Unit: 3

Introduction of Software Design Process –

Software Design is the process of transforming user requirements into a suitable form, which helps the programmer in software coding and implementation. During the software design phase, the design document is produced, based on the customer requirements as documented in the SRS document. Hence, this phase aims to transform the SRS document into a design document.

The following items are designed and documented during the design phase:

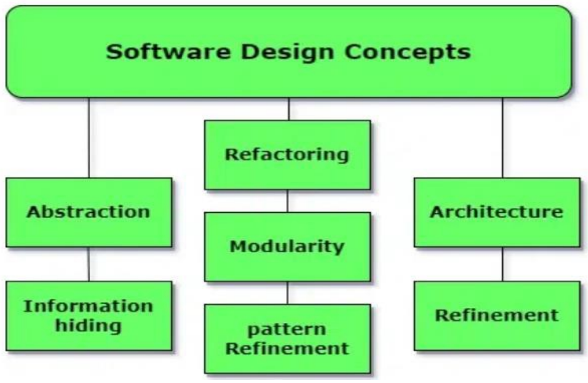
1. Different modules are required.
2. Control relationships among modules.
3. Interface among different modules.
4. Data structure among the different modules.
5. Algorithms are required to be implemented among the individual modules.

Objectives of Software Design

1. **Correctness:** A good design should be correct i.e., it should correctly implement all the functionalities of the system.
2. **Efficiency:** A good software design should address the resources, time, and cost optimization issues.
3. **Flexibility:** A good software design should have the ability to adapt and accommodate changes easily. It includes designing the software in a way, that allows for modifications, enhancements, and scalability without requiring significant rework or causing major disruptions to the existing functionality.
4. **Understandability:** A good design should be easily understandable, it should be modular, and all the modules are arranged in layers.
5. **Completeness:** The design should have all the components like data structures, modules, external interfaces, etc.
6. **Maintainability:** A good software design aims to create a system that is easy to understand, modify, and maintain over time. This involves using modular and well-structured design principles e.g.,(employing appropriate naming conventions and providing clear documentation). Maintainability in Software and design also enables developers to fix bugs, enhance features, and adapt the software to changing requirements without excessive effort or introducing new issues.

Software Design Concepts

Concepts are defined as a principal idea or invention that comes into our mind or in thought to understand something. The **software design concept** simply means the idea or principle behind the design. It describes how you plan to solve the problem of designing software, and the logic, or thinking behind how you will design software. It allows the software engineer to create the model of the system software or product that is to be developed or built. The software design concept provides a supporting and essential structure or model for developing the right software. There are many concepts of software design and some of them are given below:



Software Design Concepts

Points to be Considered While Designing Software

1. **Abstraction** (**Hide Irrelevant data**)**:** Abstraction simply means to hide the details to reduce complexity and increase 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.
2. **Modularity (subdivide the system):** Modularity simply means dividing the system or project into smaller parts to reduce the complexity of the system or project. In the same way, 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. So, modularity in design has now become a trend and is also important. If the system contains fewer components then it would mean the system is complex which requires a lot of effort (cost) but if we can divide the system into components then the cost would be small.
3. **Architecture (design a structure of something):** 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.
4. **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 a process of developing or presenting the software or system in a detailed manner which means elaborating a system or software. Refinement is very necessary to find out any error if present and then to reduce it.
5. **Pattern (a repeated form):** A 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.
6. **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.
7. **Refactoring (Reconstruct something):** Refactoring simply means reconstructing something in such a way that it does not affect the behavior of any other features. Refactoring in software design means reconstructing the design to reduce complexity and simplify it without impacting the behavior or its functions. Fowler has defined refactoring as "the process of changing a software system in a way that it won't impact the behavior of the design and improves the internal structure".

Different levels of Software Design

There are three different levels of software design. They are:

1. **Architectural Design:** The architecture of a system can be viewed as the overall structure of the system and 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.

2. **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.
3. **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.

Architectural Design in Software Engineering

Architectural design is the highest level of design activity in software engineering. It's the process of defining the fundamental structure of a software system, outlining its major components, their relationships, and how they interact with each other and with the external environment. Think of it as drawing the master blueprint for a building, detailing its main sections, their purpose, and how utilities flow through them, long before the walls are put up.

This phase is crucial because the architectural decisions made here have a profound impact on the software's quality attributes, such as performance, security, scalability, maintainability, and usability, which are often very difficult and costly to change later in the development cycle.

Why is Architectural Design Important?

- **Foundation for Development:** It provides a clear roadmap for the development team, guiding them on how to build the system's various parts.
- **Stakeholder Communication:** It serves as a common language for discussing the system's structure with various stakeholders (developers, clients, managers).
- **Quality Attributes:** It's where critical decisions are made to ensure non-functional requirements (e.g., performance, security) are met.
- **Complexity Management:** It helps in breaking down a complex system into manageable, independent components.
- **Risk Mitigation:** Early architectural assessment can identify potential technical risks and allow for proactive mitigation.
- **Reusability:** A well-defined architecture can promote the reuse of components and patterns across different projects.

Key Activities in Architectural Design

The process of creating an architectural design is often iterative and involves several key activities:

1. Understanding Requirements and Constraints

Before designing, a deep understanding of both functional (what the system does) and non-functional requirements (how well the system performs, its security needs, scalability, etc.) is essential. Constraints, such as budget, timeline, existing technologies, or regulatory compliance, also heavily influence design choices.

2. Identifying System Context

Define how the software system interacts with external entities (users, other systems, hardware). This helps in establishing clear boundaries and interfaces. Tools like **Context Diagrams** are often used here.

3. Decomposing the System (Structural View)

This involves breaking down the overall system into logical, cohesive, and loosely coupled components or subsystems. This step aims to manage complexity by dividing the problem into smaller, more manageable parts.

- **Example:** For an an e-commerce application, decomposition might yield components like User Management, Product Catalog, Shopping Cart, Order Processing, Payment Gateway Integration, and Reporting.

4. Defining Component Responsibilities and Interfaces

For each identified component, clearly define its specific responsibilities (what it does) and its interfaces (how it communicates with other components). Clear interfaces are vital for promoting modularity and reducing dependencies.

5. Selecting Architectural Styles/Patterns

Architects often draw upon established **architectural styles or patterns** that provide proven solutions to common design problems. Examples include:

- **Client-Server:** A central server provides resources and services to multiple client applications.
- **Layered Architecture:** Organizes components into horizontal layers, where each layer provides services to the layer above it and uses services from the layer below (e.g., Presentation, Business Logic, Data Access layers).

- **Microservices:** An application is built as a suite of small, independently deployable services, each running in its own process and communicating with lightweight mechanisms.
- **Event-Driven Architecture:** Components communicate by publishing and subscribing to events.
- **Model-View-Controller (MVC):** Separates application logic into three interconnected components: Model (data/business logic), View (user interface), and Controller (handles input).

The choice of style is heavily influenced by non-functional requirements and project context.

6. Designing Data Flow and Control Flow (Behavioural View)

Determine how data moves through the system and how control is passed between components. This involves considering synchronous versus asynchronous communication, message queues, and other integration patterns.

7. Documenting the Architecture

Architectural decisions must be clearly documented to serve as a reference for developers, testers, and maintainers. Common documentation artifacts include:

- **Architectural Design Document (ADD):** Describes the system's architecture, chosen patterns, design decisions, and rationale.
- **UML Diagrams:** Unified Modeling Language diagrams (e.g., Component Diagrams, Deployment Diagrams, Sequence Diagrams) are widely used to visualize architectural elements and interactions.
- **Architecture Decision Records (ADRs):** Short documents capturing a single architectural decision, its context, alternatives considered, and rationale.

8. Evaluating and Refining the Architecture

Once an initial architectural design is proposed, it undergoes rigorous evaluation. This may involve:

- **Reviews:** Formal or informal walkthroughs with stakeholders and peers to identify flaws, omissions, or potential issues.
- **Prototyping/Proof-of-Concept:** Building small, experimental implementations of critical or risky parts of the architecture to validate design choices.
- **Scenario-based Evaluation:** Assessing how well the architecture supports specific use cases or critical scenarios (e.g., "What happens if 10,000 users try to log in simultaneously?").
- **Trade-off Analysis:** Recognizing that architectural choices often involve trade-offs (e.g., performance vs. security, simplicity vs. flexibility) and making informed decisions.

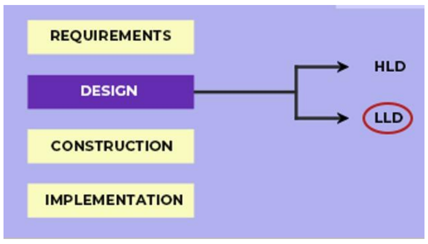
Conclusion

Architectural design is a foundational activity in software engineering. It requires a blend of technical expertise, problem-solving skills, and a deep understanding of project requirements and constraints. A well-conceived architecture sets the stage for successful development, leading to software that is not only functional but also adaptable, robust, and capable of meeting future demands. It is an iterative process, refined through constant evaluation and feedback, ensuring the software system is built on a solid and sustainable foundation.

Low Level Design or LLD?
Low-Level Design (LLD) plays a crucial role in software development, transforming high-level abstract concepts into detailed, actionable components that developers can use to build the system.

- LLD is the blueprint that guides developers on how to implement specific components of a system, such as classes, methods, algorithms, and data structures.
- Whether we are working on a micro service architecture, a web application, or a mobile app, understanding LLD is essential for building scalable, maintainable, and efficient systems.

How is LLD different from HLD?
While both **HLD (High-Level Design)** and **LLD (Low-Level Design)** are critical in system architecture, they serve different purposes.
1. High-Level Design (HLD): Focuses on the **overall architecture** of the system. It addresses questions such as which frameworks to use, what databases are suitable, how to integrate different components, and how the



system will function at a broader level.
2. Low-Level Design (LLD): Once the high-level structure is in place, **LLD takes over** by focusing on specific **components**, **modules**, and **interactions**. It provides detailed design diagrams (like UML diagrams) and breaks down **how each component** should behave, how it will interact with others, and what algorithms and data structures will be used.

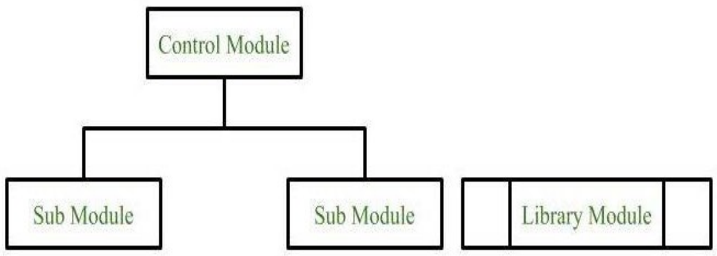
Structure Charts - Software Engineering
Structure Chart represents the hierarchical structure of modules. It breaks down the entire system into the lowest functional modules and describes the functions and sub-functions of each module of a system in greater detail. This article focuses on discussing Structure Charts in detail.
What is a Structure Chart?
Structure Chart partitions the system into black boxes (functionality of the system is known to the users, but inner details are unknown).

1. Inputs are given to the black boxes and appropriate outputs are generated.
2. Modules at the top level are called modules at low level.
3. Components are read from top to bottom and left to right.
4. When a module calls another, it views the called module as a black box, passing the required parameters and receiving results.

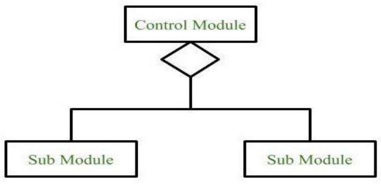
Symbols in Structured Chart
1. Module
It represents the process or task of the system. It is of three types:

- **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.

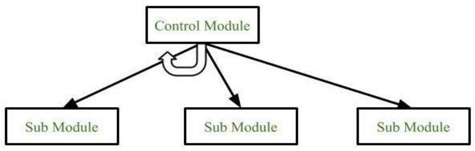
2. Conditional Call



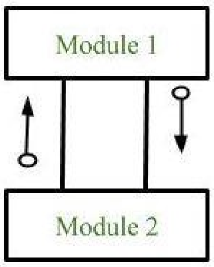
It represents that control module can select any of the sub module on the basis of some condition.



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.

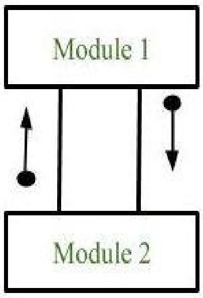


All the submodules cover by the loop repeat execution of module.
4. Data Flow
It represents the flow of data between the modules. It is represented by a directed arrow with an



empty circle at the end.
5. Control Flow

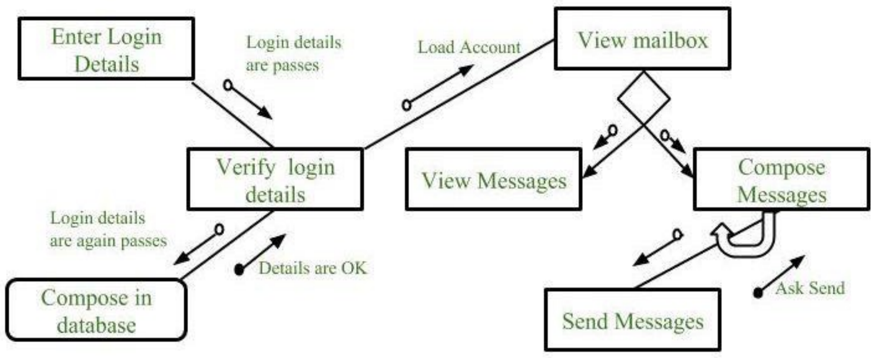
It represents the flow of control between the modules. It is represented by a directed arrow with a



filled circle at the end.
6. Physical Storage

Physical Storage

It is that where all the information are to be stored.
Example
Structure chart for an Email server



Types of Structure Chart

1. **Transform Centered Structure:** 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

What is PseudoCode

A **Pseudocode** is defined as a step-by-step description of an algorithm. Pseudocode does not use any programming language in its representation instead it uses the simple English language text as it is intended for human understanding rather than machine reading. Pseudocode is the **intermediate state between an idea and its implementation (code)** in a high-level language.

What is the need for Pseudo Code?

Pseudocode is an important part of designing an algorithm, it helps the programmer in planning the solution to the problem as well as the reader in understanding the approach to the problem. Pseudocode is an intermediate state between algorithm and program that plays supports the transition of the algorithm into the program.

Algorithm
Pseudocode
Program

Pseudocode is an intermediate state between algorithm and program How to

write Pseudocode?

Before writing the pseudocode of any algorithm the following points must be kept in mind.

- Organize the sequence of tasks and write the pseudocode accordingly.
- At first, establishes the main goal or the aim. **Example:**

This program will print first N numbers of Fibonacci series.

- Use standard programming structures such as **if-else**, **for**, **while**, and **cases** the way we use them in programming. Indent the statements if-else, for, while loops as they are indented in a program, it helps to comprehend the decision control and execution mechanism. It also improves readability to a great extent. **Example:**
- Check whether all the sections of a pseudo code are complete, finite, and clear to understand and comprehend. Also, explain everything that is going to happen in the actual code.
- Don't write the pseudocode in a programming language. It is necessary that the pseudocode is simple and easy to understand even for a layman or client, minimizing the use of technical terms.

Difference between Algorithm and Pseudocode

Algorithm	**Pseudocode**
An Algorithm is used to provide a solution to a particular problem in form of a well-defined step-based form.	A Pseudocode is a step-by-step description of an algorithm in code-like structure using plain English text.
An algorithm only uses simple English words	Pseudocode also uses reserved keywords like if-else, for, while, etc.
These are a sequence of steps of a solution to a problem	These are fake codes as the word pseudo means fake, using code like structure and plain English text
There are no rules to writing algorithms	There are certain rules for writing pseudocode

Algorithm	Pseudocode
Algorithms can be considered pseudocode	Pseudocode cannot be considered an algorithm
It is difficult to understand and interpret	It is easy to understand and interpret

Difference between Flowchart and Pseudocode

Flowchart	Pseudocode
A Flowchart is pictorial representation of flow of an algorithm.	A Pseudocode is a step-by-step description of an algorithm in code like structure using plain English text.
A Flowchart uses standard symbols for input, output decisions and start stop statements. Only uses different shapes like box, circle and arrow.	Pseudocode uses reserved keywords like if-else, for, while, etc.
This is a way of visually representing data, these are nothing but the graphical representation of the algorithm for a better understanding of the code	These are fake codes as the word pseudo means fake, using code like structure but plain English text instead of programming language
Flowcharts are good for documentation	Pseudocode is better suited for the purpose of understanding

Introduction to Flowcharts
The flowcharts are simple visual tools that help us understand and represent processes very easily. They use shapes like arrows, rectangles, and diamonds to show steps and decisions clearly. If someone is making a project or explaining a complex task, flowcharts can make complex ideas easier to understand.
Introduction to Flowcharts
Table of Content
What are Flowcharts?
Flowcharts are the visual representations of an algorithm or a process. Flowcharts use symbols/shapes like arrows, rectangles, and diamonds to properly explain the sequence of steps involved in the algorithm or process. Flowcharts have their use cases in various fields such as software development, business process modeling, and engineering.
Why use Flowcharts?
Flowcharts are used due to the numerous amount of benefits they provide. Below are some of the important reasons to use flowcharts:

- They provide clarity and simplification to the complex processes and algorithms, which in turn helps other people to understand them easily.
- Flowcharts provide a universal visual language that can be understood by anyone across different teams and helps reduce miscommunications.
- They are an optimal solution for documenting standard operating procedures, workflows, or business

processes. This makes it easier to train new employees.
- Flowcharts help in increasing the visualization of the problem being solved which enables more informed and data-driven choices.

Types of Flowcharts

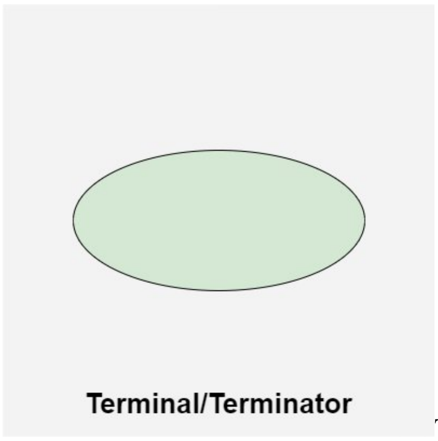
There are many types of flowcharts, each is designed to represent different kinds of processes and information. Some common types of flowcharts are:

- **Process Flowchart:** It represents the sequence of steps in a process. They are frequently used in business process modeling, manufacturing, and project management
- **Swimlane Flowchart:** It organizes the process into different lanes, each representing a different person or department and is used for illustrating how different teams or departments collaborate within a process
- **Workflow Diagram:** It represents how tasks, documents, or information move through a system and is commonly used in office processes or software development
- **Data Flow Diagram (DFD):** It focuses on detailing the inputs, processes, and outputs. Used in system design and analysis to model the flow of data within a system
- **Decision Flowchart:** It focuses on mapping out decision points within a process and the possible outcomes of each decision. It is used in decision-making scenarios

Symbols used in Flowchart Designs

1. Terminal/Terminator

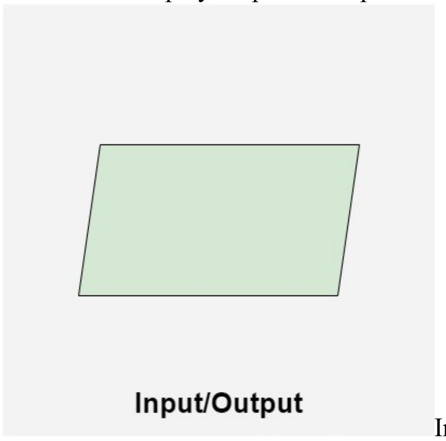
The oval symbol indicates Start, Stop and Halt in a program's logic flow. A pause/halt is generally used in a program logic under some error conditions. Terminal is the first and last symbols in the flowchart.



Terminal/Terminator

2. Input/Output

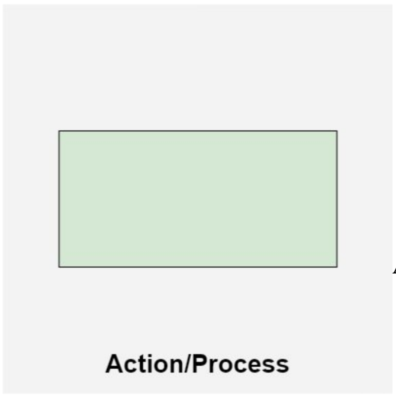
A parallelogram denotes any function of input/output type. Program instructions that take input from input devices and display output on output devices are indicated with parallelogram in a flowchart.



Input/Output

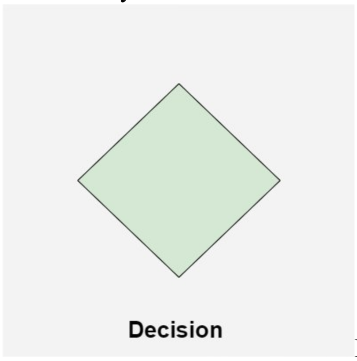
3. Action/Process

A box represents arithmetic instructions, specific action or operation that occurs as a part of the process. All arithmetic processes such as adding, subtracting, multiplication and division are indicated by action/process symbol.



Action/Process

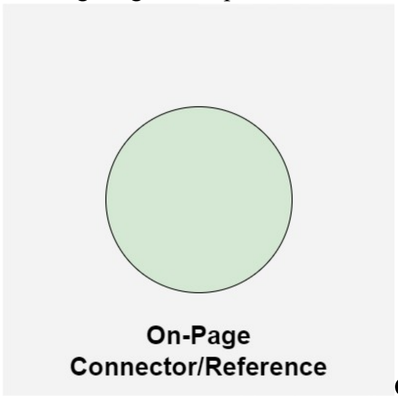
Diamond symbol represents a decision point. Decision based operations such as yes/no question or true/false are indicated by diamond in flowchart.



Decision

5. On-Page Connector/Reference

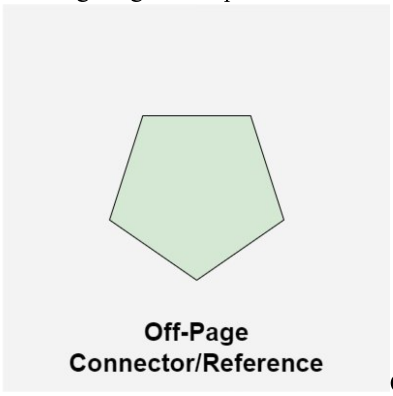
Whenever flowchart becomes complex or it spreads over more than one page, it is useful to use connectors to avoid any confusions. connectors are used to indicate a jump from one part of the flowchart to another without drawing long or complicated lines. On-Page Connector is represented by a small circle



On-Page Connector/Reference

6. Off-Page Connector/Reference

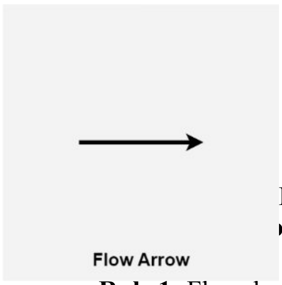
Whenever flowchart becomes complex or it spreads over more than one page, it is useful to use connectors to avoid any confusions. connectors are used to indicate a jump from one part of the flowchart to another without drawing long or complicated lines. Off-Page Connector is represented by a pentagon.



Off-Page Connector/Reference

7. Flow lines

Flow lines indicate the exact sequence in which instructions are executed. Arrows represent the direction of flow of control and relationship among different symbols of flowchart.



Flow lines

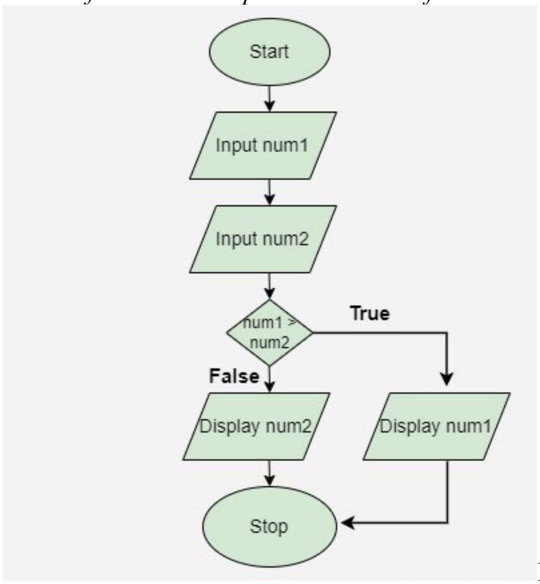
wchart

representation of an algorithm. It should follow some rules while creating a flowchart

- **Rule 1:** Flowchart opening statement must be 'start' keyword.
- **Rule 2:** Flowchart ending statement must be 'end' keyword.
- **Rule 3:** All symbols in the flowchart must be connected with an arrow line.
- **Rule 4:** Each decision point should have two or more distinct outcomes.
- **Rule 5:** Flow should generally move from top to bottom or left to right.

Example of a Flowchart

Draw a flowchart to input two numbers from the user and display the largest of two numbers.



Example Flowchart Below

is the explanation of the above flowchart:

- **Start**: The process begins with the **Start** symbol, indicating the start of the program.
- **Input num1**: The first number, represented as **num1**, is entered.
- **Input num2**: The second number, represented as **num2**, is entered.

- **Decision (num1 > num2)**: A decision point checks if **num1** is greater than **num2**.
    - If **True,** the process moves to the next step where **num1** will be displayed.
    - If **False**, the process moves to display **num2**.
- **Stop**: The process ends with the **Stop** symbol, signaling the conclusion of the program.

Advantages of using a Flowchart

- Flowcharts are a better way of communicating the logic of the system.
- Flowcharts act as a guide for blueprint during program designed.
- Flowcharts help in debugging process.
- With the help of flowcharts programs can be easily analyzed.
- They provide better documentation.
- Flowcharts serve as a good proper documentation.

Disadvantages of using a Flowchart

- It is difficult to draw flowcharts for large and complex programs.
- There is no standard to determine the amount of detail.
- It is very difficult to modify the Flowchart.
- Making a flowchart is costly.
- If changes are done in software, then the flowchart must be redrawn

System Design Strategy - Software Engineering

A good system design is to organize the program modules in such a way that they are easy to develop and change. Structured design techniques help developers to deal with the size and complexity of programs. Analysts create instructions for the developers about how code should be written and how pieces of code should fit together to form a program.

System Design Strategies

System design strategies provide well-structured methodologies to approach the complex task of designing systems. These strategies help in organizing the design process, ensuring that the resulting system is robust, scalable, and maintainable. By employing different design strategies, developers can address specific challenges and requirements effectively.

Structured Design

Structured Design is a traditional approach to system design that emphasizes the separation of a system into a hierarchy of functional modules. This strategy focuses on the flow of control and data through the system, using structured programming techniques to create clear and maintainable code.

Key Characteristics:

- **Modularity:** The system is divided into distinct modules, each with a specific function.
- **Top-Down Approach:** Design starts from the highest level of abstraction and breaks down into smaller, more detailed components.
- **Control Structures:** Uses structured control constructs such as sequences, decisions, and loops.

Advantages

1. Clarity and Simplicity (Promotes clear and straightforward code)
2. Ease of Debugging (Simplifies the process of identifying and fixing errors)
3. Reusability (Modules can be reused in different parts of the system)

Disadvantages

1. Rigidity (Top-down approach can be inflexible, making it difficult to accommodate changes)
2. Complexity in Large Systems (As the system grows, managing the interdependencies between modules can become complex)

Functional-Oriented Design

Functional-Oriented Design concentrates on functions or processes the system should perform. It focuses on the decomposition of the system into functional components whoch are responsible for a specific task.

Key Characteristics:

- **Function Decomposition:** The system is broken down into functions with the ability to play certain task.
- **Data Flow:** It focuses the flow of data and also how the data is transforming or changing.
- **Functional Independence:** Each function is designed to be as independent as possible, minimizing interdependencies.

Advantages:

- **Focus on Functionality:** This approach ensures that each part of the system is designed to perform a specific function efficiently.
- **Ease of Testing:** Functions can be tested independently, simplifying the testing process.

Disadvantages:

- **Potential for Redundancy:** Independent functions may lead to redundant code if not managed properly.
- **Integration Challenges:** Ensuring that all functions work together seamlessly can be challenging.
- **Modularity:** Promotes the creation of modular code, which can be reused and maintained easily.

Object-Oriented Design

Object-Oriented Design (OOD) is a design strategy that focuses on organizing software design around objects instead of functions and logic. Think of an object as a little package that holds both data and the methods that work with that data.

Key Characteristics:

- **Encapsulation:** Bundling data and methods that operate on the data within a single unit (object).
- **Inheritance:** Objects can inherit properties and methods from other objects, promoting code reuse.
- **Polymorphism:** The ability of objects to take on different forms, allowing for more flexible and dynamic code.
- **Abstraction:** Hiding complex implementation details and exposing only the necessary parts.

Advantages:

- Modularity and Reusability (Objects can be reused across different parts of the system or in different projects)
- Scalability (Object-oriented systems are easier to scale and modify as requirements change)
- Maintainability (Encapsulation and abstraction make the code easier to maintain and update)

Disadvantages:

- Complexity (Object-oriented design can introduce complexity, especially in large systems with many interacting objects)
- Learning Curve (Developers need to understand object-oriented concepts, which can be challenging for beginners)

Software Design Approaches

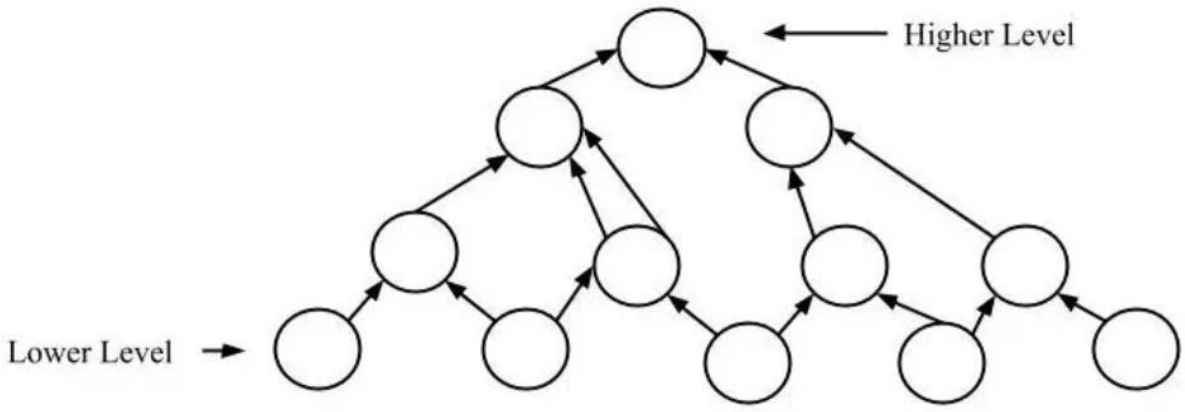
The design of a system is essentially a blueprint or a plan for a solution for the system. The design process for software systems often has two levels. At the first level the focus is on deciding which modules are needed for the system, the specifications of these modules and how the modules should be interconnected. The design of a system is correct if a system built precisely according to the design satisfies the requirements of that system. The goal of the design process is not simply to produce a design for the system. Instead, the goal is to find the best possible design within the limitations imposed by the requirements and the physical and social environment in which the system will operate.

There are several strategies that can be used to design software systems, including the following:

1. **Top-Down Design:** This strategy starts with a high-level view of the system and gradually breaks it down into smaller, more manageable components.
2. **Bottom-Up Design:** This strategy starts with individual components and builds the system up, piece by piece.
3. **Iterative Design:** This strategy involves designing and implementing the system in stages, with each stage building on the results of the previous stage.
4. **Incremental Design:** This strategy involves designing and implementing a small part of the system at a time, adding more functionality with each iteration.
5. **Agile Design:** This strategy involves a flexible, iterative approach to design, where requirements and design evolve through collaboration between self-organizing and cross-functional teams.

Bottom-Up Approach

The design starts with the lowest level components and subsystems. By using these components, the next immediate higher-level components and subsystems are created or composed. The process is continued till all the components and subsystems are composed into a single component, which is considered as the complete system. The amount of abstraction grows high as the design moves to more high levels.



Advantages of Bottom-up Approach

- The economics can result when general solutions can be reused.
- It can be used to hide the low-level details of implementation and be merged with the top-down technique.
- Simplifies the integration process by ensuring that low-level components are thoroughly tested and validated before being combined into higher-level modules.

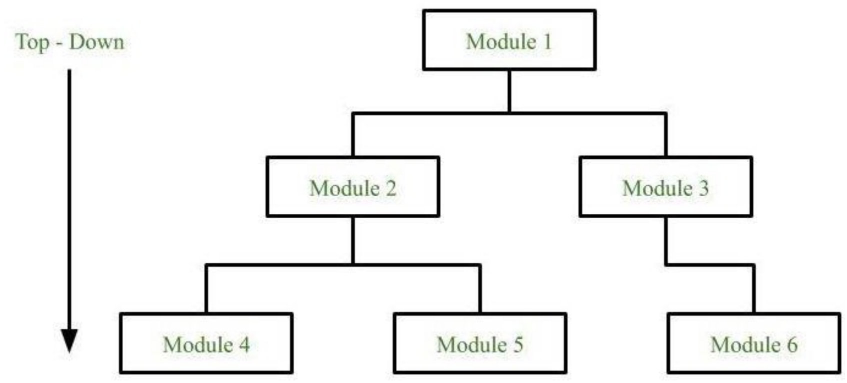
Disadvantages of Bottom-up Approach

- It is not so closely related to the structure of the problem.
- High-quality bottom-up solutions are very hard to construct.
- It leads to the proliferation of 'potentially useful' functions rather than the most appropriate ones.

Top-Down Approach

Each system is divided into several subsystems and components. Each of the subsystems is further divided into a set of subsystems and components. This process of division facilitates forming a system hierarchy structure. The complete software system is considered a single entity and in relation to the characteristics, the system is split into sub-systems and components. The same is done with each of the sub-systems.

This process is continued until the lowest level of the system is reached. The design is started initially by defining the system as a whole and then keeps on adding definitions of the subsystems and components. When all the definitions are combined, it turns out to be a complete system.



Top-down approach

Advantages of Top-Down Approach

- The main advantage of the top-down approach is that its strong focus on requirements helps to make a design responsive according to its requirements.
- Simplifies complex problem-solving by breaking down the system into smaller sub-problems.
- Enhances clarity and understanding with a high-level overview.

Disadvantages of Top-Down Approach

- Project and system boundaries tend to be application specification-oriented. Thus, it is more likely that the advantages of component reuse will be missed.
- The system is likely to miss, the benefits of a well-structured, simple architecture.
- It is a combination of both top-down and bottom-up design strategies. In this, we can reuse the modules.

Advantages of using a System Design Strategy

1. **Improved quality:** A well-designed system can improve the overall quality of the software, as it provides a clear and organized structure for the software.
2. **Ease of maintenance:** A well-designed system can make it easier to maintain and update the software, as the design provides a clear and organized structure for the software.
3. **Improved efficiency:** A well-designed system can make the software more efficient, as it provides a clear and organized structure for the software that reduces the complexity of the code.
4. **Better communication:** A well-designed system can improve communication between stakeholders, as it provides a clear and organized structure for the software that makes it easier for stakeholders to understand and agree on the design of the software.
5. **Faster development:** A well-designed system can speed up the development process, as it provides a clear and organized structure for the software that makes it easier for developers to understand the requirements and implement the software.

Disadvantages of using a System Design Strategy

1. **Time-consuming:** Designing a system can be time-consuming, especially for large and complex systems, as it requires a significant amount of documentation and analysis.
2. **Inflexibility:** Once a system has been designed, it can be difficult to make changes to the design, as the process is often highly structured and documentation-intensive.
3. **Resource-Intensive:** Requires substantial resources, including skilled personnel, time, and tools, which can strain budgets and timelines.

Halstead's Software Metrics - Software Engineering

Halstead's Software metrics are a set of measures proposed by Maurice Halstead to evaluate the complexity of a software program. These metrics are based on the number of distinct operators and operands in the program and are used to estimate the effort required to develop and maintain the program. These metrics provide a quantitative assessment of software complexity, aiding in software development, maintenance, and quality assurance processes. They include measures such as program length, vocabulary, volume, difficulty, and effort, calculated based on the number of unique operators and operands in a program. Halstead's metrics help developers understand and manage software complexity, identify potential areas for optimization, and improve overall software quality.

What is Halstead's Software Metrics?

Halstead's Software Metrics, developed by Maurice Halstead in 1977, are a set of measures used to quantify various aspects of software programs. According to Halstead's, "A computer program is an implementation of an algorithm considered to be a collection of tokens which can be classified as either operators or operand". This means that the program consists of various symbols and data elements that are either performing actions (operators) or upon which actions are performed (operands). This distinction helps in understanding and analyzing the structure and behavior of the program.

Token Count

In Halstead's Software metrics, a computer program is defined as a collection of tokens that can be described as operators or operands. These tokens are used to analyze the complexity and volume of a program. Operators are symbols that represent actions, while operands are the entities on which the operators act. All software science metrics can be specified using these basic symbols. These symbols are referred to as tokens. By counting and analyzing these tokens, Halstead's metrics provide insights into the complexity, effort, and quality of software code.

In Halstead's Software Metrics:

n1 = Number of distinct operators. n2

= Number of distinct operands.

N1 = Total number of occurrences of operators. N2

= Total number of occurrences of operands. Global variables used in different modules of the same program are counted as multiple occurrences of the same variable.

1. Local variables with the same name in different functions are counted as unique operands.
2. Functions calls are considered operators.
3. All looping statements e.g., do {…} while ( ), while ( ) {…}, for ( ) {…}, all control statements e.g., if ( ) {…}, if ( ) {…} else {…}, etc. are considered as operators.
4. In control construct switch ( ) {case:…}, switch as well as all the case statements are considered as operators.
5. The reserve words like return, default, continue, break, size, etc., are considered operators.
6. All the brackets, commas, and terminators are considered operators.
7. GOTO is counted as an operator and the label is counted as an operand.
8. The unary and binary occurrences of "+" and "-" are dealt with separately. Similarly "*" (multiplication operator) is dealt with separately.
9. In the array variables such as "array-name [index]" "array-name" and "index" are considered as operands and [ ] is considered as operator.
10. 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.
11. All the hash directives are ignored.

Let's examine the following C program

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;
}
```

Explanation

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

Operators	Occurrences	Operands	Occurrences
n1=14	N1=53	n2=10	N2=38

Here are the calculated Halstead metrics for the given C program:

Program Length (N) = 91 Vocabulary (n) = 24 Volume (V) = 417.23 bits Estimated Program Length (N^) = 86.51 Unique Operands Used as Both Input and Output (n2 = 3 (x: array holding integer to be sorted. This is used both as input and output) Potential Volume (V*) = 11.6 Program Level (L) = 0.027 Difficulty (D) = 37.03 Estimated Program Level (L^) = 0.038 Effort (T) = 610 seconds*

Advantages of Halstead Metrics

- It is simple to calculate.
- It measures the overall quality of the programs.
- It predicts the rate of error.
- It predicts maintenance effort.
- It does not require a full analysis of the programming structure.
- It is useful in scheduling and reporting projects.
- It can be used for any programming language.
- Easy to use: The metrics are simple and easy to understand and can be calculated quickly using automated tools.
- Quantitative measure: The metrics provide a quantitative measure of the complexity and effort required to develop and maintain a software program, which can be useful for project planning and estimation.
- Language independent: The metrics can be used for different programming languages and development environments.
- Standardization: The metrics provide a standardized way to compare and evaluate different software programs.

Disadvantages of Halstead Metrics

- It depends on the complete code.
- It has no use as a predictive estimating model.
- Limited scope: The metrics focus only on the complexity and effort required to develop and maintain a

software program, and do not take into account other important factors such as reliability, maintainability, and usability.

- Limited applicability: The metrics may not be applicable to all types of software programs, such as those with a high degree of interactivity or real-time requirements.
- Limited accuracy: The metrics are based on a number of assumptions and simplifications, which may limit their accuracy in certain situations.

Conclusion

Halstead's software metrics offer a quantitative approach to assessing a program's complexity. They include measures like program length (total operators and operands), vocabulary (unique operators and operands), and volume (a measure of the program's size in bits). These metrics help understand the diversity and size of the code, aiding in evaluating its complexity and potential maintainability. Overall, Halstead's metrics provide valuable insights into the code's structure and complexity.

What is Functional Point Analysis?

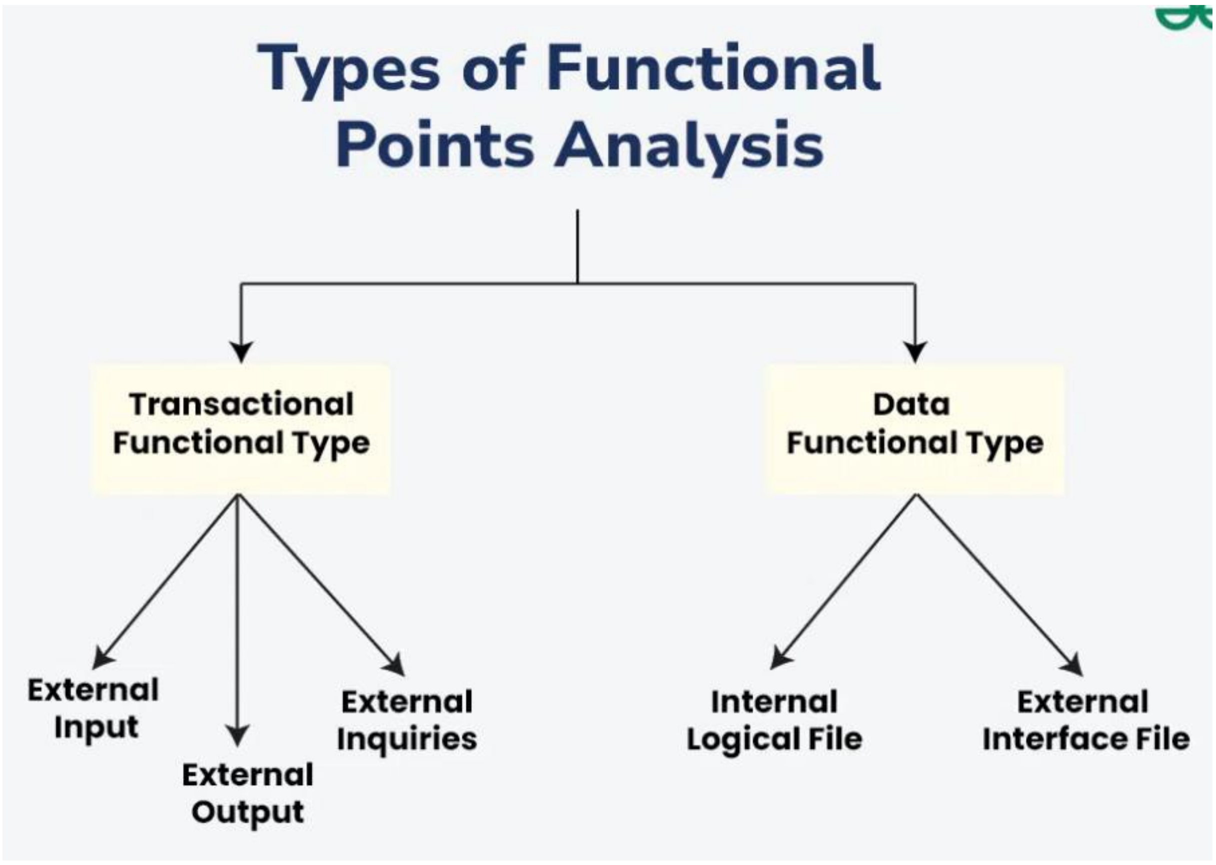
Function Point Analysis was initially developed by Allan J. Albrecht in 1979 at IBM and has been further modified by the **International Function Point User's Group (IFPUG) i**n 1984, to clarify rules, establish standards, and encourage their use and evolution. Allan J. Albrecht gave the initial definition, **Functional Point** Analysis gives a dimensionless number defined in function points which we have found to be an effective relative measure of function value delivered to our customer. A systematic approach to measuring the different functionalities of a software application is offered by function point metrics. Function point metrics evaluate functionality from the perspective of the user, that is, based on the requests and responses they receive.

Objectives of Functional Point Analysis

1. **Encourage Approximation:** FPA helps in the estimation of the work, time, and materials needed to develop a software project. Organizations can plan and manage projects more accurately when a common measure of functionality is available.
2. **To assist with project management:** Project managers can monitor and manage software development projects with the help of FPA. Managers can evaluate productivity, monitor progress, and make well-informed decisions about resource allocation and project timeframes by measuring the software's functional points.
3. **Comparative analysis:** By enabling benchmarking, it gives businesses the ability to assess how their software projects measure up to industry standards or best practices in terms of size and complexity. This can be useful for determining where improvements might be made and for evaluating how well development procedures are working.
4. **Improve Your Cost-Benefit Analysis:** It offers a foundation for assessing the value provided by the program concerning its size and complexity, which helps with cost-benefit analysis. Making educated judgements about project investments and resource allocations can benefit from having access to this information.
5. **Comply with Business Objectives:** It assists in coordinating software development activities with an organization's business objectives. It guarantees that software development efforts are directed toward providing value to end users by concentrating on user-oriented functionality.

Types of Functional Point Analysis

There are two types of Functional Point Analysis:



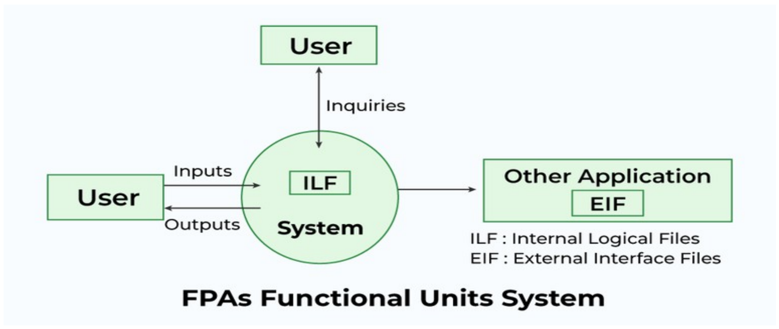
Types of Functional Point Analysis

1. Transactional Functional Type

- **External Input (EI):** EI processes data or control information that comes from outside the application's boundary. The EI is an elementary process.
- **External Output (EO):** EO is an elementary process that generates data or control information sent outside the application's boundary.
- **External Inquiries (EQ):** EQ is an elementary process made up of an input-output combination that results in data retrieval.

2. Data Functional Type

- **Internal Logical File (ILF):** A user-identifiable group of logically related data or control information maintained within the boundary of the application.
- **External Interface File (EIF):** A group of users recognizable logically related data allusion to the software but maintained within the boundary of another software.


User
Inquiries
Inputs
User
Outputs
ILF
System
Other Application
EIF
ILF : Internal Logical Files
EIF : External Interface Files
FPAs Functional Units System
Functional Point Analysis

Benefits of Functional Point Analysis

Following are the benefits of Functional Point Analysis:

1. **Technological Independence:** It calculates a software system's functional size independent of the underlying technology or programming language used to implement it. As a result, it is a technology-neutral metric that makes it easier to compare projects created with various technologies.
2. **Better Accurate Project Estimation:** It helps to improve project estimation accuracy by measuring user interactions and functional needs. Project managers can improve planning and budgeting by using the results of the FPA to estimate the time, effort and resources required for development.
3. **Improved Interaction:** It provides a common language for business analysts, developers, and project managers to communicate with one another and with other stakeholders. By communicating the size and complexity of software in a way that both technical and non-technical audiences can easily understand this helps close the communication gap.
4. **Making Well-Informed Decisions:** FPA assists in making well-informed decisions at every stage of the software development life cycle. Based on the functional requirements, organizations can use the results of the FPA to make decisions about resource allocation, project prioritization, and technology selection.
5. **Early Recognition of Changes in Scope**: Early detection of changes in project scope is made easier with the help of FPA. Better scope change management is made possible by the measurement of functional requirements, which makes it possible to evaluate additions or changes for their effect on the project's overall size.

Disadvantage of Functional Point Analysis

Given below are some disadvantages of Functional Point Analysis:

1. **Subjective Judgement**: One of the main disadvantages of Functional Point Analysis is it's dependency on subjective judgement i.e. relying on personal opinions and interpretations instead of just using clear, measurable standards.
2. **Low Accuracy:** It has low evaluation accuracy as it's dependency on subjective judgement.
3. **Time Consuming:** Functional Point Analysis is a time consuming process, particularly during the initial stages of implementation.
4. **Steep Learning Curve:** Learning FPA can be challenging due to its complexity and the length of time required to gain proficiency.
5. **Less Research Data:** Compared to LOC-based metrics, there is relatively less research data available on function points.
6. **Costly:** The need for thorough analysis and evaluation can result in increased project timelines and associated costs.

Characteristics of Functional Point Analysis

- **We can calculate the functional point** with the help of the number of functions and types of functions used in applications. These are classified into five types:

Types of FP Attributes or Information Domain Characteristics

Measurement Parameters	**Examples**
Number of External Inputs (EI)	Input screen and tables
Number of External Output (EO)	Output screens and reports
Number of external inquiries (EQ)	Prompts and interrupts
Number of internal files (ILF)	Databases and directories
Number of external interfaces (EIF)	Shared databases and shared routines

- Functional Point helps in describing system complexity and also shows project timelines.
- It is majorly used for business systems like information systems.
- FP is language and technology independent, meaning it can be applied to software systems developed using any programming language or technology stack.
- All the factors mentioned above are given weights, and these weights are determined through practical experiments in the following table.

What is Cyclomatic Complexity?

The cyclomatic complexity of a code section is the quantitative measure of the number of linearly independent paths in it. It is a software metric used to indicate the complexity of a program. It is computed using the control flow graph of the program. The nodes in the graph indicate the smallest group of commands of a program, and a directed edge in it connects the two nodes i.e. if the second command might immediately follow the first command.

For example, if the source code contains no control flow statement then its cyclomatic complexity will be 1, and the source code contains a single path in it. Similarly, if the source code contains one **if condition** then cyclomatic complexity will be 2 because there will be two paths one for true and the other for false.