Unit 3: Software Design

[**Syllabus 1**](#_1ogac7tm956f)

[**Coupling 2**](#_ajrwxhisdxpg)

[**Cohesion 2**](#_6aw1b16q42jl)

[**Conceptual design of the system: 3**](#_uhbyurtggi4z)

[**Technical Design of the System: 3**](#_afzfi7yso9rq)

[**Modularization 3**](#_im382dscjinz)

[**Coupling 4**](#_vv0r0bk4a5kj)

[Types of Coupling 5](#_h9dii4dup2x6)

[**Cohesion 6**](#_5mv8ab547lc7)

[Types of Cohesion 6](#_xd1rf9nvjl32)

[Advantages of low coupling 8](#_4gre18nveqxj)

[Advantages of high cohesion: 8](#_p56y71a3394i)

[Disadvantages of high coupling 8](#_x3o2hkvoxbed)

[Disadvantages of low cohesion: 8](#_rpv6h7jscv7i)

[**Function Oriented Design 9**](#_xycvj4auuncp)

[Generic Procedure 9](#_huwa8z9pp236)

[Problem in Top-Down design method 9](#_7v1ldzwacuxz)

[Solution to the problem: 9](#_mll3mhkjzyk8)

[Function Oriented Design Strategies 10](#_tnqm6y60t2k8)

[**Object-oriented design 10**](#_c8a7y5ktzxd1)

[Structured Programming 11](#_9nxvu5hxmhg9)

[Procedural Programming Approach 11](#_mio2y774pe4o)

[The Object-Oriented Programming (OOP) Approach 12](#_9okl7jb1o6z)

[What Are Class & Object? 13](#_7hhnlx1dwx9y)

[Field of Halstead Metrics 18](#)

[Halstead Metrics 19](#)

[Advantages of Halstead Metrics 20](#)

[Disadvantages of Halstead Metrics 21](#)

# 

# Syllabus

UNIT – III Software Design: Cohesion & Coupling, Classification of Cohesiveness & Coupling, Function Oriented Design, Object Oriented Design, Software Metrics: Software measurements: What & Why, Token Count, Halstead Software Science Measures, Design Metrics, Data Structure Metrics Software Implementation: Relationship between design and implementation, Implementation issues and programming support environment, Coding the procedural design, Good coding style.

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

# **Coupling**

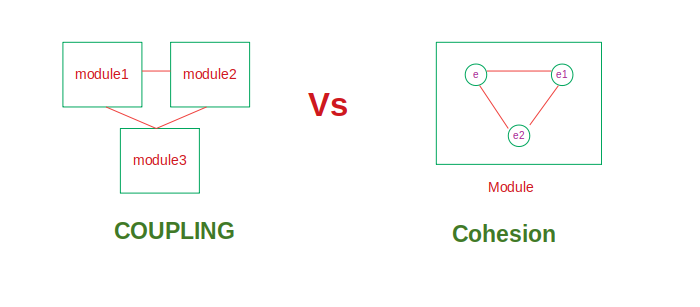
Coupling refers to the degree of interdependence between software modules. High coupling means that modules are closely connected and changes in one module may affect other modules. Low coupling means that modules are independent and changes in one module have little impact on other modules.

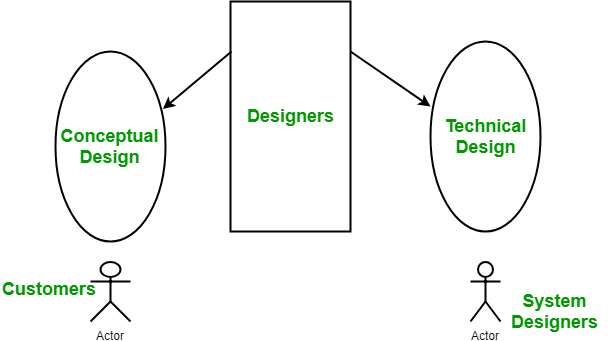
# Cohesion

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

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.

Basically, design is a two-part iterative process. First part is Conceptual Design that tells the customer what the system will do. Second is Technical Design that allows the system builders to understand the actual hardware and software needed to solve a customer's problem.





# **Conceptual design of** **the system:**

* Written in simple language i.e. customer understandable language.
* Detailed explanation about system characteristics.
* Describes the functionality of the system.
* It is independent of implementation.
* Linked with requirement documents.

# **Technical Design of the System:**

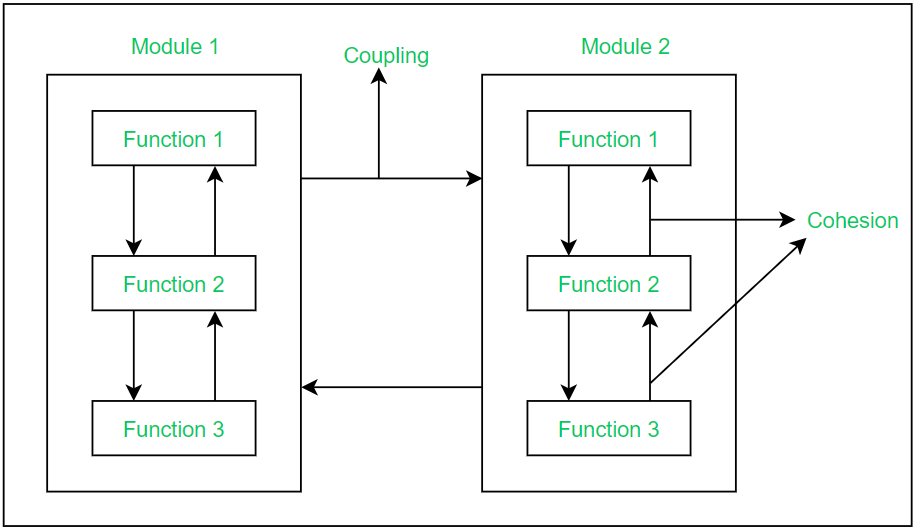
* Hardware component and design.
* Functionality and hierarchy of software components.
* Software architecture
* Network architecture
* Data structure and flow of data.
* I/O component of the system.
* Shows interface.

# 

# **Modularization**

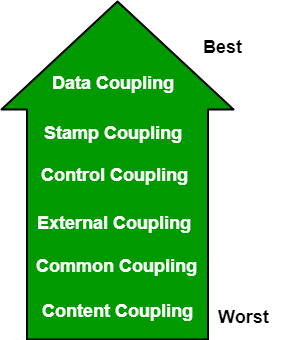
Modularization is the process of dividing a software system into multiple independent modules where each module works independently. There are many advantages of Modularization in software engineering. Some of these are given below:

* 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.

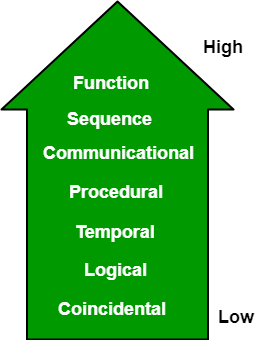


## **Types of Coupling**

* **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 behaviour 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 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.
* **Temporal Coupling:** Temporal coupling occurs when two modules depend on the timing or order of events, such as one module needing to execute before another. This type of coupling can result in design issues and difficulties in testing and maintenance.
* **Sequential Coupling:** Sequential coupling occurs when the output of one module is used as the input of another module, creating a chain or sequence of dependencies. This type of coupling can be difficult to maintain and modify.
* **Communicational Coupling:**Communicational coupling occurs when two or more modules share a common communication mechanism, such as a shared message queue or database. This type of coupling can lead to performance issues and difficulty in debugging.
* **Functional Coupling:**Functional coupling occurs when two modules depend on each other’s functionality, such as one module calling a function from another module. This type of coupling can result in tightly-coupled code that is difficult to modify and maintain.
* **Data-Structured Coupling:** Data-structured coupling occurs when two or more modules share a common data structure, such as a database table or data file. This type of coupling can lead to difficulty in maintaining the integrity of the data structure and can result in performance issues.
* **Interaction Coupling:** Interaction coupling occurs due to the methods of a class invoking methods of other classes. Like with functions, the worst form of coupling here is if methods directly access internal parts of other methods. Coupling is lowest if methods communicate directly through parameters.
* **Component Coupling:**Component coupling refers to the interaction between two classes where a class has variables of the other class. Three clear situations exist as to how this can happen. A class C can be component coupled with another class C1, if C has an instance variable of type C1, or C has a method whose parameter is of type C1,or if C has a method which has a local variable of type C1. It should be clear that whenever there is component coupling, there is likely to be interaction coupling.

# 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 the next line and reverse the characters of a string in a single component.
* Procedural Cohesion: This type of cohesion occurs when elements or tasks are grouped together in a module based on their sequence of execution, such as a module that performs a set of related procedures in a specific order. Procedural cohesion can be found in structured programming languages.
* Communicational Cohesion: Communicational cohesion occurs when elements or tasks are grouped together in a module based on their interactions with each other, such as a module that handles all interactions with a specific external system or module. This type of cohesion can be found in object-oriented programming languages.
* Temporal Cohesion: Temporal cohesion occurs when elements or tasks are grouped together in a module based on their timing or frequency of execution, such as a module that handles all periodic or scheduled tasks in a system. Temporal cohesion is commonly used in real-time and embedded systems.
* Informational Cohesion: Informational cohesion occurs when elements or tasks are grouped together in a module based on their relationship to a specific data structure or object, such as a module that operates on a specific data type or object. Informational cohesion is commonly used in object-oriented programming.
* Functional Cohesion: This type of cohesion occurs when all elements or tasks in a module contribute to a single well-defined function or purpose, and there is little or no coupling between the elements. Functional cohesion is considered the most desirable type of cohesion as it leads to more maintainable and reusable code.
* Layer Cohesion: Layer cohesion occurs when elements or tasks in a module are grouped together based on their level of abstraction or responsibility, such as a module that handles only low-level hardware interactions or a module that handles only high-level business logic. Layer cohesion is commonly used in large-scale software systems to organize code into manageable layers.

## 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:

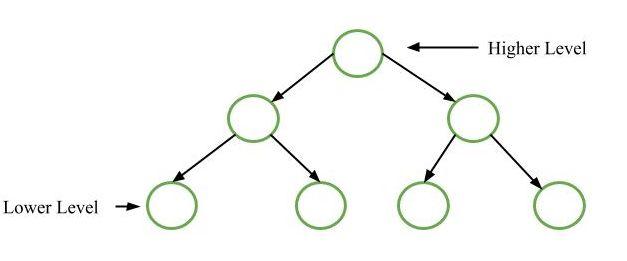
* Increased code duplication: Low cohesion can lead to the duplication of code, as elements that belong together are split into separate modules.
* Reduced functionality: Low cohesion can result in modules that lack a clear purpose and contain elements that don’t belong together, reducing their functionality 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.

# Function Oriented Design

Function Oriented Design is an approach to software design where the design is decomposed into a set of interacting units where each unit has a clearly defined function.

## Generic Procedure

Start with a high level description of what the software / program does. Refine each part of the description one by one by specifying in greater details the functionality of each part. These points lead to Top-Down Structure.

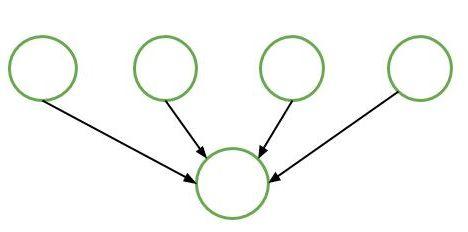


## Problem in Top-Down design method

Mostly each module is used by at most one other module and that module is called its Parent module.

### **Solution to the problem:**

Designing of reusable modules. It means modules use several modules to do their required functions.



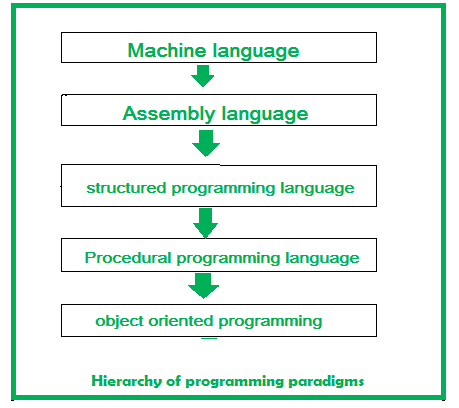
## **Function Oriented Design Strategies**

Function Oriented Design Strategies are as follows:

1. **Data Flow Diagram (DFD):**  
   A data flow diagram (DFD) maps out the flow of information for any process or system. It uses defined symbols like rectangles, circles and arrows, plus short text labels, to show data inputs, outputs, storage points and the routes between each destination.
2. **Data Dictionaries:**  
   Data dictionaries are simply repositories to store information about all data items defined in DFDs. At the requirement stage, data dictionaries contain data items. Data dictionaries include Name of the item, Aliases (Other names for items), Description / purpose, Related data items, Range of values, Data structure definition / form.
3. **Structure Charts:**  
   It is the hierarchical representation of a system which partitions the system into black boxes (functionality is known to users but inner details are unknown). Components are read from top to bottom and left to right. When a module calls another, it views the called module as black box, passing required parameters and receiving results.
4. **Pseudo Code:**  
   Pseudo Code is system description in short English like phrases describing the function. It uses keywords and indentation. Pseudo codes are used as replacement for flow charts. It decreases the amount of documentation required.

# Object-oriented design

Object-oriented design started right from the moment computers were invented. Programming was there, and programming approaches came into the picture. Programming is basically giving certain instructions to the computer.   
At the beginning of the computing era, programming was usually limited to machine language programming. Machine language means those sets of instructions that are specific to a particular machine or processor, which are in the form of 0’s and 1’s. These are sequences of bits (0100110…). But it’s quite difficult to write a program or develop software in machine language.  
It’s actually impossible to develop software used in today’s scenarios with sequences of bits. This was the main reason programmers moved on to the next generation of programming languages, developing assembly languages, which were near enough to the English language to easily understand. These assembly languages were used in microprocessors. With the invention of the microprocessor, assembly languages flourished and ruled over the industry, but it was not enough. Again, programmers came up with something new, i.e., structured and procedural programming.

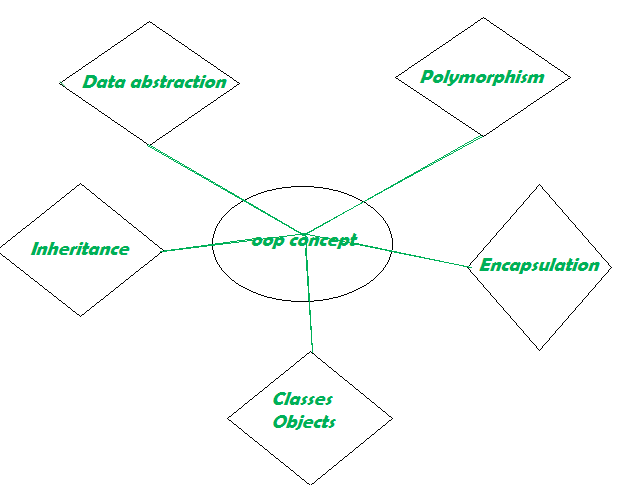


## Structured Programming

The basic principle of the structured programming approach is to divide a program into functions and modules. The use of modules and functions makes the program more understandable and readable. It helps to write cleaner code and to maintain control over the functions and modules. This approach gives importance to functions rather than data. It focuses on the development of large software applications, for example, C was used for modern operating system development. The programming languages: PASCAL (introduced by Niklaus Wirth) and C (introduced by Dennis Ritchie) follow this approach.

## Procedural Programming Approach

This approach is also known as the top-down approach. In this approach, a program is divided into functions that perform specific tasks. This approach is mainly used for medium-sized applications. Data is global, and all the functions can access global data. The basic drawback of the procedural programming approach is that data is not secured because data is global and can be accessed by any function. Program control flow is achieved through function calls and goto statements. The programming languages: FORTRAN (developed by IBM) and COBOL (developed by Dr. Grace Murray Hopper) follow this approach.   
These programming constructs were developed in the late 1970s and 1980s. There were still some issues with these languages, though they fulfilled the criteria of well-structured programs, software, etc. They were not as structured as the requirements were at that time. They seem to be over-generalized and don’t correlate with real-time applications.  
To solve such kinds of problems, OOP, an object-oriented approach was developed as a solution.

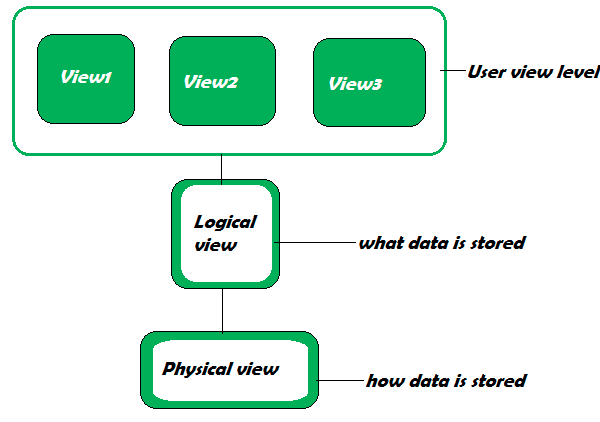


## The Object-Oriented Programming (OOP) Approach

The OOP concept was basically designed to overcome the drawback of the above programming methodologies, which were not so close to real-world applications. The demand increased, but still, conventional methods were used. This new approach brought a revolution in the programming methodology field.  
Object-oriented programming (OOP) is nothing but that which allows the writing of programs with the help of certain classes and real-time objects. We can say that this approach is very close to the real-world and its applications because the state and behaviour of these classes and objects are almost the same as real-world objects.  
Let’s go deeper into the general concepts of OOP, which are given below:

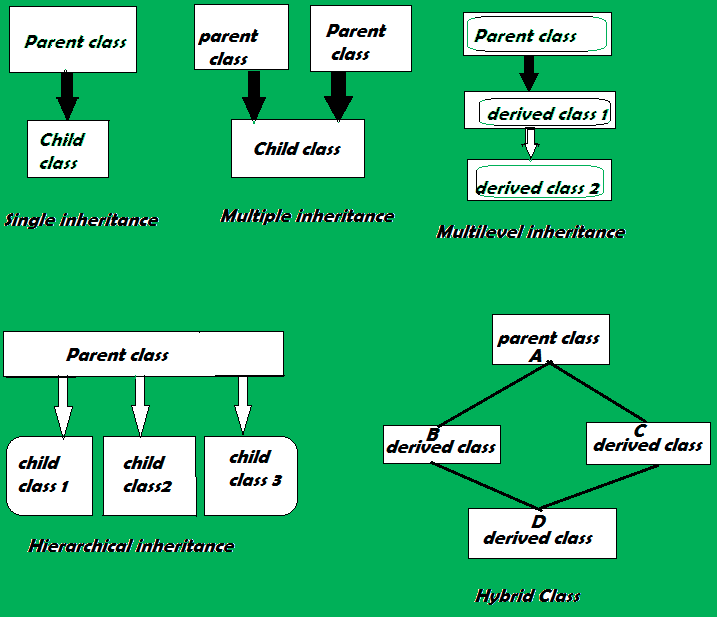
## What Are Class & Object?

It is the basic concept of OOP; an extended concept of the structure used in C. It is an abstract and user-defined data type. It consists of several variables and functions. The primary purpose of the class is to store data and information. The members of a class define the behaviour of the class. A class is the blueprint of the object, but also, we can say the implementation of the class is the object. The class is not visible to the world, but the object is.  
Data Abstraction –   
Abstraction refers to the act of representing important and special features without including the background details or explanation about that feature. Data abstraction simplifies database design.

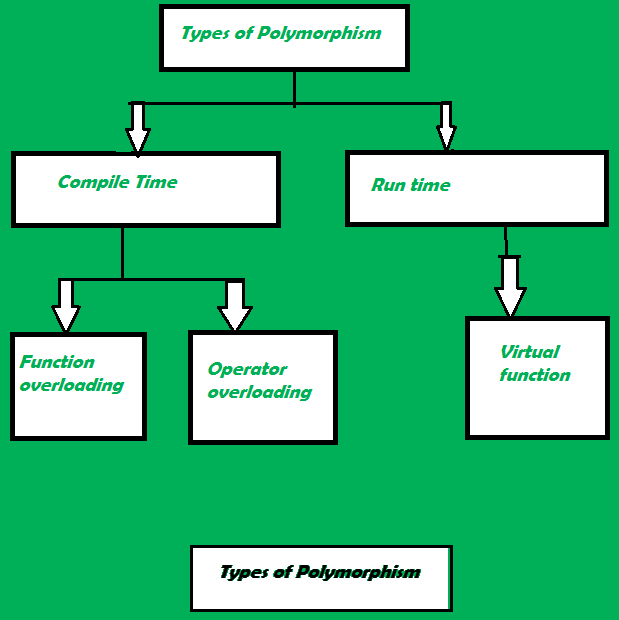


1. **Physical Level:**   
   It describes how the records are stored, which are often hidden from the user. It can be described with the phrase, “block of storage.”
2. **Logical Level:**   
   It describes data stored in the database and the relationships between the data. The programmers generally work at this level as they are aware of the functions needed to maintain the relationships between the data.
3. **View Level:**   
   Application programs hide details of data types and information for security purposes. This level is generally implemented with the help of GUI, and details that are meant for the user are shown.

**Encapsulation –**   
Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit, e.g., a class in Java. This concept is often used to hide the internal state representation of an object from the outside.  
**Inheritance –**   
Inheritance is the ability of one class to inherit capabilities or properties of another class, called the parent class. When we write a class, we inherit properties from other classes. So when we create a class, we do not need to write all the properties and functions again and again, as these can be inherited from another class that possesses it. Inheritance allows the user to reuse the code whenever possible and reduce its redundancy. 



**Polymorphism –**   
Polymorphism is the ability of data to be processed in more than one form. It allows the performance of the same task in various ways. It consists of method overloading and method overriding, i.e., writing the method once and performing a number of tasks using the same method name.



**Some important points to know about OOP:** 

1. OOP treats data as a critical element.
2. Emphasis is on data rather than procedure.
3. Decomposition of the problem into simpler modules.
4. Doesn’t allow data to freely flow in the entire system, ie localized control flow.
5. Data is protected from external functions.

**Advantages of OOPs –** 

* It models the real world very well.
* With OOP, programs are easy to understand and maintain.
* OOP offers code reusability. Already created classes can be reused without having to write them again.
* OOP facilitates the quick development of programs where parallel development of classes is possible.
* With OOP, programs are easier to test, manage and debug.

**Disadvantages of OOP –** 

* With OOP, classes sometimes tend to be over-generalized.
* The relations among classes become superficial at times.
* The OOP design is tricky and requires appropriate knowledge. Also, one needs to do proper planning and design for OOP programming.
* To program with OOP, the programmer needs proper skills such as design, programming, and thinking in terms of objects and classes, etc.

**Software Measurement:** A measurement is a manifestation of the size, quantity, amount, or dimension of a particular attribute of a product or process. Software measurement is a titrate impute of a characteristic of a software product or the software process. It is an authority within software engineering. The software measurement process is defined and governed by ISO Standard.

**Software Measurement Principles:**

The software measurement process can be characterized by five activities-

1. **Formulation:**The derivation of software measures and metrics appropriate for the representation of the software that is being considered.
2. **Collection:** The mechanism used to accumulate data required to derive the formulated metrics.
3. **Analysis:** The computation of metrics and the application of mathematical tools.
4. **Interpretation:**The evaluation of metrics resulting in insight into the quality of the representation.
5. **Feedback:**Recommendation derived from the interpretation of product metrics transmitted to the software team.

**Need for Software Measurement:**

Software is measured to:

* Create the quality of the current product or process.
* Anticipate future qualities of the product or process.
* Enhance the quality of a product or process.
* Regulate the state of the project in relation to budget and schedule.
* Enable data-driven decision-making in project planning and control.
* Identify bottlenecks and areas for improvement to drive process improvement activities.
* Ensure that industry standards and regulations are followed.
* Give software products and processes a quantitative basis for evaluation.
* Enable the ongoing improvement of software development practices.

**Classification of Software Measurement:**

There are 2 types of software measurement:

1. **Direct Measurement:** In direct measurement, the product, process, or thing is measured directly using a standard scale.
2. **Indirect Measurement:** In indirect measurement, the quantity or quality to be measured is measured using related parameters i.e. by use of reference.

**Metrics:**

A metric is a measurement of the level at which any impute belongs to a system product or process.

Software metrics will be useful only if they are characterized effectively and validated so that their worth is proven. There are 4 functions related to software metrics:

1. Planning
2. Organizing
3. Controlling
4. Improving

**Characteristics of software Metrics:**

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

**Classification of Software Metrics:**

There are 3 types of software metrics:

1. **Product Metrics:** Product metrics are used to evaluate the state of the product, tracing risks and undercover prospective problem areas. The ability of the team to control quality is evaluated. Examples include lines of code, cyclomatic complexity, code coverage, defect density, and code maintainability index.
2. **Process Metrics:** Process metrics pay particular attention to enhancing the long-term process of the team or organization. Examples include effort variance, schedule variance, defect injection rate, and lead time.
3. **Project Metrics:** The project matrix describes the project characteristic and execution process. Examples include effort estimation accuracy, schedule deviation, cost variance, and productivity.
   * Number of software developer
   * Staffing patterns over the life cycle of software
   * Cost and schedule
   * Productivity

**Advantages of Software Metrics :**

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

**Disadvantages of Software Metrics :**

1. It is expensive and difficult to implement the metrics in some cases.
2. Performance of the entire team or an individual from the team can’t be determined. Only the performance of the product is determined.
3. Sometimes the quality of the product is not met with the expectation.
4. It leads to measuring unwanted data which is a waste of time.
5. Measuring the incorrect data leads to wrong decision making.

 Halstead’s Software metrics are a set of measures proposed by Maurice Halstead to evaluate the complexity of a [software program](https://www.geeksforgeeks.org/difference-between-software-and-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.

### Field of Halstead Metrics

* **Program length (N):** This is the total number of operator and operand occurrences in the program.
* **Vocabulary size (n):** This is the total number of distinct operators and operands in the program.
* **Program volume (V):** This is the product of program length (N) and the logarithm of vocabulary size (n), i.e., V = N\*log2(n).
* **Program level (L):** This is the ratio of the number of operator occurrences to the number of operand occurrences in the program, i.e., L = n1/n2, where n1 is the number of operator occurrences and n2 is the number of operand occurrences.
* **Program difficulty (D):** This is the ratio of the number of unique operators to the total number of operators in the program, i.e., D = (n1/2) \* (N2/n2).
* **Program effort (E):** This is the product of program volume (V) and program difficulty (D), i.e., E = V\*D.
* **Time to implement (T):** This is the estimated time required to implement the program, based on the program effort (E) and a constant value that depends on the programming language and development environment.

Halstead’s software metrics can be used to estimate the size, complexity, and effort required to develop and maintain a software program. However, they have some limitations, such as the assumption that all operators and operands are equally important, and the assumption that the same set of metrics can be used for different programming languages and development environments.

Overall, Halstead’s software metrics can be a useful tool for software developers and project managers to estimate the effort required to develop and maintain software programs.

n1 = Number of distinct operators.   
n2 = Number of distinct operands.   
N1 = Total number of occurrences of operators.   
N2 = Total number of occurrences of operands.

## Halstead Metrics

Halstead metrics are:

* **Halstead Program Length:** The total number of operator occurrences and the total number of operand occurrences.   
  N = N1 + N2   
  And estimated program length is, N^ = n1log2n1 + n2log2n2   
  The following alternate expressions have been published to estimate program length:
  + NJ = log2(n1!) + log2(n2!)
  + NB = n1 \* log2n2 + n2 \* log2n1
  + NC = n1 \* sqrt(n1) + n2 \* sqrt(n2)
  + NS = (n \* log2n) / 2
* **Halstead Vocabulary:** The total number of unique operators and unique operand occurrences.   
  n = n1 + n2
* **Program Volume:** Proportional to program size, represents the size, in bits, of space necessary for storing the program. This parameter is dependent on specific algorithm implementation. The properties V, N, and the number of lines in the code are shown to be linearly connected and equally valid for measuring relative program size.   
  V = Size \* (log2 vocabulary) = N \* log2(n)   
  The unit of measurement of volume is the common unit for size “bits”. It is the actual size of a program if a uniform binary encoding for the vocabulary is used. And error = Volume / 3000
* **Potential Minimum Volume:** The potential minimum volume V\* is defined as the volume of the most succinct program in which a problem can be coded.   
  V\* = (2 + n2\*) \* log2(2 + n2\*)   
  Here, n2\* is the count of unique input and output parameters
* **Program Level:** To rank the programming languages, the level of abstraction provided by the programming language, Program Level (L) is considered. The higher the level of a language, the less effort it takes to develop a program using that language.   
  L = V\* / V   
  The value of L ranges between zero and one, with L=1 representing a program written at the highest possible level (i.e., with minimum size).   
  And estimated program level is L^ =2 \* (n2) / (n1)(N2)
* **Program Difficulty:** This parameter shows how difficult to handle the program is.   
  D = (n1 / 2) \* (N2 / n2)   
  D = 1 / L   
  As the volume of the implementation of a program increases, the program level decreases and the difficulty increases. Thus, programming practices such as redundant usage of 6++operands, or the failure to use higher-level control constructs will tend to increase the volume as well as the difficulty.
* **Programming Effort:** Measures the amount of mental activity needed to translate the existing algorithm into implementation in the specified program language.   
  E = V / L = D \* V = Difficulty \* Volume
* **Language Level:** Shows the algorithm implementation program language level. The same algorithm demands additional effort if it is written in a [low-level program language](https://www.geeksforgeeks.org/difference-between-high-level-and-low-level-languages/). For example, it is easier to program in Pascal than in[Assembler](https://www.geeksforgeeks.org/introduction-of-assembler/).   
  L’ = V / D / D   
  lambda = L \* V\* = L2 \* V
* **Intelligence Content:** Determines the amount of intelligence presented (stated) in the program This parameter provides a measurement of program complexity, independently of the programming language in which it was implemented.   
  I = V / D
* **Programming Time:** Shows time (in minutes) needed to translate the existing algorithm into implementation in the specified program language.   
  T = E / (f \* S)   
  The concept of the processing rate of the human brain, developed by psychologist John Stroud, is also used. Stroud defined a moment as the time required by the human brain to carry out the most elementary decision. The Stroud number S is therefore Stroud's moments per second with:   
  5 <= S <= 20. Halstead uses 18. The value of S has been empirically developed from psychological reasoning, and its recommended value for programming applications is 18.   
  Stroud number S = 18 moments / second   
  seconds-to-minutes factor f = 60

## **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.