

UNIT 3

UNIT 3 SOFTWARE DESIGN

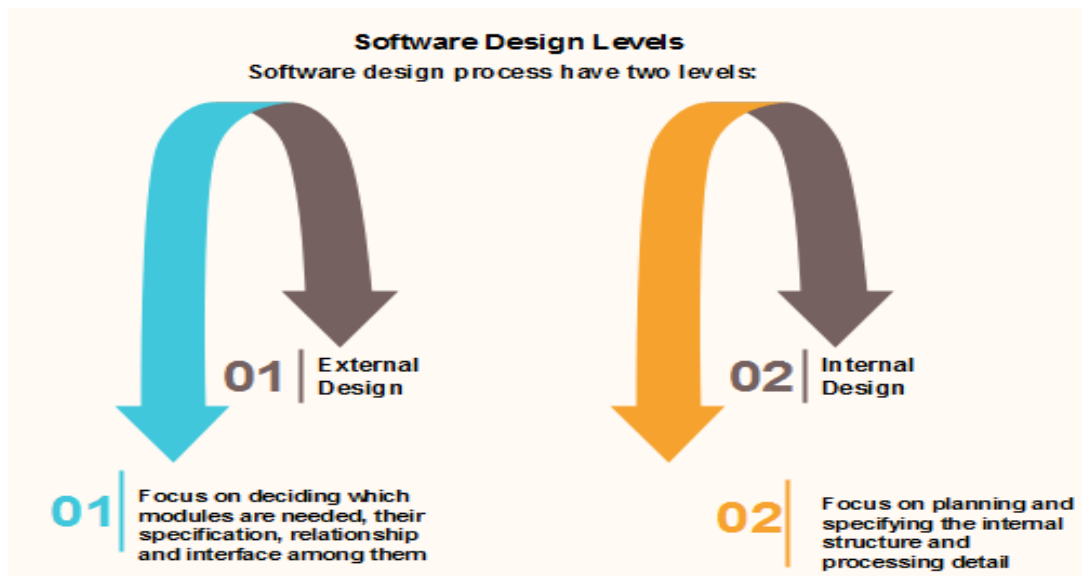
9 Hrs.

Design process - Modular design - Design heuristic - Design model and document - Architectural design - Software architecture - Data design - Architecture data - Transform and transaction mapping - User interface design - User interface design principles.

Design Process:

Software design is a mechanism to transform user requirements into some suitable form, which helps the programmer in software coding and implementation. It deals with representing the client's requirement, as described in SRS (Software Requirement Specification) document, into a form, i.e., easily implementable using programming language.

The software design phase is the first step in SDLC (Software Design Life Cycle), which moves the concentration from the problem domain to the solution domain. In software design, we consider the system to be a set of components or modules with clearly defined behaviors & boundaries.



Objectives of Software Design

Following are the purposes of Software design:

UNIT 3

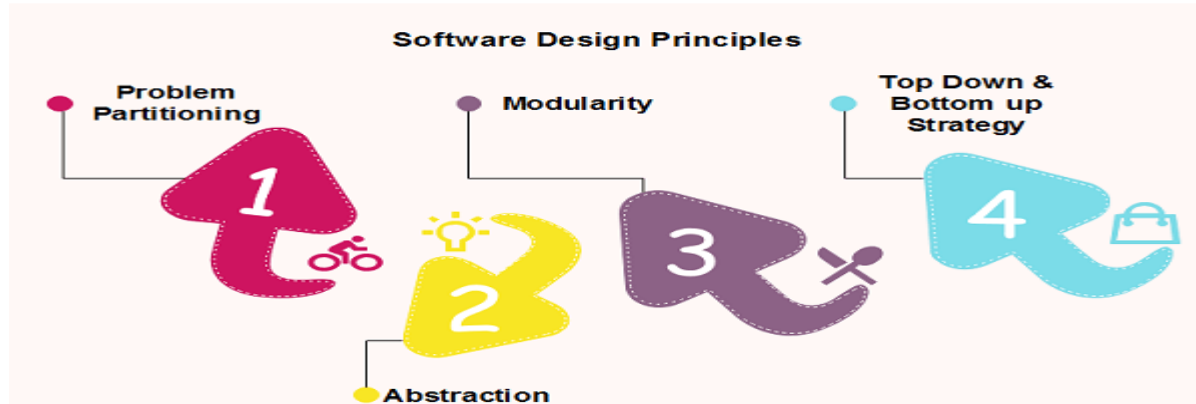


- **Correctness:**Software design should be correct as per requirement.
- **Completeness:**The design should have all components like data structures, modules, and external interfaces, etc.
- **Efficiency:**Resources should be used efficiently by the program.
- **Flexibility:**Able to modify on changing needs.
- **Consistency:**There should not be any inconsistency in the design.
- **Maintainability:** The design should be so simple so that it can be easily maintainable by other designers

Software Design Principles

Software design principles are concerned with providing means to handle the complexity of the design process effectively. Effectively managing the complexity will not only reduce the effort needed for design but can also reduce the scope of introducing errors during design.

UNIT 3



Problem Partitioning

For small problem, we can handle the entire problem at once but for the significant problem, divide the problems and conquer the problem it means to divide the problem into smaller pieces so that each piece can be captured separately.

For software design, the goal is to divide the problem into manageable pieces.

Benefits of Problem Partitioning

1. Software is easy to understand
2. Software becomes simple
3. Software is easy to test
4. Software is easy to modify
5. Software is easy to maintain
6. Software is easy to expand

These pieces cannot be entirely independent of each other as they together form the system. They have to cooperate and communicate to solve the problem. This communication adds complexity.

Abstraction

An abstraction is a tool that enables a designer to consider a component at an abstract level without bothering about the internal details of the implementation. Abstraction can be used for existing element as well as the component being designed.

Here, there are two common abstraction mechanisms

UNIT 3

1. Functional Abstraction
2. Data Abstraction

Functional Abstraction

- i. A module is specified by the method it performs.
- ii. The details of the algorithm to accomplish the functions are not visible to the user of the function.

Functional abstraction forms the basis for **Function oriented design approaches**.

Data Abstraction

Details of the data elements are not visible to the users of data. Data Abstraction forms the basis for **Object Oriented design approaches**.

Modular Design:

Modularity

Modularity specifies to the division of software into separate modules which are differently named and addressed and are integrated later on in to obtain the completely functional software. It is the only property that allows a program to be intellectually manageable. Single large programs are difficult to understand and read due to a large number of reference variables, control paths, global variables, etc.

The desirable properties of a modular system are:

- Each module is a well-defined system that can be used with other applications.
- Each module has single specified objectives.
- Modules can be separately compiled and saved in the library.
- Modules should be easier to use than to build.
- Modules are simpler from outside than inside.

Modularity

Modularity specifies to the division of software into separate modules which are differently named and addressed and are integrated later on in to obtain the

UNIT 3

completely functional software. It is the only property that allows a program to be intellectually manageable. Single large programs are difficult to understand and read due to a large number of reference variables, control paths, global variables, etc.

The desirable properties of a modular system are:

- Each module is a well-defined system that can be used with other applications.
- Each module has single specified objectives.
- Modules can be separately compiled and saved in the library.
- Modules should be easier to use than to build.
- Modules are simpler from outside than inside.

Advantages and Disadvantages of Modularity

Advantages of Modularity

There are several advantages of Modularity

- It allows large programs to be written by several or different people
- It encourages the creation of commonly used routines to be placed in the library and used by other programs.
- It simplifies the overlay procedure of loading a large program into main storage.
- It provides more checkpoints to measure progress.
- It provides a framework for complete testing, more accessible to test
- It produced the well designed and more readable program.

Disadvantages of Modularity

There are several disadvantages of Modularity

- Execution time maybe, but not certainly, longer
- Storage size perhaps, but is not certainly, increased
- Compilation and loading time may be longer
- Inter-module communication problems may be increased
- More linkage required, run-time may be longer, more source lines must be written, and more documentation has to be done

UNIT 3

Modular Design

Modular design reduces the design complexity and results in easier and faster implementation by allowing parallel development of various parts of a system. We discuss a different section of modular design in detail in this section:

1. Functional Independence: Functional independence is achieved by developing functions that perform only one kind of task and do not excessively interact with other modules. Independence is important because it makes implementation more accessible and faster. The independent modules are easier to maintain, test, and reduce error propagation and can be reused in other programs as well. Thus, functional independence is a good design feature which ensures software quality.

It is measured using two criteria:

- **Cohesion:** It measures the relative function strength of a module.
- **Coupling:** It measures the relative interdependence among modules.

2. Information hiding: The fundamental of Information hiding suggests that modules can be characterized by the design decisions that protect from the others, i.e., In other words, modules should be specified that data include within a module is inaccessible to other modules that do not need for such information.

The use of information hiding as design criteria for modular system provides the most significant benefits when modifications are required during testing's and later during software maintenance. This is because as most data and procedures are hidden from other parts of the software, inadvertent errors introduced during modifications are less likely to propagate to different locations within the software.

Strategy of Design

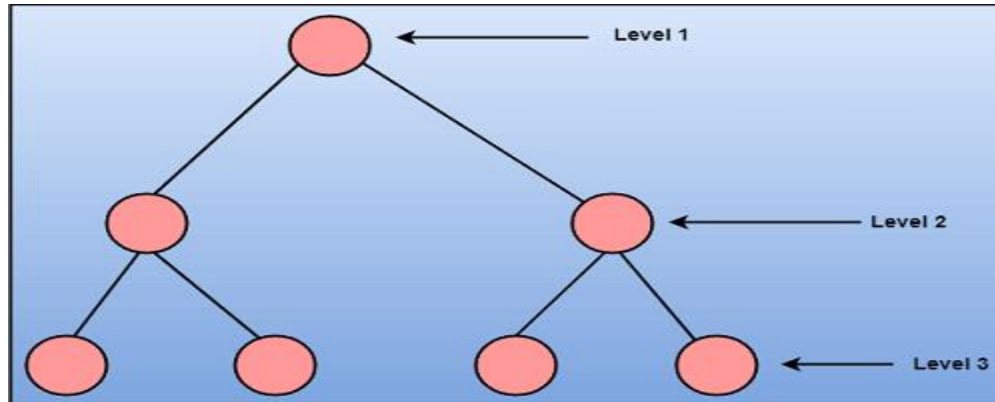
A good system design strategy is to organize the program modules in such a method that are easy to develop and latter too, change. Structured design methods help developers to deal with the size and complexity of programs. Analysts generate instructions for the developers about how code should be composed and how pieces of code should fit together to form a program.

To design a system, there are two possible approaches:

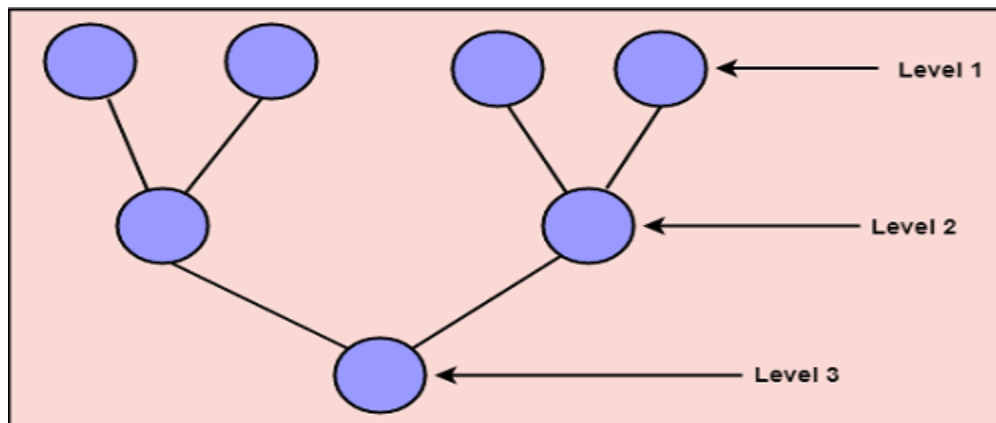
1. Top-down Approach
2. Bottom-up Approach

UNIT 3

1. **Top-down Approach:** This approach starts with the identification of the main components and then decomposing them into their more detailed sub-components.



2. **Bottom-up Approach:** A bottom-up approach begins with the lower details and moves towards up the hierarchy, as shown in fig. This approach is suitable in case of an existing system.



Coupling and Cohesion

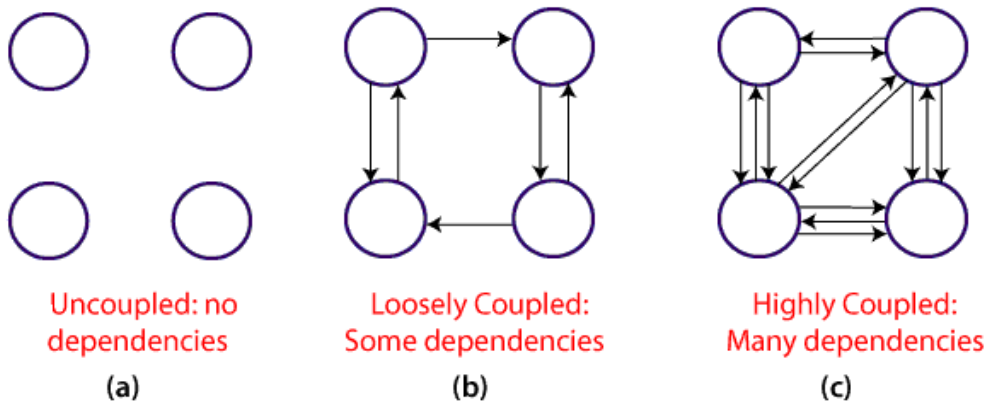
Module Coupling

In software engineering, the coupling is the degree of interdependence between software modules. Two modules that are tightly coupled are strongly dependent on each other. However, two modules that are loosely coupled are not dependent on each other. **Uncoupled modules** have no interdependence at all within them.

The various types of coupling techniques are shown in fig

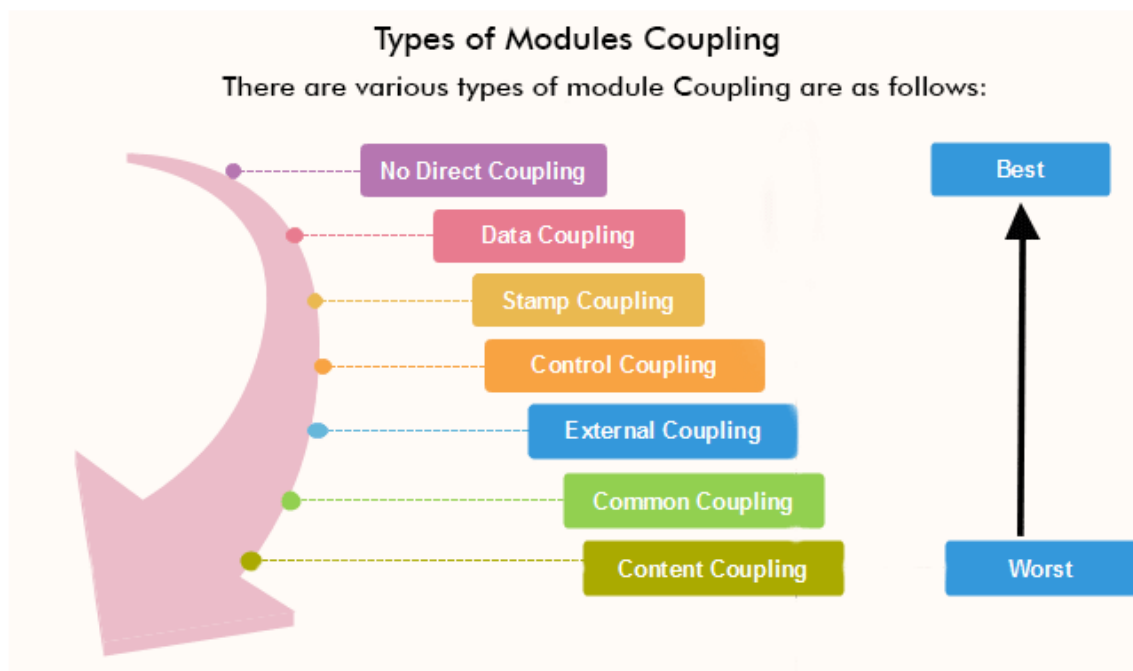
UNIT 3

Module Coupling



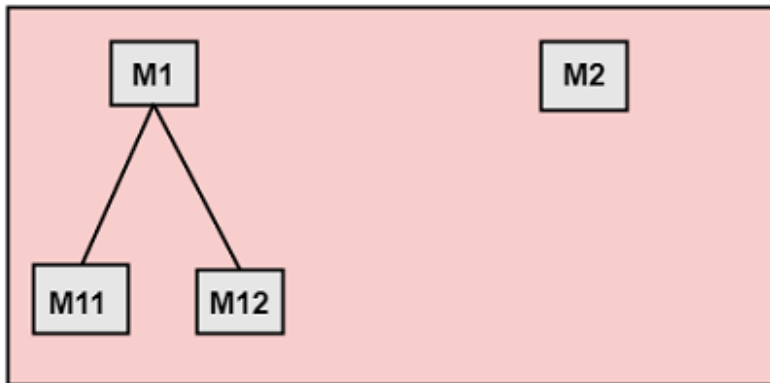
A good design is the one that has low coupling. Coupling is measured by the number of relations between the modules. That is, the coupling increases as the number of calls between modules increase or the amount of shared data is large. Thus, it can be said that a design with high coupling will have more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

Types of Module Coupling



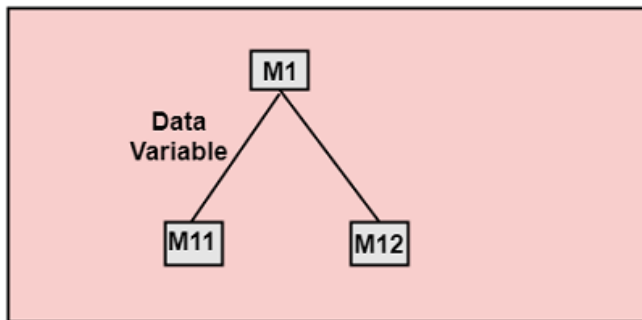
UNIT 3

1. No Direct Coupling: There is no direct coupling between M1 and M2.



In this case, modules are subordinates to different modules. Therefore, no direct coupling.

2. Data Coupling: When data of one module is passed to another module, this is called data coupling.



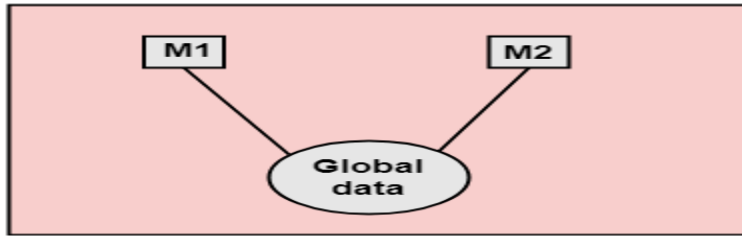
3. Stamp Coupling: Two modules are stamp coupled if they communicate using composite data items such as structure, objects, etc. When the module passes non-global data structure or entire structure to another module, they are said to be stamp coupled. For example, passing structure variable in C or object in C++ language to a module.

4. Control Coupling: Control Coupling exists among two modules if data from one module is used to direct the structure of instruction execution in another.

5. External Coupling: External Coupling arises when two modules share an externally imposed data format, communication protocols, or device interface. This is related to communication to external tools and devices.

6. Common Coupling: Two modules are common coupled if they share information through some global data items.

UNIT 3

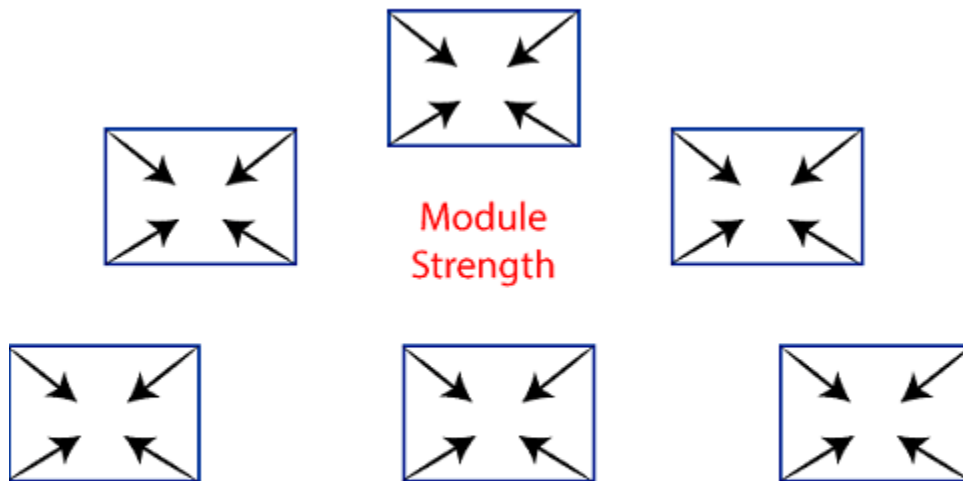


7. **Content Coupling:** Content Coupling exists among two modules if they share code, e.g., a branch from one module into another module.

Module Cohesion

In computer programming, cohesion defines to the degree to which the elements of a module belong together. Thus, cohesion measures the strength of relationships between pieces of functionality within a given module. For example, in highly cohesive systems, functionality is strongly related.

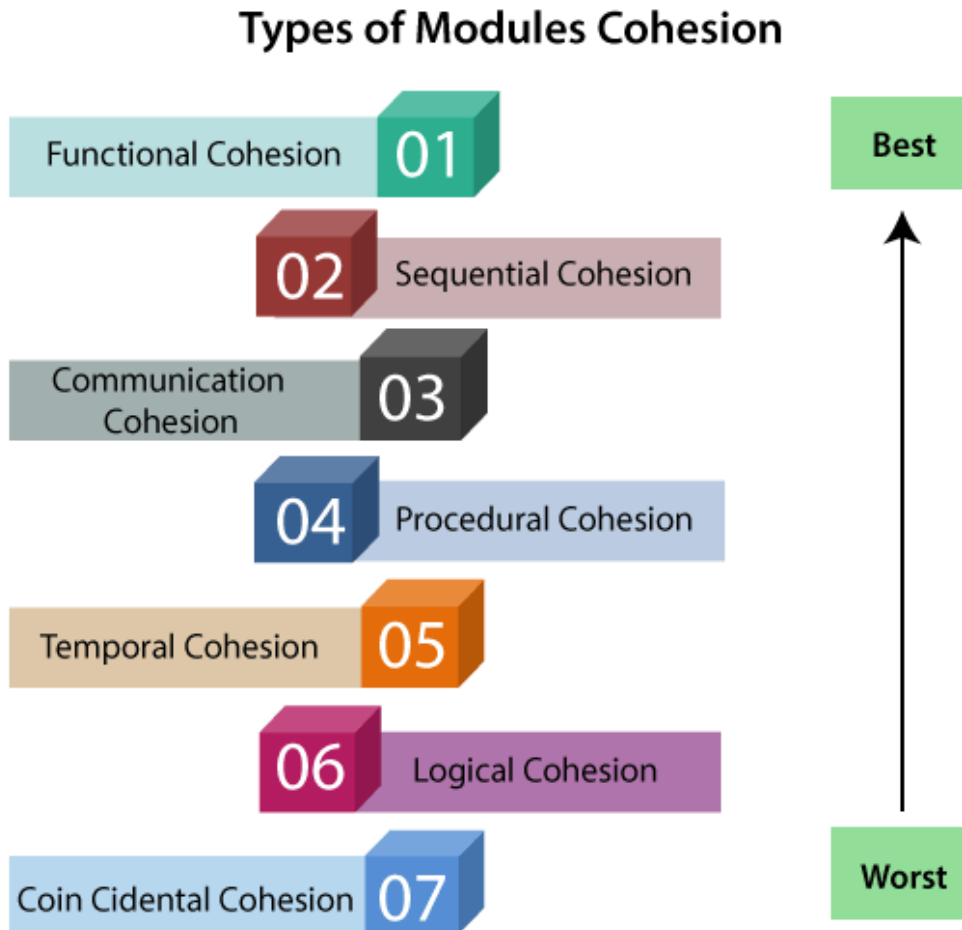
Cohesion is an **ordinal** type of measurement and is generally described as "high cohesion" or "low cohesion."



Cohesion= Strength of relations within Modules

UNIT 3

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 module is said to be procedural cohesion if the set of purpose of the module are all parts of a procedure in which particular sequence of steps has to be carried out for achieving a goal, e.g., the algorithm for decoding a message.

UNIT 3

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. **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.
7. **Coincidental Cohesion:** A module is said to have coincidental cohesion if it performs a set of tasks that are associated with each other very loosely, if at all

Differentiate between Coupling and Cohesion

Coupling	Cohesion
Coupling is also called Inter-Module Binding.	Cohesion is also called Intra-Module Binding.
Coupling shows the relationships between modules.	Cohesion shows the relationship within the module.
Coupling shows the relative independence between the modules.	Cohesion shows the module's relative functional strength.
While creating, you should aim for low coupling, i.e., dependency among modules should be less.	While creating you should aim for high cohesion, i.e., a cohesive component/ module focuses on a single function (i.e., single-mindedness) with little interaction with other modules of the system.
In coupling, modules are linked to the other modules.	In cohesion, the module focuses on a single thing.

Active
Go to S

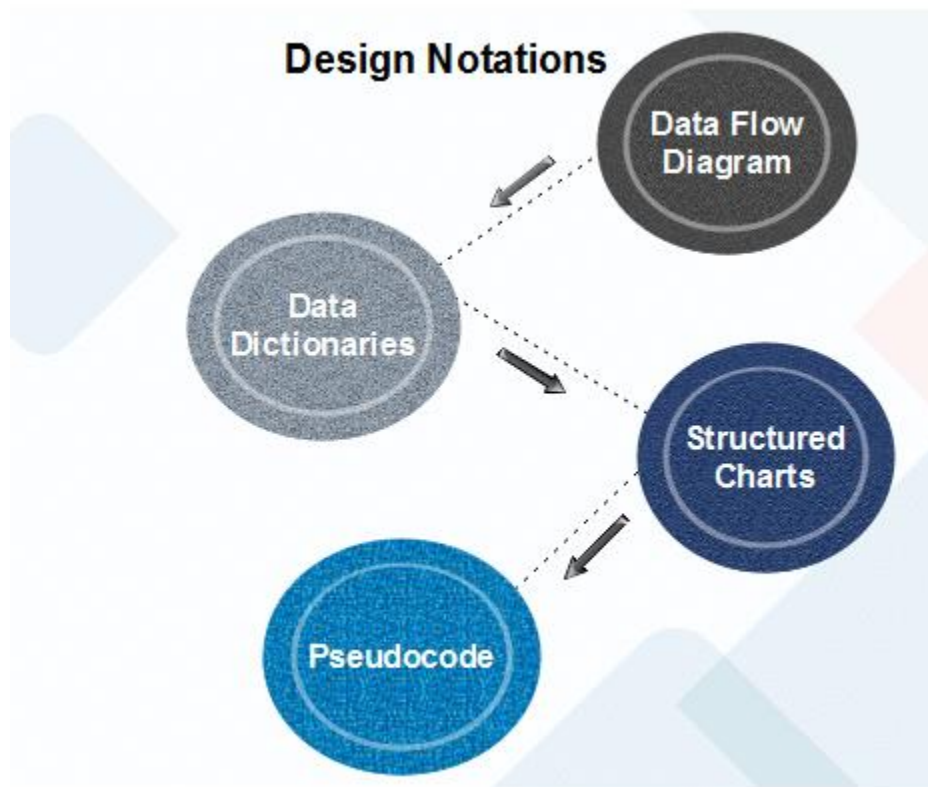
Function Oriented Design

Function Oriented design is a method to software design where the model is decomposed into a set of interacting units or modules where each unit or module has a clearly defined function. Thus, the system is designed from a functional viewpoint.

UNIT 3

Design Notations

Design Notations are primarily meant to be used during the process of design and are used to represent design or design decisions. For a function-oriented design, the design can be represented graphically or mathematically by the following:



Data Flow Diagram





Data-flow design is concerned with designing a series of functional transformations that convert system inputs into the required outputs. The design is described as data-flow diagrams. These diagrams show how data flows through a system and how the output is derived from the input through a series of functional transformations.

Data-flow diagrams are a useful and intuitive way of describing a system. They are generally understandable without specialized training, notably if control information is excluded. They show end-to-end processing. That is the flow of processing from when data enters the system to where it leaves the system can be traced.

UNIT 3

Data-flow design is an integral part of several design methods, and most CASE tools support data-flow diagram creation. Different ways may use different icons to represent data-flow diagram entities, but their meanings are similar.

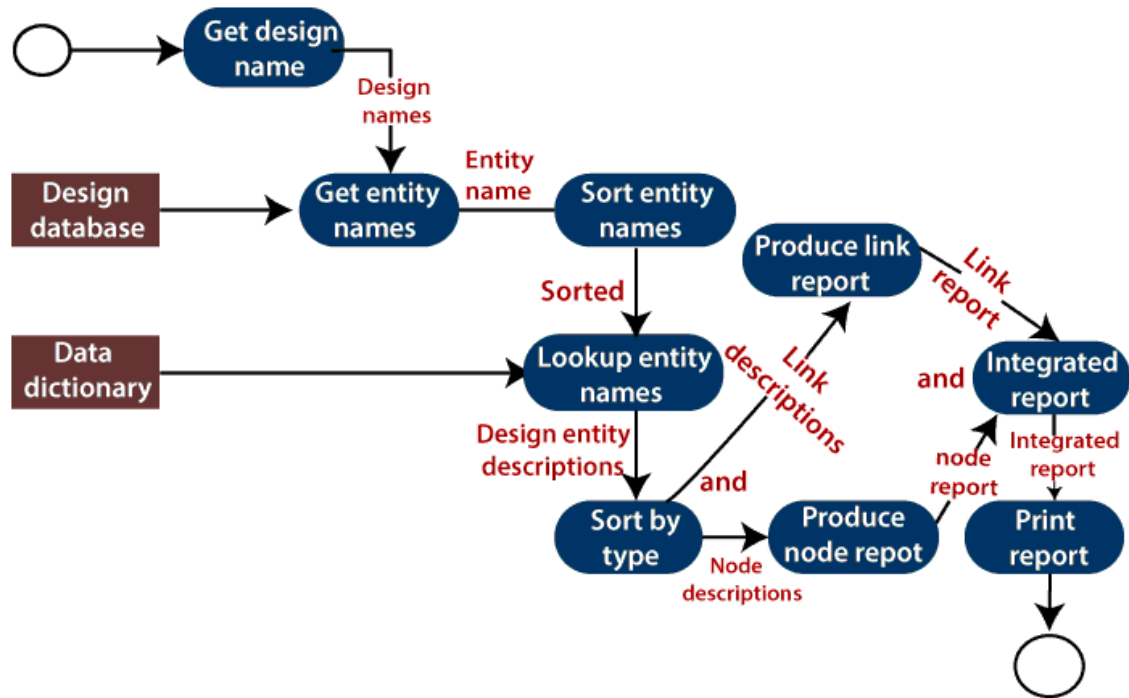
The notation which is used is based on the following symbols:

Symbol	Name	Meaning
	Rounded Rectangle	It represents functions which transforms input to output. The transformation name indicates its function.
	Rectangle	It represents data stores. Again, they should give a descriptive name.
	Circle	It represents user interactions with the system that provides input or receives output.
	Arrows	It shows the direction of data flow. Their name describes the data flowing along the path.
"and" and "or"	Keywords	The keywords "and" and "or". These have their usual meanings in boolean expressions. They are used to link data flows when more than one data flow may be input or output from a transformation.

Data flow diagram of a design
report generator

Acti
Go to

UNIT 3



The report generator produces a report which describes all of the named entities in a data-flow diagram. The user inputs the name of the design represented by the diagram. The report generator then finds all the names used in the data-flow diagram. It looks up a data dictionary and retrieves information about each name. This is then collated into a report which is output by the system.

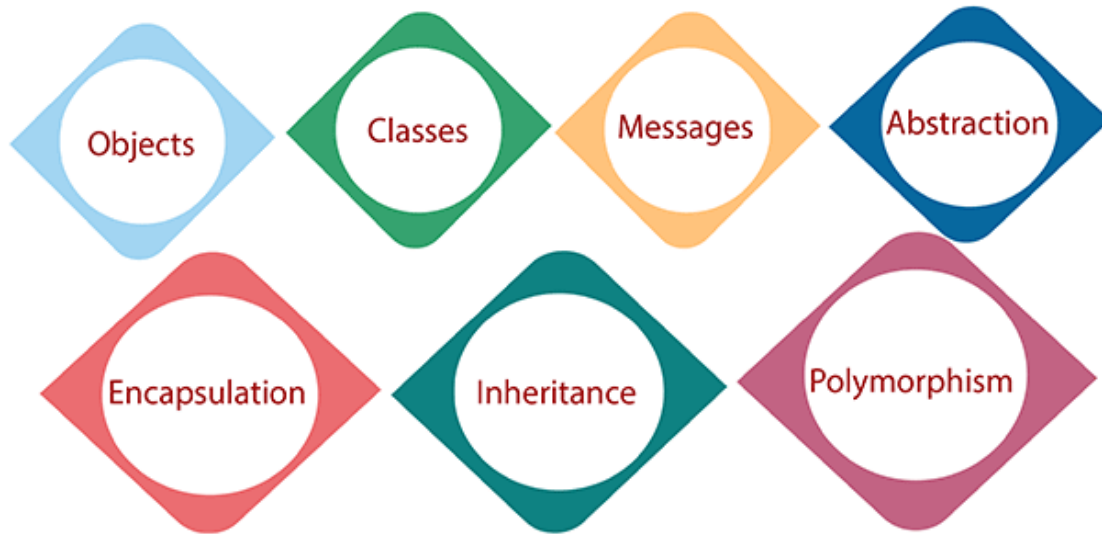
Object-Oriented Design

In the object-oriented design method, the system is viewed as a collection of objects (i.e., entities). The state is distributed among the objects, and each object handles its state data. For example, in a Library Automation Software, each library representative may be a separate object with its data and functions to operate on these data. The tasks defined for one purpose cannot refer or change data of other objects. Objects have their internal data which represent their state. Similar objects create a class. In other words, each object is a member of some class. Classes may inherit features from the superclass.

UNIT 3

The different terms related to object design are:

Object Oriented Design



1. **Objects:** All entities involved in the solution design are known as objects. For example, person, banks, company, and users are considered as objects. Every entity has some attributes associated with it and has some methods to perform on the attributes.
2. **Classes:** A class is a generalized description of an object. An object is an instance of a class. A class defines all the attributes, which an object can have and methods, which represents the functionality of the object.
3. **Messages:** Objects communicate by message passing. Messages consist of the integrity of the target object, the name of the requested operation, and any other action needed to perform the function. Messages are often implemented as procedure or function calls.
4. **Abstraction** In object-oriented design, complexity is handled using abstraction. Abstraction is the removal of the irrelevant and the amplification of the essentials.
5. **Encapsulation:** Encapsulation is also called an information hiding concept. The data and operations are linked to a single unit. Encapsulation not only bundles essential information of an object together but also restricts access to the data and methods from the outside world.
6. **Inheritance:** OOD allows similar classes to stack up in a hierarchical manner where the lower or sub-classes can import, implement, and re-use allowed variables and functions from their immediate superclasses. This

UNIT 3

property of OOD is called an inheritance. This makes it easier to define a specific class and to create generalized classes from specific ones.

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

Architectural Design:

Need of Architecture:

The architecture is not the operational software. Rather, it is a representation that enables a

software engineer to:

- (1) analyze the effectiveness of the design in meeting its stated requirements,
- (2) consider architectural alternatives at a stage when making design changes is still relatively
easy, and
- (3) reduce the risks associated with the construction of the software.

Importance :

- Software architecture representations enable communications among stakeholders
- Architecture highlights early design decisions that will have a profound impact on the
ultimate success of the system as an operational entity
- The architecture constitutes an intellectually graspable model of how the system is
structured and how its components work together

UNIT 3

Architectural Styles – 1:

- **Data centered**

- file or database lies at the center of this architecture and is accessed frequently by other components that modify data

Architectural Styles – 2:

- **Data flow**

- input data is transformed by a series of computational components into output data
- Pipe and filter pattern has a set of components called filters, connected by pipes that transmit data from one component to the next.
- If the data flow degenerates into a single line of transforms, it is termed batch sequential

- **Object-oriented**

- components of system encapsulate data and operations, communication between components is by message passing

Layered

- several layers are defined
- each layer performs operations that become closer to the machine instruction set in the lower layers

UNIT 3

Architectural Styles – 3:

Call and return

– program structure decomposes function into control hierarchy with main program

invoking several subprograms

Software Architecture Design – 1:

- Software to be developed must be put into context
- model external entities and define interfaces
- Identify architectural archetypes
- collection of abstractions that must be modeled if the system is to be constructed

Object oriented Architecture :

- The components of a system encapsulate data and the operations that must be applied to

manipulate the data. Communication and coordination between components is accomplished via message passing

Software Architecture Design – 2:

- Specify structure of the system
- define and refine the software components needed to implement each archet ype
- Continue the process iteratively until a complete architectural structure has been derived

Layered Architecture:

- Number of different layers are defined, each accomplishing operations that progressively

UNIT 3

become closer to the machine instruction set

- At the outer layer –components service user interface operations.
- At the inner layer – components perform operating system interfacing.
- Intermediate layers provide utility services and application software function

Architecture Tradeoff Analysis – 1:

1. Collect scenarios
2. Elicit requirements, constraints, and environmental description
3. Describe architectural styles/patterns chosen to address scenarios and requirements
 - module view
 - process view
 - data flow view

Architecture Tradeoff Analysis – 2:

4. Evaluate quality attributes independently (e.g. reliability, performance, security, maintainability, flexibility, testability, portability, reusability, interoperability)
5. Identify sensitivity points for architecture
 - any attributes significantly affected by changing in the architecture

Refining Architectural Design:

- Processing narrative developed for each module
- Interface description provided for each module
- Local and global data structures are defined
- Design restrictions/limitations noted

UNIT 3

- Design reviews conducted

Refinement considered if required and justified

Architectural Design

- An early stage of the system design process.
- Represents the link between specification and design processes.
- Often carried out in parallel with some specification activities.
- It involves identifying major system components and their communications.

Advantages of explicit architecture

- Stakeholder communication
 - Architecture may be used as a focus of discussion by system stakeholders.
- System analysis
 - Means that analysis of whether the system can meet its non-functional requirements is possible.
- Large-scale reuse
 - The architecture may be reusable across a range of systems.

Architecture and system characteristics

- Performance
 - Localise critical operations and minimise communications. Use large rather than finegrain components.
- Security

UNIT 3

- Use a layered architecture with critical assets in the inner layers.
- Safety
- Localise safety-critical features in a small number of sub-systems.
- Availability
- Include redundant components and mechanisms for fault tolerance.
- Maintainability
- Use fine-grain, replaceable components.

Architectural conflicts

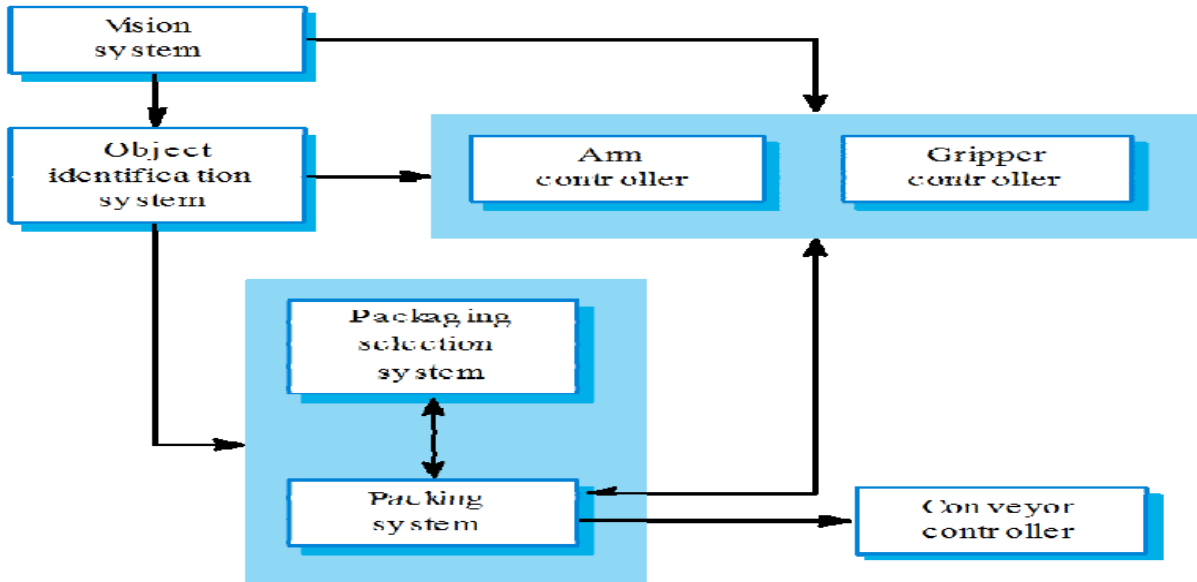
- Using large-grain components improves performance but reduces maintainability.
- Introducing redundant data improves availability but makes security more difficult.
- Localising safety-related features usually means more communication so degraded performance.

System structuring

- Concerned with decomposing the system into interacting sub-systems.
- The architectural design is normally expressed as a block diagram presenting an overview of the system structure.
- More specific models showing how sub-systems share data, are distributed and interface with each other may also be developed.

Packing robot control system

UNIT 3



Box and line diagrams

- Very abstract - they do not show the nature of component relationships nor the externally

visible properties of the sub-systems.

- However, useful for communication with stakeholders and for project planning.

Architectural design decisions

- Architectural design is a creative process so the process differs depending on the type of system being developed.

Architecture reuse

- Systems in the same domain often have similar architectures that reflect domain concepts.
- Application product lines are built around a core architecture with variants that satisfy particular customer requirements.

UNIT 3

Architectural styles

- The architectural model of a system may conform to a generic architectural model or style.
- An awareness of these styles can simplify the problem of defining system architectures.
- However, most large systems are heterogeneous and do not follow a single architectural style.

Architectural models

- Used to document an architectural design.

Static structural model that shows the major system components.

- Dynamic process model that shows the process structure of the system.
- Interface model that defines sub-system interfaces.
- Relationships model such as a data-flow model that shows sub-system relationships.
- Distribution model that shows how sub-systems are distributed across computers.

System organisation

- Reflects the basic strategy that is used to structure a system.
- Three organisational styles are widely used:
 - A shared data repository style;
 - A shared services and servers style;
 - An abstract machine or layered style.

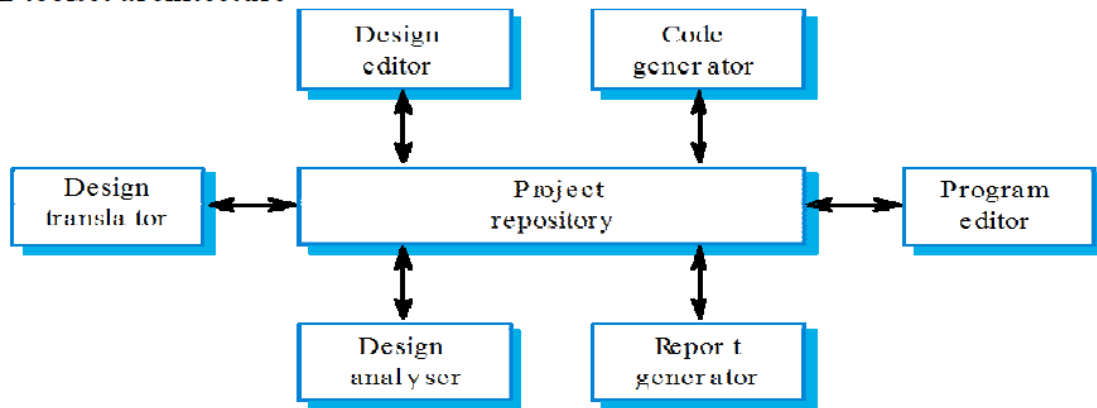
The repository model

- Sub-systems must exchange data. This may be done in two ways:

UNIT 3

- Shared data is held in a central database or repository and may be accessed by all subsystems;
- Each sub-system maintains its own database and passes data explicitly to other subsystems.
- When large amounts of data are to be shared, the repository model of sharing is most commonly used.

CASE toolset architecture



Repository model characteristics

Advantages

- Efficient way to share large amounts of data;
- Sub-systems need not be concerned with how data is produced Centralised management
e.g. backup, security, etc.
- Sharing model is published as the repository schema.

Repository model characteristics

Advantages

- Efficient way to share large amounts of data;
- Sub-systems need not be concerned with how data is produced Centralised management

UNIT 3

e.g. backup, security, etc.

- Sharing model is published as the repository schema.

Disadvantages

- Sub-systems must agree on a repository data model. Inevitably a compromise;
- Data evolution is difficult and expensive;
- No scope for specific management policies;
- Difficult to distribute efficiently.

Transform and Transcation Mapping:

Transform Mapping (Analysis)

A structure chart is produced by the conversion of a DFD diagram; this conversion is described as 'transform mapping (analysis)'. It is applied through the 'transforming' of input data flow into output data flow.

Transform analysis establishes the modules of the system, also known as the primary functional components, as well as the inputs and outputs of the identified modules in the DFD. Transform analysis is made up of a number of steps that need to be carried out. The first one is the dividing of the DFD into 3 parts:

- Input
- Logical processing
- Output

The '**input**' part of the DFD covers operations that change high level input data from physical to logical form e.g. from a keyboard input to storing the characters typed into a database. Each individual instance of an input is called an 'afferent branch'.

The '**output**' part of the DFD is similar to the 'input' part in that it acts as a conversion process. However, the conversion is concerned with the logical output of the system into a physical one e.g. text stored in a database converted into a printed version through a printer. Also, similar to the 'input', each individual instance of an output is called as 'efferent branch'. The remaining part of a DFD is called the central transform.

UNIT 3

Once the above step has been conducted, transform analysis moves onto the second step; the structure chart is established by identifying one module for the central transform, the afferent and efferent branches. These are controlled by a 'root module' which acts as an 'invoking' part of the DFD.

In order to establish the highest input and output conversions in the system, a 'bubble' is drawn out. In other words, the inputs are mapped out to their outputs until an output is found that cannot be traced back to its input. Central transforms can be classified as processes that manipulate the inputs/outputs of a system e.g. sorting input, prioritizing it or filtering data. Processes which check the inputs/outputs or attach additional information to them cannot be classified as central transforms. Inputs and outputs are represented as boxes in the first level structure chart and central transforms as single boxes.

Moving on to the third step of transform analysis, sub-functions (formed from the breaking up of high-level functional components, a process called 'factoring') are added to the structure chart. The factoring process adds sub-functions that deal with error-handling and sub-functions that determine the start and end of a process.

Transform analysis is a set of design steps that allows a DFD with transform flow characteristics to be mapped into specific architectural style. These steps are as follows:

- Step1: Review the fundamental system model
- Step2: Review and refine DFD for the SW
- Step3: Assess the DFD in order to decide the usage of transform or transaction flow.
- Step4: Identify incoming and outgoing boundaries in order to establish the transform center.
- Step5: Perform "first-level factoring".
- Step6: Perform "second-level factoring".
- Step7: Refine the first-iteration architecture using design heuristics for improved SW quality.

Transaction Mapping (Analysis)

Similar to 'transform mapping', transaction analysis makes use of DFDs diagrams to establish the various transactions involved and producing a structure chart as a result.

Transaction mapping Steps:

- Review the fundamental system model
- Review and refine DFD for the SW

UNIT 3

- Assess the DFD in order to decide the usage of transform or transaction flow.
- Identify the transaction center and the flow characteristics along each action path
- Find transaction center
- Identify incoming path and isolate action paths
- Evaluate each action path for transform vs. transaction characteristics
- Map the DFD in a program structure agreeable to transaction processing
- Carry out the 'factoring' process
- Refine the first iteration program/ architecture using design heuristics for improved SW quality

In transaction mapping a single data item triggers one of a number of information flows that affect a function implied by the triggering data item. The data item implies a transaction.

User Interface Design

The visual part of a computer application or operating system through which a client interacts with a computer or software. It determines how commands are given to the computer or the program and how data is displayed on the screen.

Types of User Interface

There are two main types of User Interface:

- Text-Based User Interface or Command Line Interface
- Graphical User Interface (GUI)

Text-Based User Interface: This method relies primarily on the keyboard. A typical example of this is UNIX.

Advantages

- Many and easier to customizations options.
- Typically capable of more important tasks.

Disadvantages

- Relies heavily on recall rather than recognition.
- Navigation is often more difficult.

UNIT 3

Graphical User Interface (GUI): GUI relies much more heavily on the mouse. A typical example of this type of interface is any versions of the Windows operating systems.

GUI Characteristics

Characteristics	Descriptions
Windows	Multiple windows allow different information to be displayed simultaneously on the user's screen.
Icons	Icons different types of information. On some systems, icons represent files. On other icons describes processes.
Menus	Commands are selected from a menu rather than typed in a command language.
Pointing	A pointing device such as a mouse is used for selecting choices from a menu or indicating items of interests in a window.
Graphics	Graphics elements can be mixed with text or the same display.

Advantages

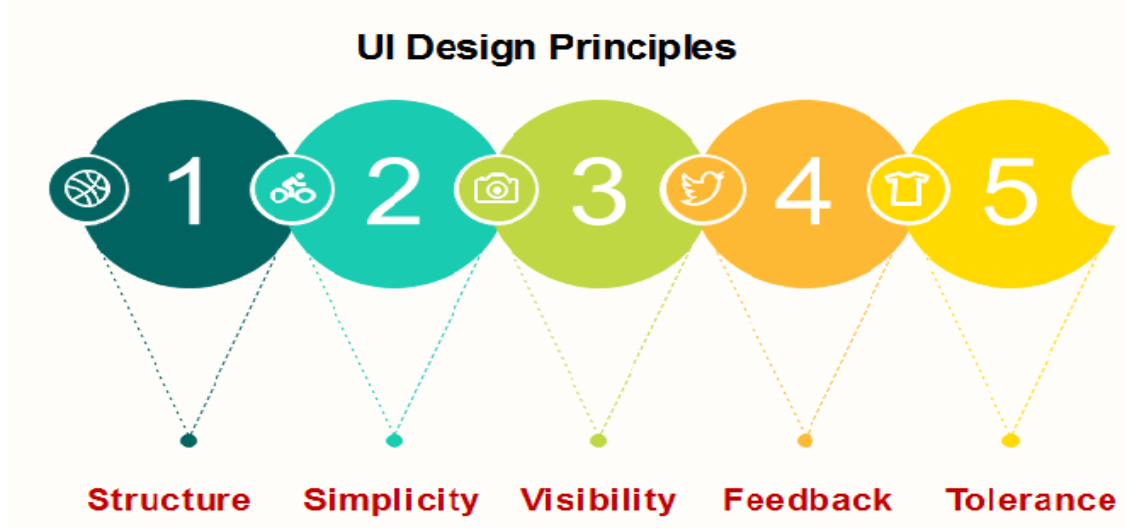
- Less expert knowledge is required to use it.
- Easier to Navigate and can look through folders quickly in a guess and check manner.
- The user may switch quickly from one task to another and can interact with several different applications.

Disadvantages

- Typically decreased options.
- Usually less customizable. Not easy to use one button for tons of different variations.

UNIT 3

UI Design Principles



Structure: Design should organize the user interface purposefully, in the meaningful and usual based on precise, consistent models that are apparent and recognizable to users, putting related things together and separating unrelated things, differentiating dissimilar things and making similar things resemble one another. The structure principle is concerned with overall user interface architecture.

Simplicity: The design should make the simple, common task easy, communicating clearly and directly in the user's language, and providing good shortcuts that are meaningfully related to longer procedures.

Visibility: The design should make all required options and materials for a given function visible without distracting the user with extraneous or redundant data.

Feedback: The design should keep users informed of actions or interpretation, changes of state or condition, and bugs or exceptions that are relevant and of interest to the user through clear, concise, and unambiguous language familiar to users.

Tolerance: The design should be flexible and tolerant, decreasing the cost of errors and misuse by allowing undoing and redoing while also preventing bugs wherever possible by tolerating varied inputs and sequences and by interpreting all reasonable actions.

UNIT 3