Project Evaluation and Review Technique (PERT)

Project Evaluation and Review Technique (PERT) is a procedure through which activities of a project are represented in its appropriate sequence and timing. It is a scheduling technique used to schedule, organize and integrate tasks within a project. PERT is basically a mechanism for management planning and control which provides blueprint for a particular project. All of the primary elements or events of a project have been finally identified by the PERT.
In this technique, a PERT Chart is made which represent a schedule for all the specified tasks in the project. The reporting levels of the tasks or events in the PERT Charts is somewhat same as defined in the work breakdown structure (WBS).

Characteristics of PERT:
The main characteristics of PERT are as following :

- It serves as a base for obtaining the important facts for implementing the decision-making.
- It forms the basis for all the planning activities.
- PERT helps management in deciding the best possible resource utilization method.
- PERT take advantage by using time network analysis technique.
- PERT presents the structure for reporting information.
- It helps the management in identifying the essential elements for the completion of the project within time.

Advantages of PERT

It has the following advantages :

- Estimation of completion time of project is given by the PERT.
- It supports the identification of the activities with slack time.
- The start and dates of the activities of a specific project is determined.
- It helps project manager in identifying the critical path activities.
- PERT makes well organized diagram for the representation of large amount of data.

Disadvantages of PERT

It has the following disadvantages :

- The complexity of PERT is more which leads to the problem in implementation.
- The estimation of activity time are subjective in PERT which is a major disadvantage.
- Maintenance of PERT is also expensive and complex.
- The actual distribution of may be different from the PERT beta distribution which causes wrong assumptions.
- It under estimates the expected project completion time as there is chances that other paths can become the critical path if their related activities are deferred.

Coupling and Cohesion

The purpose of Design phase in the Software Development Life Cycle is to produce a solution to a problem given in the SRS(Software Requirement Specification) document. The output of the design phase is Software Design Document (SDD).

Coupling and Cohesion are two key concepts in software engineering that are used to measure the quality of a software system's design.

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.

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.

Both coupling and cohesion are important factors in determining the maintainability, scalability, and reliability of a software system. High coupling and low cohesion can make a system difficult to change and test, while low coupling and high cohesion make a system easier to maintain and improve.

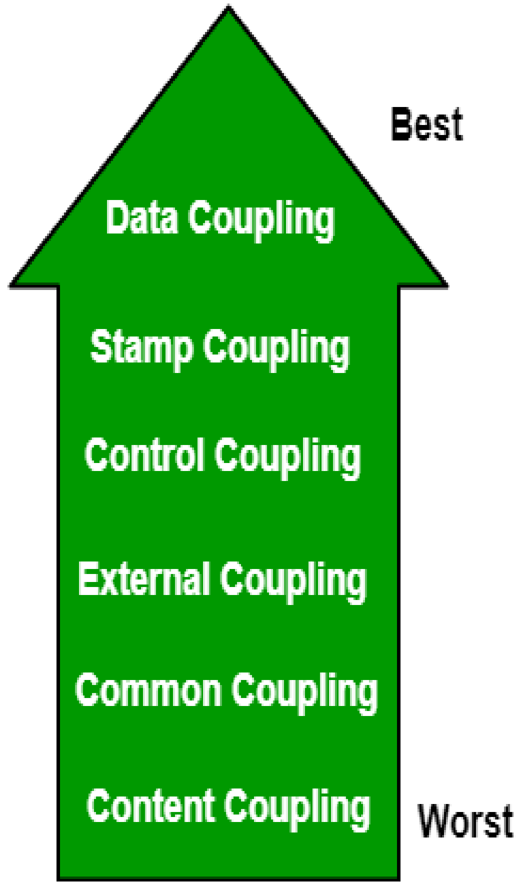
Modularization: Modularization is the process of dividing a software system into multiple independent modules where each module works independently. There are many advantages of Modularization in software engineering. Some of these are given below:

- Easy to understand the system.
- System maintenance is easy.
- A module can be used many times as their requirements. No need to write it again and again

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

Types of Coupling:

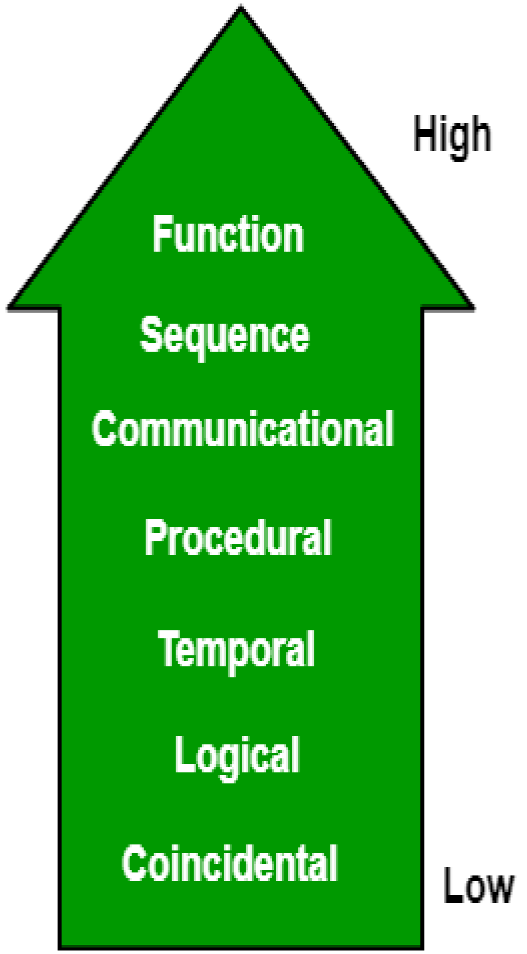
- **Data Coupling:** If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent of each other and communicate through data. Module communications don't contain tramp 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:

- **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.



ADVANTAGES OR DISADVANTAGES:

Advantages of low coupling:

- Improved maintainability: Low coupling reduces the impact of changes in one module on other modules, making it easier to modify or replace individual components without affecting the entire system.
- Enhanced modularity: Low coupling allows modules to be developed and tested in isolation, improving the modularity and reusability of code.
- Better scalability: Low coupling facilitates the addition of new modules and the removal of existing ones, making it easier to scale the system as needed.

Advantages of high cohesion:

- Improved readability and understandability: High cohesion results in clear, focused modules with a single, well-defined purpose, making it easier for developers to understand the code and make changes.
- Better error isolation: High cohesion reduces the likelihood that a change in one part of a module will affect other parts, making it easier to
- isolate and fix errors. Improved reliability: High cohesion leads to modules that are less prone to errors and that function more consistently,
- leading to an overall improvement in the reliability of the system.

Disadvantages of high coupling:

- Increased complexity: High coupling increases the interdependence between modules, making the system more complex and difficult to understand.
- Reduced flexibility: High coupling makes it more difficult to modify or replace individual components without affecting the entire system.
- Decreased modularity: High coupling makes it more difficult to develop and test modules in isolation, reducing the modularity and reusability of code.

Disadvantages of low cohesion:

- Increased code duplication: Low cohesion can lead to the duplication of code, as elements that belong together are split into separate modules.
- Reduced functionality: Low cohesion can result in modules that lack a clear purpose and contain elements that don’t belong together, reducing their functionality and making them harder to maintain.
- Difficulty in understanding the module: Low cohesion can make it harder for developers to understand the purpose and behavior of a module, leading to errors and a lack of clarity.

Differentiate between Coupling and Cohesion

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

Re-engineering

Software Re-engineering is a process of software development which is done to improve the maintainability of a software system. Re-engineering is the examination and alteration of a system to reconstitute it in a new form. This process encompasses a combination of sub-processes like reverse engineering, forward engineering, reconstructing etc.

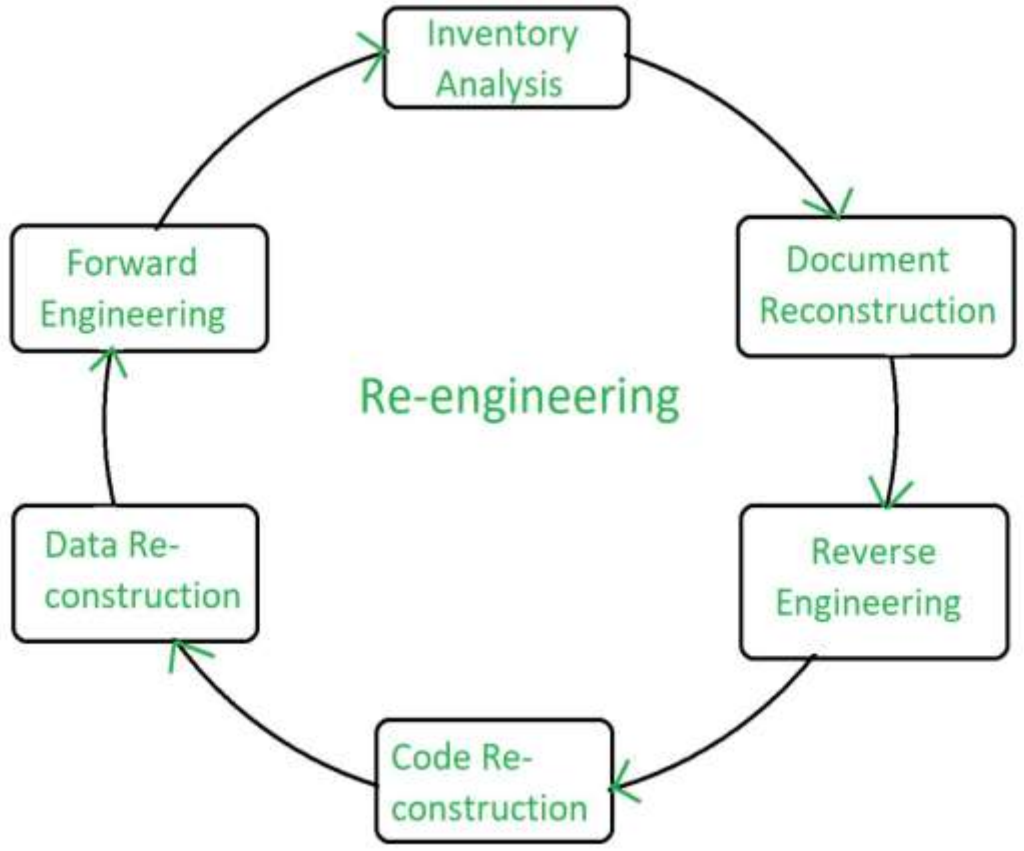
Re-engineering is the reorganizing and modifying existing software systems to make them more maintainable.

Objectives of Re-engineering:

- To describe a cost-effective option for system evolution.
- To describe the activities involved in the software maintenance process.
- To distinguish between software and data re-engineering and to explain the problems of data re-engineering.

Steps involved in Re-engineering:

- Inventory Analysis
- Document Reconstruction
- Reverse Engineering
- Code Reconstruction
- Data Reconstruction
- Forward Engineering



Re-engineering Cost Factors:

- The quality of the software to be re-engineered
- The tool support available for re-engineering
- The extent of the required data conversion
- The availability of expert staff for re-engineering

Advantages of Re-engineering:

- **Reduced Risk:** As the software is already existing, the risk is less as compared to new software development. Development problems, staffing problems and specification problems are the lots of problems which may arise in new software development.
- **Reduced Cost:** The cost of re-engineering is less than the costs of developing new software.
- **Revelation of Business Rules:** As a system is re-engineered , business rules that are embedded in the system are rediscovered.
- **Better use of Existing Staff:** Existing staff expertise can be maintained and extended accommodate new skills during re-engineering.

Disadvantages of Re-engineering:

- Practical limits to the extent of re-engineering.
- Major architectural changes or radical reorganizing of the systems data management has to be done manually.
- Re-engineered system is not likely to be as maintainable as a new system developed using modern software Re-engineering methods.

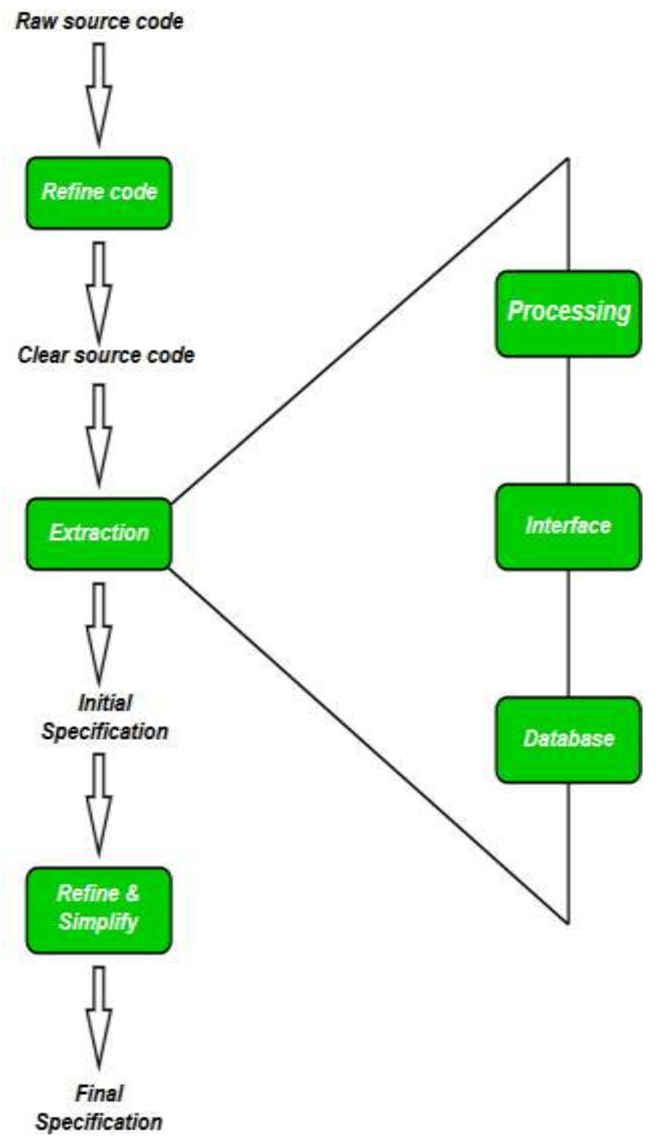
Reverse Engineering

Software Reverse Engineering is a process of recovering the design, requirement specifications and functions of a product from an analysis of its code. It builds a program database and generates information from this.

The purpose of reverse engineering is to facilitate the maintenance work by improving the understandability of a system and to produce the necessary documents for a legacy system.

Reverse Engineering Goals:

- Cope with Complexity.
- Recover lost information.
- Detect side effects.
- Synthesise higher abstraction.
- Facilitate Reuse.



Steps of Software Reverse Engineering:

- **Collection Information:**
  This step focuses on collecting all possible information (i.e., source design documents etc.) about the software.
- **Examining the information:**
  The information collected in step-1 as studied so as to get familiar with the system.
- **Extracting the structure:**
  This step concerns with identification of program structure in the form of structure chart where each node corresponds to some routine.
- **Recording the functionality:**
  During this step processing details of each module of the structure, charts are recorded using structured language like decision table, etc.
- **Recording data flow:**
  From the information extracted in step-3 and step-4, set of data flow diagrams are derived to show the flow of data among the processes.
- **Recording control flow:**
  High level control structure of the software is recorded.
- **Review extracted design:**
  Design document extracted is reviewed several times to ensure consistency and correctness. It also ensures that the design represents the program.
- **Generate documentation:**
  Finally, in this step, the complete documentation including SRS, design document, history, overview, etc. are recorded for future use.

Reverse Engineering Tools:

Reverse engineering if done manually would consume lot of time and human labour and hence must be supported by automated tools. Some of tools are given below:

- **CIAO and CIA:** A graphical navigator for software and web repositories along with a collection of Reverse Engineering tools.
- **Rigi:** A visual software understanding tool.
- **Bunch:** A software clustering/modularization tool.
- **GEN++:** An application generator to support development of analysis tools for the C++ language.
- **PBS:** Software Bookshelf tools for extracting and visualizing the architecture of programs.

Forward Engineering:

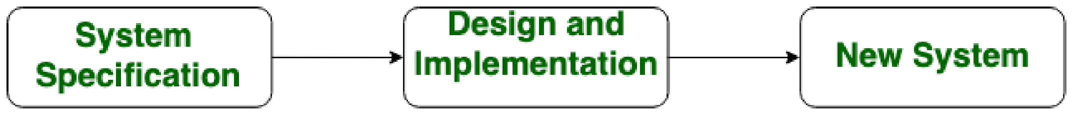
Forward Engineering is a method of creating or making an application with the help of the given requirements. Forward engineering is also known as Renovation and Reclamation. Forward engineering requires high proficiency skills. It takes more time to construct or develop an application. Forward engineering is a technique of creating high-level models or designs to make in complexities and low-level information. Therefore this kind of engineering has completely different principles in numerous package and information processes. Forward Engineering applies of all the software engineering process which contains SDLC to recreate associate existing application. It is near to full fill new needs of the users into re-engineering.

Characteristics of forward engineering:

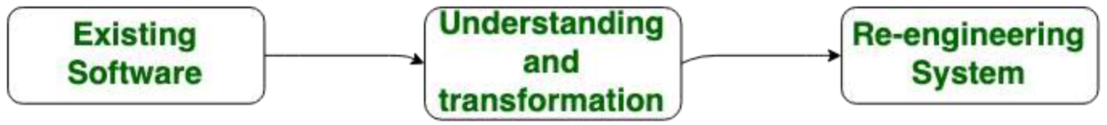
- Forward engineering is a variety of engineering that has different principles in numerous package and information processes.
- Forward engineering is vital in IT as a result of it represents the 'normal' development process.
- Forward engineering deals with the conversion of business processes, services, and functions into applications.
- In this method business model is developed first. Then, a top-to-down approach is followed to urge the package from the model developed.
- Forward engineering tools are accustomed move from implementation styles and logic to the event of supply code.
- It essentially permits the user to develop a business model which may then be translated into data system components.
- These tools basically follow the top-to down approach. System creator and visual Analyst is a forward engineering CASE tool.

Difference between Forward Engineering and Reverse Engineering:

S.NO	Forward Engineering	Reverse Engineering
1.	In forward engineering, the application are developed with the given requirements.	In reverse engineering or backward engineering, the information are collected from the given application.
2.	Forward Engineering is a high proficiency skill.	Reverse Engineering or backward engineering is a low proficiency skill.
3.	Forward Engineering takes more time to develop an application.	While Reverse Engineering or backward engineering takes less time to develop an application.
4.	The nature of forward engineering is Prescriptive.	The nature of reverse engineering or backward engineering is Adaptive.
5.	In forward engineering, production is started with given requirements.	In reverse engineering, production is started by taking the existing products.
6.	The example of forward engineering is the construction of electronic kit, construction of DC MOTOR , etc.	An example of backward engineering is research on Instruments etc.



Forward Engineering



Reverse Engineering

System Investigation

System Investigation is the process of finding out what the system is being built to do and if the system is feasible.

Preliminary investigation is the first step in the system development project. It is a way of handling the user's request to change, improve or enhance an existing system. System investigation includes the following two stages:

1. Problem definition
2. Feasibility study

1. Problem definition:

The first responsibility of a system analyst is to prepare a written statement of the objectives of the problem. Based on interviews with the user, the analyst writes a brief description of his/her understanding of the problem and reviews it with both the groups. People respond to written statements. They ask for clarifications and they correct obvious errors or misunderstandings. That is why a clear statement of objectives is important. In other words, proper understanding of the problem is essential to discover the cause of the problem and to plan a directed investigation by asking questions like what is being done. Why? Is there an underlying reason different from the one the user identifies? Following are some possible definitions of problems:

a. The existing system has a poor response time
b. It is unable to handle the workload.
c. The problem of cost, that is the economic system is not feasible.
d. The problem of accuracy and reliability
e. The required information is not produced by the existing system
f. The problem of security.

2. Feasibility study:

The actual meaning of feasibility is viability. This study is undertaken to know the likelihood of the system being useful to the organization. The aim of feasibility study is to assess alternative systems and to propose the most feasible and desirable system for development.

Thus, feasibility study provides an overview of the problem and acts as an important checkpoint that should be completed before committing more resources. The feasibility of a proposed system can be assessed in terms of four major categories as given below:

a) Organizational feasibility: the extent to which a proposed information system supports the objective of the organization's strategic plan for information systems determines the organizational feasibility of the system project.
b) Economic feasibility: In this study, costs and returns are evaluated to know whether returns justify the investment in the system project.
c) Technical feasibility: whether reliable hardware and software, capable of meeting the needs of the proposed system can be acquired or developed by the organizations in the required time is a major concern of the technical feasibility.
d) Operational feasibility: the willingness and ability of the management, employees, customers, suppliers, etc to operate, use and support a proposed system come under operational feasibility. In other words, the test of operational feasibility asks if the system will work when it is developed and installed.

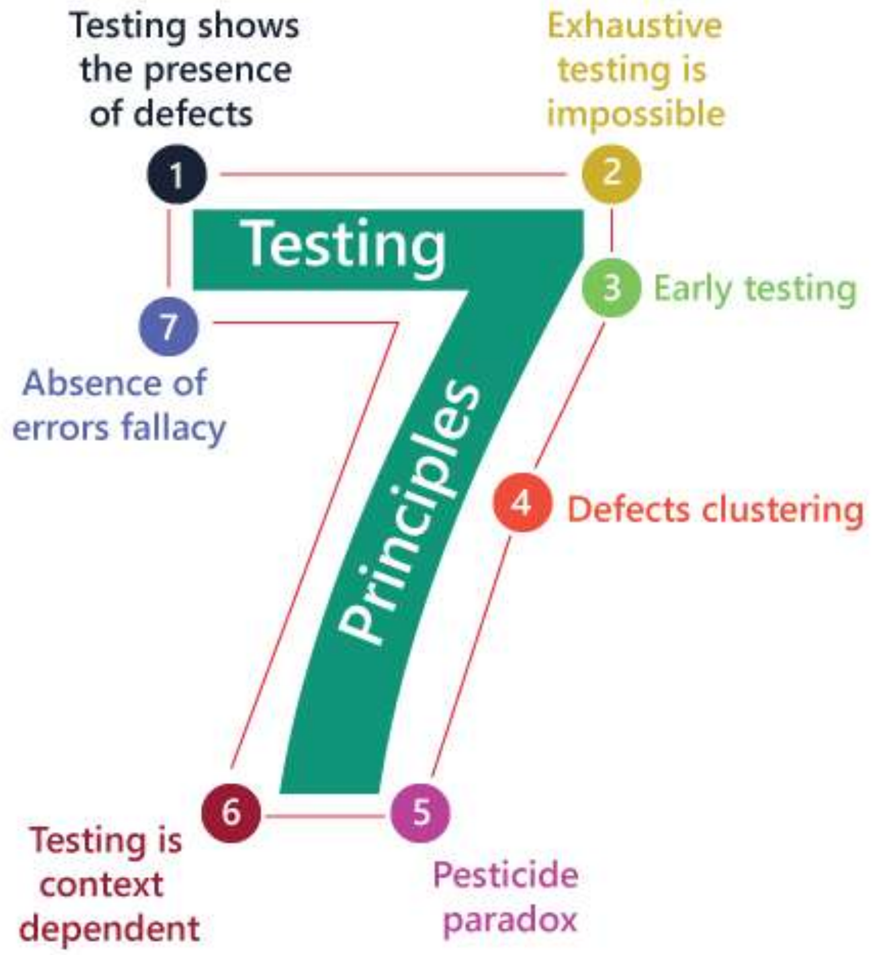
Black Box Testing	White Box Testing
Internal workings of an application are not required.	Knowledge of the internal workings is a must.
Also known as closed box/data-driven testing.	Also known as clear box/structural testing.
End users, testers, and developers.	Normally done by testers and developers.
This can only be done by a trial and error method.	Data domains and internal boundaries can be better tested.

Principles of Software Testing

There are seven principles in software testing:

- Testing shows the presence of defects
- Exhaustive testing is not possible
- Early testing
- Defect clustering
- Pesticide paradox
- Testing is context-dependent
- Absence of errors fallacy

- **Testing shows the presence of defects:** The goal of software testing is to make the software fail. Software testing reduces the presence of defects. Software testing talks about the presence of defects and doesn't talk about the absence of defects. Software testing can ensure that defects are present but it can not prove that software is defect-free. Even multiple testing can never ensure that software is 100% bug-free. Testing can reduce the number of defects but not remove all defects.
- **Exhaustive testing is not possible:** It is the process of testing the functionality of the software in all possible inputs (valid or invalid) and pre-conditions is known as exhaustive testing. Exhaustive testing is impossible means the software can never test at every test case. It can test only some test cases and assume that the software is correct and it will produce the correct output in every test case. If the software will test every test case then it will take more cost, effort, etc., which is impractical.
- **Early Testing:** To find the defect in the software, early test activity shall be started. The defect detected in the early phases of SDLC will be very less expensive. For better performance of software, software testing will start at the initial phase i.e. testing will perform at the requirement analysis phase.


Testing shows the presence of defects
1
Exhaustive testing is impossible
2
Testing
3
Early testing
7
Absence of errors fallacy
Principles
4
Defects clustering
6
5
Testing is context dependent
Pesticide paradox

- **Defect clustering:** In a project, a small number of modules can contain most of the defects. Pareto Principle to software testing state that 80% of software defect comes from 20% of modules.
- **Pesticide paradox:** Repeating the same test cases, again and again, will not find new bugs. So it is necessary to review the test cases and add or update test cases to find new bugs.
- **Testing is context-dependent:** The testing approach depends on the context of the software developed. Different types of software need to perform different types of testing. For example, The testing of the e-commerce site is different from the testing of the Android application.
- **Absence of errors fallacy:** If a built software is 99% bug-free but it does not follow the user requirement then it is unusable. It is not only necessary that software is 99% bug-free but it is also mandatory to fulfill all the customer requirements.

When to Start Testing?

An early start to testing reduces the cost and time to rework and produce error-free software that is delivered to the client. However in Software Development Life Cycle (SDLC), testing can be started from the Requirements Gathering phase and continued till the deployment of the software.

It also depends on the development model that is being used. For example, in the Waterfall model, formal testing is conducted in the testing phase; but in the incremental model, testing is performed at the end of every increment/iteration and the whole application is tested at the end.

Testing is done in different forms at every phase of SDLC −

- During the requirement gathering phase, the analysis and verification of requirements are also considered as testing.
- Reviewing the design in the design phase with the intent to improve the design is also considered as testing.
- Testing performed by a developer on completion of the code is also categorized as testing.

When to Stop Testing?

It is difficult to determine when to stop testing, as testing is a never-ending process and no one can claim that a software is 100% tested. The following aspects are to be considered for stopping the testing process –

- Testing Deadlines
- Completion of test case execution
- Completion of functional and code coverage to a certain point
- Bug rate falls below a certain level and no high-priority bugs are identified
- Management decision

Common myths about software testing

Myth 1: Testing is Too Expensive
Reality − There is a saying, pay less for testing during software development or pay more for maintenance or correction later. Early testing saves both time and cost in many aspects, however reducing the cost without testing may result in improper design of a software application rendering the product useless.

Myth 2: Testing is Time-Consuming
Reality − During the SDLC phases, testing is never a time-consuming process. However diagnosing and fixing the errors identified during proper testing is a time-consuming but productive activity.

Myth 3: Only Fully Developed Products are Tested
Reality − No doubt, testing depends on the source code but reviewing requirements and developing test cases is independent from the developed code. However iterative or incremental approach as a development life cycle model may reduce the dependency of testing on the fully developed software.

Myth 4: Complete Testing is Possible
Reality − It becomes an issue when a client or tester thinks that complete testing is possible. It is possible that all paths have been tested by the team but occurrence of complete testing is never possible. There might be some scenarios that are never executed by the test team or the client during the software development life cycle and may be executed once the project has been deployed.

Myth 5: A Tested Software is Bug-Free
Reality − This is a very common myth that the clients, project managers, and the management team believes in. No one can claim with absolute certainty that a software application is 100% bug-free even if a tester with superb testing skills has tested the application.

Myth 6: Missed Defects are due to Testers
Reality − It is not a correct approach to blame testers for bugs that remain in the application even after testing has been performed. This myth relates to Time, Cost, and Requirements changing Constraints. However the test strategy may also result in bugs being missed by the testing team.

Myth 7: Testers are Responsible for Quality of Product
Reality − It is a very common misinterpretation that only testers or the testing team should be responsible for product quality. Testers' responsibilities include the identification of bugs to the stakeholders and then it is their decision whether they will fix the bug or release the software. Releasing the software at the time puts more pressure on the testers, as they will be blamed for any error.

Myth 8: Test Automation should be used wherever possible to Reduce Time
Reality − Yes, it is true that Test Automation reduces the testing time, but it is not possible to start test automation at any time during software development. Test automaton should be started when the software has been manually tested and is stable to some extent. Moreover, test automation can never be used if requirements keep changing.

Myth 9: Anyone can Test a Software Application
Reality − People outside the IT industry think and even believe that anyone can test a software and testing is not a creative job. However testers know very well that this is a myth. Thinking alternative scenarios, try to crash a software with the intent to explore potential bugs is not possible for the person who developed it.

Myth 10: A Tester's only Task is to Find Bugs
Reality − Finding bugs in a software is the task of the testers, but at the same time, they are domain experts of the particular software. Developers are only responsible for the specific component or area that is assigned to them but testers understand the overall workings of the software, what the dependencies are, and the impacts of one module on another module.

Quality Assurance	Quality Control	Testing
QA includes activities that ensure the implementation of processes, procedures and standards in context to verification of developed software and intended requirements.	It includes activities that ensure the verification of a developed software with respect to documented (or not in some cases) requirements.	It includes activities that ensure the identification of bugs/error/defects in a software.
Focuses on processes and procedures rather than conducting actual testing on the system.	Focuses on actual testing by executing the software with an aim to identify bug/defect through implementation of procedures and process.	Focuses on actual testing.
Process-oriented activities.	Product-oriented activities.	Product-oriented activities.
Preventive activities.	It is a corrective process.	It is a preventive process.
It is a subset of Software Test Life Cycle (STLC).	QC can be considered as the subset of Quality Assurance.	Testing is the subset of Quality Control.

COCOMO Model

COCOMO (Constructive Cost Model) is a regression model based on LOC, i.e **number of Lines of Code**. It is a procedural cost estimate model for software projects and often used as a process of reliably predicting the various parameters associated with making a project such as size, effort, cost, time and quality. It was proposed by Barry Boehm in 1970 and is based on the study of 63 projects, which make it one of the best-documented models.

The key parameters which define the quality of any software products, which are also an outcome of the COCOMO are primarily Effort & Schedule:

- **Effort:** Amount of labor that will be required to complete a task. It is measured in person-months units.
- **Schedule:** Simply means the amount of time required for the completion of the job, which is, of course, proportional to the effort put. It is measured in the units of time such as weeks, months.

Different models of C0C0MO have been proposed to predict the cost estimation at different levels, based on the amount of accuracy and correctness required. All of these models can be applied to a variety of projects, whose characteristics determine the value of constant to be used in subsequent calculations. These characteristics pertaining to different system types are mentioned below.

Boehm’s definition of organic, semidetached, and embedded systems:

- **Organic –** A software project is said to be an organic type if the team size required is adequately small, the problem is well understood and has been solved in the past and also the team members have a nominal experience regarding the problem.
- **Semi-detached –** A software project is said to be a Semi-detached type if the vital characteristics such as team-size, experience, knowledge of the various programming environment lie in between that of organic and Embedded. The projects classified as Semi-Detached are comparatively less familiar and difficult to develop compared to the organic ones and require more experience and better guidance and creativity. Eg: Compilers or different Embedded Systems can be considered of Semi-Detached type.
- **Embedded –** A software project with requiring the highest level of complexity, creativity, and experience requirement fall under this category. Such software requires a larger team size than the other two models and also the developers need to be sufficiently experienced and creative to develop such complex models.

All the above system types utilize different values of the constants used in Effort Calculations.

Types of Models:

COCOMO consists of a hierarchy of three increasingly detailed and accurate forms. Any of the three forms can be adopted according to our requirements. These are types of COCOMO model:

1. Basic COCOMO Model
2. Intermediate COCOMO Model
3. Detailed COCOMO Model

The first level, **Basic COCOMO** can be used for quick and slightly rough calculations of Software Costs. Its accuracy is somewhat restricted due to the absence of sufficient factor considerations.

Intermediate COCOMO takes these Cost Drivers into account and **Detailed COCOMO** additionally accounts for the influence of individual project phases, i.e in case of Detailed it accounts for both these cost drivers and also calculations are performed phase wise henceforth producing a more accurate result.

Estimation of Effort: Calculations –
1. Basic Model –

$$E= a(KLOC)^b$$

The above formula is used for the cost estimation of for the basic COCOMO model, and also is used in the subsequent models. The constant values a and b for the Basic Model for the different categories of system:

Software Projects	a	b
Organic	2.4	1.05
Semi Detached	3.0	1.12
Embedded	3.6	1.20

The effort is measured in Person-Months and as evident from the formula is dependent on Kilo-Lines of code. These formulas are used as such in the Basic Model calculations, as not much consideration of different factors such as reliability, expertise is taken into account, henceforth the estimate is rough.

2. Intermediate Model –

The basic COCOMO model assumes that the effort is only a function of the number of lines of code and some constants evaluated according to the different software system. However, in reality, no system's effort and schedule can be solely calculated on the basis of Lines of Code. For that, various other factors such as reliability, experience, Capability. These factors are known as Cost Drivers and the Intermediate Model utilizes 15 such drivers for cost estimation.

Classification of Cost Drivers and their attributes:

(i) Product attributes –

- Required software reliability extent
- Size of the application database
- The complexity of the product

(ii) Hardware attributes –

- Run-time performance constraints
- Memory constraints
- The volatility of the virtual machine environment
- Required turnabout time

(iii) Personnel attributes –

- Analyst capability
- Software engineering capability
- Applications experience
- Virtual machine experience
- Programming language experience

(iv) Project attributes –

- Use of software tools
- Application of software engineering methods
- Required development schedule

Cost Drivers	Very Low	Low	Nominal	High	Very High
Product Attributes					
Required Software Reliability	0.75	0.88	1.00	1.15	1.40
Size of Application Database		0.94	1.00	1.08	1.16
Complexity of The Product	0.70	0.85	1.00	1.15	1.30
Hardware Attributes					
Runtime Performance Constraints			1.00	1.11	1.30
Memory Constraints			1.00	1.06	1.21
Volatility of the virtual machine environment		0.87	1.00	1.15	1.30
Required turnabout time		0.94	1.00	1.07	1.15
Personnel attributes					
Analyst capability	1.46	1.19	1.00	0.86	0.71
Applications experience	1.29	1.13	1.00	0.91	0.82
Software engineer capability	1.42	1.17	1.00	0.86	0.70
Virtual machine experience	1.21	1.10	1.00	0.90	
Programming language experience	1.14	1.07	1.00	0.95	
Project Attributes					
Application of software engineering methods	1.24	1.10	1.00	0.91	0.82
Use of software tools	1.24	1.10	1.00	0.91	0.83
Required development schedule	1.23	1.08	1.00	1.04	1.10

The project manager is to rate these 15 different parameters for a particular project on a scale of one to three. Then, depending on these ratings, appropriate cost driver values are taken from the above table. These 15 values are then multiplied to calculate the EAF (Effort Adjustment Factor). The Intermediate COCOMO formula now takes the form:

$$\mathrm{E}= (\mathrm{a}(\mathrm{KLOC})^{b}) * EAF$$

The values of a and b in case of the intermediate model are as follows:

Software Projects	a	b
Organic	3.2	1.05
Semi Detached	3.0	1.12
Embeddedc	2.8	1.20

3. Detailed Model –

Detailed COCOMO incorporates all characteristics of the intermediate version with an assessment of the cost driver's impact on each step of the software engineering process. The detailed model uses different effort multipliers for each cost driver attribute. In detailed COCOCMO, the whole software is divided into different modules and then we apply COCOMO in different modules to estimate effort and then sum the effort.

The Six phases of detailed COCOMO are:

- Planning and requirements
- System design
- Detailed design
- Module code and test
- Integration and test
- Cost Constructive model

The effort is calculated as a function of program size and a set of cost drivers are given according to each phase of the software lifecycle.

Software Metrics

A software metric is a measure of software characteristics which are measurable or countable. Software metrics are valuable for many reasons, including measuring software performance, planning work items, measuring productivity, and many other uses.
Within the software development process, many metrics are that are all connected. Software metrics are similar to the four functions of management: Planning, Organization, Control, or Improvement.

Characteristics of software Metrics:

- **Quantitative:** Metrics must possess quantitative nature. It means metrics can be expressed in values.
- **Understandable:** Metric computation should be easily understood, and the method of computing metrics should be clearly defined.
- **Applicability:** Metrics should be applicable in the initial phases of the development of the software.
- **Repeatable:** The metric values should be the same when measured repeatedly and consistent in nature.
- **Economical:** The computation of metrics should be economical.
- **Language Independent:** Metrics should not depend on any programming language.

Classification of Software Metrics

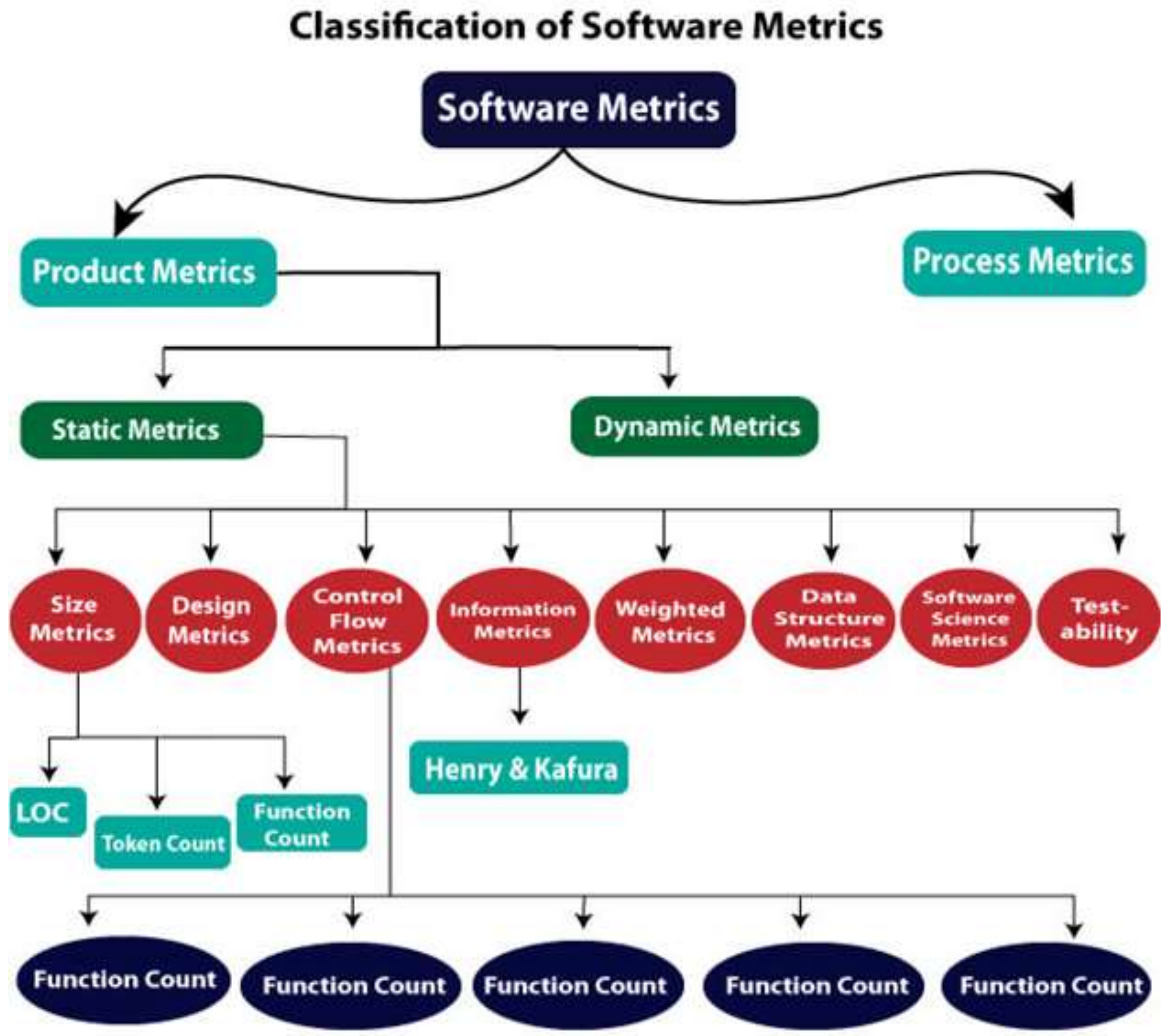
Software metrics can be classified into two types as follows:

1. Product Metrics: These are the measures of various characteristics of the software product. The two important software characteristics are:

- Size and complexity of software.
- Quality and reliability of software.

These metrics can be computed for different stages of SDLC.

2. Process Metrics: These are the measures of various characteristics of the software development process. For example, the efficiency of fault detection. They are used to measure the characteristics of methods, techniques, and tools that are used for developing software.



Types of Metrics

- **Internal metrics:** Internal metrics are the metrics used for measuring properties that are viewed to be of greater importance to a software developer. For example, Lines of Code (LOC) measure.
- **External metrics:** External metrics are the metrics used for measuring properties that are viewed to be of greater importance to the user, e.g., portability, reliability, functionality, usability, etc.
- **Hybrid metrics:** Hybrid metrics are the metrics that combine product, process, and resource metrics. For example, cost per FP where FP stands for Function Point Metric.
- **Project metrics:** Project metrics are the metrics used by the project manager to check the project's progress. Data from the past projects are used to collect various metrics, like time and cost; these estimates are used as a base of new software. Note that as the project proceeds, the project manager will check its progress from time-to-time and will compare the effort, cost, and time with the original effort, cost and time. Also understand that these metrics are used to decrease the development costs, time efforts and risks. The project quality can also be improved. As quality improves, the number of errors and time, as well as cost required, is also reduced.

Advantages of Software Metrics :

- Reduction in cost or budget.
- It helps to identify the particular area for improvising.
- It helps to increase the product quality.
- Managing the workloads and teams.
- Reduction in overall time to produce the product,.
- It helps to determine the complexity of the code and to test the code with resources.
- It helps in providing effective planning, controlling and managing of the entire product.

Disadvantages of Software Metrics :

- It is expensive and difficult to implement the metrics in some cases.
- Performance of the entire team or an individual from the team can’t be determined. Only the performance of the product is determined.
- Sometimes the quality of the product is not met with the expectation.
- It leads to measure the unwanted data which is wastage of time.
- Measuring the incorrect data leads to make wrong decision making.

<u>Size Oriented Metrics</u>

LOC Metrics

It is one of the earliest and simpler metrics for calculating the size of the computer program. It is generally used in calculating and comparing the productivity of programmers. These metrics are derived by normalizing the quality and productivity measures by considering the size of the product as a metric.

Following are the points regarding LOC measures:

- In size-oriented metrics, LOC is considered to be the normalization value.
- It is an older method that was developed when FORTRAN and COBOL programming were very popular.
- Productivity is defined as KLOC / EFFORT, where effort is measured in person-months.
- Size-oriented metrics depend on the programming language used.
- As productivity depends on KLOC, so assembly language code will have more productivity.
- LOC measure requires a level of detail which may not be practically achievable.
- The more expressive is the programming language, the lower is the productivity.
- LOC method of measurement does not apply to projects that deal with visual (GUI-based) programming. As already explained, Graphical User Interfaces (GUIs) use forms basically. LOC metric is not applicable here.
- It requires that all organizations must use the same method for counting LOC. This is so because some organizations use only executable statements, some useful comments, and some do not. Thus, the standard needs to be established.
- These metrics are not universally accepted.

Based on the LOC/KLOC count of software, many other metrics can be computed:

- Errors/KLOC.
- $/ KLOC.
- Defects/KLOC.
- Pages of documentation/KLOC.
- Errors/PM.
- Productivity = KLOC/PM (effort is measured in person-months).
- $/ Page of documentation.

Advantages of LOC

- Simple to measure

Disadvantage of LOC

- It is defined on the code. For example, it cannot measure the size of the specification.
- It characterizes only one specific view of size, namely length, it takes no account of functionality or complexity
- Bad software design may cause an excessive line of code
- It is language dependent
- Users cannot easily understand it

Functional Point (FP) Analysis

Allan J. Albrecht initially developed function Point Analysis in 1979 at IBM and it has been further modified by the International Function Point Users Group (IFPUG). FPA is used to make estimate of the software project, including its testing in terms of functionality or function size of the software product. However, functional point analysis may be used for the test estimation of the product. The functional size of the product is measured in terms of the function point, which is a standard of measurement to measure the software application.

Objectives of FPA

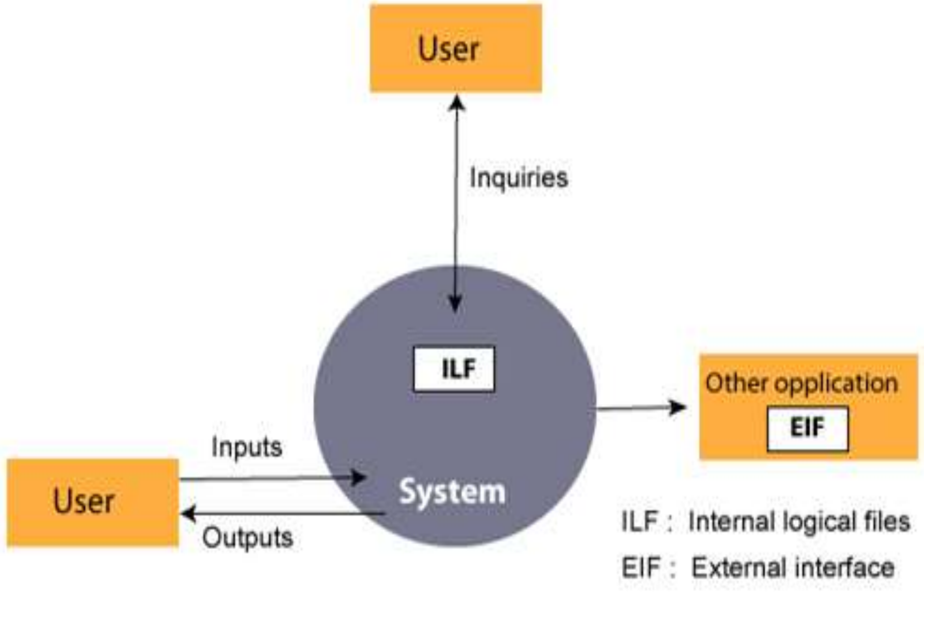
The basic and primary purpose of the functional point analysis is to measure and provide the software application functional size to the client, customer, and the stakeholder on their request. Further, it is used to measure the software project development along with its maintenance, consistently throughout the project irrespective of the tools and the technologies.

1. FPs of an application is found out by counting the number and types of functions used in the applications. Various functions used in an application can be put under five types, as shown in Table:

Types of FP Attributes

Measurements Parameters	Examples
1.Number of External Inputs(EI)	Input screen and tables
2. Number of External Output (EO)	Output screens and reports
3. Number of external inquiries (EQ)	Prompts and interrupts.
4. Number of internal files (ILF)	Databases and directories
5. Number of external interfaces (EIF)	Shared databases and shared routines.

All these parameters are then individually assessed for complexity.



FPAs Functional Units System

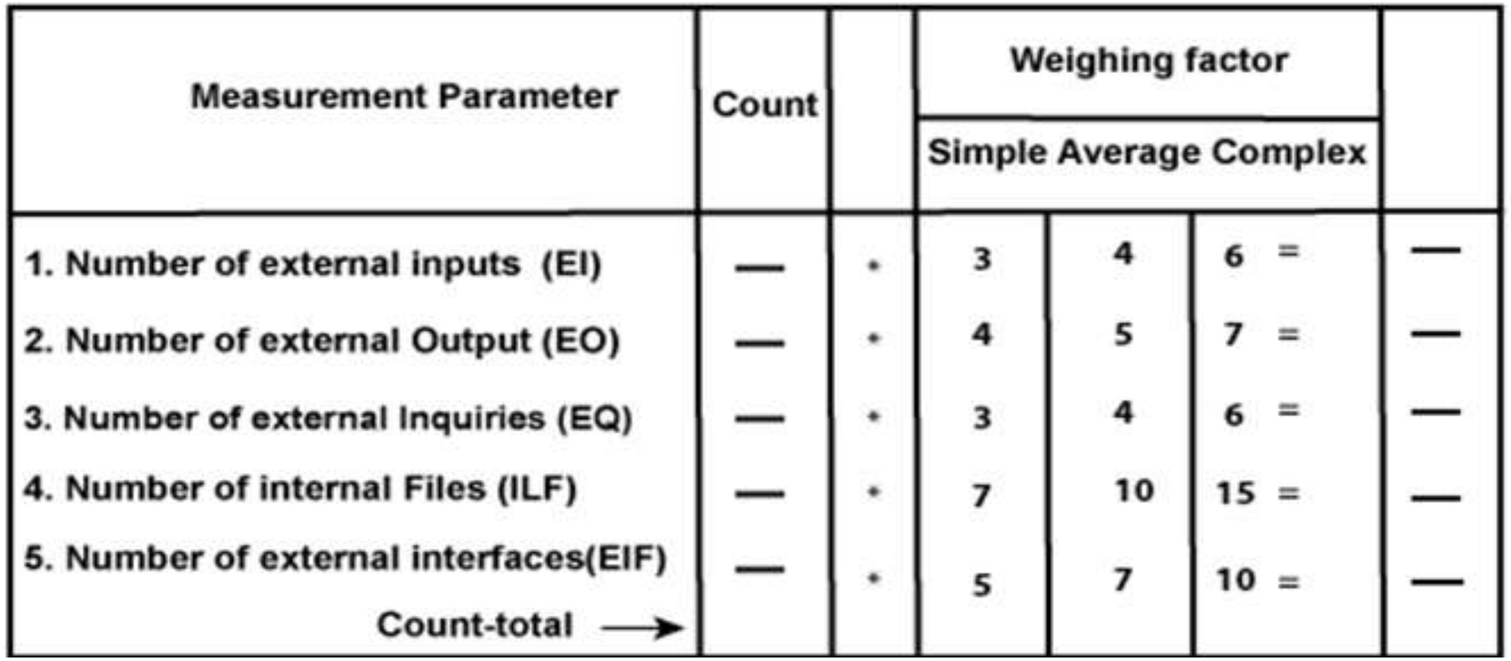
2. FP characterizes the complexity of the software system and hence can be used to depict the project time and the manpower requirement.
3. The effort required to develop the project depends on what the software does.
4. FP is programming language independent.
5. FP method is used for data processing systems, business systems like information systems.
6. The five parameters mentioned above are also known as information domain characteristics.
7. All the parameters mentioned above are assigned some weights that have been experimentally determined and are shown in Table:

Measurement Parameter	**Low**	**Average**	**High**
1. Number of external inputs (EI)	7	10	15
2. Number of external outputs (EO)	5	7	10
3. Number of external inquiries (EQ)	3	4	6
4. Number of internal files (ILF)	4	5	7
5. Number of external interfaces (EIF)	3	4	6

Weights of 5-FP Attributes

The functional complexities are multiplied with the corresponding weights against each function, and the values are added up to determine the UFP (Unadjusted Function Point) of the subsystem.

Computing FPs



Measurement Parameter	Count		Weighing factor			
			Simple	Average	Complex	
1. Number of external inputs (EI)	—	*	3	4	6 =	—
2. Number of external Output (EO)	—	*	4	5	7 =	—
3. Number of external Inquiries (EQ)	—	*	3	4	6 =	—
4. Number of internal Files (ILF)	—	*	7	10	15 =	—
5. Number of external interfaces(EIF)	—	*	5	7	10 =	—
Count-total →						

Here that weighing factor will be simple, average, or complex for a measurement parameter type.

The Function Point (FP) is thus calculated with the following formula.

FP = Count-total * [0.65 + 0.01 * $\sum(f_i)$]

= Count-total * CAF

where Count-total is obtained from the above Table.

CAF = [0.65 + 0.01 *$\sum(f_i)$]

and **$\sum(f_i)$** is the sum of all 14 questionnaires and show the complexity adjustment value/ factor-CAF (where i ranges from 1 to 14). Usually, a student is provided with the value of $\sum(f_i)$

Also note that **$\sum(f_i)$** ranges from 0 to 70, i.e.,

$0 <= \sum(f_i) <=70$

and CAF ranges from 0.65 to 1.35 because

- When **$\sum(f_i)$** = 0 then CAF = 0.65
- When **$\sum(f_i)$** = 70 then CAF = 0.65 + (0.01 * 70) = 0.65 + 0.7 = 1.35

Based on the FP measure of software many other metrics can be computed:

- Errors/FP
- $/FP.
- Defects/FP
- Pages of documentation/FP
- Errors/PM.
- Productivity = FP/PM (effort is measured in person-months).
- $/Page of Documentation.

8. LOCs of an application can be estimated from FPs. That is, they are inter convertible. **This process is known as backfiring**. For example, 1 FP is equal to about 100 lines of COBOL code.

9. FP metrics is used mostly for measuring the size of Management Information System (MIS) software.

10. But the function points obtained above are unadjusted function points (UFPs). These (UFPs) of a subsystem are further adjusted by considering some more General System Characteristics (GSCs). It is a set of 14 GSCs that need to be considered. The procedure for adjusting UFPs is as follows:

- Degree of Influence (DI) for each of these 14 GSCs is assessed on a scale of 0 to 5. (b) If a particular GSC has no influence, then its weight is taken as 0 and if it has a strong influence then its weight is 5.
- The score of all 14 GSCs is totaled to determine Total Degree of Influence (TDI).
- Then Value Adjustment Factor (VAF) is computed from TDI by using the formula: **VAF = (TDI * 0.01) + 0.65**

Remember that the value of VAF lies within 0.65 to 1.35 because

- When TDI = 0, VAF = 0.65
- When TDI = 70, VAF = 1.35
- VAF is then multiplied with the UFP to get the final FP count: **FP = VAF * UFP**

Differentiate between FP and LOC

FP	LOC
1. FP is specification based.	1. LOC is an analogy based.
2. FP is language independent.	2. LOC is language dependent.
3. FP is user-oriented.	3. LOC is design-oriented.
4. It is extendible to LOC.	4. It is convertible to FP (backfiring)

Halstead's Software Metrics

A software metric is a quantitative or countable measure of programmed properties.

Software metrics are helpful for various purposes, including assessing software performance, planning work items, and determining productivity.

The purpose of recording and analyzing software metrics is to figure out how good a product or process is now, improve it, and predict how good it will be once we finish the project.

All the Software development teams can use software metrics to convey project status, pinpoint and handle issues, and monitor, improve, and better manage their workflow.
Here we will look at why Software Metrics are required and Halsted’s Software Metrics.

Requirement of Software Metrics

- Assess the existing product or process's quality.
  Any product's quality is determined by its ability to meet its standards (both expressed and implicit needs). The product or service must complete customer needs.
- Predict a product's or process's attributes.
  Quality means that the product's performance and process are designed to satisfy specified goals in a manufacturing process. As a result, reliable quality prediction is critical for delivering high-quality products and increasing competitiveness.
- Enhance the quality of a product or process
  Some ways to improve the quality of product or process:
  → Automation allows you to test early and often.
  → Encourage creativity by implementing quality controls from the start.
  → Producing high-quality software necessitates long-term planning and strategy.
  → Make a list of your deliverables.
  → Review, revise, and consider the fact.

Halstead’s Software Metrics

Halstead's hypothesis is a method for calculating software products' size, development effort, and development cost. To generate the formulas for overall program length, possible minimal volume, actual volume, effort, and development time, Halstead's used a few primitive program parameters.

According to Halstead’s statement, **"A computer program is an implementation of an algorithm that is believed to be a collection of tokens that can be classed as either operators or operands.“**

Count of Token

A computer program is defined as a collection of tokens classed as operators or operands in these metrics. These basic symbols can be used to determine all software science measurements. Tokens are the symbols that are used to represent something.

The fundamental metrics are:
- **n1** is the number of distinct operators.
- **n2** is the number of separate operands.
- **N1** is the total number of instances of each operator.
- **N2** is the total number of occurrences of operands.

The program's size can be stated as N = N1 + N2 in terms of unlimited tokens used.
In addition to the preceding, Halstead specifies:
- n1* denotes the number of possible operators.
- n2* is the number of possible operands.

The minimal feasible number of operators and operands for a module and a program, according to Halstead, are n1* and n2*.

Halstead’s Software Metrics

1. **The Volume of the programme (V)**
   The standard unit for size "bits" is the unit of measurement for volume. If a standard binary encoding for the vocabulary is employed, it is the actual size of the programme.
   V=N*log2n

2. **Level of the Program (L)**
   L is a number that ranges from zero to one, with L=1 denoting a programme written at the most excellent level possible (i.e., with minimum size).
   L=V*/V

3. **The Difficulty of the Program**
   The number of unique operators in the programme determines the program's complexity level or error-proneness (D).
   (n1/2) * (N2/n2) = D

4. **Effort Involved in Programming (E)**
   Fundamental mental discriminations are the unit of measurement for E.
   E=V/L=D*V

.

5. Program Length Approximated

The first hypothesis of software science, according to Halstead, is that the length of a well-structured programme is only a function of the number of unique operators and operands.

N=N1+N2

→ N^ stands for the estimated length of the programme.

→ N^ is equal to n1log2n1 + n2log2n2.

Alternative expressions for estimating programme duration have been published:

→ NJ is equal to log2 (n1!) + log2 (n2!)

→ NB = n1 * log2n2 + n2 * log2n1

→ NC = n1 * sqrt(n1) + n2 * sqrt(n2)

→ NS = (n * log2n) / 2

6. Minimum Volume Potential

The volume of the shortest programme in which a problem can be coded is denoted as the potential minimal volume V*.

V* = (2 + n2*) * log2 (2 + n2*)

n2* is the number of distinct input and output parameters.

7. The Size of Vocabulary (n)

The size of a program's vocabulary, which is comprised of the number of distinct tokens required to construct the programme, is defined as:

n=n1+n2

Where n denotes a program's vocabulary.

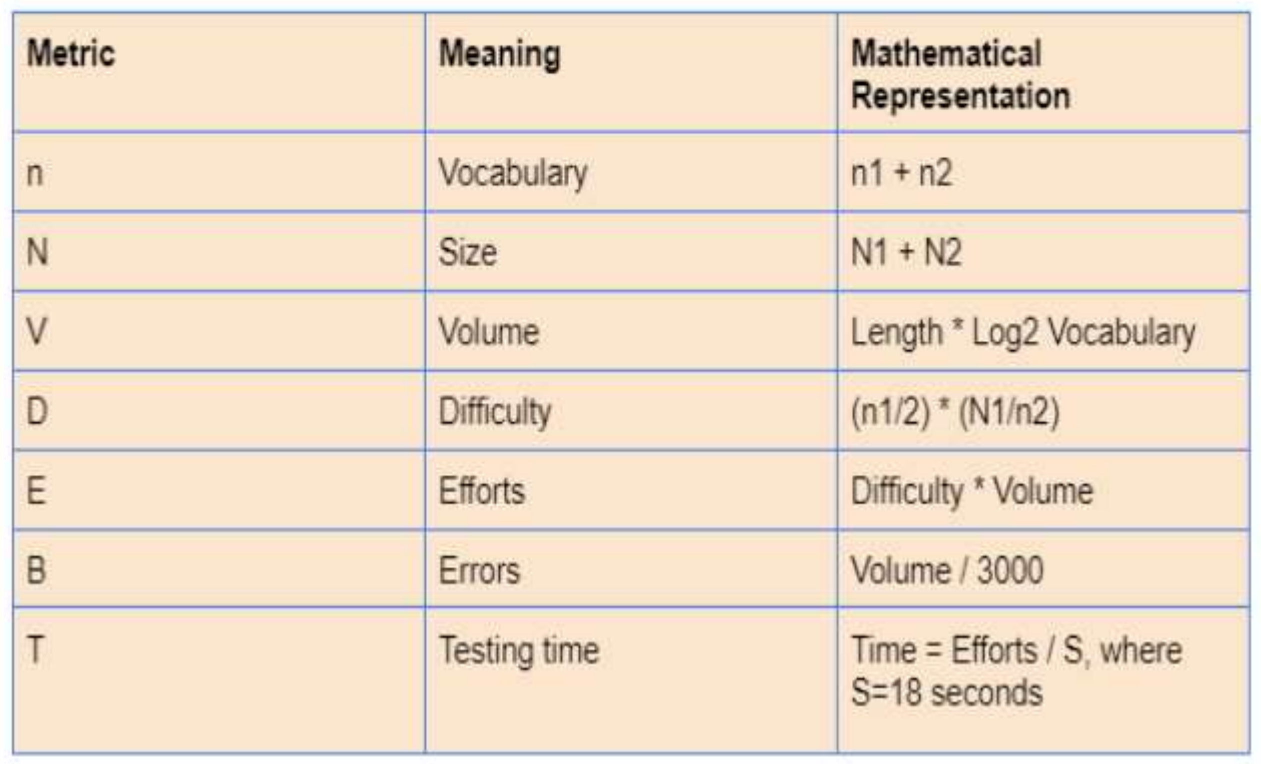
n1 denotes the number of distinct operators, while n2 denotes the number of separate operands

Complexity Measures by Halstead

Howard Halstead pioneered the use of metrics to assess the complexity of software.
Halstead's metrics are based on the program's actual implementation and measures directly derived from the operators and operands in the source code.
It allows you to analyze C/C++/Java source code testing time, vocabulary, size, difficulty, errors, and attempts. Halstead metrics consider a program a series of operators and their operands. He defines several measures to assess the module's difficulty.

- n1 The number of distinct operators
- n2 The number of separate operands
- N1 Total number of occurrences of operators
- N2 Total number of circumstances of operands

When we choose a source file in Metric Viewer to check its complexity data, we get the following result in Metric Report:



Metric	Meaning	Mathematical Representation
n	Vocabulary	n1 + n2
N	Size	N1 + N2
V	Volume	Length * Log2 Vocabulary
D	Difficulty	(n1/2) * (N1/n2)
E	Efforts	Difficulty * Volume
B	Errors	Volume / 3000
T	Testing time	Time = Efforts / S, where S=18 seconds

Advantages of Halstead’s Software Metrics

- Predicts the rate of mistake.
- Indicates the amount of work required for upkeep.
- The Calculation is simple.
- Assess the overall quality of the product.
- Any language can be used.

The Disadvantage of Halstead’s Software Metrics

- Complete code is required.
- As the program level drops, the complexity rises.
- The Calculation is complex.

Cyclomatic Complexity

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 second command might immediately follow the first command.

For example, if source code contains no control flow statement then its cyclomatic complexity will be 1 and 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.

Mathematically, for a structured program, the directed graph inside control flow is the edge joining two basic blocks of the program as control may pass from first to second.
So, cyclomatic complexity M would be defined as,

M = E – N + 2P
where,
E = the number of edges in the control flow graph
N = the number of nodes in the control flow graph
P = the number of connected components

Steps that should be followed in calculating cyclomatic complexity and test cases design are:

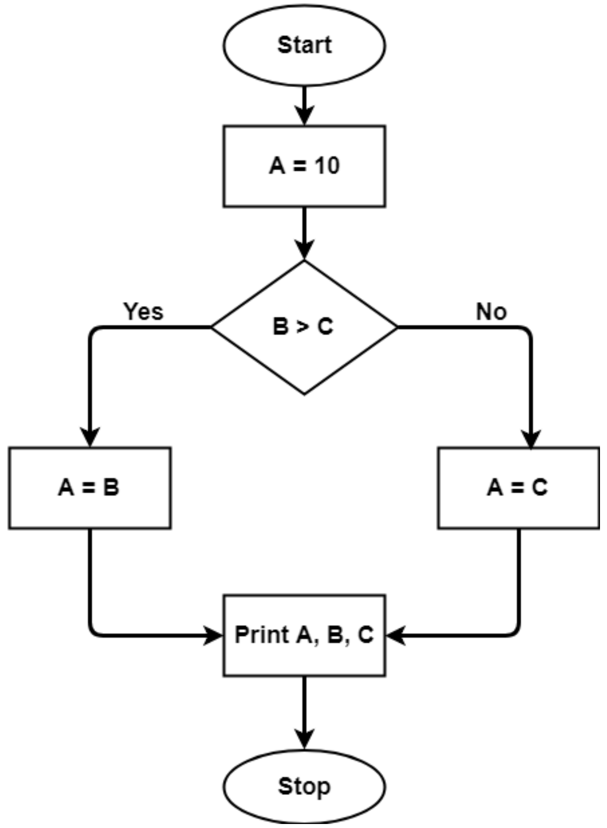
Construction of graph with nodes and edges from code.

- Identification of independent paths.
- Cyclomatic Complexity Calculation
- Design of Test Cases

Let a section of code as such:

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

Control Flow Graph of side code:



The cyclomatic complexity calculated for above code will be from control flow graph. The graph shows seven shapes (nodes), seven lines(edges), hence cyclomatic complexity is 7-7+2 = 2.

Use of Cyclomatic Complexity:

- Determining the independent path executions thus proven to be very helpful for Developers and Testers.
- It can make sure that every path have been tested at least once.
- Thus help to focus more on uncovered paths.
- Code coverage can be improved.
- Risk associated with program can be evaluated.
- These metrics being used earlier in the program helps in reducing the risks.

Advantages of Cyclomatic Complexity:.

- It can be used as a quality metric, gives relative complexity of various designs.
- It is able to compute faster than the Halstead's metrics.
- It is used to measure the minimum effort and best areas of concentration for testing.
- It is able to guide the testing process.
- It is easy to apply.

Disadvantages of Cyclomatic Complexity:

- It is the measure of the programs's control complexity and not the data complexity.
- In this, nested conditional structures are harder to understand than non-nested structures.
- In case of simple comparisons and decision structures, it may give a misleading figure.