Software Design

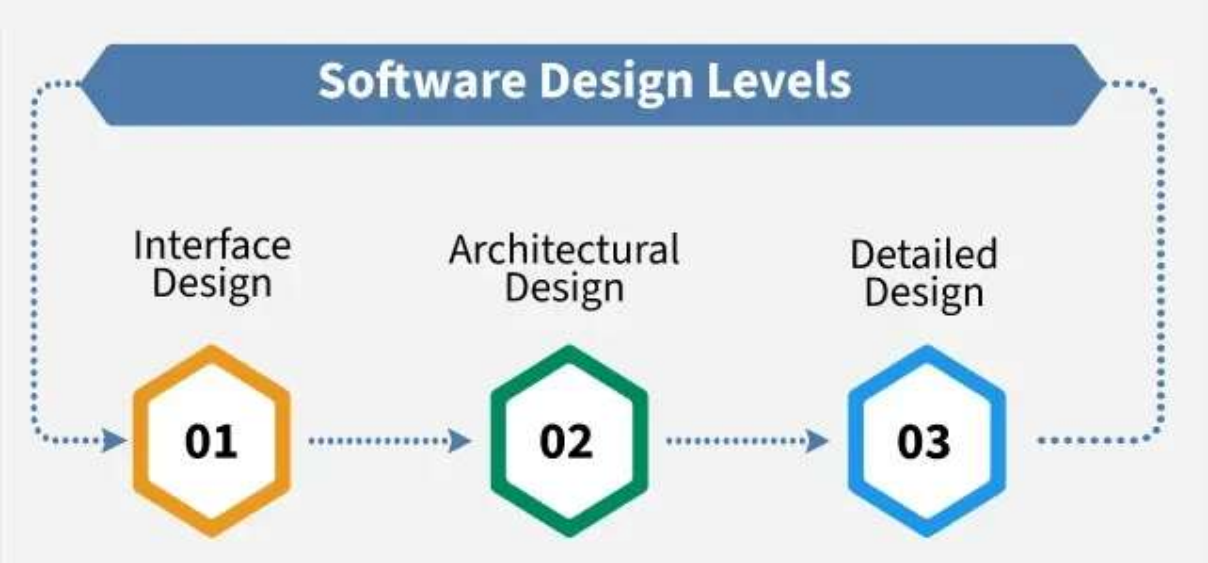
What is the Software Design Process?

Software design is the step where developers **plan** how to build a software system. Think of it as creating a **blueprint** before building a house. Instead of writing code right away, developers first:

- Understand the needs of users.
- Break big tasks into smaller parts.
- Decide how all parts will work together.

Levels of Software Design

Software design happens in **three main levels**:



1. Interface Design

- Focuses on **how the system interacts** with users, devices, and other systems.
- At this stage, the **inside logic** of the system is ignored.

- The system is treated like a **black box** — only input and output are considered.

It includes:

- Events or messages the system should respond to.
- What the system should send back.
- Data format and flow in and out.
- Timing and order of actions.

---

2. Architectural Design

- Focuses on **big parts** (components) of the system and their **roles**.
- Shows how parts **connect and work together**.
- Internal details are still ignored.

Key points:

- Divide system into components.
- Assign jobs to each component.
- Define interfaces and communication between parts.
- Consider performance and reliability.

---

3. Detailed Design

- Now we go **inside each component**.
- Decide what each part does, how it does it, and how they connect.

It includes:

- Breaking components into smaller units.
- Writing algorithms and choosing data structures.
- Designing user interfaces.

- Managing how data is shared and stored.

Phases of the Software Design Process

Here are the steps teams usually follow:

1. Understand Requirements

- Learn what users need and what the business wants to achieve.

2. Research, Analysis & Planning

- Use surveys, interviews, and data to understand users better.
- Plan the software with real needs in mind.

3. Create Initial Designs

- Make sketches (wireframes), user stories, and flow diagrams.
- Get feedback and improve the design.

4. Technical Design

- Write a detailed plan of how the system will work technically.
- Define components and how they connect.

5. User Interface (UI) Design

- Focus on how the software looks and feels.
- Make sure it's **easy to use** and **visually clear**.

6. Prototyping

- Build early versions (prototypes) to test the design.
- These help gather feedback before actual development.

Elements of Software Design

Here are the core building blocks of any software design:

1. **Architecture**
   The big picture — how everything is structured and connected.
2. **Modules**
   Small parts that each do one specific job. Easier to test and maintain.
3. **Components**
   Groups of related modules. Help in organizing large systems.
4. **Interfaces**
   Define how different parts of the system **talk to each other**.
5. **Data**
   Information that flows through the system. Managing it properly is crucial.

How Software Design Fits into SDLC (Software Development Life Cycle)

- **Design** happens after requirements are gathered.
- Before writing code, the team plans everything using diagrams, wireframes, and models.
- A good design ensures **smooth development** and **fewer errors**.

Software Design Principles

Good design follows some key rules:

- **Modularity**: Break software into small, separate parts.
- **Low Coupling**: Keep parts independent so changes don't affect everything.
- **Abstraction**: Hide unnecessary details; show only what's needed.
- **Anticipate Change**: Make the design flexible for future updates.
- **Simplicity**: Keep the design clear and easy to understand.
- **Sufficiency & Completeness**: Cover all needs without adding unnecessary things.

Structured Analysis and Structured Design (SA/SD) –

This is a diagrammatic notation that is designed to help people understand the system. The basic goal of SA/SD is to improve quality and reduce the risk of system failure.

Main Phases of SA/SD

1. Structured Analysis (SA)

- Focus: Understanding what the system should do.
- Tools used:
    - Data Flow Diagram (DFD): Shows how data moves through the system.
    - Data Dictionary: Defines all data elements used in the system.
    - State Transition Diagram: Shows object states and transitions due to events.
    - Entity-Relationship (ER) Diagram: Shows relationships between data/entities.

2. Structured Design (SD)

- Focus: How the system will be built.

- Tools used:
    - Structure Chart: Breaks down system into modules/tasks.
    - Pseudocode: Describes logic in human-readable form (no coding language needed).

SA/SD is combined known as SAD and it mainly focuses on the following 3 points:

- **System** – What is to be built.
- **Process** – How the data flows and is handled.
- **Technology** – Platforms and tools used for implementation.

Advantages of SA/SD

Advantage	Explanation
Clarity & Simplicity	Breaks complex systems into easy parts.
Better Communication	Diagrams and documents help all team members understand.
Improved Maintainability	Clear structure means easier future updates.
Better Testability	Defined inputs/outputs make testing easier.

Disadvantages of SA/SD

Disadvantage	Explanation
Time-Consuming	Heavy documentation and planning.
Inflexible	Hard to change after design is done.
Limited Iteration	Not suitable for modern, agile methods.

Steps in SA/SD Process

1. **Requirements Gathering** – Talk to users and stakeholders.
2. **Structured Analysis** – Use DFD, ER diagrams, etc., to analyze.
3. **Data Modeling** – Define how data is structured.
4. **Process Modeling** – Define how the system works via diagrams.
5. **I/O Design** – Define inputs (UI, forms) and outputs (reports).
6. **Structured Design** – Design modules and system architecture.
7. **Implementation & Testing** – Build and test the system.

SA/SD involves 2 phases:

1. **Analysis Phase:** It uses Data Flow Diagram, Data Dictionary, State Transition diagram and ER diagram.
2. **Design Phase:** It uses Structure Chart and Pseudo Code.

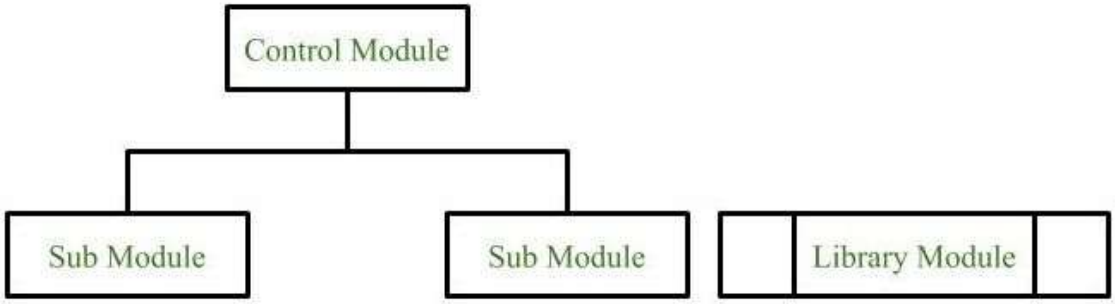
Design Structure Charts –

- Structure Chart represents the hierarchical structure of modules.
- Structure Chart partitions the system into black boxes.

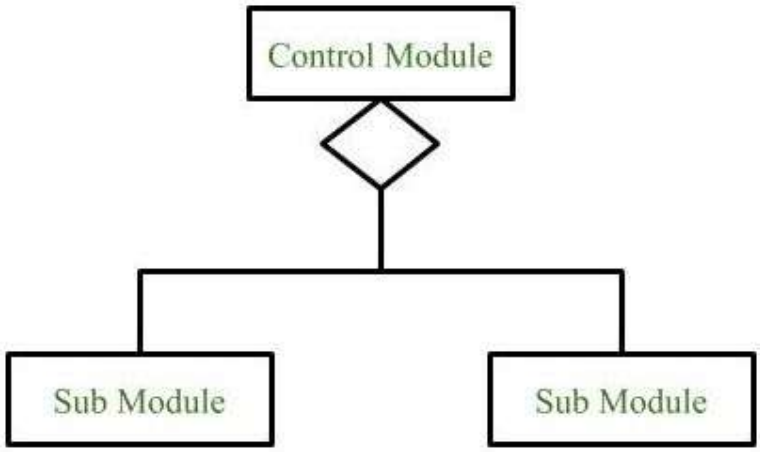
Symbols in Structural Chart –

1. Module -
   It represents the process or task of the system. It is of three types:
   - **Control Module**: A control module branches to more than one submodule.
   - **Sub Module:** Sub Module is a module which is the part (Child) of another module.

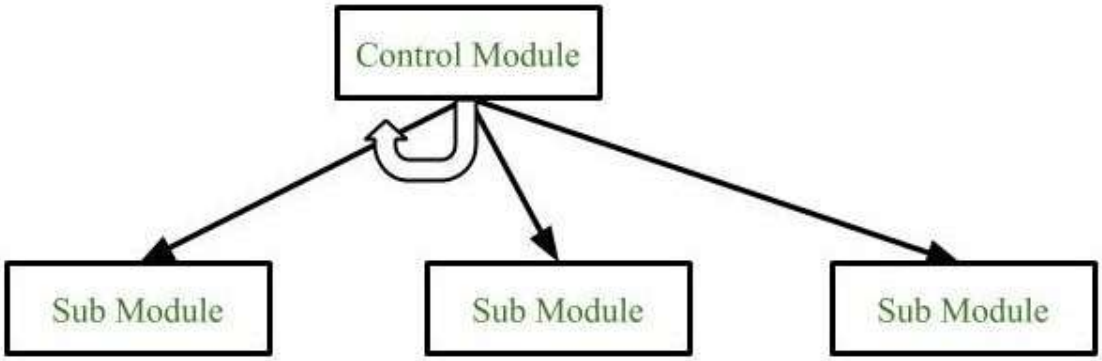
- **Library Module:** Library Module are reusable and invokable from any module.



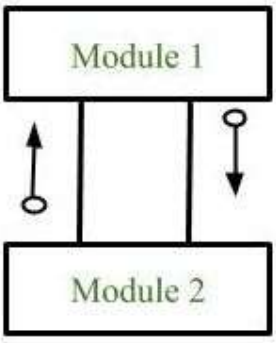
2. Conditional Call
   It represents that control module can select any of the sub module on the basis of some condition.

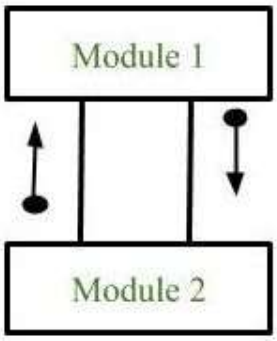


3. Loop (Repetitive Call of Module)
   It represents the repetitive execution of module by the sub module. A curved arrow represents a loop in the module.



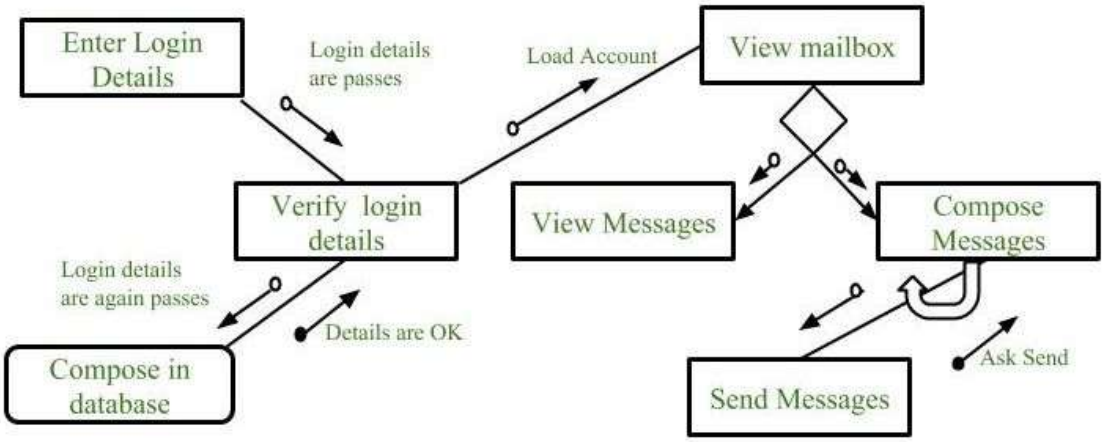
4. Data flow -
   It represents the flow of data between the modules. It is represented by a directed arrow with an empty circle at the end.





5. **Control Flow**
   It represents the flow of control between the modules. It is represented by a directed arrow with a filled circle at the end.
6. Physical Storage
   It is that where all the information are to be stored.

Structure Chart for an Email Server



Pseudocode :
Pseudocode is a simplified , informal representation of an algorithm or a program that uses a mix of natural language and programming constructs.

It is used to illustrate the high – level structure and logic of an algorithm without the syntactic details of a specific programming language.

Key Features of Pseudocode:

- **Language-agnostic**: Doesn't follow rules of any programming language (like Python or Java).
- **Readable**: Written in plain, structured English.
- **Descriptive**: Focuses on logic and steps, not code formatting.
- **Step-by-step**: Breaks down a process into clear actions.

Object – Oriented Design (OOD) –

Object-Oriented Design (OOD) is a software development technique that models a system as a collection of interacting objects, each defined by its class , state (data), and behavior (methods).

It uses real-world concepts like:

- Classes
- Objects
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Goal: Create a modular, scalable, and maintainable system architecture.

Importance of OOD in System Design

Benefit	Description
Modularity	Breaks complex systems into manageable pieces (classes).
Reusability	Objects and classes can be reused in other parts of the system or projects.
Scalability	Easily add new features by introducing new objects.
Maintainability	Easier to debug and modify due to encapsulation and modularity.
Real-World Mapping	Easier to conceptualize because it's modeled after real-world entities.

Benefit	Description
Flexibility	Changes in the system are easier to implement through polymorphism/inheritance.

Key Principles of OOD

1. Encapsulation

- Combines data and methods into a class.
- Hides internal implementation details.
- Prevents direct access to internal states.

```
class Car {
  private int speed;
  void accelerate() { speed += 10; }
}
```

2. Abstraction

- Shows only **essential details** to the user.
- Hides unnecessary implementation complexities.

3. Inheritance

- Enables a class to inherit properties and behavior from another class.
- Promotes **code reuse** and creates a **hierarchical relationship**.

4. Polymorphism

- Same interface, different implementations.
- Methods can act differently based on the object calling them.

```
Animal a = new Dog();
a.makeSound(); // Calls Dog's makeSound
```

5. Composition

- Classes are made by combining other class objects.
- More flexible than inheritance.

SOLID Principles of OOD

Principle	Explanation
S – Single Responsibility	Each class should have only one reason to change.
O – Open/Closed	Open for extension, closed for modification.
L – Liskov Substitution	Subtypes must be replaceable by base types.
I – Interface Segregation	Prefer many small interfaces over one large one.
D – Dependency Inversion	Depend on abstractions, not concrete classes.

Object-Oriented Design Patterns

Design patterns are like templates or blueprints for solving common software design problems. They are categorized into three types:

Creational Patterns – Handling Object Creation

- **Singleton**: Ensures that only **one object** of a class is created in the whole system.

Structural Patterns – How Objects Are Organized or Connected

- **Adapter**: Converts one type of interface to another so they can work together.

Behavioral Patterns – How Objects Communicate

- **Observer**: One object changes, and many others get notified.

UML Diagrams in OOD

Diagram Type	Purpose
Class Diagram	Shows class structure, attributes, and relationships.
Sequence Diagram	Shows object interactions over time.
State Diagram	Depicts object states and transitions.

Common Challenges in OOD

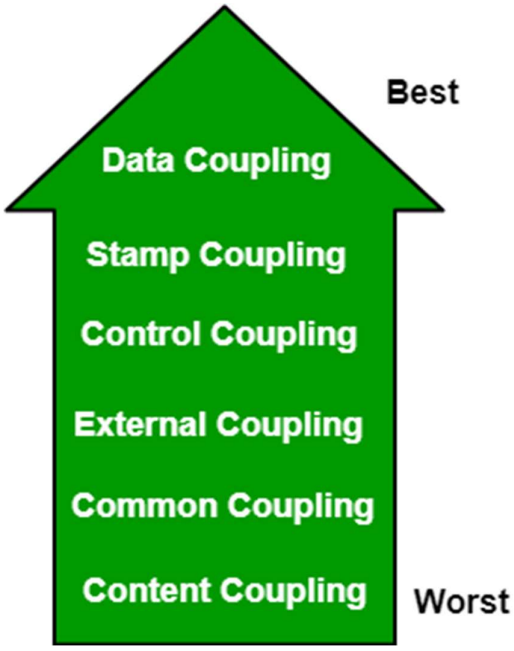
Problem	Simple Fix
Overengineering	Don’t build for "maybe" future needs — solve **today’s** real problems.
Not Flexible Enough	Follow **SOLID** principles to make changes easier later.
Slow from Too Much Abstraction	Optimize only the **important/slow** parts.
Wrong Abstraction Level	Don’t make things too complex or too simple — find the right **balance**.

Common Anti-Patterns in OOD

Anti-Pattern	Problem
God Object	One class tries to **do everything** — hard to manage or update.
Spaghetti Code	Code has **no structure**, everything is tangled — hard to fix.
Lava Flow	Old, **unused code** still exists — clutters the system.
Object Orgy	Objects share **too much data** — no boundaries, breaks encapsulation.

Coupling :

Coupling refers to the degree of interdependence between software modules. High coupling means that modules are closely connected and changes in one module may affect other modules. Low coupling means that modules are independent, and changes in one module have little impact on other modules.



1. Data Coupling - In data coupling, the components are independent of each other and communicate through data. E.g. addition by using call by value

2. Stamp Coupling - In stamp coupling, the complete data structure is passed from one module to another module. Therefore, it involves tramp data.
E.g. – Call By Reference

3. Control Coupling - If the modules communicate by passing control information, then they are said to be control coupled.

4. 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.

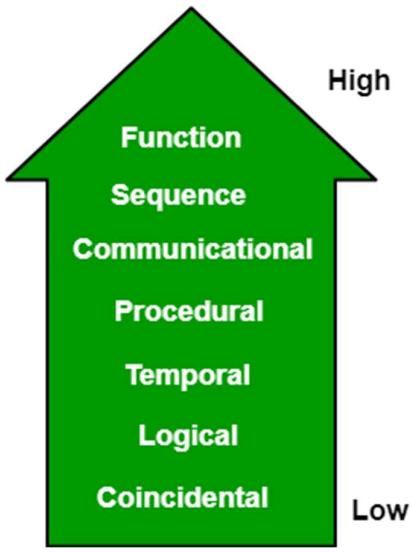
5. Common Coupling - The modules have shared data such as global data structures. E.g. – Synchronization issue between the processes.

6. 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.

One module is a part or context of another module.

Cohesion -

Cohesion refers to the degree to which elements within a module work together to fulfill a single, well-defined purpose. High cohesion means that elements are closely related and focused on a single purpose, while low cohesion means that elements are loosely related and serve multiple purposes.



1. 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.

2. Logical Cohesion – All the elements of a module perform similar or slightly similar operations , for e.g. if mouse , printer , scanner functions are written in the same module .

3. 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.

4. Procedural Cohesion – Here functions of a module are related to each other through a flow control i.e. they are part of an algorithm or procedure. Ex- calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA.

5. 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.

6. 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.

7. 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.

High Cohesion , Low Coupling Required

System Design Strategies

1. Top-Down Design

- Starts with a **high-level view** of the system.
- Gradually breaks down into **subsystems and components**.
- Suitable when developing **software from scratch**.

Advantages:

- Strong focus on requirements.
- Design is responsive to system goals.

Disadvantages:

- May miss component reuse opportunities.
- Might result in **over-complicated** architecture.

2. Bottom-Up Design

- Begins with **individual components**.
- Builds the system **from smaller parts** to a complete solution.
- Ideal for integrating with **existing systems**.

Advantages:

- Enables **reuse of general-purpose components**.
- Hides low-level implementation details.

Disadvantages:

- Less aligned with the overall problem structure.
- May create unnecessary or unused functionality.
- Harder to produce **high-quality solutions** directly.

Estimation Modal –

Here we try to estimate about two things time and cost of development.

Majorly estimation can be divided into four types :

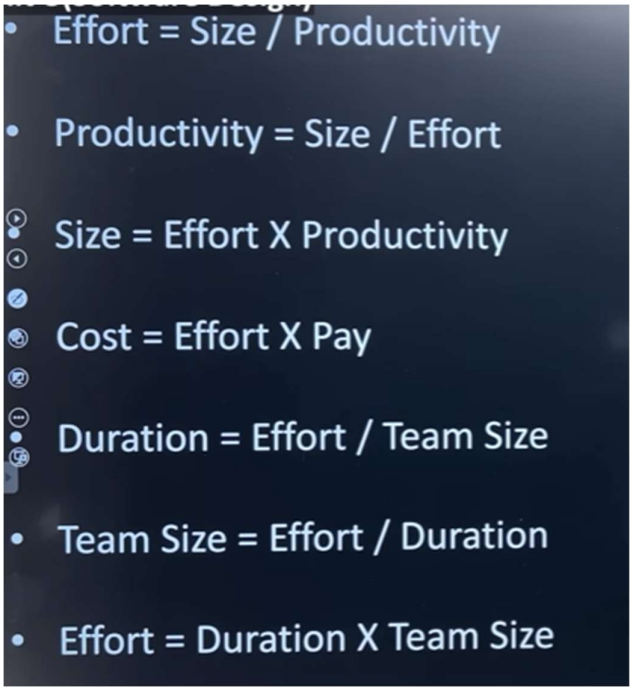
- Post Estimation
- Base Estimation
- Decomposition
- Empirical Modal

1.Post / Delayed Estimation –
In case of a friendly partly and known technology , we do not estimate either the time or the cost , because we know that cost will support as in every situation.

2. Base Estimation -
In this we predict the cost and time of the entire project based on the experience which we have gained from the previous projects.

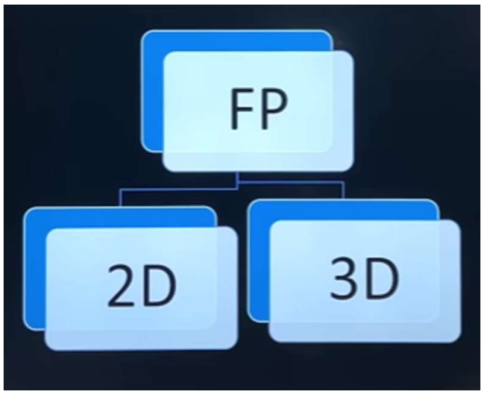
3. Decomposition Based Estimation -
It is used for large projects where decomposition of the problem into smaller problems is done , usually it is done on two bases .

Direct Estimation (White Box) : Size Oriented metrices (KLOC)
Indirect Estimation (Black Box) : Function oriented metrices (FP)



Indirect Estimation -
Here we use functional points to predict the cost of development . It is a better technique in general compare to kloc because it considers the logical complexity of the product , as in general it is not necessary that the larger the project more complex it will be code.

Here we say that the size of the software is directly dependent on the number and type of different functions it performs .



2D – in 2D We consider only information domain where we consider mainly five factors , as follows :

- No. of inputs : Each User Data input is counted
- No. of outputs : Output refers to reports , screen and error messages
- No. of inquiries : The No. of distinct interactive queries made by user which requires specific actions by system
- No. of files : Each logical file Either can be data structure or Physical Files
- No. of External interfaces : Data files on tapes , disk etc. and other interfaces that are used to transmit information to other systems are counted.

Measurement parameters	Simple	Average	Complex
i/p	3	4	6
o/p	4	5	7
Inquires	3	4	6
Files	7	10	15
External interface	5	7	10

total_count = Unadjusted Function Points (UFP) = ∑ (Number of each component × Weight)

In Adjustable case :
FP = total_count * EAF (Error adjustable factor / cost adjustment factor)
EAF = 0.65 + (0.01 × ∑fi)
in FP analysis 14 factors are indirectly involved to estimate the size of the software.

In 3D FP we consider only information domain where we consider mainly five factors , and along with function and behaviour domain.

Cyclomatic Complexity –

- It measures the number of linearly independent paths through a program's source code.
- A quantitative measure of independent paths in a program's control flow graph (CFG).
- Higher complexity = more paths = harder to test and maintain.
- Basic blocks = nodes; control transitions = edges.

Formula for Calculating Cyclomatic Complexity

General Formula:

M = E - N + 2P

Where:

- **E** = Number of edges
- **N** = Number of nodes
- **P** = Number of connected components (usually 1 for a single program/module)

For Strongly Connected Graph:

M = E - N + P

For a Single Module (P = 1):

M = E - N + 2

How to Calculate Cyclomatic Complexity?

Steps:

1. Construct Control Flow Graph (CFG)

2. Count edges (E), nodes (N), and components (P)
3. Apply the formula
4. Identify linearly independent paths
5. Design test cases for each path

Advantages

- Easy to apply and compute
- Guides testing with **minimum path coverage**
- Measures **control flow quality**
- Faster than some other metrics (e.g., Halstead)
- Pinpoints **testing focus areas**

Disadvantages

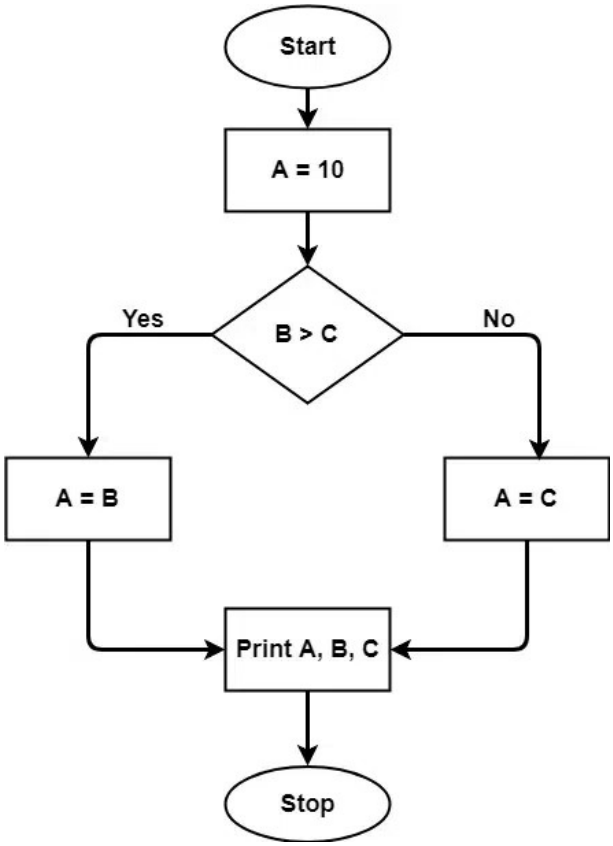
- Doesn't measure **data complexity**
- Can be misleading with **simple logic but high nesting**
- May not reflect actual **code understandability**
- Focuses only on **control structures**

Uses of Cyclomatic Complexity

- Identifies **independent execution paths**
- Ensures **adequate test coverage**
- Aids in **risk analysis**
- Helps with **code quality assurance**
- Useful in **early-stage development** for maintainability

Example

```
A = 10
IF B > C THEN
   A = B
ELSE
   A = C
ENDIF
Print A
Print B
Print C
```



Control Flow Graph Values:

- Nodes (N) = 7
- Edges (E) = 7
- M = 7 - 7 + 2 = 2

People Matrices

Definition:

People matrices are metrics used to evaluate and improve the performance, skills, morale, and productivity of individuals or teams involved in the software development process.

Goal:

To optimize human resource utilization, improve team collaboration, identify training needs, and retain top talent.

Detailed Examples:

Metric Name	Description
Productivity Rate	Lines of code or features delivered per developer per day.
Team Velocity (Agile)	Story points completed in a sprint; measures team output.
Defects per Developer	Helps identify training or process gaps in code quality.
Employee Turnover Rate	% of employees leaving over a certain period.
Training Hours per Employee	Time spent in upskilling (e.g., new tools or frameworks).
Peer Review Participation	% of reviews a person participates in; measures collaboration.
Satisfaction Surveys	Measures job satisfaction and engagement.

Why Important:

- High-performing teams deliver better products faster.
- Helps managers **align roles** with skills.
- Identifies **overloaded or underperforming** team members.
- Supports **career growth planning** and employee retention.

Process Matrices

Definition:

Process matrices are used to measure the efficiency, quality, and effectiveness of the software development lifecycle (SDLC).

Goal:

To evaluate and improve how software is being developed — from requirement gathering to deployment and maintenance.

Detailed Examples:

Metric Name	Description
Defect Removal Efficiency	% of defects removed before release vs. total defects found.
Cycle Time	Time taken for a task to move from "start" to "done".
Build Success Rate	% of successful builds; indicates code stability.
Review Effectiveness	% of defects found during reviews (vs. later in testing).
Rework Rate	% of work redone due to errors or misunderstandings.
Automation Coverage	% of test cases automated; improves process efficiency.

💡 **Why Important:**

- Helps identify process bottlenecks (e.g., slow reviews, long testing cycles).
- Reduces waste through **better planning and workflows**.
- Ensures **quality assurance** by improving defect detection early.
- Promotes **continuous improvement** in development practices.

Project Matrices

Definition:

Project matrices track how well a project is adhering to its **cost, time, scope, and resource** constraints.

Goal:

To manage and monitor **project health**, progress, budget, and timelines.

Detailed Examples:

Metric Name	Description
Schedule Variance (SV)	Difference between planned vs. actual completion dates.
Cost Variance (CV)	Planned cost vs. actual cost; detects overspending.
Earned Value (EV)	Value of work actually performed vs. planned.
Resource Utilization	% of time resources are actively working on tasks.
Risk Burndown Chart	Tracking of risks over time; lower is better.
Milestone Achievement Rate	% of milestones achieved on time.

Why Important:

- Keeps projects **on track and within budget**.
- Allows **early detection of delays or cost overruns**.
- Facilitates **communication with stakeholders** using data.
- Helps in **project forecasting and decision-making**.

Product Matrices

Definition:

Product matrices measure the **quality, reliability, usability, and performance** of the software product itself.

Goal:

To ensure that the **final delivered software** is robust, meets requirements, and performs well in real-world scenarios.

Detailed Examples:

Metric Name	Description
Number of Bugs per Release	How many bugs are reported per release or sprint.
Mean Time to Failure (MTTF)	Average time between system crashes or failures.
Code Coverage	% of code tested by automated or manual tests.
User Satisfaction Score	Ratings/feedback from end users.
Performance Metrics	Response time, CPU usage, memory consumption, etc.
Security Defects	Number of security vulnerabilities found.

Why Important:

- Directly affects **customer satisfaction and trust**.
- Reduces **maintenance cost and support tickets**.
- Ensures **product scalability and performance** under load.
- Measures **compliance with quality standards**.

Halstead's Software Metrics –

This provide a quantitative approach to analyzing software code based on its operators and operands. These metrics aim to measure the **complexity, effort, and maintainability** of software through a mathematical model.

Core Concept

A program is viewed as a collection of **tokens** categorized as:

- **Operators**: Symbols that perform operations (e.g., +, if, return)
- **Operands**: Variables or values that are manipulated (e.g., x, i, 5)

Basic Measures

Metric	Meaning
n1	Number of distinct operators
n2	Number of distinct operands
N1	Total occurrences of operators
N2	Total occurrences of operands

Derived Metrics

Metric	Formula	Description
Program Length (N)	N = N1 + N2	Total token count
Vocabulary (n)	n = n1 + n2	Total distinct tokens
Volume (V)	$V = N * \log_2(n)$	Size in bits required to represent the program
Difficulty (D)	D = (n1 / 2) * (N2 / n2)	How difficult the program is to understand
Effort (E)	E = D * V	Mental effort needed to write or understand the code
Time to Implement (T)	T = E / (18 * 60)	Time in minutes to implement (using Stroud's number)

Metric	Formula	Description
Program Level (L)	L = 1 / D or L = V* / V	Language abstraction level
Intelligence Content (I)	I = V / D	Represents inherent complexity or "smartness" of code

Other Formulas

- **Estimated Program Length (N^):**
  $N^ = n1 * \log_2(n1) + n2 * \log_2(n2)$
- **Potential Volume (V*):**
  $V^* = (2 + n2^*) * \log_2(2 + n2^*)$
  where n2* is count of unique I/O parameters.
- **Language Level (L'):**
  $L' = V / (D^2)$ or $L' = L * V^*$

Counting Rules for C Language

1. Comments are not considered.
2. The identifier and function declarations are not considered
3. All the variables and constants are considered operands.
4. Global variables used in different modules of the same program are counted as multiple occurrences of the same variable.
5. Local variables with the same name in different functions are counted as unique operands.
6. Functions calls are considered operators.

7. All looping statements e.g., do {…} while ( ), while ( ) {…}, for ( ) {…}, all control statements e.g., if ( ) {…}, if ( ) {…} else {…}, etc. are considered as operators.
8. In control construct switch ( ) {case:…}, switch as well as all the case statements are considered as operators.
9. The reserve words like return, default, continue, break, size, etc., are considered operators.
10. All the brackets, commas, and terminators are considered operators.
11. GOTO is counted as an operator and the label is counted as an operand.
12. The unary and binary occurrences of "+" and "-" are dealt with separately. Similarly "*" (multiplication operator) is dealt with separately.
13. In the array variables such as "array-name [index]" "array-name" and "index" are considered as operands and [ ] is considered as operator.
14. In the structure variables such as "struct-name, member-name" or "struct-name -> member-name", struct-name, and member-name are taken as operands, and '.', '->' are taken as operators. Some names of member elements in different structure variables are counted as unique operands.
15. All the hash directives are ignored.

```
int sort (int x[ ], int n)

{
    int i, j, save, im1;
    /*This function sorts array x in ascending order */
    If (n< 2) return 1;
    for (i=2; i< =n; i++)
    {
        im1=i-1;
        for (j=1; j< =im1; j++)
            if (x[i] < x[j])
            {
                Save = x[i];
                x[i] = x[j];
                x[j] = save;
            }
    }
    return 0;
}
```

Operators	Occurrences	Operands	Occurrences
int	4	sort	1
()	5	x	7
,	4	n	3
[]	7	i	8

Operators	Occurrences	Operands	Occurrences
if	2	j	7
<	2	save	3
;	11	im1	3
for	2	2	2
=	6	1	3
–	1	0	1
<=	2	–	–
++	2	–	–
return	2	–	–
{}	3	–	–
n1=14	N1=53	n2=10	N2=38

Advantages

- Language-independent and tool-friendly
- Helps in code comparison, complexity estimation, and effort prediction
- Simple to automate for static code analysis
- Useful for project planning, quality control, and maintenance estimation

Limitations

- Assumes equal weight for all operators/operands
- Not suited for non-procedural or object-oriented designs directly
- Doesn’t capture logical or structural complexity