

# Unit 3

# Software Engineering

Prepared By

**Abhishek Kesharwani**

**Assistant Professor ,United College of Engineering and Research**

# Index

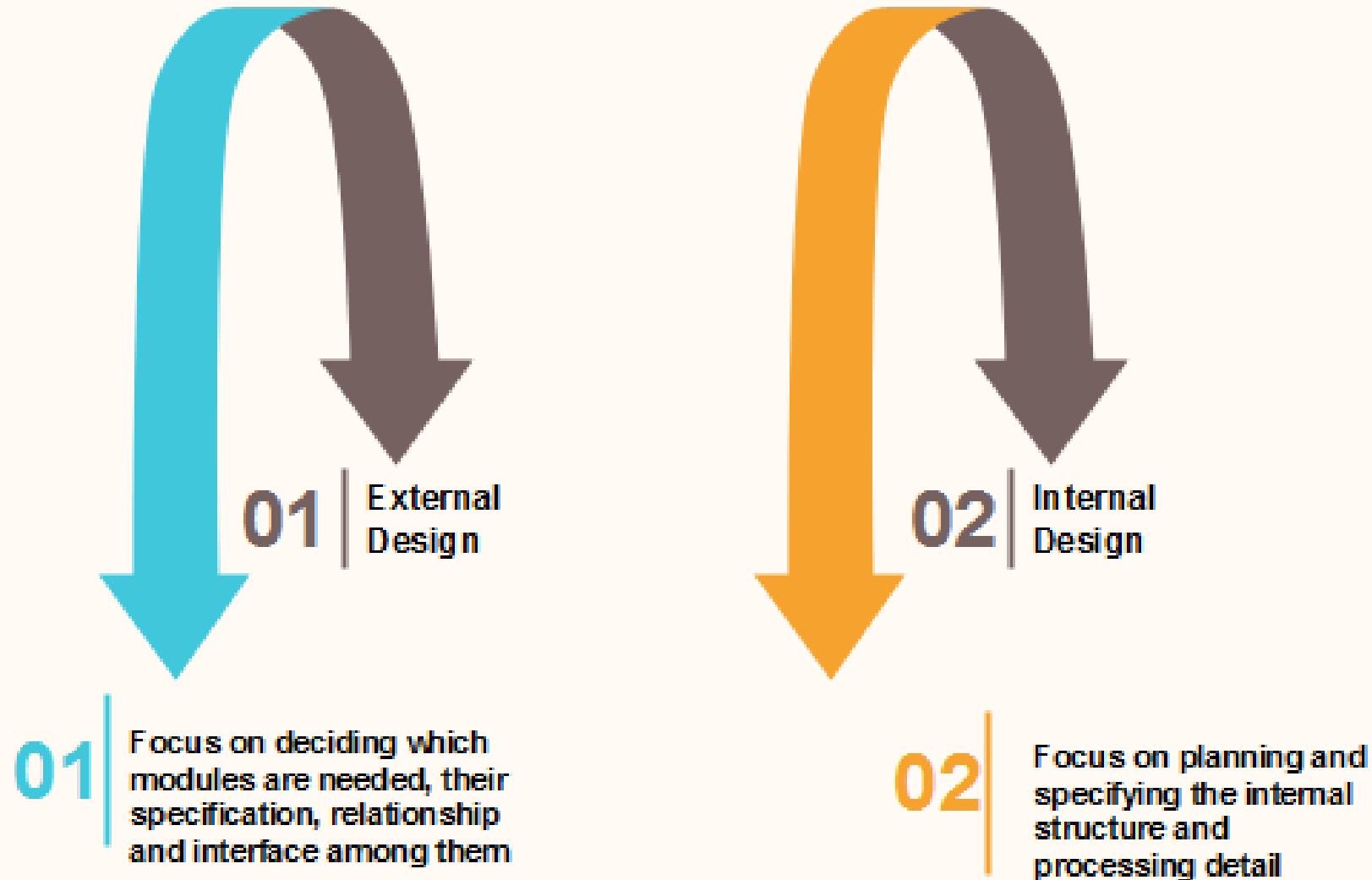
- Software Design
- Basic Concept of Software Design
- Architectural Design
- Low Level Design
- Modularization
- Design Structure Charts
- Pseudo Codes
- Flow Charts
- Coupling and Cohesion Measures

# Software Design

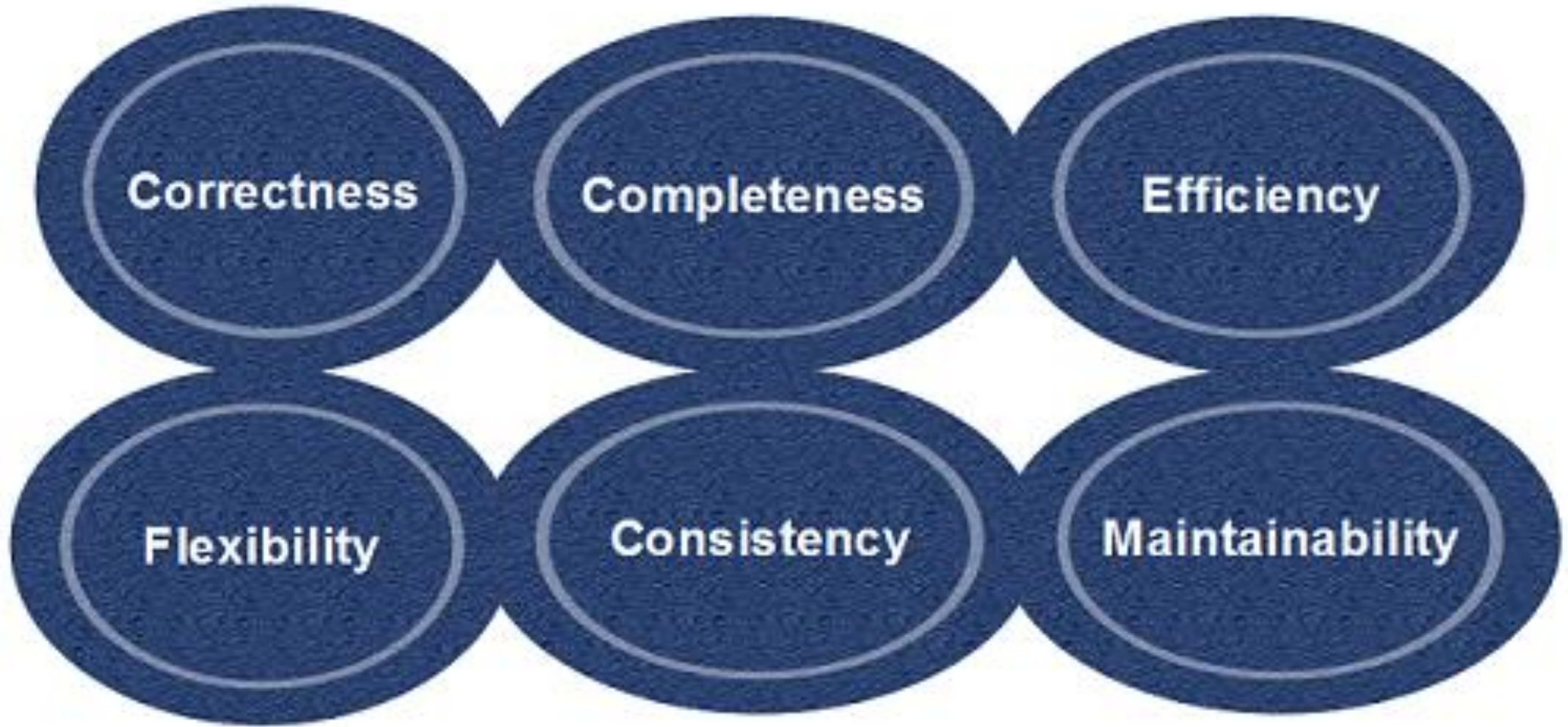
- Software design is a mechanism to transform user requirements into some suitable form, which helps the programmer in software coding and implementation.
- It deals with representing the client's requirement, as described in SRS (Software Requirement Specification) document, into a form, i.e., easily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



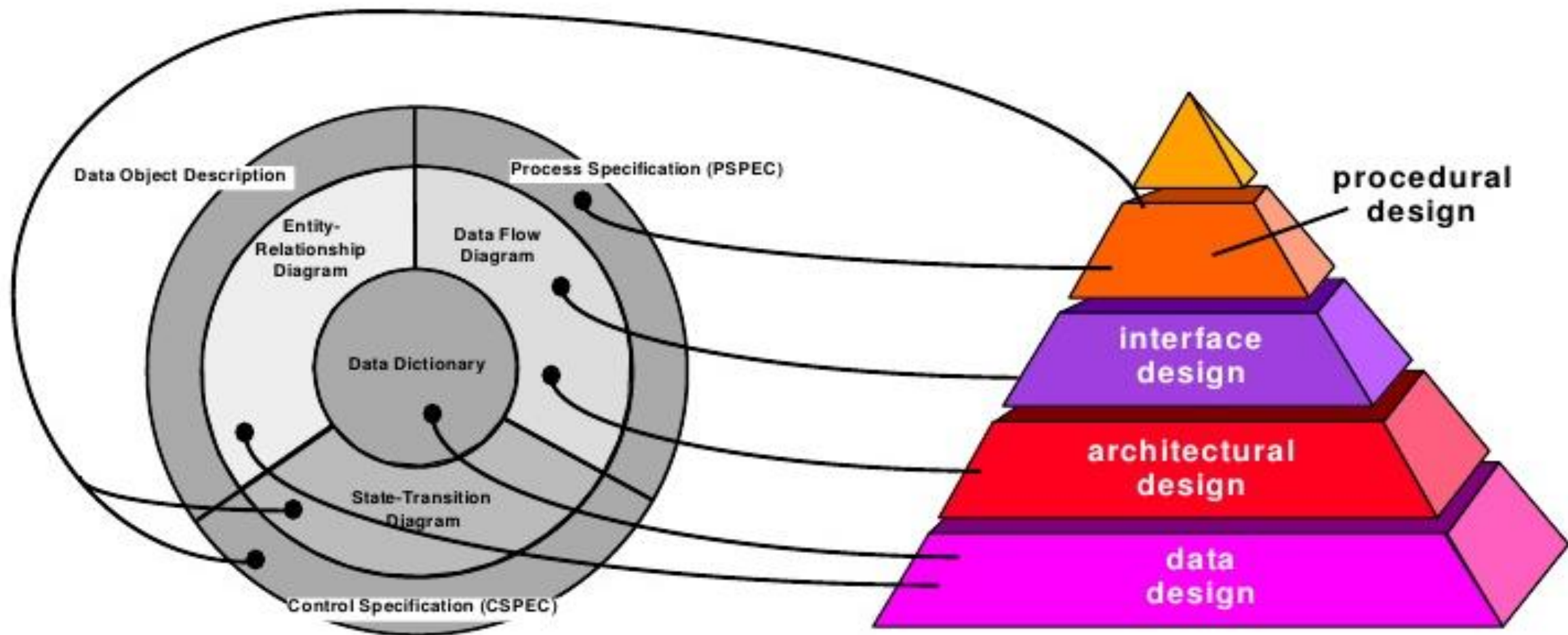
# Objectives of Software Design



**Objectives of Software Design**

Abhishek Kesharwani, Assistant Professor, United College of  
Engineering and Research

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



THE ANALYSIS MODEL

THE DESIGN MODEL



# Data/class design

The *data/class design* transforms class models into design class realizations and the requisite data structures required to implement the software. Part of class design may occur in conjunction with the design of software architecture.



# Architectural design

The *architectural design* defines the relationship between major structural elements of the software, the architectural styles and design patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented.

# The interface design

The *interface design* describes how the software communicates with systems that interoperate with it, and with humans who use it.

An interface implies a flow of information and a specific type of behavior. Therefore, usage scenarios and behavioral models provide much of the information required for interface design.

# Component-level design

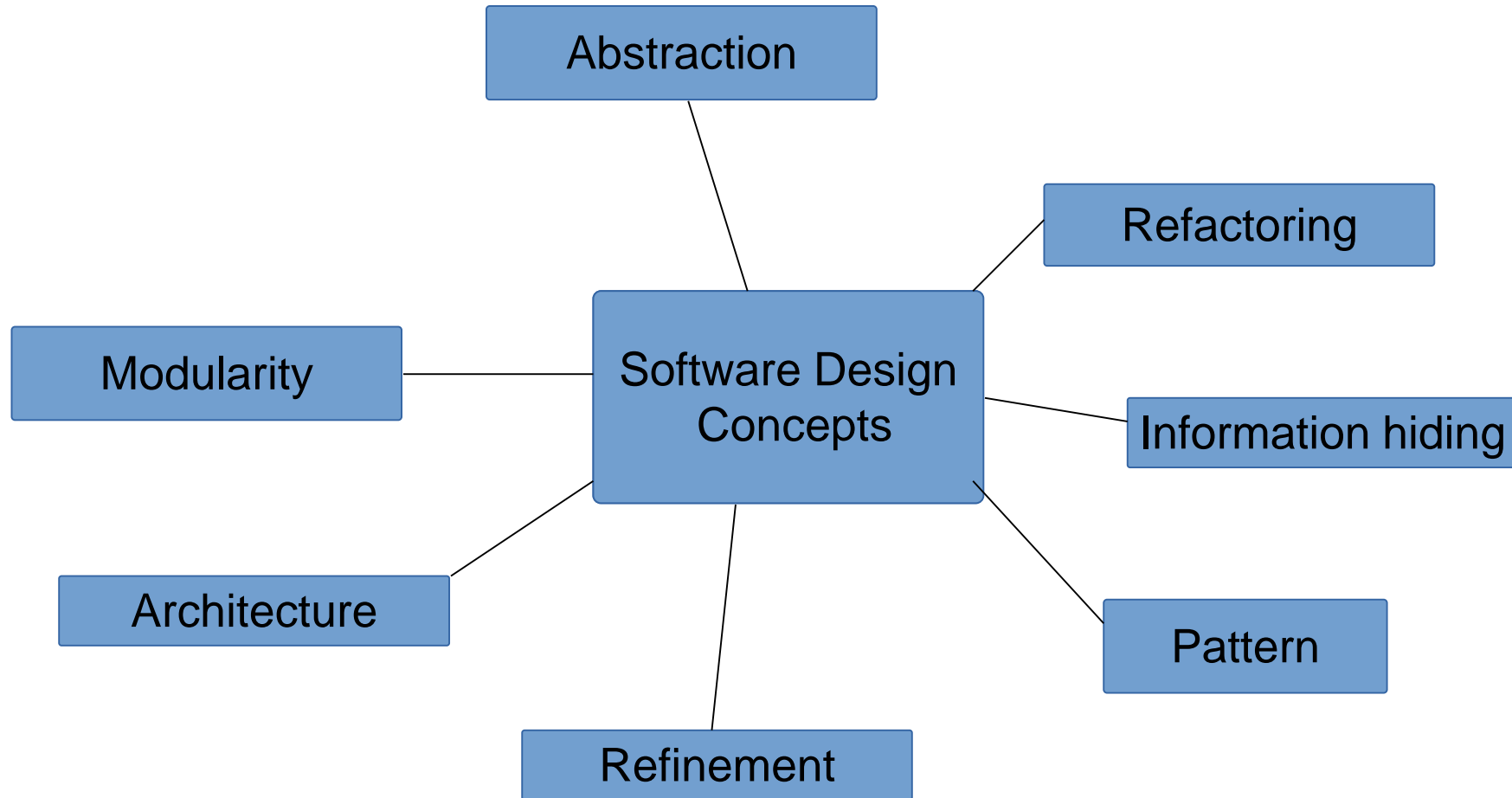
The *component-level design* transforms structural elements of the software architecture into a procedural description of software components.

Information obtained from the class-based models, flow models, and behavioral models serve as the basis for component design.

# The Design Process

- **Any design may be modelled as a directed graph** made up of entities with attributes which participate in relationships.
- The **system** should be described at several different levels of abstraction.
- **Design takes place in overlapping stages.** It is artificial to separate it into distinct phases but some separation is usually necessary.

# Software Design Concepts



# Abstraction- hide Irrelevant data

- . Abstraction simply means to hide the details to reduce complexity and increases efficiency or quality.
- . Different levels of Abstraction are necessary and must be applied at each stage of the design process so that any error that is present can be removed to increase the efficiency of the software solution and to refine the software solution.
- . The solution should be described in broad ways that cover a wide range of different things at a higher level of abstraction and a more detailed description of a solution of software should be given at the lower level of abstraction.

# Modularity- subdivide the system

- Modularity in design means subdividing a system into smaller parts so that these parts can be created independently and then use these parts in different systems to perform different functions.
- It is necessary to divide the software into components known as modules because nowadays there are different software available like Monolithic software that is hard to grasp for software engineers.
- If the system contains fewer components then it would mean the system is complex which requires a lot of effort (cost) but if we are able to divide the system into components then the cost would be small.



# Architecture

- Architecture simply means a technique to design a structure of something.
- Architecture in designing software is a concept that focuses on various elements and the data of the structure.
- These components interact with each other and use the data of the structure in architecture.

# Refinement- removes impurities

- Refinement simply means to refine something to remove any impurities if present and increase the quality.
- The refinement concept of software design is actually a process of developing or presenting the software or system in a detailed manner that means to elaborate a system or software.
- Refinement is very necessary to find out any error if present and then to reduce it.

# Pattern- a repeated form

- . The pattern simply means a repeated form or design in which the same shape is repeated several times to form a pattern.
- . The pattern in the design process means the repetition of a solution to a common recurring problem within a certain context.

# Information Hiding- hide the information

*Information hiding simply means to hide the information so that it cannot be accessed by an unwanted party.*

*In software design, information hiding is achieved by designing the modules in a manner that the information gathered or contained in one module is hidden and can't be accessed by any other modules.*

## *Refactoring- reconstruct something*

*Refactoring simply means reconstructing something in such a way that it does not affect the behaviour of any other features.*

*Refactoring in software design means reconstructing the design to reduce complexity and simplify it without affecting the behaviour or its functions.*

*Fowler has defined refactoring as “the process of changing a software system in a way that it won’t affect the behaviour of the design and improves the internal structure”*

# Levels of Software Design

## *Architectural Design:*

*The architecture of a system can be viewed as the overall structure of the system & the way in which structure provides conceptual integrity of the system. The architectural design identifies the software as a system with many components interacting with each other. At this level, the designers get the idea of the proposed solution domain.*

## *Preliminary or high-level design:*

*Here the problem is decomposed into a set of modules, the control relationship among various modules identified, and also the interfaces among various modules are identified. The outcome of this stage is called the program architecture. Design representation techniques used in this stage are structure chart and UML.*

## *Detailed design:*

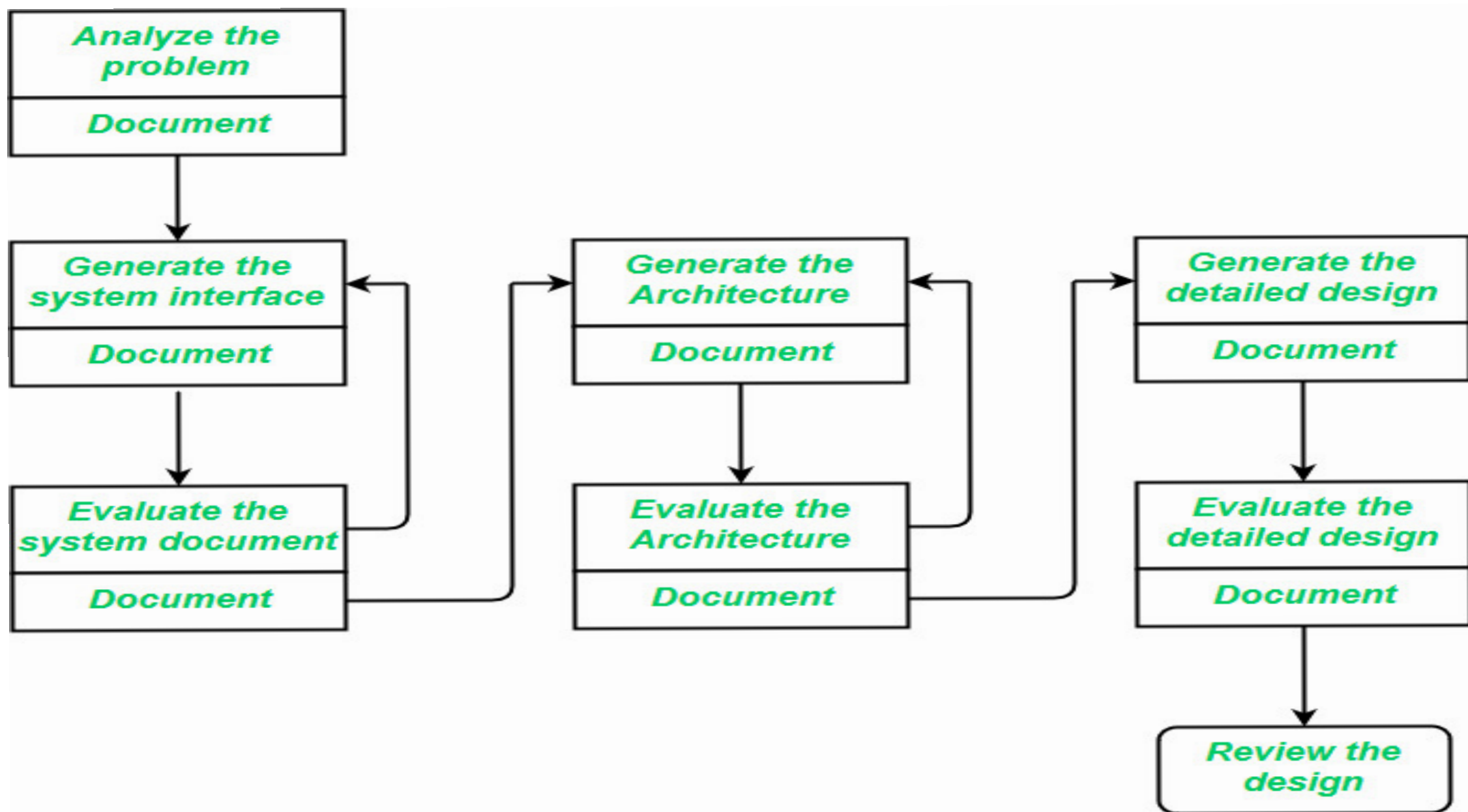
*Once the high-level design is complete, a detailed design is undertaken. In detailed design, each module is examined carefully to design the data structure and algorithms. The stage outcome is documented in the form of a module specification document.*

# Software Design process

The software design process can be divided into the following three levels of phases of design:

- Interface Design
- Architectural Design
- Detailed Design





# Interface Design

Interface design should include the following details:

- Precise description of events in the environment, or messages from agents to which the system must respond.
- Precise description of the events or messages that the system must produce.
- Specification on the data, and the formats of the data coming into and going out of the system.
- Specification of the ordering and timing relationships between incoming events or messages, and outgoing events or outputs.

# Architectural Design

Architectural design is the specification of the major components of a system, their responsibilities, properties, interfaces, and the relationships and interactions between them. In architectural design, the overall structure of the system is chosen, but the internal details of major components are ignored.

Issues in architectural design includes:

- Gross decomposition of the systems into major components.
- Allocation of functional responsibilities to components.
- Component Interfaces
- Component scaling and performance properties, resource consumption properties, reliability properties, and so forth.
- Communication and interaction between components.

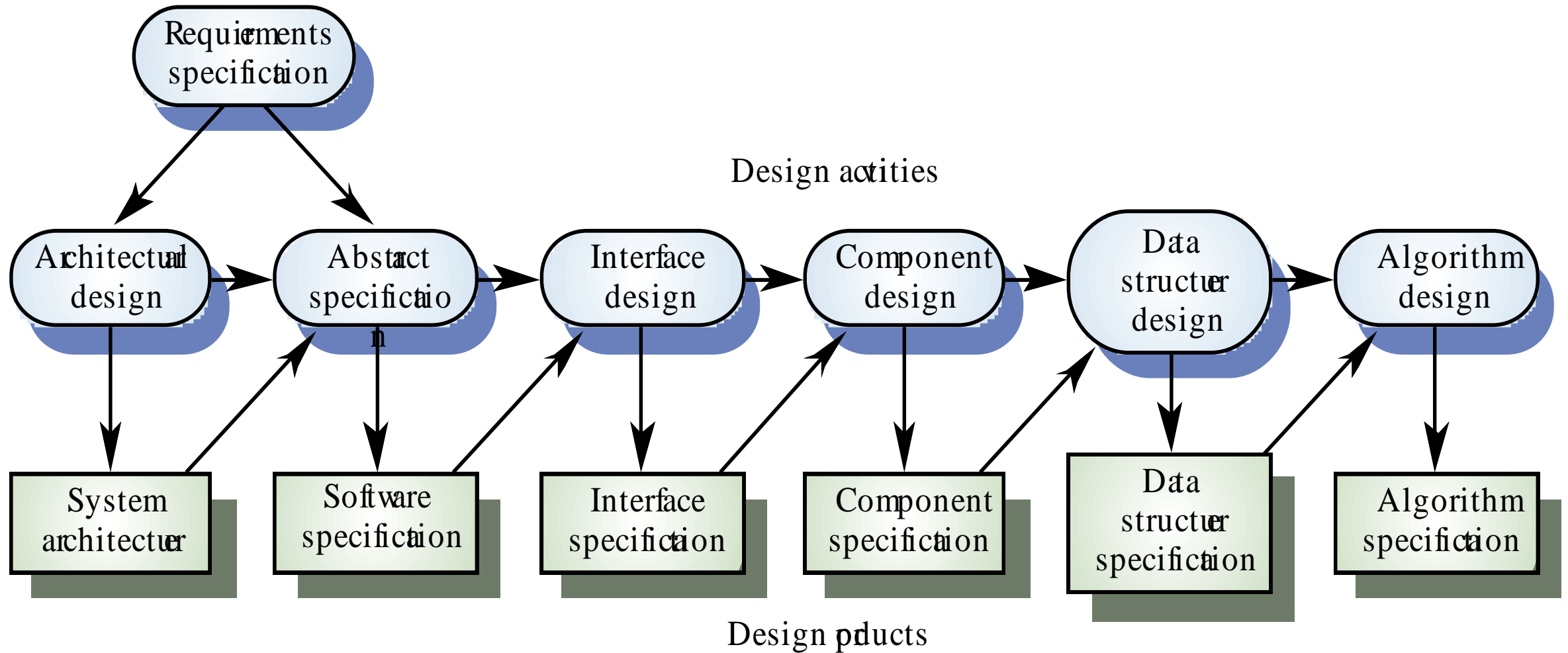
# Detailed Design

Design is the specification of the internal elements of all major system components, their properties, relationships, processing, and often their algorithms and the data structures.

The detailed design may include:

- Decomposition of major system components into program units.
- Allocation of functional responsibilities to units.
- User interfaces
- Unit states and state changes
- Data and control interaction between units
- Data packaging and implementation, including issues of scope and visibility of program elements
- Algorithms and data structures

# Phases in the Design Process



# Modular programming

- Computer systems are not monolithic: they are usually composed of multiple, interacting modules.
- Modularity has long been seen as a key to cheap, high quality software.

The goal of system design is to decode:

- What the modules are;
- What the modules should be;
- How the modules interact with one-another

# Modular programming

In the early days, **modular programming** was taken to mean constructing programs out of small pieces: “subroutines”

But **modularity** cannot bring benefits unless the modules are

- autonomous,
- coherent and
- robust



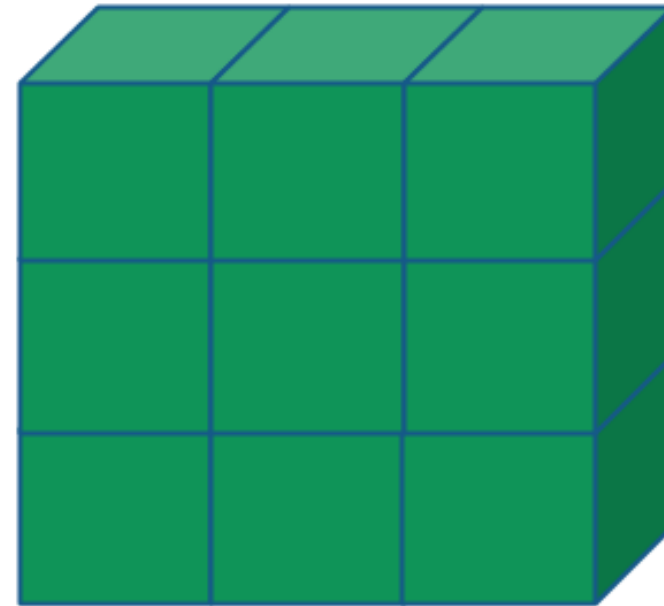
# Modularization

Modularization is the process of separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality.

With modularization, we can easily work on adding separate and smaller modules to a program without being hindered by the complexity of its other functions. In short, it's about being flexible and fast in adding more software functions to a program. In a software engineering team, we could easily work independently on each module without affecting others' work.



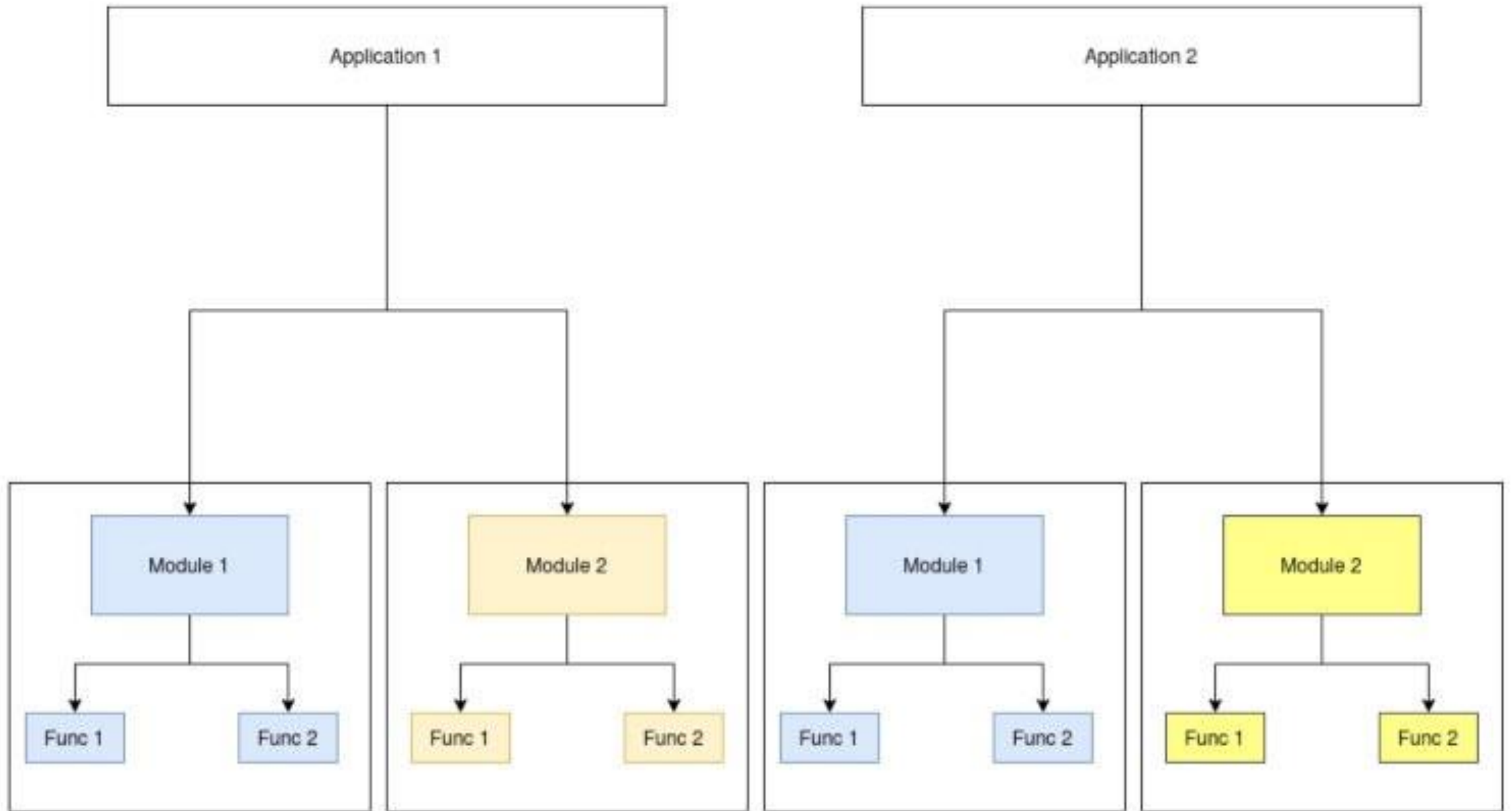
Monolith



Modular

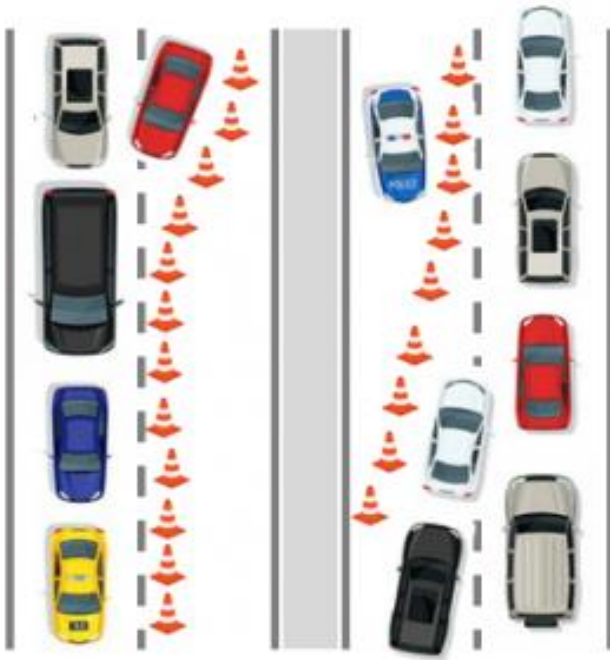
# Modularization

- Modularization is a technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out task(s) independently. These modules may work as basic constructs for the entire software. Designers tend to design modules such that they can be executed and/or compiled separately and independently.
- Modular design unintentionally follows the rules of 'divide and conquer' problem-solving strategy this is because there are many other benefits attached with the modular design of a software.

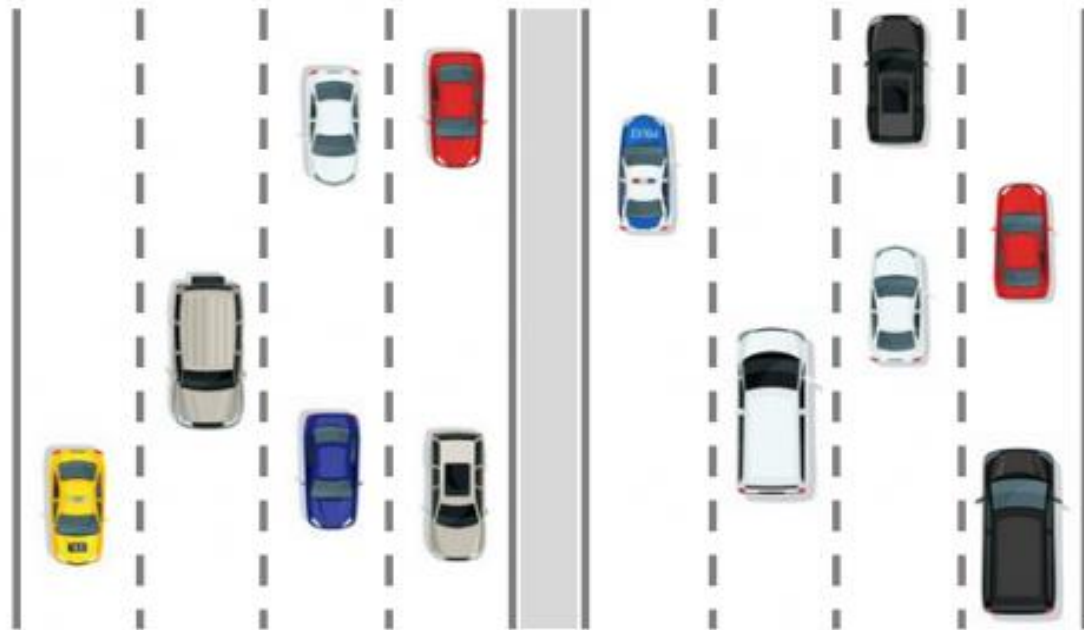


Without modularization, this will lead to an increase in development time, the number of bugs and the duration it takes to test and release a program.

### Monolithic Software



### Modular Software



# Advantage of modularization

- Smaller components are easier to maintain
- Program can be divided based on functional aspects
- Desired level of abstraction can be brought in the program
- Components with high cohesion can be re-used again
- Concurrent execution can be made possible
- Desired from security aspect

# Structure Chart

- **Structure Chart** represent hierarchical structure of modules.
- It breaks down the entire system into lowest functional modules, describe functions and sub-functions of each module of a system to a greater detail.
- Structure Chart partitions the system into black boxes (functionality of the system is known to the users but inner details are unknown).
- Inputs are given to the black boxes and appropriate outputs are generated.
- Modules at top level called modules at low level.
- Components are read from top to bottom and left to right.
- When a module calls another, it views the called module as black box, passing required parameters and receiving results.



# Symbols used in construction of structured chart

## 1. Module

It represents the process or task of the system. It is of three types.

### 1. Control Module

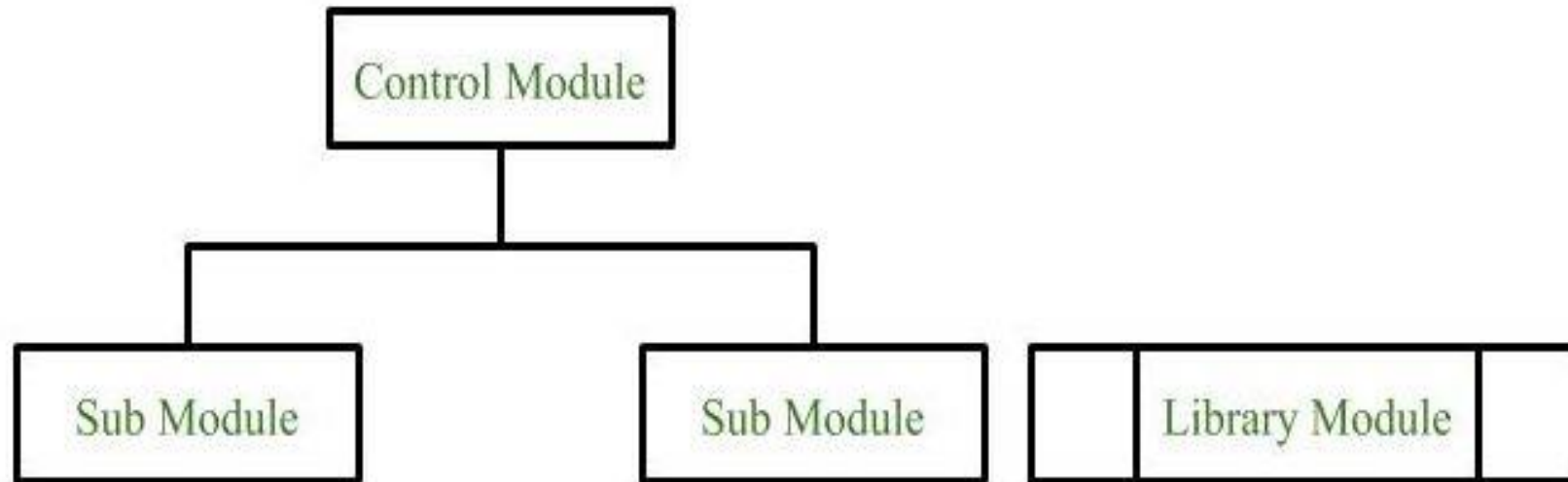
A control module branches to more than one sub module.

### 2. Sub Module

Sub Module is a module which is the part (Child) of another module.

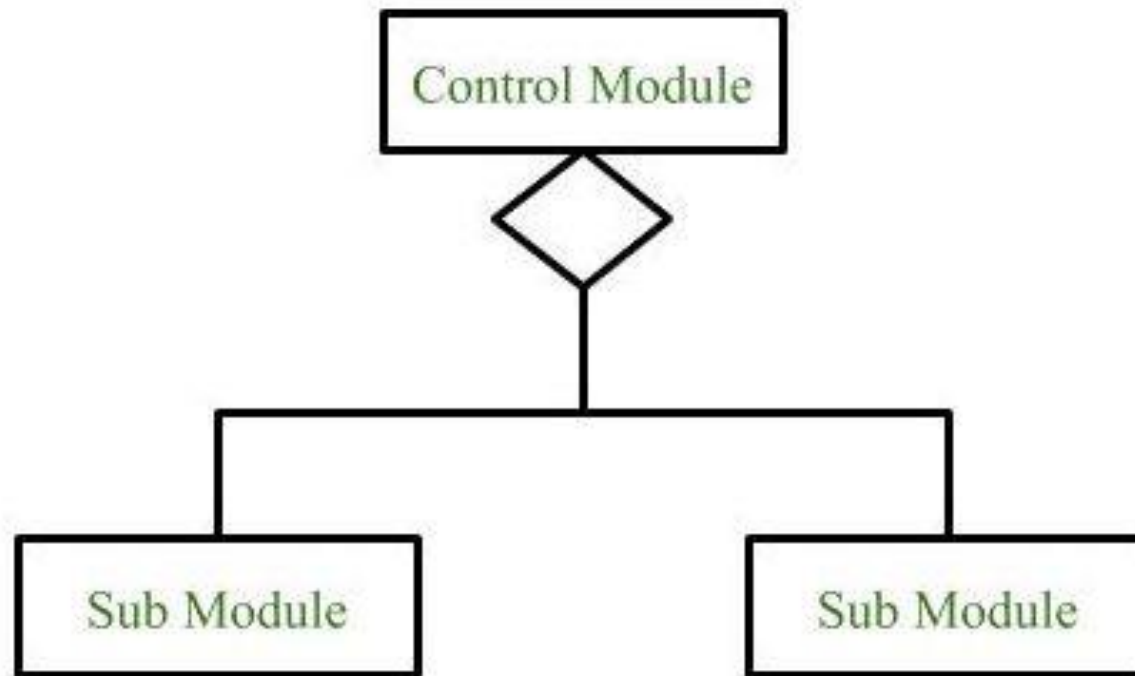
### 3. Library Module

Library Module are reusable and invocable from any module.



## 2. Conditional Call

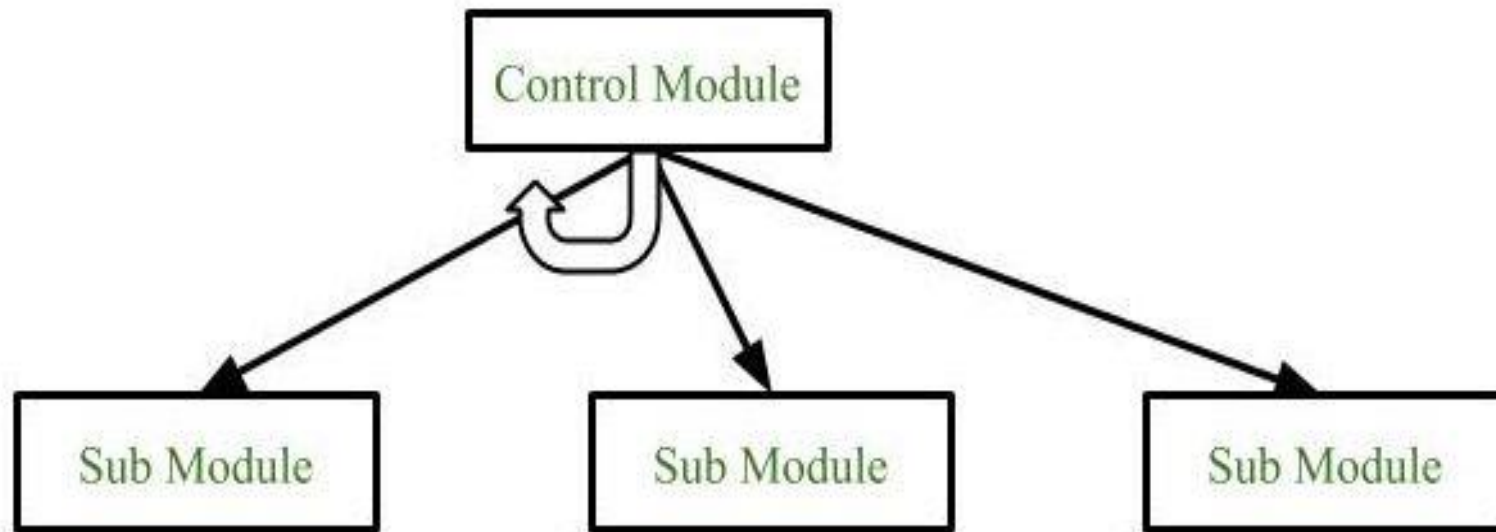
It represents that control module can select any of the sub module on the basis of some condition.



### 3. Loop (Repetitive call of module)

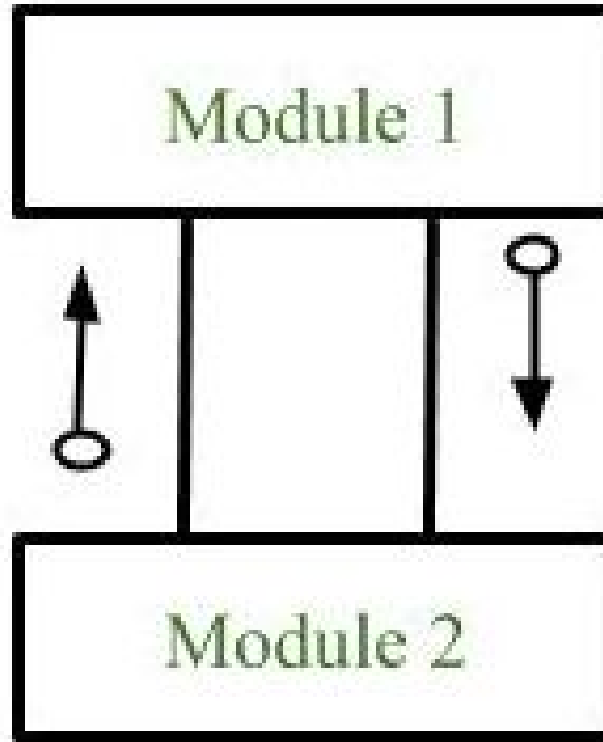
It represents the repetitive execution of module by the sub module.

A curved arrow represents loop in the module.



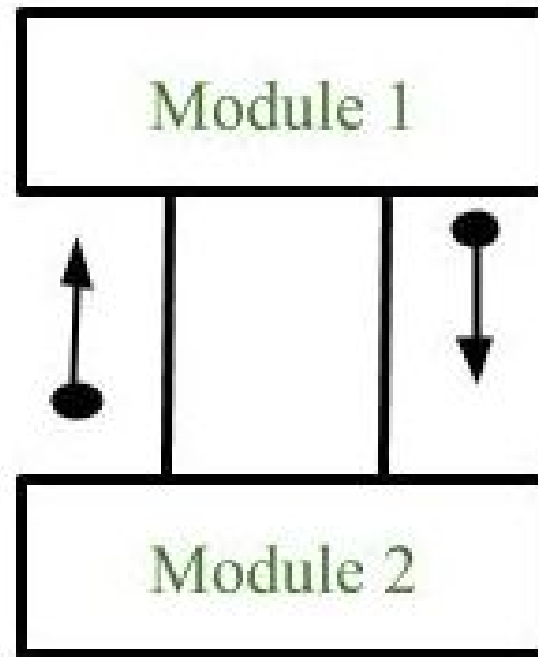
## 4. Data Flow

It represents the flow of data between the modules. It is represented by directed arrow with empty circle at the end.



## 5. Control Flow

It represents the flow of control between the modules. It is represented by directed arrow with filled circle at the end.



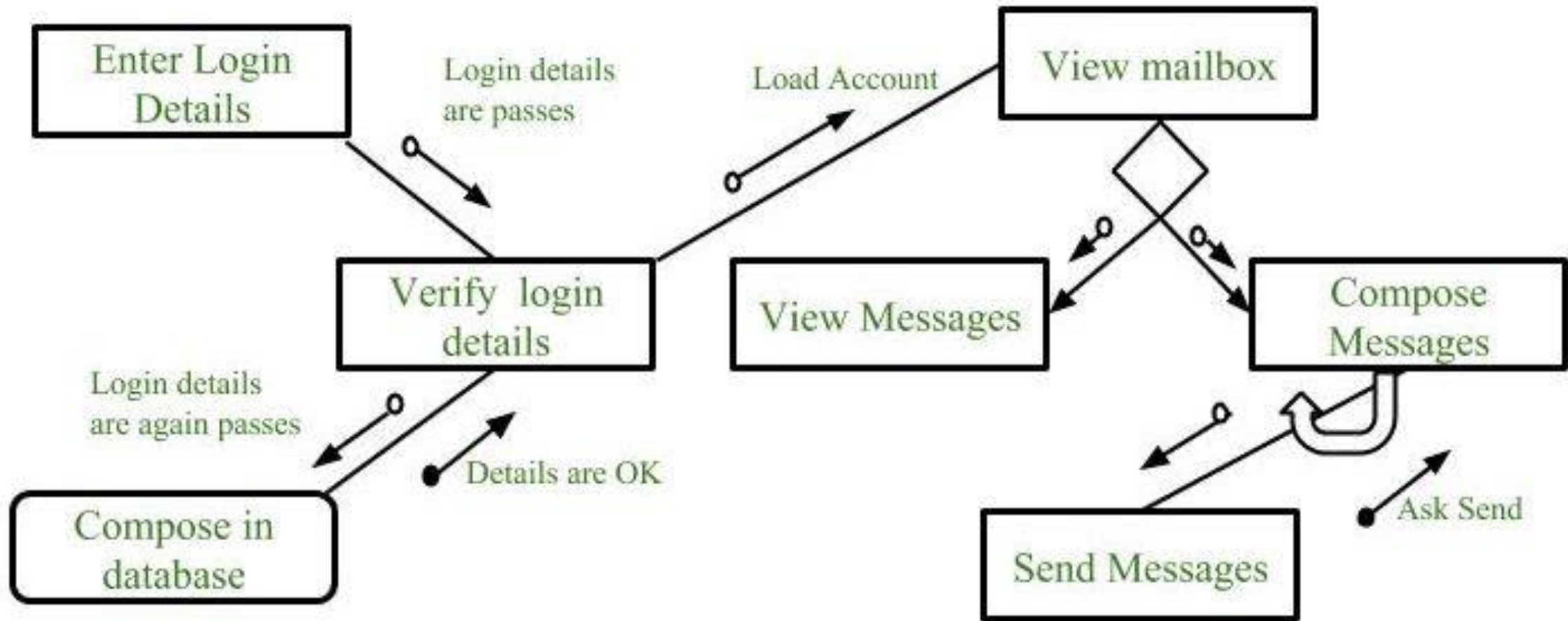
## 6. Physical Storage

Physical Storage is that where all the information are to be stored.



Physical Storage

# Example : Structure chart for an Email server



# Types of Structure Chart

## 1.Transform Centered Structured:

These type of structure chart are designed for the systems that receives an **input** which **is transformed by a sequence of operations being carried out by one module.**

## 2.Transaction Centered Structure:

These structure describes a system that **processes a number of different types of transaction.**



# Pseudo code

- **Pseudo code** is a term which is often used in programming and algorithm based fields.
- It is a methodology that allows the programmer to represent the implementation of an algorithm.
- Simply, we can say that it's the cooked up representation of an algorithm.
- Often at times, algorithms are represented with the help of pseudo codes as they can be interpreted by programmers no matter what their programming background or knowledge is.

# Pseudo code

Pseudo code: It's simply an implementation of an algorithm in the form of annotations and informative text written in plain English. It has no syntax like any of the programming language and thus can't be compiled or interpreted by the computer.

# Advantages of Pseudocode

- Improves the readability of any approach. It's one of the best approaches to start implementation of an algorithm.
- Acts as a bridge between the program and the algorithm or flowchart. Also works as a rough documentation, so the program of one developer can be understood easily when a pseudo code is written out. In industries, the approach of documentation is essential. And that's where a pseudo-code proves vital.
- The main goal of a pseudo code is to explain what exactly each line of a program should do, hence making the code construction phase easier for the programmer.

# How to write a Pseudo-code

1. Arrange the sequence of tasks and write the pseudocode accordingly.
2. Start with the statement of a pseudo code which establishes the main goal or the aim.

## **Example:**

This program will allow the user to check the number whether it's even or odd.

3. The way the if-else, for, while loops are indented in a program, indent the statements likewise, as it helps to comprehend the decision control and execution mechanism. They also improve the readability to a great extent.

Example:

```
if "1"  
    print response  
    "I am case 1"
```

```
if "2"  
    print response  
    "I am case 2"
```

4. Use appropriate naming conventions. The human tendency follows the approach to follow what we see. If a programmer goes through a pseudo code, his approach will be the same as per it, so the naming must be simple and distinct.
5. Use appropriate sentence casings, such as CamelCase for methods, upper case for constants and lower case for variables.
6. Elaborate everything which is going to happen in the actual code. Don't make the pseudo code abstract.

7. Use standard programming structures such as 'if-then', 'for', 'while', 'cases' the way we use it in programming.
8. Check whether all the sections of a pseudo code is complete, finite and clear to understand and comprehend.
9. Don't write the pseudo code in a complete programmatic manner. It is necessary to be simple to understand even for a layman or client, hence don't incorporate too many technical terms.

**Do's :**

- . Use control structures
- . Use proper naming convention
- . Indentation and white spaces are the key
- . Keep it simple.
- . Keep it concise.

**Don'ts :**


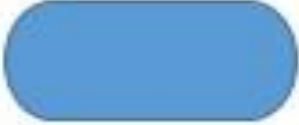



- . Don't make the pseudo code abstract.
- . Don't be too generalized.
- .






# Flowchart





- A flowchart is a diagram that depicts a process, system or computer algorithm.
- They are widely used in multiple fields to document, study, plan, improve and communicate often complex processes in clear, easy-to-understand diagrams.
- They can range from simple, hand-drawn charts to comprehensive computer-drawn diagrams depicting multiple steps and routes.
- If we consider all the various forms of flowcharts, they are one of the most common diagrams on the planet, used by both technical and non-technical people in numerous fields.
- Flowcharts are sometimes called by more specialized names such as Process Flowchart, Process Map, Functional Flowchart, Business Process Mapping, Business Process Modeling and Notation (BPMN), or Process Flow Diagram (PFD).




# Flowchart symbols

Symbol	Symbol Name	Description
	Flow lines	Flow lines are used to connect symbols used in flowchart and indicate direction of flow.
	Terminal (START / STOP)	This is used to represent start and end of the flowchart.
	Input / Output	It represents information which the system reads as input or sends as output.
	Processing	Any process is represented by this symbol. For example, arithmetic operation, data movement.
	Decision	This symbol is used to check any condition or take decision for which there are two answers. Yes (True) or No (False).

# Flowchart symbols

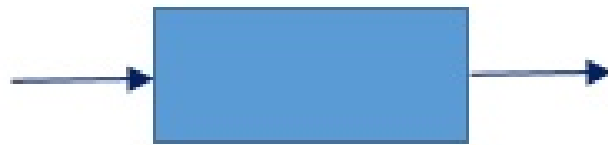
	Connector	It is used to connect or join flow lines.
	Off-page Connector	This symbol indicates the continuation of flowchart on the next page.
	Document	It represents a paper document produced during the flowchart process.

	Annotation	It is used to provide additional information about another flowchart symbol which may be in the form of descriptive comments, remarks or explanatory notes.
	Manual Input	It represents input to be given by a developer or programmer.
	Manual Operation	This symbol indicates that the process has to be done by a developer or programmer.
	Online Storage	It represents online data storage such as hard disks, magnetic drums or other storage devices.

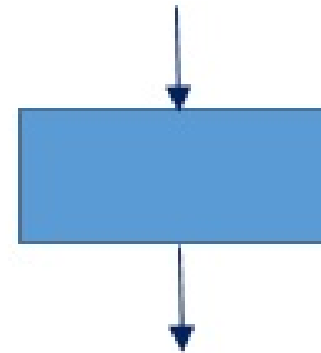
	Offline Storage	It represents offline data storage such as sales on OCR, data on punched cards.
	Communication Link	It represents the data received or to be transmitted from an external system.
	Magnetic Disk	It represents data input or output from and to a magnetic disk.

# Guidelines for preparing proper flowcharts

- The flowchart should be neat and easy to follow so that it will be clearly understood.
- A logical start and end must be given to the flowchart.
- The flowchart should include necessary steps in logical order.
- 

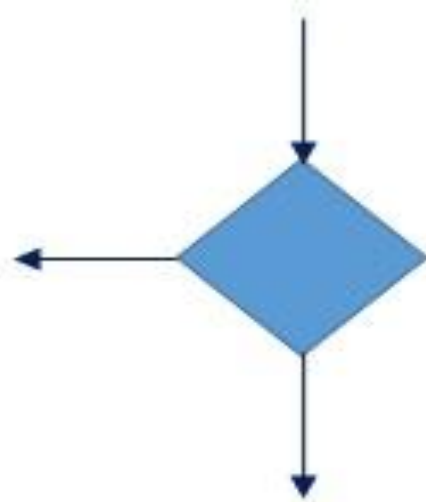


**Left to right**



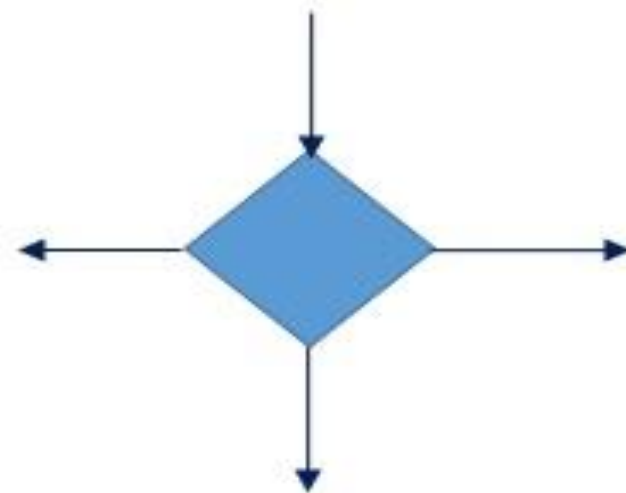
**Top to bottom**

- The only one flow line should come out from a process symbol.
- The decision making symbol should have only one incoming flow line. However, it may have two or three out-going flowlines.



**Two outgoing flow-lines**

OR

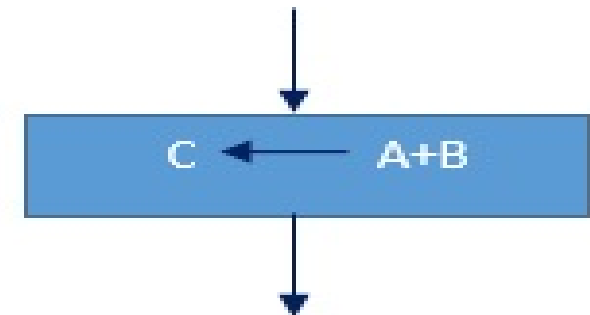


**Three outgoing flow-lines**

- The terminal symbols, that is, Start and Stop / End symbols should have only one flow line.

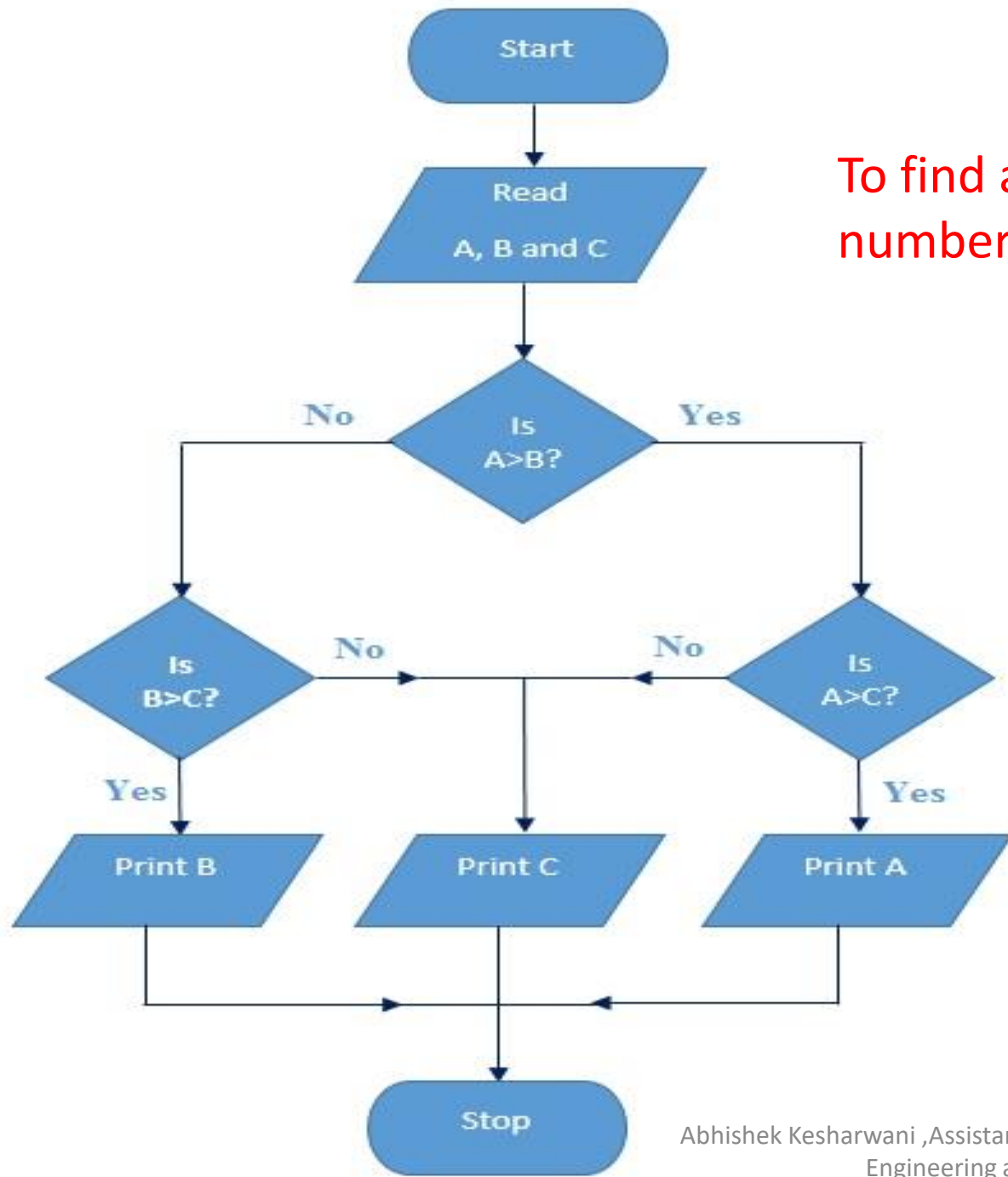


- The symbol should contain the information (process, data or text) clearly to carry out the required action.
- For example, to add two numbers A and B and storing results as C, draw the following block.





- The number of flow-lines can be reduced using connector symbol. The connectors are mainly required in complex flowcharts.
- The intersected flow-lines should be avoided. This makes the flowchart effective and represents communication clearly.
- The correctness of the flowchart can be tested by passing the test data through it. It becomes the validity test of the flowchart.

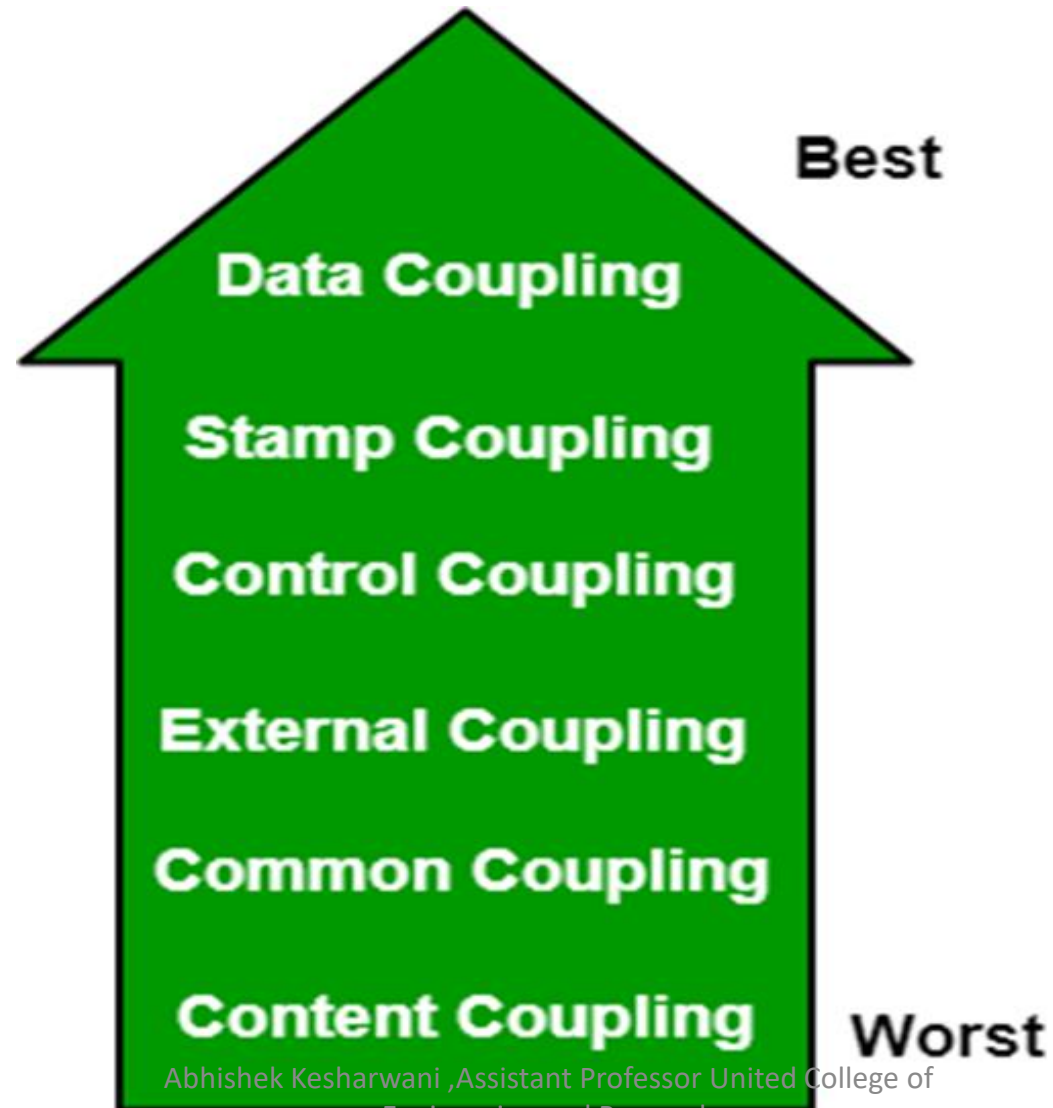


To find and print the largest of the given three numbers A, B & C

# Coupling and Cohesion Measures

Coupling: Coupling is the measure of the degree of interdependence between the modules. A good software will have low coupling.

# Types of Coupling:



# Types of Coupling

## Data Coupling:

If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled.

In data coupling, the components are independent of each other and communicate through data. Module communications don't contain tramp data.

Example-customer billing system.

## Stamp Coupling

In stamp coupling, the complete data structure is passed from one module to another module. Therefore, it involves tramp data. It may be necessary due to efficiency factors- this choice was made by the insightful designer, not a lazy programmer.

## Control Coupling:

If the modules communicate by passing control information, then they are said to be control coupled. It can be bad if parameters indicate completely different behavior and good if parameters allow factoring and reuse of functionality. Example- sort function that takes comparison function as an argument.

**External Coupling:** In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Ex- protocol, external file, device format, etc.

**Common Coupling:** The modules have shared data such as global data structures. The changes in global data mean tracing back to all modules which access that data to evaluate the effect of the change. So it has got disadvantages like difficulty in reusing modules, reduced ability to control data accesses, and reduced maintainability.

**Content Coupling:** In a content coupling, one module can modify the data of another module, or control flow is passed from one module to the other module. This is the worst form of coupling and should be avoided.

# Cohesion

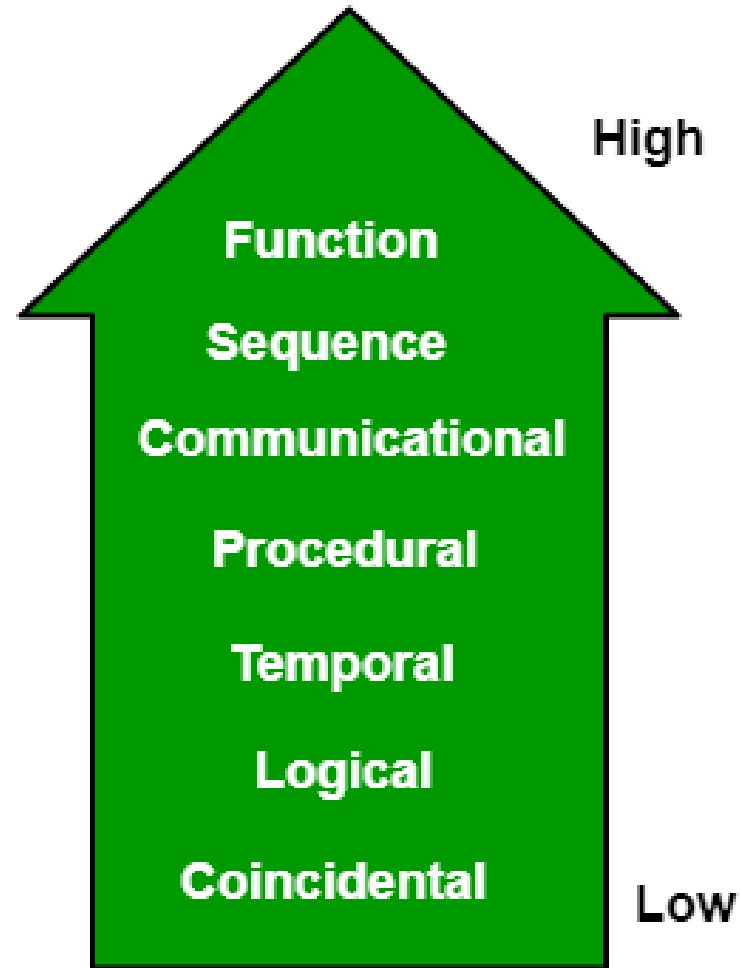
Cohesion is a measure of the degree to which the elements of the module are functionally related.

It is the degree to which all elements directed towards performing a single task are contained in the component. Basically, cohesion is the internal glue that keeps the module together.

A good software design will have high cohesion.



# Types of Cohesion



# Types of Cohesion

**Functional Cohesion:** Every essential element for a single computation is contained in the component. A functional cohesion performs the task and functions. It is an ideal situation.

**Sequential Cohesion:** An element outputs some data that becomes the input for other element, i.e., data flow between the parts. It occurs naturally in functional programming languages.

**Communicational Cohesion:** Two elements operate on the same input data or contribute towards the same output data. Example- update record in the database and send it to the printer.

**Procedural Cohesion:** Elements of procedural cohesion ensure the order of execution. Actions are still weakly connected and unlikely to be reusable. Ex- calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA.

**Temporal Cohesion:** The elements are related by their timing involved. A module connected with temporal cohesion all the tasks must be executed in the same time span. This cohesion contains the code for initializing all the parts of the system. Lots of different activities occur, all at unit time.

**Logical Cohesion:** The elements are logically related and not functionally.

Ex- A component reads inputs from tape, disk, and network. All the code for these functions is in the same component. Operations are related, but the functions are significantly different.

**Coincidental Cohesion:** The elements are not related(unrelated). The elements have no conceptual relationship other than location in source code. It is accidental and the worst form of cohesion.

Ex- print next line and reverse the characters of a string in a single component.