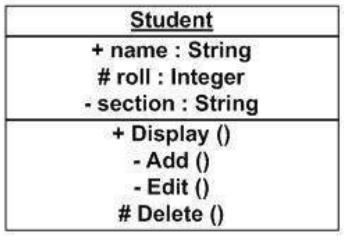
Structural things define the static part of the model. They represent the physical and conceptual elements. Following are the brief descriptions of the structural things.

Class Notation − Class represents a set of objects having similar responsibilities. The diagram is divided into four parts.

- The top section is used to name the class.
- The second one is used to show the attributes of the class.
- The third section is used to describe the operations performed by the class.
- The fourth section is optional to show any additional components.



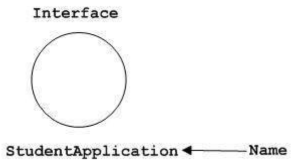
Object Notation - The *object* is represented in the same way as the class. The only difference is the *name* which is underlined as shown in the following figure.



As the object is an actual implementation of a class, which is known as the instance of a class. Hence, it has the same usage as the class.

Interface Notation

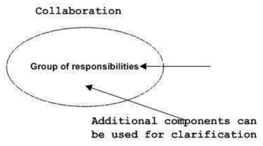
Interface is represented by a circle as shown in the following figure. It has a name which is generally written below the circle.



Interface is used to describe the functionality without implementation. Interface is just like a template where you define different functions, not the implementation. When a class implements the interface, it also implements the functionality as per requirement.

Collaboration Notation

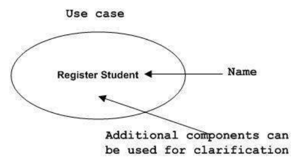
Collaboration is represented by a dotted eclipse as shown in the following figure. It has a name written inside the eclipse.


Collaboration
Group of responsibilities
Additional components can be used for clarification

Collaboration represents responsibilities. Generally, responsibilities are in a group.

Use Case Notation

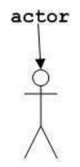
Use case is represented as an eclipse with a name inside it. It may contain additional responsibilities.



Use case is used to capture high level functionalities of a system.

Actor Notation

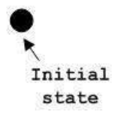
An actor can be defined as some internal or external entity that interacts with the system.



An actor is used in a use case diagram to describe the internal or external entities.

Initial State Notation

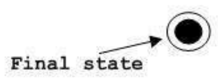
Initial state is defined to show the start of a process. This notation is used in almost all diagrams.



The usage of Initial State Notation is to show the starting point of a process.

Final State Notation

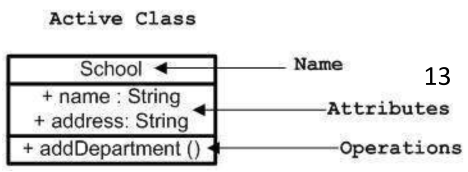
Final state is used to show the end of a process. This notation is also used in almost all diagrams to describe the end.



The usage of Final State Notation is to show the termination point of a process.

Active Class Notation

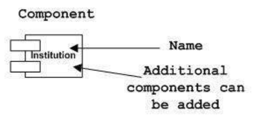
Active class looks similar to a class with a solid border. Active class is generally used to describe the concurrent behavior of a system.



Active class is used to represent the concurrency in a system.

Component Notation

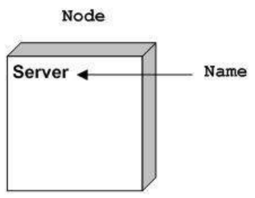
A component in UML is shown in the following figure with a name inside. Additional elements can be added wherever required.



Component is used to represent any part of a system for which UML diagrams are made.

Node Notation

A node in UML is represented by a square box as shown in the following figure with a name. A node represents the physical component of the system.

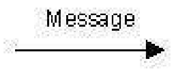


Node is used to represent the physical part of a system such as the server, network, etc.

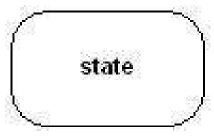
Behavioral Things

A behavioral thing consists of the dynamic parts of UML models. Following are the behavioral things −

Interaction − Interaction is defined as a behavior that consists of a group of messages exchanged among elements to accomplish a specific task.



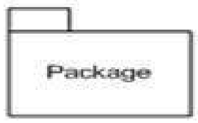
State machine − State machine is useful when the state of an object in its life cycle is important. It defines the sequence of states an object goes through in response to events. Events are external factors responsible for state change



Grouping Things:-

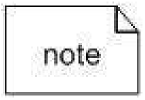
Grouping things can be defined as a mechanism to group elements of a UML model together. There is only one grouping thing available −

Package − Package is the only one grouping thing available for gathering structural and behavioral things.



Annotational Things

Annotational things can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements. **Note** - It is the only one Annotational thing available. A note is used to render comments, constraints, etc. of an UML element.



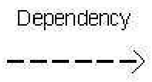
Relationship

Relationship is another most important building block of UML. It shows how the elements are associated with each other and this association describes the functionality of an application.

There are four kinds of relationships available.

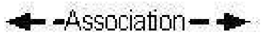
<u>Dependency:-</u>

Dependency is a relationship between two things in which change in one element also affects the other.



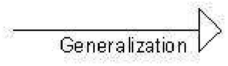
Association

Association is basically a set of links that connects the elements of a UML model. It also describes how many objects are taking part in that relationship.



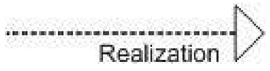
Generalization

Generalization can be defined as a relationship which connects a specialized element with a generalized element. It basically describes the inheritance relationship in the world of objects.



Realization

Realization can be defined as a relationship in which two elements are connected. One element describes some responsibility, which is not implemented and the other one implements them. This relationship exists in case of interfaces.



UML Diagrams

A diagram is the graphical presentation of a set of elements, most often rendered as a connected
graph of vertices (things) and paths (relationships).

A diagram represents an elided view of the elements that make up a system.

In theory, a diagram may contain any combination of things and relationships.

In practice, a small number of common combinations arise, which are consistent with the five most useful views that comprise the architecture of a software intensive system

The UML includes Nine kinds of diagrams:

1. Class diagram
2. Object diagram
3. Use case diagram
4. Sequence diagram
5. Collaboration diagram
6. Statechart diagram
7. Activity diagram
8. Component diagram
9. Deployment diagram

1. Class diagram shows a set of classes, interfaces, and collaborations and their relationships. These diagrams are the most common diagram found in modeling object-oriented systems. Class diagrams address the static design view of a system. Class diagrams that include active classes address the static process view of a system
2. Object diagram shows a set of objects and their relationships. Object diagrams represent static snapshots of instances of the things found in class diagrams. These diagrams address the static design view or static process view of a system as do class diagrams.
3. Use case diagram shows a set of use cases and actors (a special kind of class) and their relationships. Use case diagrams address the static use case view of a system.
4. Sequence diagram is an interaction diagram that emphasizes the time-ordering of messages;
5. Collaboration diagram a communication diagram is an interaction diagram that emphasizes the structural organization of the objects or roles that send and receive messages.
6. Statechart diagram shows a state machine, consisting of states, transitions, events, and activities. A state diagrams shows the dynamic view of an object.
7. Activity diagram shows the structure of a process or other computation as the flow of control and data from step to step within the computation. Activity diagrams address the dynamic view of a system.

8. Component diagram is shows an encapsulated class and its interfaces, ports, and internal structure consisting of nested components and connectors. Component diagrams address the static design implementation view of a system.
9. Deployment diagram shows the configuration of run-time processing nodes and the components that live on them. Deployment diagrams address the static deployment view of an architecture

Characteristics of a good User Interface

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.
- A good user interface should not require its users to memorize commands.
- Neither should the user be asked to remember information from one screen to another
    - Use of metaphors and intuitive command names: Speed of learning an interface is greatly facilitated if these are based on some dayto-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.
    - 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.
    - 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.
- The speed of learning characteristic of a user interface can be determined by measuring the training time and practice that users require before they can effectively use the software.

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.
- 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.
- This can be achieved through careful design of the interface.
- The most frequently used commands should have the smallest length or be available at the top of a menu.

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.
- The interface should prevent the user from entering wrong values

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.

Feedback:

- A good user interface must provide feedback to various user actions.
- Especially, if any user request takes more than a 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.

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.
- The skill level of users improves as they keep using a software product and they look for commands to suit their skill levels.

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:

User Guidance and Online 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.

1. **Online 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 online 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.

2. **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.

3. **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.

Mode-based and modeless interfaces:

- 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.
- A mode-based interface can be represented using a state transition diagram.

Graphical User Interface versus Text-based User Interface:

- In a GUI multiple windows with different information can simultaneously be displayed on the user screen. user has the flexibility to simultaneously interact with several related items at any time
- Iconic information representation and symbolic information manipulation is possible in a GUI.
- 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.
- A GUI requires special terminals with graphics capabilities for running and also requires special input devices such a mouse.
- 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, They have become affordable.

Types of User Interfaces:

- Broadly speaking, user interfaces can be classified into the following three categories:
    - Command language-based interfaces.
    - Menu-based interfaces.
    - Direct manipulation interfaces.

  - Each of these categories of interfaces has its own characteristic advantages and disadvantages.
- Each of these categories of interfaces has its own characteristic advantages and disadvantages.

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.
- The command language interface allows for the most efficient command issue procedure requiring minimal typing.
- a command language-based interface can be implemented even on cheap alphanumeric terminals.
- 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
- command language-based interfaces suffer from several drawbacks.
- command language-based interfaces are difficult to learn and require the user to memorize the set of primitive commands.
- Most users make errors while formulating commands.
- All interactions with the system are through a key-board and cannot take advantage of effective interaction devices such as a mouse.

Issues in designing a command language based interface:

- The designer has to decide what mnemonics (command names) to use for the different commands.
- The designer has to decide whether the users will be allowed to redefine the command names to suit their own preferences.
- The designer has to decide whether it should be possible to compose primitive commands to form more complex commands.

Menu-Based Interfaces:

- 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.
- In a menu-based interface the typing effort is minimal.
- A major challenge in the design of a menu-based interface is to structure a large number of menu choices into manageable forms.
- 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.
        - 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.
    - Walking menu:
        - Walking menu is very commonly used to structure a large collection of menu items.
        - 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.
    - Hierarchical menu:
        - This type of menu is suitable for small screens with limited display area such as that in mobile phones.
        - 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.

Direct Manipulation Interfaces:

- Direct manipulation interfaces present the interface to the user in the form of visual models.

- Direct manipulation interfaces are sometimes called 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 recognised by the users very easily, and that icons are language independent.
- However, experienced users find direct manipulation interfaces very useful too.
- Also, it is difficult to give complex commands using a direct manipulation interface.

User Interface Design Methodology:

- At present, no step-by-step methodology is available which can be followed by rote to come up with a good user interface.
- What we present in this section is a set of recommendations which you can use to complement your ingenuity.

A GUI Design Methodology:

- The GUI design methodology we present here is based on the seminal work of Frank Ludolph.
- Our user interface design methodology consists of the following important steps:
    - Examine the use case model of the software.
    - Interview, discuss, and review the GUI issues with the end-users.
    - Task and object modeling.
    - Metaphor selection.
    - Interaction design and rough layout.
    - Detailed presentation and graphics design.
    - GUI construction. ○ Usability evaluation.

Examining the use case model

- The starting point for GUI design is the use case model.
- This captures the important tasks the users need to perform using the software.
- Metaphors help in interface development at lower effort and reduced costs for training the users.
- Metaphors can also be based on physical objects such as a visitor's book, a catalog, a pen, a brush, a scissor, etc.
- A solution based on metaphors is easily understood by the users, reducing learning time and training costs.

Task and object Modeling:

- A task is a human activity intended to achieve some goals.
- Examples of task goals can be as follows:
  - Reserve an airline seat
  - Buy an item Transfer money from one account to another
  - Book a cargo for transmission to an address.
- A task model is an abstract model of the structure of a task.
- A task model should show the structure of the subtasks that the user needs to perform to achieve the overall task goal.
- Each task can be modeled as a hierarchy of subtasks.
- A task model can be drawn using a graphical notation similar to the activity network model.
- A user object model is a model of business objects which the end-users believe that they are interacting with.
- The objects in a library software may be books, journals, members, etc.

Metaphor selection:

- The first place one should look for while trying to identify the candidate metaphors is the set of parallels to objects, tasks, and terminologies of the use cases.
- If no obvious metaphors can be found, then the designer can fall back on the metaphors of the physical world of concrete objects.
- The appropriateness of each candidate metaphor should be tested by restating the objects and tasks of the user interface model in terms of the metaphor.

Interaction design and rough layout:

- The interaction design involves mapping the subtasks into appropriate controls, and other widgets such as forms, text box, etc.
- This involves making a choice from a set of available components that would best suit the subtask.
- Rough layout concerns how the controls, and other widgets to be organized in windows

Detailed presentation and graphics design:

- Each window should represent either an object or many objects that have a clear relationship to each other.
- At one extreme, each object view could be in its own window. But, this is likely to lead to too much window opening, closing, moving, and resizing.

- At the other extreme, all the views could be placed in one window side-by-side, resulting in a very large window.
- This would force the user to move the cursor around the window to look for different objects.

GUI construction:

- Some of the windows have to be defined as modal dialogs.
- When a window is a modal dialog, no other windows in the application are accessible until the current window is closed.
- When a modal dialog is closed, the user is returned to the window from which the modal dialog was invoked.
- Modal dialogs are commonly used when an explicit confirmation or authorisation step is required for an action.

User interface inspection:

- Nielson studied common usability problems and built a check list of points which can be easily checked for an interface. The following checklist is based on the work of Nielson:
    - Visibility of the system status
    - Match between the system and the real world
    - Undoing mistakes
    - Consistency
    - Recognition rather than recall
    - Support for multiple skill levels
    - Aesthetic and minimalist design
    - Help and error messages
    - ○ Error prevention