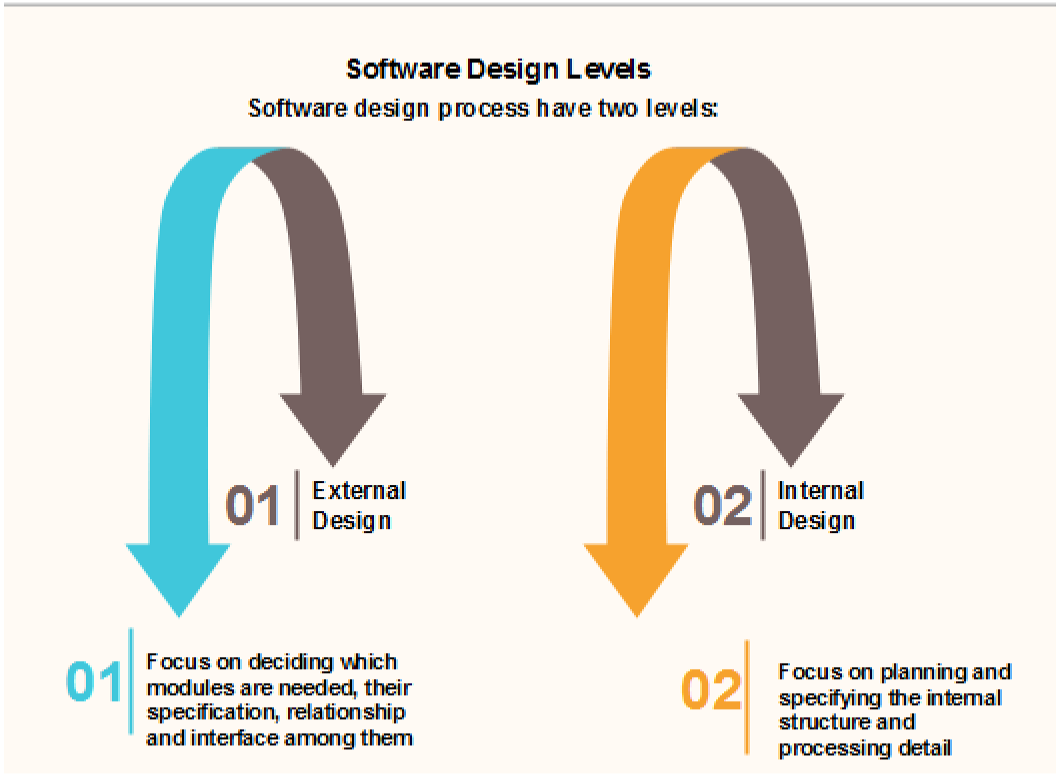
Unit-1

Topics covered: *Software Design - Software Design Fundamentals, Design Standards - Design Type, Design model – Architectural design, Software architecture, Software Design Methods, Top Down , Bottom Up, Module Division (Refactoring), Module Coupling, Component level design, User Interface Design, Pattern oriented design, Design Reuse, Concurrent Engineering in Software Design.*

1. Software Design

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.


Software Design Levels
Software design process have two levels:
01 External Design
02 Internal Design
01 Focus on deciding which modules are needed, their specification, relationship and interface among them
02 Focus on planning and specifying the internal structure and processing detail

Objectives of Software Design

Following are the purposes of Software design:

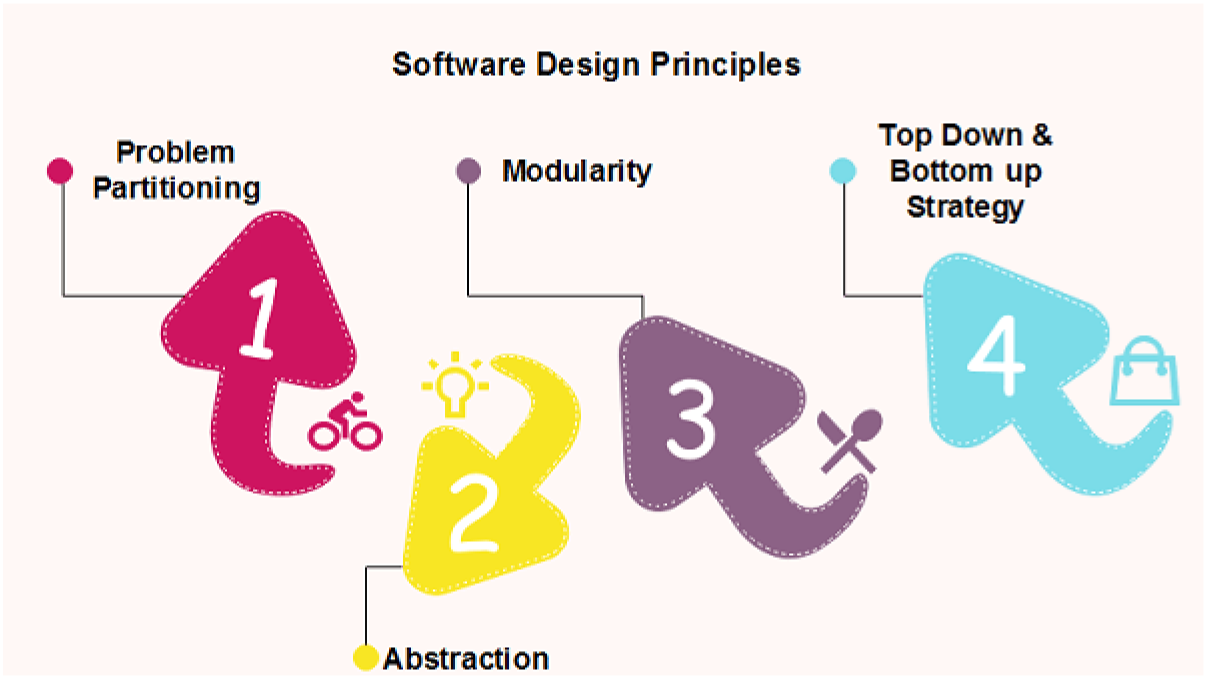
Correctness
Completeness
Efficiency
Flexibility
Consistency
Maintainability

Objectives of Software Design

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

2. Software Design Fundamentals/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.



- **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

  - Software is easy to understand
  - Software becomes simple
  - Software is easy to test
  - Software is easy to modify
  - Software is easy to maintain
  - 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

- **Functional Abstraction**
- **Data Abstraction**

Functional Abstraction

- A module is specified by the method it performs.
- 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**.

- **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.

- Advantages and Disadvantages of Modularity
- In this topic, we will discuss various advantage and disadvantage 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

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:

a. **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.

b. **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.

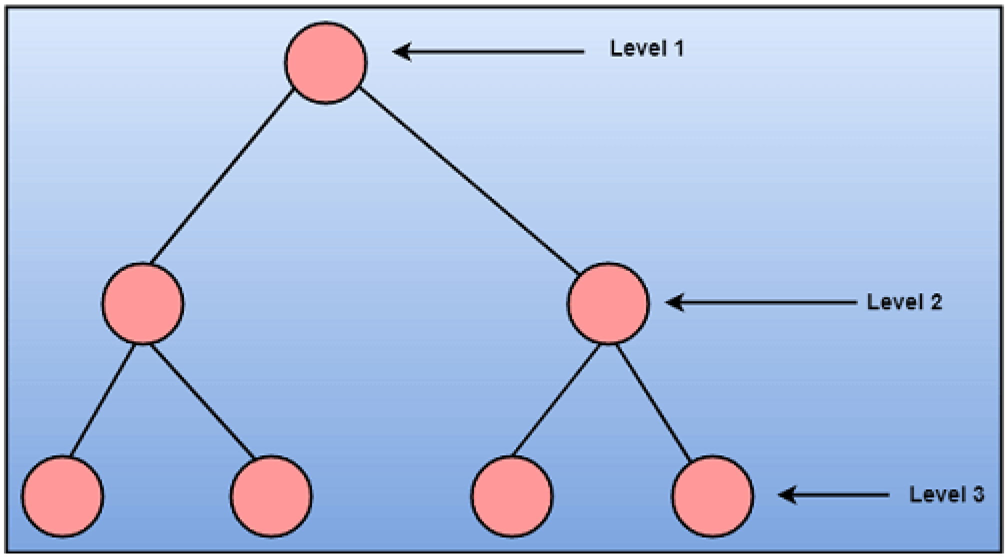
4. Strategy of Design

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.

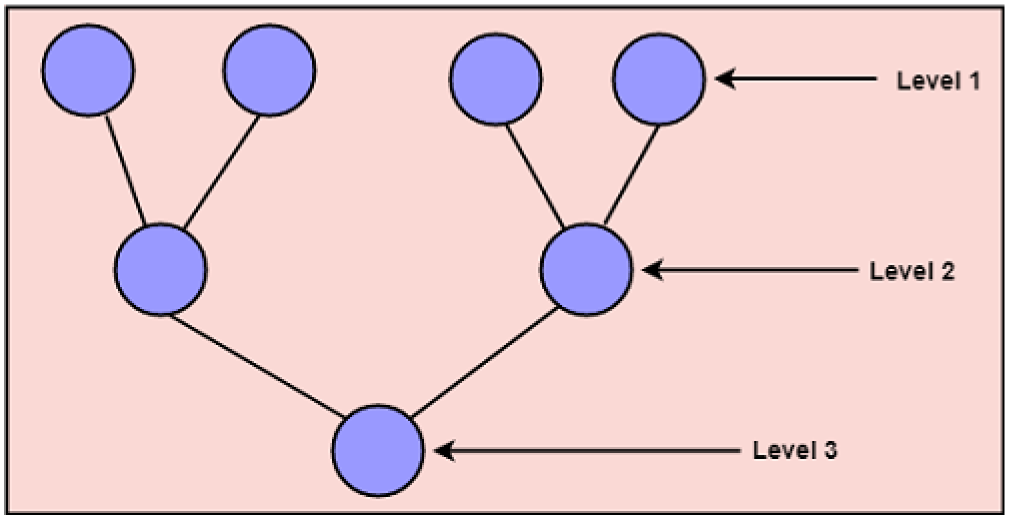
To design a system, there are two possible approaches:

- **Top-down Approach**
- **Bottom-up Approach**

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



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.

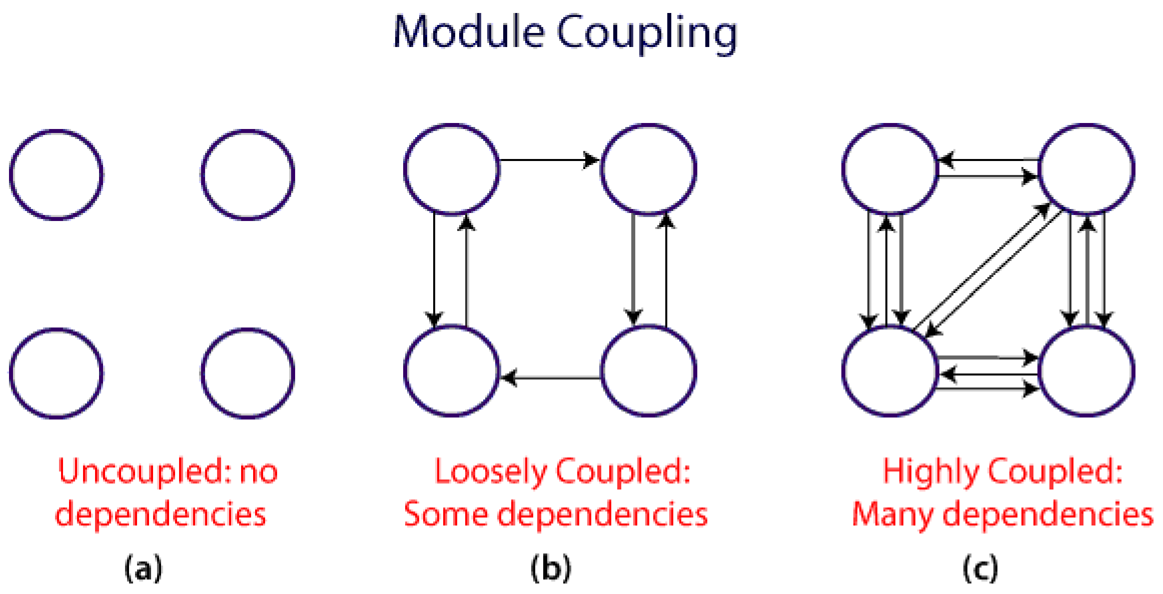


5. Coupling and Cohesion

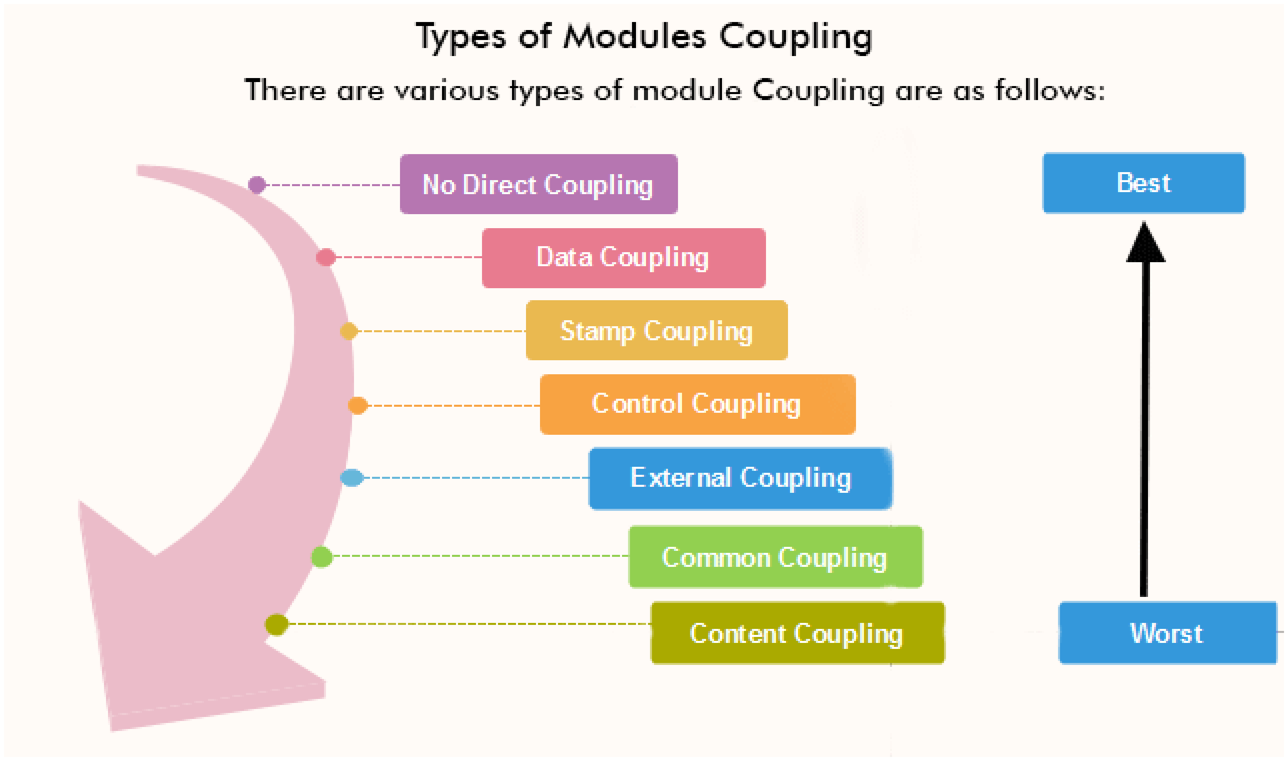
Module Coupling

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.

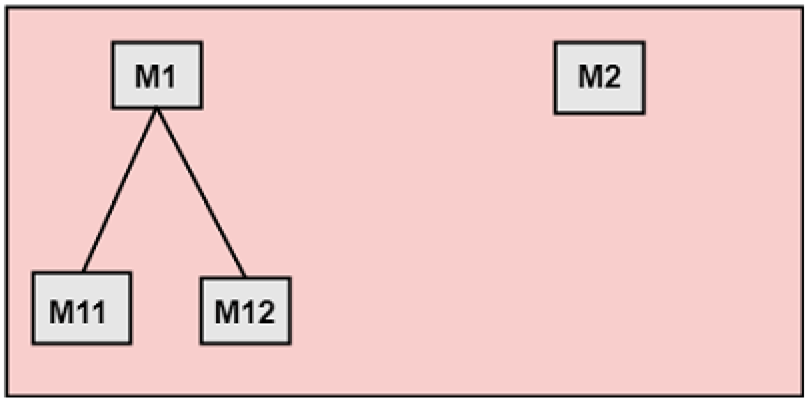
The various types of coupling techniques are shown in fig:


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.

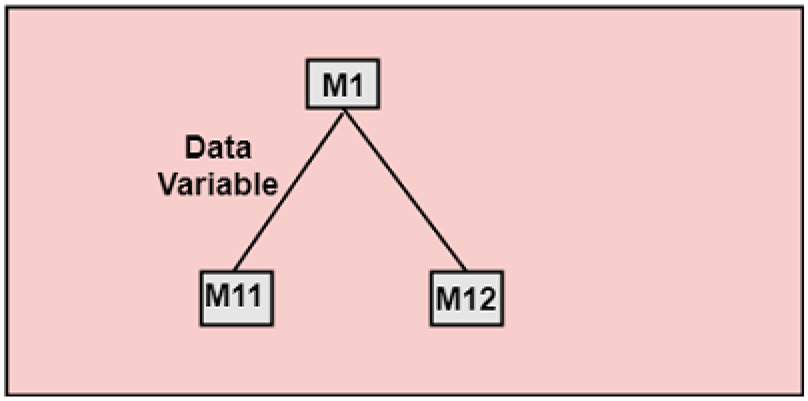


- **No Direct Coupling:** There is no direct coupling between M1 and M2.

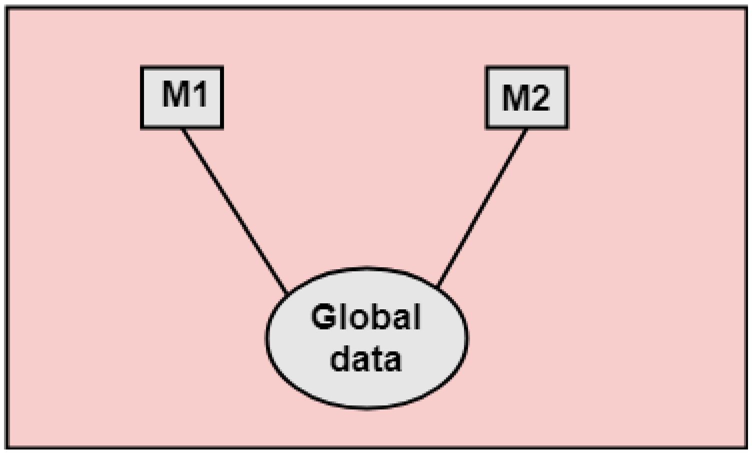


In this case, modules are subordinates to different modules. Therefore, no direct coupling is there.

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



- **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.
- **Control Coupling:** Control Coupling exists among two modules if data from one module is used to direct the structure of instruction execution in another.
- **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.
- **Common Coupling:** Two modules are common coupled if they share information through some global data items.

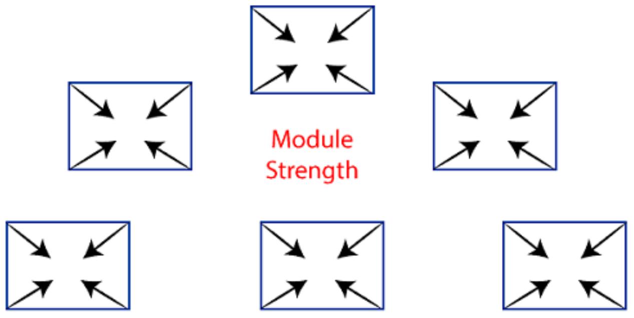


- **Content Coupling:** Content Coupling exists among two modules if they share code, e.g., a branch from one module into another module.

- **Module Cohesion**

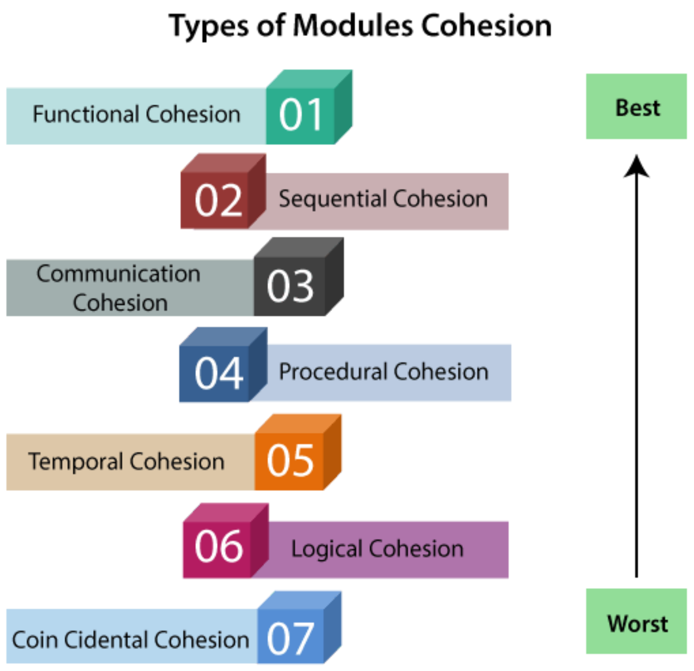
In computer programming, cohesion defines to the degree to which the elements of a module belong together. Thus, cohesion measures the strength of relationships between pieces of functionality within a given module. For example, in highly cohesive systems, functionality is strongly related.

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



Cohesion= Strength of relations within Modules

- **Types of Modules Cohesion**



- **Functional Cohesion:** Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.
- **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.
- **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.
- **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.
- **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.
- **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.
- **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	Cohesion shows the module's relative **functional** strength.

modules.	
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.

6. Design model – Architectural design

A design model in software engineering is **an object-based picture or pictures that represent the use cases for a system**. Or to put it another way, it's the means to describe a system's implementation and source code in a diagrammatic fashion.

- **Architectural design**

The software needs the architectural design to represents the design of software. IEEE defines architectural design as "the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system." The software that is built for computer-based systems can exhibit one of these many architectural styles. Each style will describe a system category that consists of:

- A set of components (eg: a database, computational modules) that will perform a function required by the system.
- The set of connectors will help in coordination, communication, and cooperation between the components.
- Conditions that how components can be integrated to form the system.
- Semantic models that help the designer to understand the overall properties of the system.
- The use of architectural styles is to establish a structure for all the components of the system.

Taxonomy of Architectural styles:

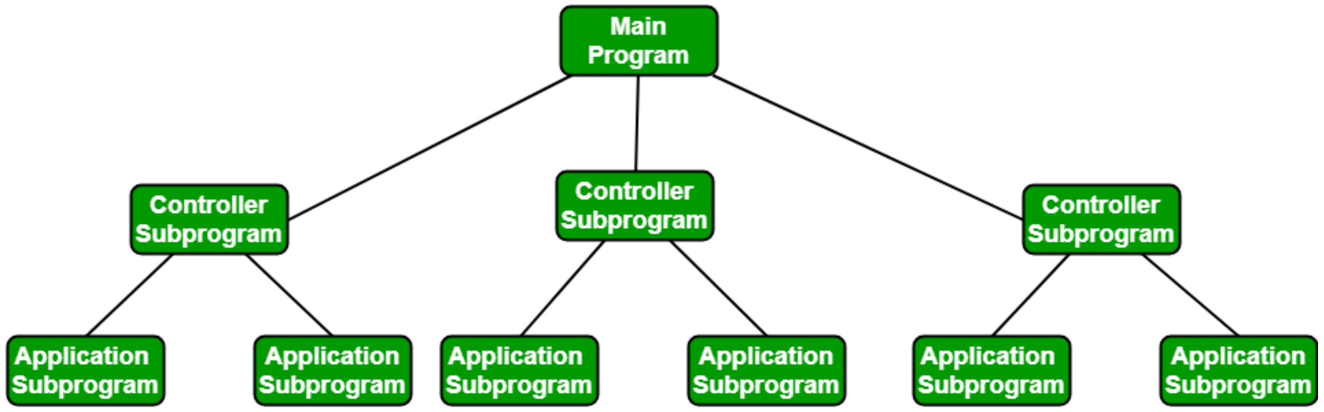
- **Data centered architectures:** A data store will reside at the center of this architecture and is accessed frequently by the other components that update, add, delete or modify the data present within the store.

- The client software access a central repository. Variations of this approach are used to transform the repository into a blackboard when data related to client or data of interest for the client change the notifications to client software.
- This data-centered architecture will promote integrability. This means that the existing components can be changed and new client components can be added to the architecture without the permission or concern of other clients.
- Data can be passed among clients using blackboard mechanism.

- **Data flow architectures:**

- This kind of architecture is used when input data to be transformed into output data through a series of computational manipulative components.
- The figure represents pipe-and-filter architecture since it uses both pipe and filter and it has a set of components called filters connected by pipes.
- Pipes are used to transmit data from one component to the next.
- Each filter will work independently and is designed to take data input of a certain form and produces data output to the next filter of a specified form. The filters don't require any knowledge of the working of neighboring filters.
- If the data flow degenerates into a single line of transforms, then it is termed as batch sequential. This structure accepts the batch of data and then applies a series of sequential components to transform it.

- **Call and Return architectures:** It is used to create a program that is easy to scale and modify. Many sub-styles exist within this category. Two of them are explained below.
- **Remote procedure call architecture:** This component is used to present in a main program or sub program architecture distributed among multiple computers on a network.
- **Main program or Subprogram architectures:** The main program structure decomposes into number of subprograms or function into a control hierarchy. Main program contains number of subprograms that can invoke other components.



- **Object Oriented architecture:** The components of a system encapsulate data and the operations that must be applied to manipulate the data. The coordination and communication between the components are established via the message passing.
- **Layered architecture:** A number of different layers are defined with each layer performing a well-defined set of operations.
  - Each layer will do some operations that become closer to machine instruction set progressively.
  - At the outer layer, components will receive the user interface operations and at the inner layers, components will perform the operating system interfacing(communication and coordination with OS)
  - Intermediate layers to utility services and application software functions.

7. Component level design

- Component-based architecture focuses on the decomposition of the design into individual functional or logical components that represent well-defined communication interfaces containing methods, events, and properties.
- It provides a higher level of abstraction and divides the problem into sub-problems, each associated with component partitions.
- The primary objective of component-based architecture is to ensure **component reusability**.
- A component encapsulates functionality and behaviors of a software element into a reusable and self-deployable binary unit.
- There are many standard component frameworks such as COM/DCOM, JavaBean, EJB, CORBA, .NET, web services, and grid services. These technologies are widely used in local desktop GUI application design such as graphic JavaBean components, MS ActiveX components, and COM components which can be reused by simply drag and drop operation.
- Component-oriented software design has many advantages over the traditional object-oriented approaches such as –
  - ❖ Reduced time in market and the development cost by reusing existing components.
  - ❖ Increased reliability with the reuse of the existing components.

What is a Component?

- A component is a modular, portable, replaceable, and reusable set of well-defined functionality that encapsulates its implementation and exporting it as a higher-level interface.

- A component is a software object, intended to interact with other components, encapsulating certain functionality or a set of functionalities. It has an obviously defined interface and conforms to a recommended behavior common to all components within an architecture.
- A software component can be defined as a unit of composition with a contractually specified interface and explicit context dependencies only. That is, a software component can be deployed independently and is subject to composition by third parties.

- **Views of a Component**

A component can have three different views − object-oriented view, conventional view, and process-related view.

- **Object-oriented view**

  A component is viewed as a set of one or more cooperating classes. Each problem domain class (analysis) and infrastructure class (design) are explained to identify all attributes and operations that apply to its implementation. It also involves defining the interfaces that enable classes to communicate and cooperate.

- **Conventional view**

  It is viewed as a functional element or a module of a program that integrates the processing logic, the internal data structures that are required to implement the processing logic and an interface that enables the component to be invoked and data to be passed to it.

- **Process-related view**

- In this view, instead of creating each component from scratch, the system is building from existing components maintained in a library. As the software architecture is formulated, components are selected from the library and used to populate the architecture.
- A user interface (UI) component includes grids, buttons referred as controls, and utility components expose a specific subset of functions used in other components.
- Other common types of components are those that are resource intensive, not frequently accessed, and must be activated using the just-in-time (JIT) approach.
- Many components are invisible which are distributed in enterprise business applications and internet web applications such as Enterprise JavaBean (EJB), .NET components, and CORBA components.

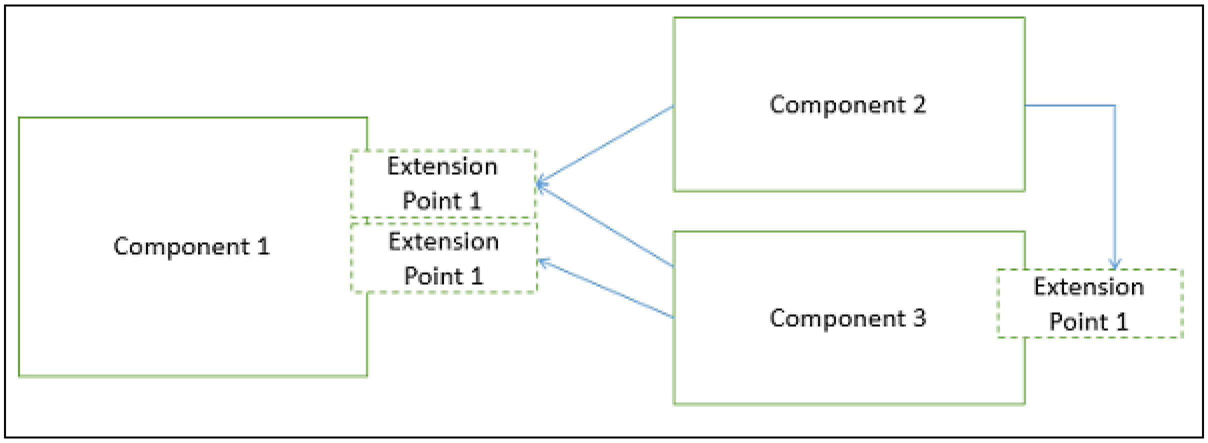
- **Characteristics of Components**

- **Reusability** – Components are usually designed to be reused in different situations in different applications. However, some components may be designed for a specific task.
- **Replaceable** – Components may be freely substituted with other similar components.
- **Not context specific** – Components are designed to operate in different environments and contexts.
- **Extensible** – A component can be extended from existing components to provide new behavior.
- **Encapsulated** – A A component depicts the interfaces, which allow the caller to use its functionality, and do not expose details of the internal processes or any internal variables or state.
- **Independent** – Components are designed to have minimal dependencies on other components.

- **Principles of Component–Based Design**

A component-level design can be represented by using some intermediary representation (e.g. graphical, tabular, or text-based) that can be translated into source code. The design of data structures, interfaces, and algorithms should conform to well-established guidelines to help us avoid the introduction of errors.

- The software system is decomposed into reusable, cohesive, and encapsulated component units.
- Each component has its own interface that specifies required ports and provided ports; each component hides its detailed implementation.
- A component should be extended without the need to make internal code or design modifications to the existing parts of the component.
- Depend on abstractions component do not depend on other concrete components, which increase difficulty in expendability.
- Connectors connected components, specifying and ruling the interaction among components. The interaction type is specified by the interfaces of the components.
- Components interaction can take the form of method invocations, asynchronous invocations, broadcasting, message driven interactions, data stream communications, and other protocol specific interactions.
- For a server class, specialized interfaces should be created to serve major categories of clients. Only those operations that are relevant to a particular category of clients should be specified in the interface.
- A component can extend to other components and still offer its own extension points. It is the concept of plug-in based architecture. This allows a plugin to offer another plugin API.



- **Component-Level Design Guidelines**

Creates a naming convention for components that are specified as part of the architectural model and then refines or elaborates as part of the component-level model.

- Attains architectural component names from the problem domain and ensures that they have meaning to all stakeholders who view the architectural model.
- Extracts the business process entities that can exist independently without any associated dependency on other entities.
- Recognizes and discover these independent entities as new components.
- Uses infrastructure component names that reflect their implementation-specific meaning.
- Models any dependencies from left to right and inheritance from top (base class) to bottom (derived classes).
- Model any component dependencies as interfaces rather than representing them as a direct component-to-component dependency.

- **Conducting Component-Level Design**

Recognizes all design classes that correspond to the problem domain as defined in the analysis model and architectural model.

- Recognizes all design classes that correspond to the infrastructure domain.
- Describes all design classes that are not acquired as reusable components, and specifies message details.
- Identifies appropriate interfaces for each component and elaborates attributes and defines data types and data structures required to implement them.
- Describes processing flow within each operation in detail by means of pseudo code or UML activity diagrams.

- Describes persistent data sources (databases and files) and identifies the classes required to manage them.
- Develop and elaborates behavioral representations for a class or component. This can be done by elaborating the UML state diagrams created for the analysis model and by examining all use cases that are relevant to the design class.
- Elaborates deployment diagrams to provide additional implementation detail.
- Demonstrates the location of key packages or classes of components in a system by using class instances and designating specific hardware and operating system environment.
- The final decision can be made by using established design principles and guidelines. Experienced designers consider all (or most) of the alternative design solutions before settling on the final design model.

- **Advantages**

- **Ease of deployment –** as new compatible versions become available, it is easier to replace existing versions with no impact on the other components or the system as a whole.
- **Reduced cost –** the use of third-party components allows you to spread the cost of development and maintenance.
- **Ease of development –** Components implement well-known interfaces to provide defined functionality, allowing development without impacting other parts of the system.
- **Reusable –** the use of reusable components means that they can be used to spread the development and maintenance cost across several applications or systems.
- **Modification of technical complexity –** A component modifies the complexity through the use of a component container and its services.
- **Reliability –** the overall system reliability increases since the reliability of each individual component enhances the reliability of the whole system via reuse.
- **System maintenance and evolution –** Easy to change and update the implementation without affecting the rest of the system.
- **Independent –** Independency and flexible connectivity of components. Independent development of components by different group in parallel. Productivity for the software development and future software development.

8. User Interface Design

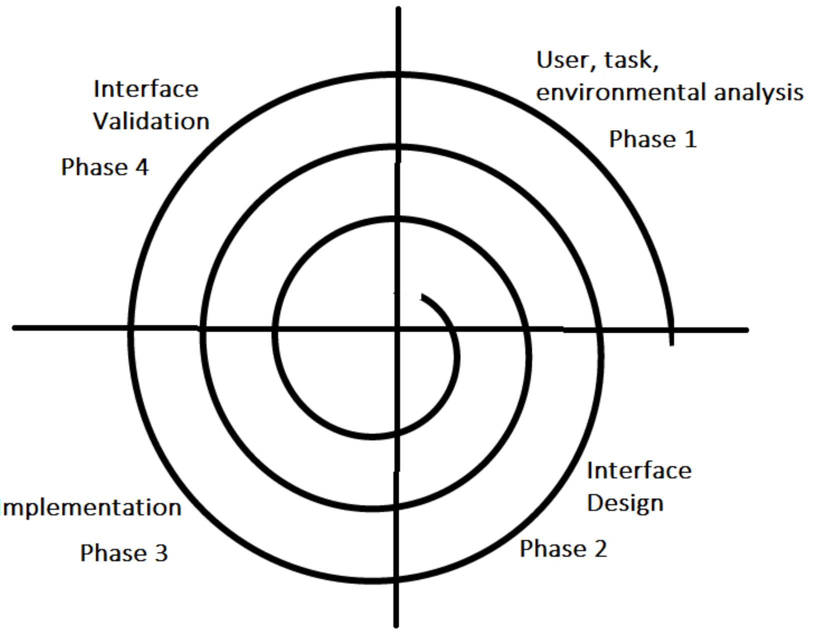
User interface is the front-end application view to which user interacts in order to use the software. The software becomes more popular if its user interface is:

- Attractive
- Simple to use
- Responsive in short time
- Clear to understand
- Consistent on all interface screens

There are two types of User Interface:

I. **Command Line Interface:** Command Line Interface provides a command prompt, where the user types the command and feeds to the system. The user needs to remember the syntax of the command and its use.
II. **Graphical User Interface:** Graphical User Interface provides the simple interactive interface to interact with the system. GUI can be a combination of both hardware and software. Using GUI, user interprets the software.

User Interface Design Process:



The analysis and design process of a user interface is iterative and can be represented by a spiral model. The analysis and design process of user interface consists of four framework activities.

I. **User, task, environmental analysis, and modeling:** Initially, the focus is based on the profile of users who will interact with the system, i.e. understanding, skill and knowledge, type of user, etc, based on the user's profile users are made into categories. From each category requirements are gathered. Based on the requirements developer understand how to develop the interface. Once all the requirements are gathered a detailed analysis is conducted. In the analysis part, the tasks that the user performs to establish the goals of the system are identified, described and elaborated. The analysis of the user environment focuses on the physical work environment. Among the questions to be asked are:

   a. Where will the interface be located physically?
   b. Will the user be sitting, standing, or performing other tasks unrelated to the interface?
   c. Does the interface hardware accommodate space, light, or noise constraints?
   d. Are there special human factors considerations driven by environmental factors?

II. **Interface Design:** The goal of this phase is to define the set of interface objects and actions i.e. Control mechanisms that enable the user to perform desired tasks. Indicate how these control mechanisms affect the system. Specify the action sequence of tasks and subtasks, also called a user scenario. Indicate the state of the system when the user performs a particular task. Always follow the three golden rules stated by Theo Mandel. Design issues such as response time, command and action structure, error handling, and help facilities are considered as the design model is refined. This phase serves as the foundation for the implementation phase.

III. **Interface construction and implementation:** The implementation activity begins with the creation of prototype (model) that enables usage scenarios to be evaluated. As iterative design process continues a User Interface toolkit that allows the creation of windows, menus, device interaction, error messages, commands, and many other elements of an interactive environment can be used for completing the construction of an interface.

IV. **Interface Validation:** This phase focuses on testing the interface. The interface should be in such a way that it should be able to perform tasks correctly and it should be able to handle a variety of tasks. It should achieve all the user's requirements. It should be easy to use and easy to learn. Users should accept the interface as a useful one in their work.

The following are the golden rules stated by Theo Mandel that must be followed during the design of the interface.

Place the user in control:

- Define the interaction modes in such a way that does not force the user into unnecessary or undesired actions: The user should be able to easily enter and exit the mode with little or no effort.
- Provide for flexible interaction: Different people will use different interaction mechanisms, some might use keyboard commands, some might use mouse, some might use touch screen, etc, and hence all interaction mechanisms should be provided.
- Allow user interaction to be interruptible and undoable: When a user is doing a sequence of actions the user must be able to interrupt the sequence to do some other work without losing the work that had been done. The user should also be able to do undo operation.

- Streamline interaction as skill level advances and allow the interaction to be customized: Advanced or highly skilled user should be provided a chance to customize the interface as user wants which allows different interaction mechanisms so that user doesn't feel bored while using the same interaction mechanism.
- Hide technical internals from casual users: The user should not be aware of the internal technical details of the system. He should interact with the interface just to do his work.
- Design for direct interaction with objects that appear on screen: The user should be able to use the objects and manipulate the objects that are present on the screen to perform a necessary task. By this, the user feels easy to control over the screen.

Reduce the user's memory load:

- Reduce demand on short-term memory: When users are involved in some complex tasks the demand on short-term memory is significant. So the interface should be designed in such a way to reduce the remembering of previously done actions, given inputs and results.
- Establish meaningful defaults: Always initial set of defaults should be provided to the average user, if a user needs to add some new features then he should be able to add the required features.
- Define shortcuts that are intuitive: Mnemonics should be used by the user. Mnemonics means the keyboard shortcuts to do some action on the screen.
- The visual layout of the interface should be based on a real-world metaphor: Anything you represent on a screen if it is a metaphor for real-world entity then users would easily understand.
- Disclose information in a progressive fashion: The interface should be organized hierarchically i.e. on the main screen the information about the task, object or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick.

Make the interface consistent:

- Allow the user to put the current task into a meaningful context: Many interfaces have dozens of screens. So it is important to provide indicators consistently so that the user know about the doing work. The user should also know from which page has navigated to the current page and from the current page where can navigate.
- Maintain consistency across a family of applications: The development of some set of applications all should follow and implement the same design, rules so that consistency is maintained among applications.
- If past interactive models have created user expectations do not make changes unless there is a compelling reason.

9. Pattern oriented design

Design patterns are used to represent some of the best practices adapted by experienced object-oriented software developers. A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples.

Usage of Design Pattern

Design Patterns have two main usages in software development.

- **Common platform for developers**

  Design patterns provide a standard terminology and are specific to particular scenario. For example, a singleton design pattern signifies use of single object so all developers familiar with single design pattern will make use of single object and they can tell each other that program is following a singleton pattern.

- **Best Practices**

  Design patterns have been evolved over a long period of time and they provide best solutions to certain problems faced during software development. Learning these patterns helps inexperienced developers to learn software design in an easy and faster way.

- **Types of Design Patterns**

  As per the design pattern reference book **Design Patterns - Elements of Reusable Object-Oriented Software**, there are 23 design patterns which can be classified in three categories: Creational, Structural and Behavioral patterns. We'll also discuss another category of design pattern: J2EE design patterns.

S.N.	Pattern & Description
1	**Creational Patterns** These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.
2	**Structural Patterns** These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.
3	**Behavioral Patterns** These design patterns are specifically concerned with communication between objects.

4	**J2EE Patterns** These design patterns are specifically concerned with the presentation tier. These patterns are identified by Sun Java Center.

10. Design Reuse

- ○ Design reuse is the process of building new software applications and tools by reusing previously developed designs. New features and functionalities may be added by incorporating minor changes.
- ○ Design reuse involves the use of designed modules, such as logic and data, to build a new and improved product. The reusable components, including code segments, structures, plans and reports, minimize implementation time and are less expensive. This avoids reinventing existing software by using techniques already developed and to create and test the software.

- Design reuse involves many activities utilizing existing technologies to cater to new design needs. The ultimate goal of design reuse is to help the developers create better products maximizing it's value with minimal resources, cost and effort.

  Today, it is almost impossible to develop an entire product from scratch. Reuse of design becomes necessary to maintain continuity and connectivity. In the software field, the reuse of the modules and data helps save implementation time and increases the possibility of eliminating errors due to prior testing and use.

  Design reuse requires that a set of designed products already exist and the design information pertaining to the product is accessible. Large software companies usually have a range of designed products. Hence the reuse of design facilitates making new and better software products. Many software companies have incorporated design reuse and have seen considerable success. The effectiveness of design reuse is measured in terms of production, time, cost and quality of the product. These key factors determine whether a company has been successful in making design reuse a solution to its new software needs and demands. With proper use of existing technology and resources, a company can benefit in terms of cost, time, performance and product quality.

  A proper process requires an intensive design reuse process model. There are two interrelated process methodologies involved in the systematic design reuse process model.

  The data reuse process is as follows:

a) Gathering Information: This involves the collection of information, processing and modeling to fetch related data.
b) Information Reuse: This involves the effective use of data.

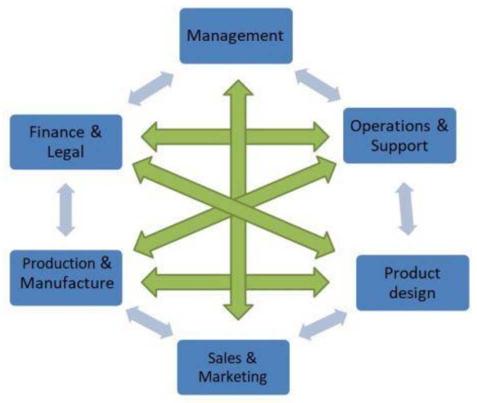
The design reuse process has four major issues:

a) Retrieve
b) Reuse
c) Repair
d) Recover

These are generally referred to as the four Rs. In spite of these challenges, companies have used the design reuse concept as a successfully implemented concept in the software field at different levels, ranging from low level code reuse to high level project reuse.

11. Concurrent Engineering in Software Design

Concurrent engineering is a method of designing and developing engineering products, in which different departments work on the different stages of engineering product development simultaneously. If managed well, it helps to increase the efficiency of **product development** and **marketing** considerably reducing the time and contributing to the reduction of the **overall development cost** while improving the final product **quality**.



Concurrent Engineering interactions

- This **streamlined** approach towards an engineering product forces several teams such as **product design**, **manufacturing**, **production, marketing**, product **suppor**t, **finance,** etc., within the organization to work simultaneously on new product development.

- For instance, while engineering product designers begin to design the product, the sales team can start working on the marketing and the product support department can start thinking about the after-sale support. While the mechanical designers work on the packaging design to incorporate the PCB being developed by the electrical engineering team, the software engineers can start looking at the software code.

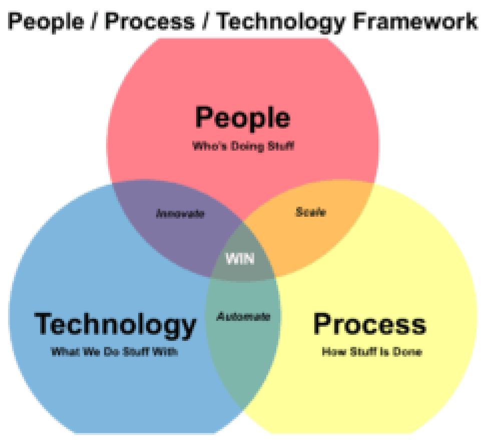
- **Concurrent engineering,** also known as **integrated product development (IPD)** or **simultaneous engineering** was introduced a few decades ago to eliminate the issues from sequential engineering or the so-called "over the wall" process**.** This systematic approach is intended to force all the stakeholders to be involved and the full engineering product cycle to be considered from concept to after-sale support. There are plenty of incentives to choose Concurrent engineering over sequential engineering product development.

- The popularity of **integrated product development** has been growing recently, thanks to the ever-increasing demand for quality products expeditiously at affordable prices.

- Although managing a **simultaneous engineering** process is very challenging, the techniques and practices followed as part of concurrent engineering benefit from several competitive advantages to the company and to the final engineering product itself.

- **Elements of concurrent engineering**

Concurrent engineering presents an environment that encourages and improves the interaction of different disciplines and departments towards a single goal of satisfying engineering product requirements. Key elements of **concurrent engineering** can be summarized using a **PPT framework** or the **Golden Triangle.**



People, process & technology framework

People, processes, and technology are crucial to any organization and essential in implementing concurrent engineering to achieve shorter development time, lower cost, improved product quality and fulfill customer needs.

- **People**

Concurrent product development is a multidisciplinary team task and it's necessary that companies utilize the right skilled personnel at the right time to accelerate product development. It is also necessary to find people with the right skills and experience along with the following key aspects;

- Multidisciplinary team to suit the product at the start of the NPD
- Teamwork culture at the core of the program

Good communication and collaboration between teams – sharing relevant and up to date information across departments and personnel

The harmonized goal across the company from the top management to the bottom of the organizational structure

- **Process**

A process is a series of product development steps that need to happen to achieve a goal. These can be project planning stages, milestone management, problem-solving methodologies, product development key stages, information sharing workflow, etc., as just people are ineffective without processes in place to support their tasks and decisions. Following are some of the processes that can be adopted in concurrent engineering;

  - Project planning processes and workflow management which include key new product development elements such as key design stages, milestones for cross-departmental interaction, etc.
  - Workflow for product data management such as sharing information, managing engineering change, control specification creep, etc.
  - Product requirement tracking and checkpoints using techniques such as Quality Function Deployment (QFD) across departments
  - Design evaluation workflow processes
  - Design analysis methodologies such as brainstorming
  - Failure Mode and Effects Analysis (FMEA) allows for a systematic investigation of the occurrence and impact of possible flaws in the new product design
  - The use of Design of Experiments (DOE) enables the systematic identification of critical product/process parameters that influence performance

- **Technology**

For concurrent engineering to be successful, the effective introduction of tools, techniques, and technologies to aid a smooth integration of people and processes is vital. Following key aspects should be considered before any implementation.

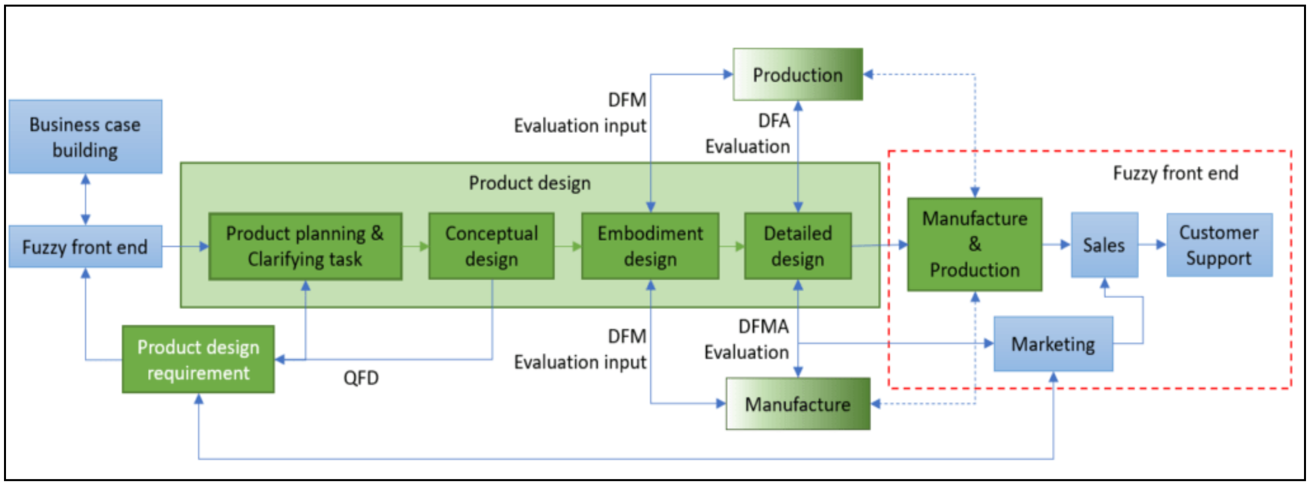
  - Identifying the correct tools and technologies that suit the company size, number of team members, processes implemented and product type
  - Identifying the training needs and training people to use the tools and technologies identified above

These are just a few of the many supportive tools that can be used in a concurrent engineering environment.

- Project management software
- Product data management & product lifecycle management suites
- Quality Function Deployment (QFD)
- 3D CAD and rapid prototyping technologies such as additive manufacturing
- Suitable FEA tools
- Evaluation tools such as DFM, DFA, DFMA and DOE
- Failure mode analysis tools such as FMEA

- **Concurrent new product development**

By concurrently engaging in multiple aspects of design and development phases across the PPT framework, a new product cycle can be decreased significantly. Figure 3 shows a typical new product development cycle with some examples of concurrent activities and functions discussed above.



Typical concurrent product development

Examples of concurrent new product development activities:

- Management consults experts from different disciplines to jointly define a product design specification (PDS). Tools such as QFD can be used to track the product requirement across departments during the initial product development stages.
- During the embodiment design stage, manufacturing is consulted to evaluate the manufacturability of the design by using tools such as DFM, QFD and DFMA.

- Production is consulted to evaluate the design for assembly, which would flag up issues with an assembly including the requirement of tools and jigs. Tools such as design for assembly (DFA) can be used to analyse the design.
- Sharing design information with production at the detailed design stages of the process would enable them to get the tools and production jigs ready for production.
- At the final stages of the design, the design team shares information such as final specifications with sales and marketing teams enabling them to prepare datasheets, brochures, package design, promotional events, etc.
- Sharing the latest information across the team in a controlled central manner is crucial at all stages.