Unit 3
Software Engineering

Prepared By

Abhishek Kesharwani

Assistant Professor ,United College of Engineering and Research

Index

Function Oriented Design

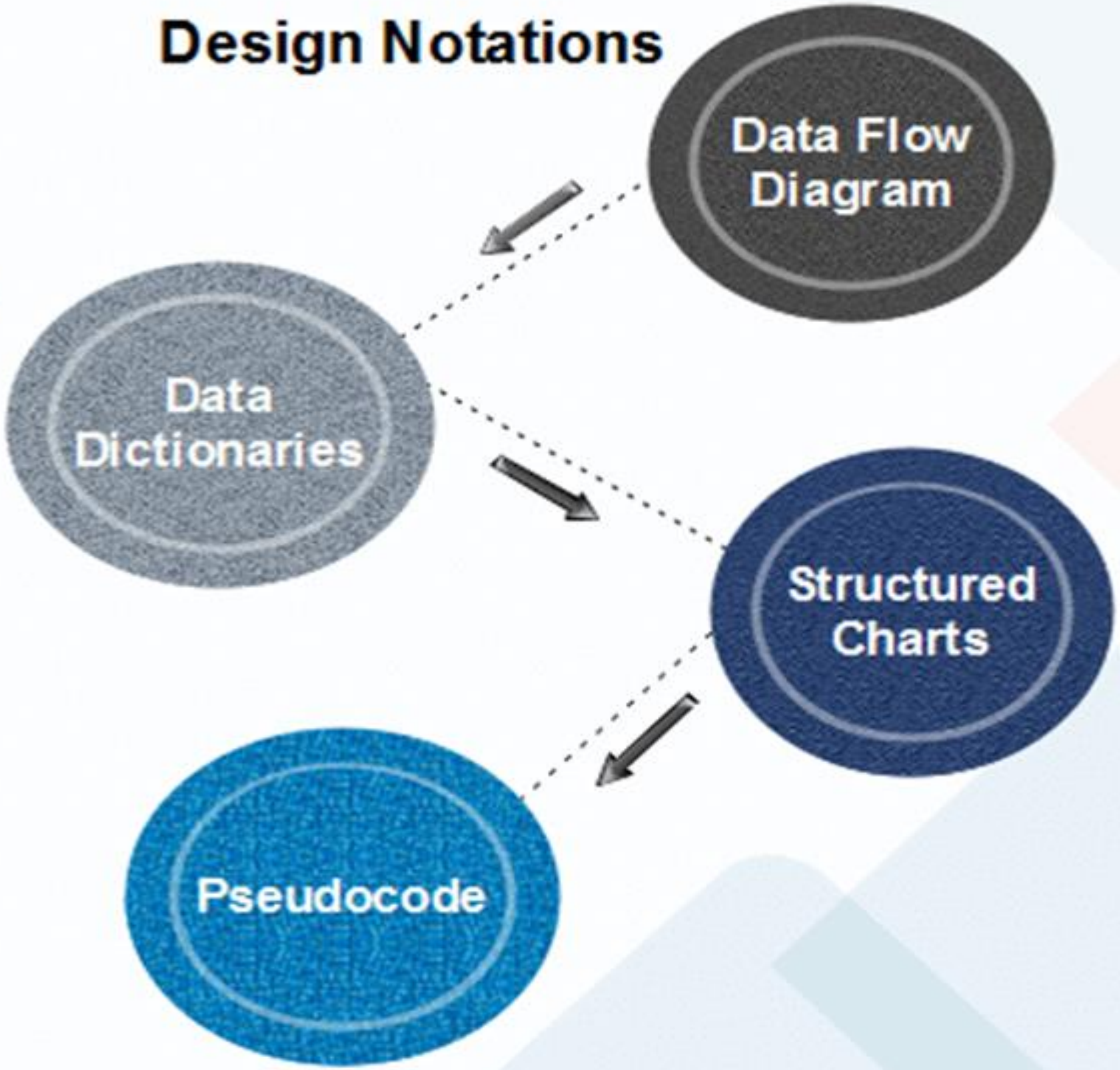
Function Oriented Design is an approach to software design where the design is decomposed into a set of interacting units where each unit has a clearly defined function.

Design Notations

Design Notations are primarily meant to be used during the process of design and are used to represent design or design decisions. For a function-oriented design, the design can be represented graphically or mathematically by the following:

- DFD
- Data Dictionary
- Structured Chart
- Pseudo Code

Design Notations



Data Flow Diagram

Data-flow design is concerned with designing a series of functional transformations that convert system inputs into the required outputs. The design is described as data-flow diagrams. These diagrams show how data flows through a system and how the output is derived from the input through a series of functional transformations.

Data Dictionaries

A data dictionary lists all data elements appearing in the DFD model of a system. The data items listed contain all data flows and the contents of all data stores looking on the DFDs in the DFD model of a system.

Structured Charts

It partitions a system into block boxes. A Black box system that functionality is known to the user without the knowledge of internal design.

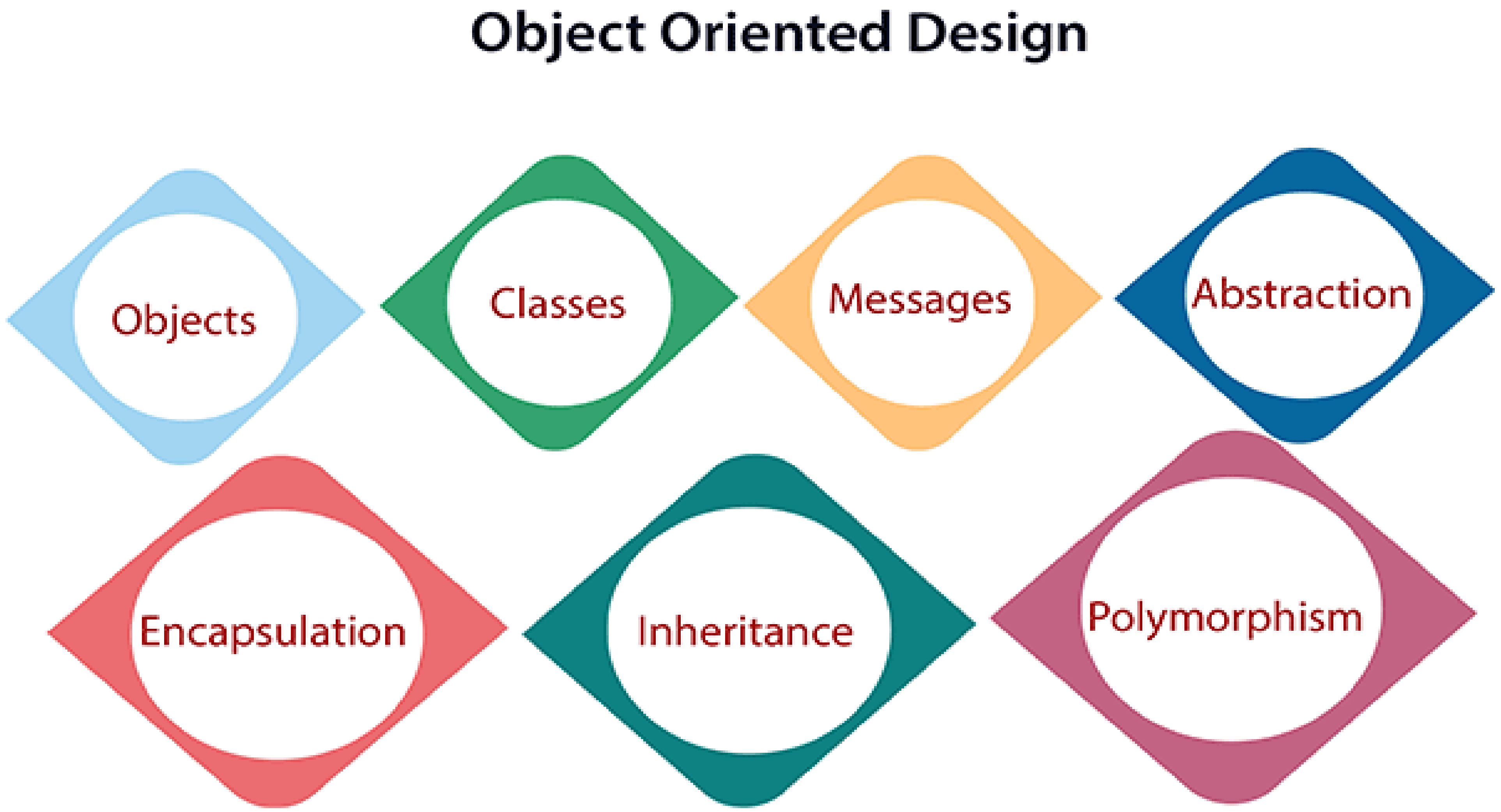
Pseudo-code

Pseudo-code notations can be used in both the preliminary and detailed design phases. Using pseudo-code, the designer describes system characteristics using short, concise, English Language phases that are structured by keywords such as If-Then-Else, While-Do, and End.

Object-Oriented Design

- In the object-oriented design method, the system is viewed as a collection of objects (i.e., entities).
- The state is distributed among the objects, and each object handles its state data.
- For example, in a Library Automation Software, each library representative may be a separate object with its data and functions to operate on these data.
- The tasks defined for one purpose cannot refer or change data of other objects. Objects have their internal data which represent their state.
- Classes may inherit features from the superclass.

The different terms related to object design are:



1. **Objects:** All entities involved in the solution design are known as objects. For example, person, banks, company, and users are considered as objects. Every entity has some attributes associated with it and has some methods to perform on the attributes.
2. **Classes:** A class is a generalized description of an object. An object is an instance of a class. A class defines all the attributes, which an object can have and methods, which represents the functionality of the object.
3. **Messages:** Objects communicate by message passing. Messages consist of the integrity of the target object, the name of the requested operation, and any other action needed to perform the function. Messages are often implemented as procedure or function calls.

4.Abstraction In object-oriented design, complexity is handled using abstraction. Abstraction is the removal of the irrelevant and the amplification of the essentials.

5.Encapsulation: Encapsulation is also called an information hiding concept. The data and operations are linked to a single unit. Encapsulation not only bundles essential information of an object together but also restricts access to the data and methods from the outside world.

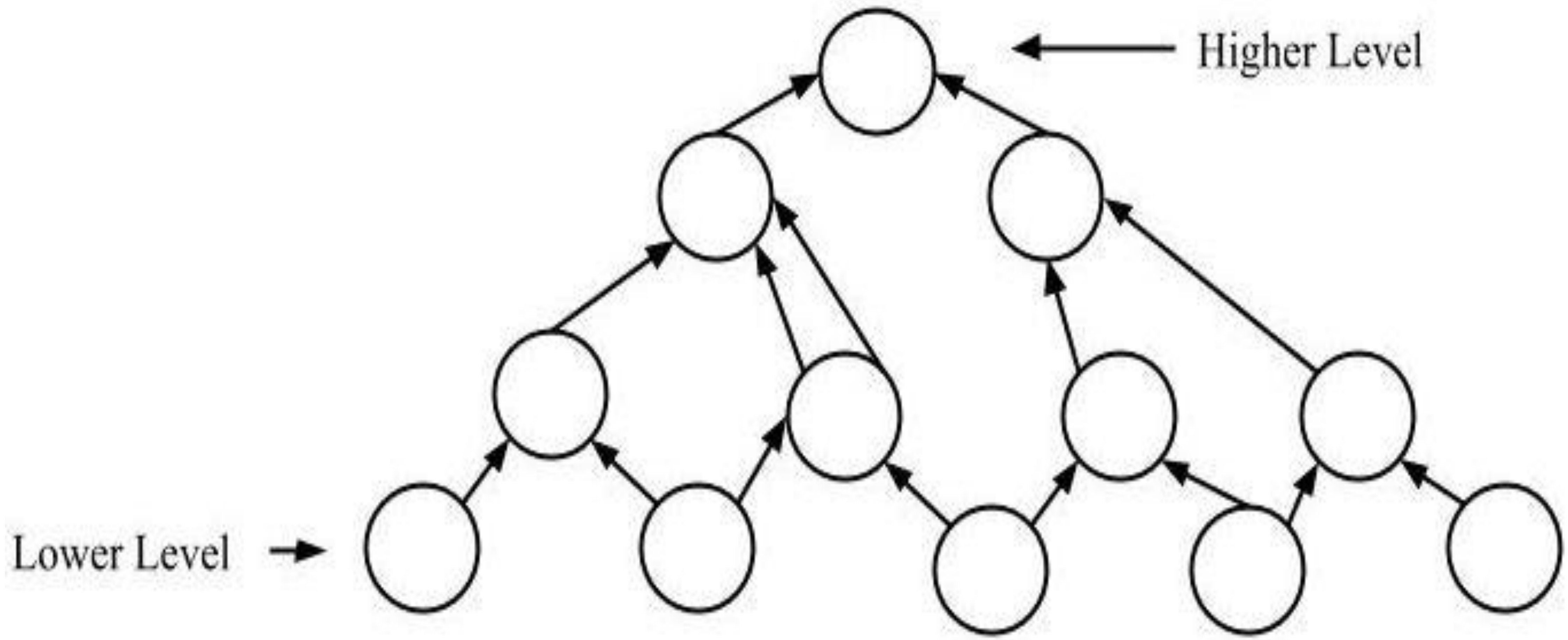
6.Inheritance: OOD allows similar classes to stack up in a hierarchical manner where the lower or sub-classes can import, implement, and re-use allowed variables and functions from their immediate superclasses.This property of OOD is called an inheritance. This makes it easier to define a specific class and to create generalized classes from specific ones.

7.Polymorphism: OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned the same name. This is known as polymorphism, which allows a single interface is performing functions for different types. Depending upon how the service is invoked, the respective portion of the code gets executed.

Top-Down and Bottom-Up Design

Bottom-up approach:

- The design starts with the lowest level components and subsystems.
- By using these components, the next immediate higher-level components and subsystems are created or composed.
- The process is continued till all the components and subsystems are composed into a single component, which is considered as the complete system.
- The amount of abstraction grows high as the design moves to more high levels.
- By using the basic information existing system, when a new system needs to be created, the bottom-up strategy suits the purpose.


Higher Level
Lower Level

Advantages:

- The economics can result when general solutions can be reused.
- It can be used to hide the low-level details of implementation and be merged with the top-down technique.

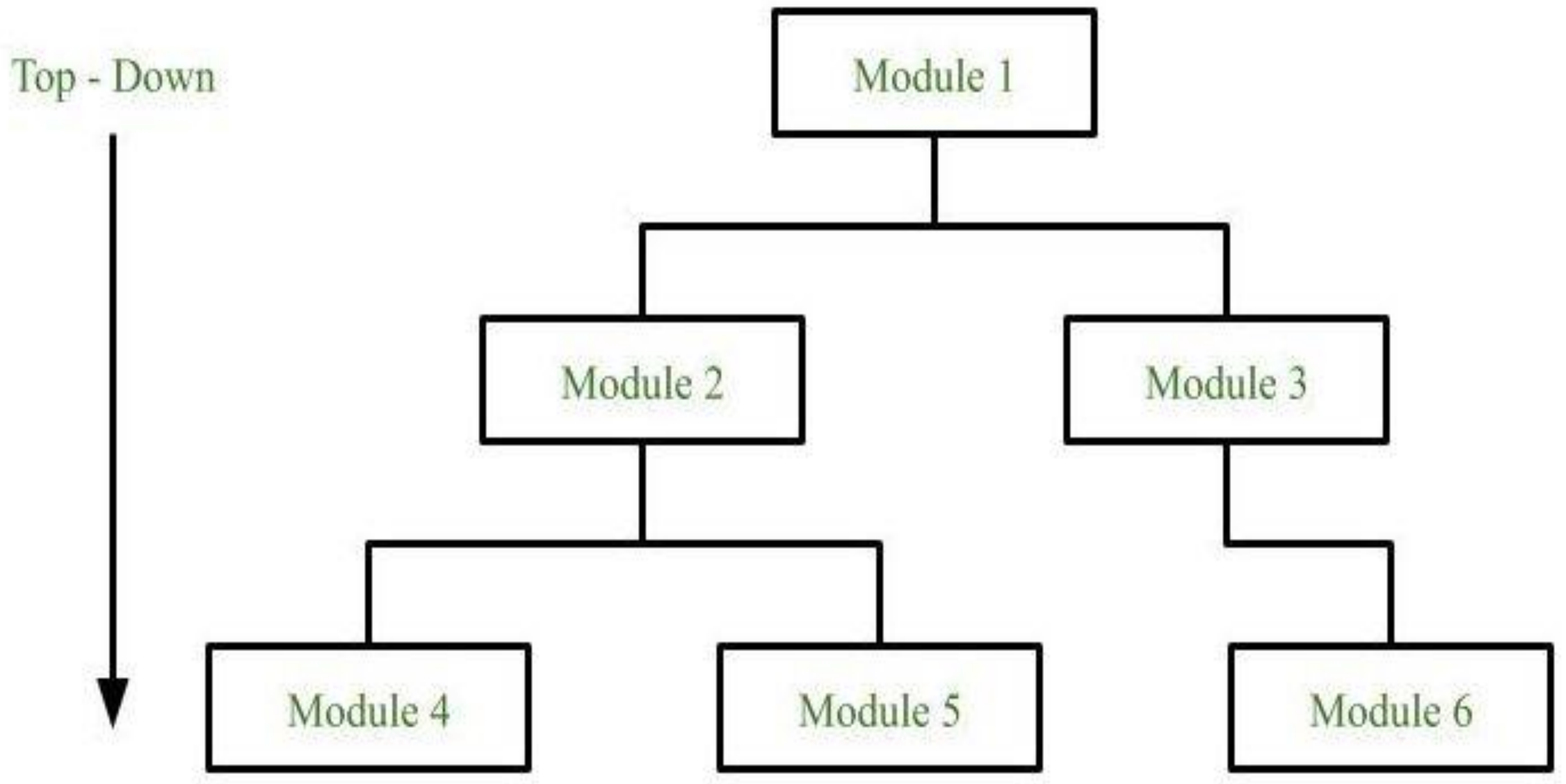
Disadvantages:

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

Top-down approach:

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

- The design is started initially by defining the system as a whole and then keeps on adding definitions of the subsystems and components. When all the definitions are combined together, it turns out to be a complete system.
- For the solutions of the software that need to be developed from the ground level, top-down design best suits the purpose.


Top - Down
Module 1
Module 2
Module 3
Module 4
Module 5
Module 6

Advantages:

- The main advantage of the top-down approach is that its strong focus on requirements helps to make a design responsive according to its requirements.

Disadvantages:

- Project and system boundaries tend to be application specification-oriented. Thus it is more likely that advantages of component reuse will be missed.
- The system is likely to miss, the benefits of a well-structured, simple architecture.

Software Measurement

Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined unambiguous rules.

Metrics and measurements are necessary aspects of managing a software development project . For effective monitoring, management needs to get information about the project:

- how far it has progressed
- how much development has taken place
- how far behind schedule it is, and
- the quality of the development so far

Based on this information, decisions, can be made about the project.

Software Metrics: What and Why ?

Software metric is defined as a quantitative measure of an attribute a software system possesses with respect to Cost, Quality, Size and Schedule.

Example-

Measure - No. of Errors

Metrics - No. of Errors found per person

These are numerical data related to software development. Metrics strongly support software project management activities.

- ❖ Pressman explained as “A **measure** provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of the product or process”.
- ❖ **Measurement** is the act of determine a measure
- ❖ The **metric** is a quantitative measure of the degree to which a system, component, or process possesses a given attribute.

Areas of Applications

The most established area of software metrics is **cost** and **size estimation techniques**.

The **prediction of quality levels** for software, often in terms of **reliability**, is another area where software metrics have an important role to play.

The use of software metrics to provide **quantitative checks on software design** is also a well established area.

Categories of Metrics

i. **Product metrics:** describe the characteristics of the product such as size, complexity, design features, performance, efficiency, reliability, portability, etc.

ii. **Process metrics:** describe the effectiveness and quality of the processes that produce the software product. Examples are:

- effort required in the process
- time to produce the product
- effectiveness of defect removal during development
- number of defects found during testing

iii. Project metrics: describe the project characteristics and execution. Examples are :

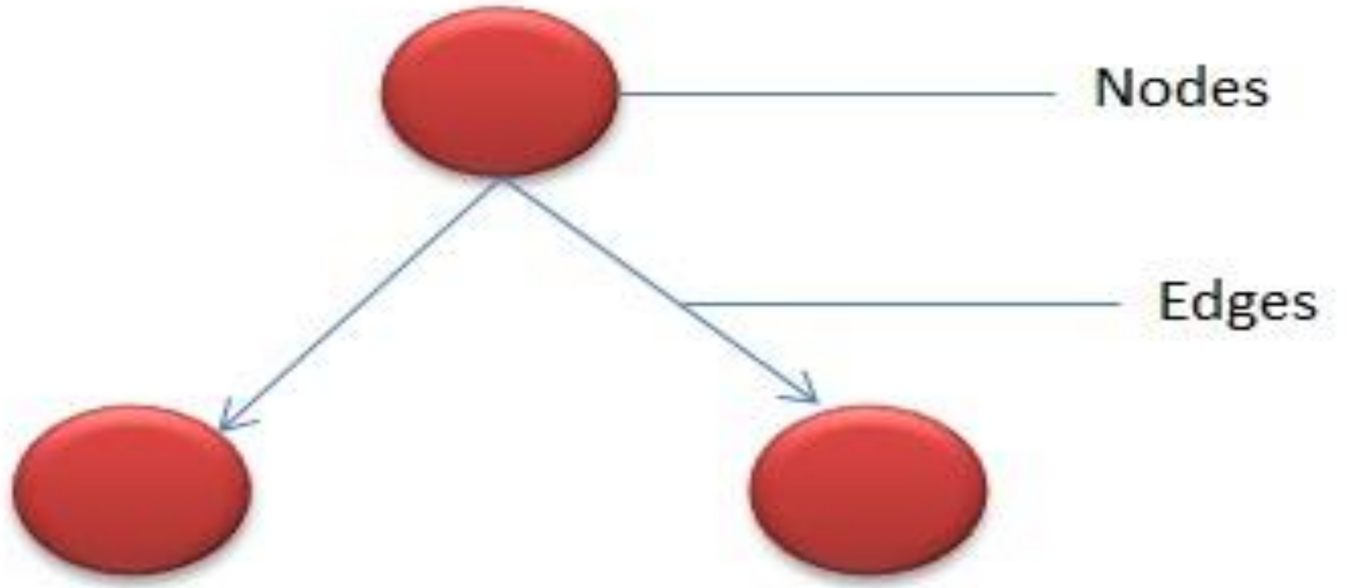
- number of software developers
- cost and schedule
- productivity

Control flow graph

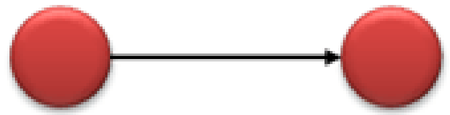
- A control flow graph (CFG) describes:
  - the sequence in which different instructions of a program get executed.
  - the way control flows through the program.

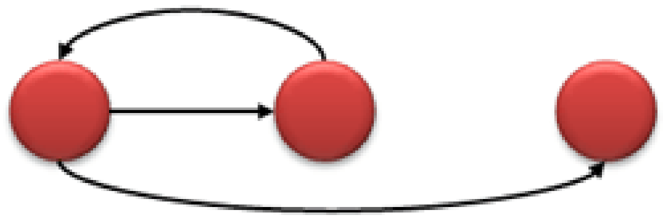
- Control flow depicts a program as a graph which consists of Nodes and Edges.

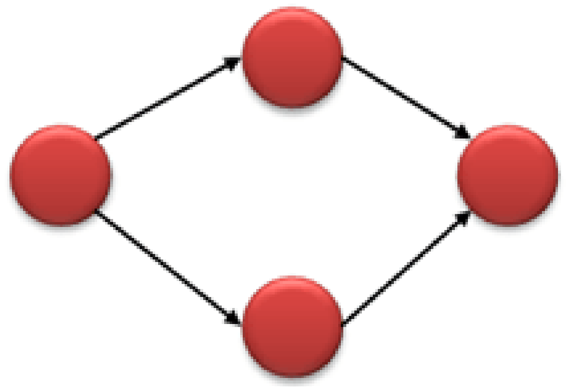
In the graph, Nodes represent processing tasks while edges represent control flow between the nodes.

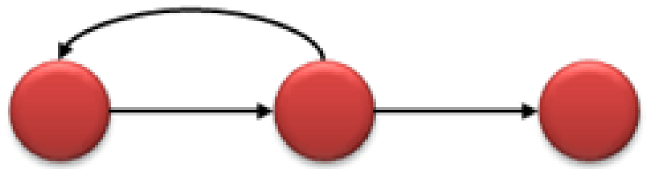


Flow graph notation for a program:


Sequence


While


If-then-else


Until

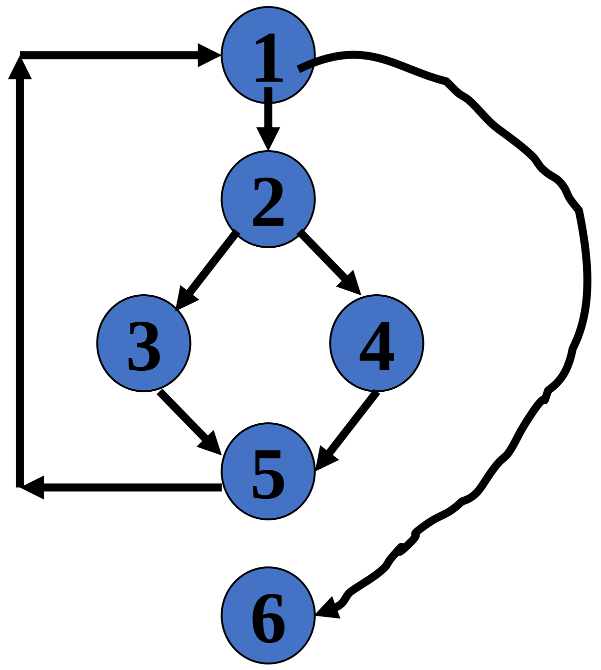
How to draw Control flow graph?

- Number all the statements of a program.
- Numbered statements:
  - - represent nodes of the control flow graph.
- An edge from one node to another node exists:
  - - if execution of the statement representing the first node can result in transfer of control to the other node.

Example

```
int f1(int x,int y){
while (x != y){
    if (x>y) then
         x=x-y;
    else y=y-x;
 }
 return x;        }
```

Example Control Flow Graph



Cyclomatic complexity = 7 – 6 + 2 = 3.

CYCLOMATIC COMPLEXITY

This metric was **developed by Thomas J. McCabe in 1976** and it is based on a control flow representation of the program.

McCabe's cyclomatic complexity is a software quality metric that **quantifies the complexity of a software program.**
Complexity is inferred by **measuring the number of linearly independent paths** through the program. The **higher the number the more complex the code**.
Independent path is defined as a path that has atleast one edge which has not been traversed before in any other paths.

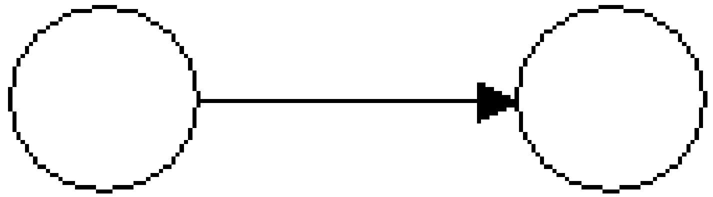
Cyclomatic complexity can be calculated with respect to functions, modules, methods or classes within a program.

Different ways to compute CC

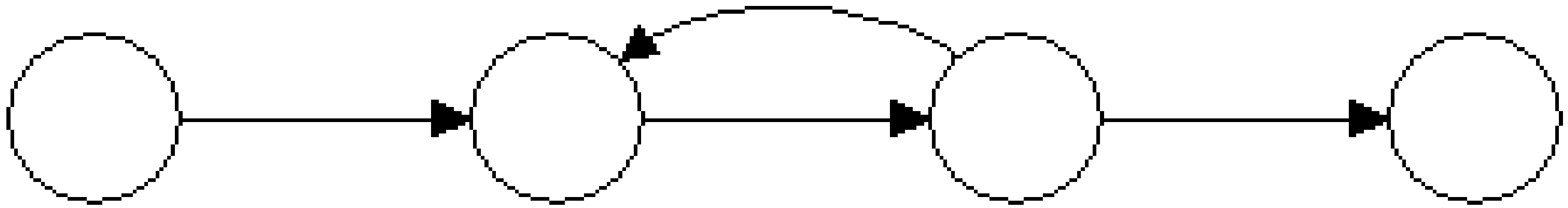
There are three methods:-

Method1: Cyclomatic complexity is derived from the control flow graph of a program as follows:
Cyclomatic complexity(CC), $V(G) = E - N + 2P$
Where:

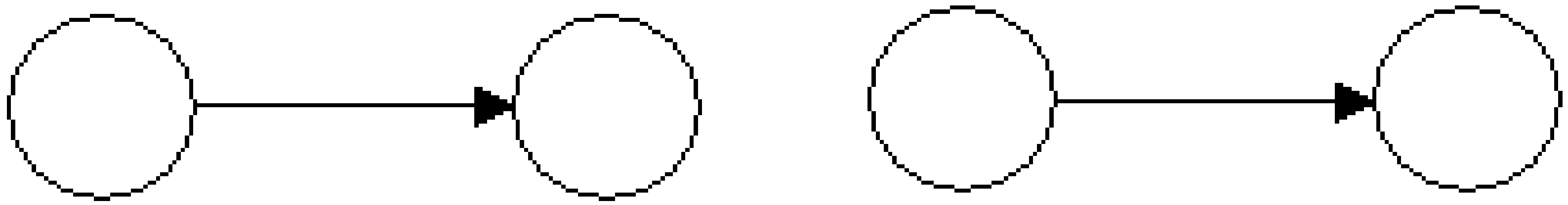
E = number of edges (transfers of control)
N = number of nodes (sequential group of statements containing only one transfer of control)
P = number of disconnected parts of the flow graph (e.g. a calling program and a subroutine) or number of strongly connected components



E = 1 N = 2 P=1 CC = 1-2+2 = 1



E = 4 N = 4 P=1 CC = 4-4+2 = 2



E = 2 N = 4 P=2 CC = 2-4+2x2 = 2

Different ways to compute CC

Method 2: An alternative way of computing the Cyclomatic complexity of a program from an inspection of its control flow graph is as follows:-

V(G) = Total number of bounded areas + 1

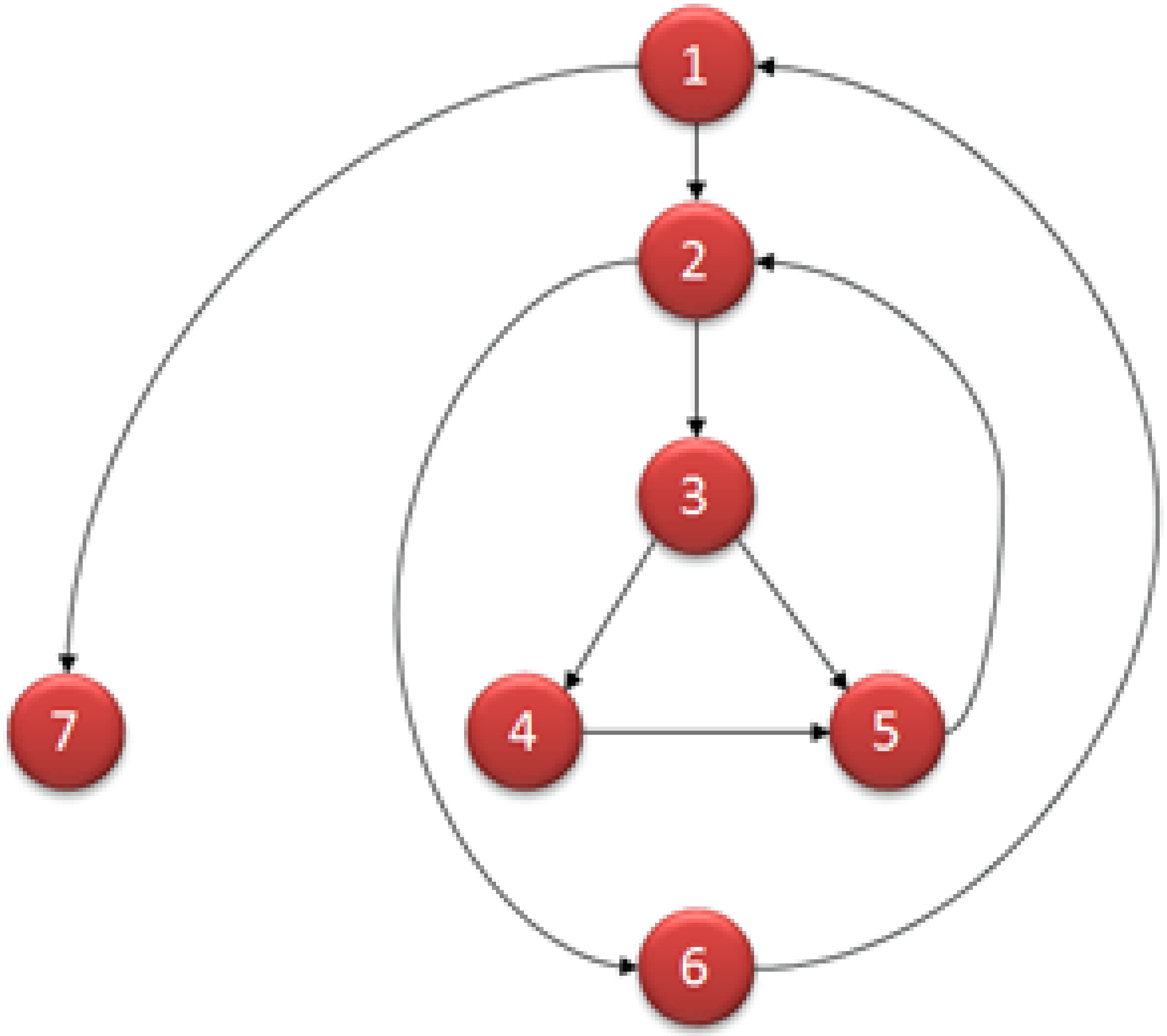
•Any region enclosed by nodes and edges can be called as a bounded area.

Method 3: Cyclomatic complexity can also be computed by computing the number of decision statements.

V (G) = Π + 1

Where **Π** = number of decision statement of a program.

Flow graph for program will be



Flow graph notation

Computing mathematically,

V(G) = 9 - 7 + 2 = 4

V(G) = 3 + 1 = 4 (Condition nodes are 1,2 and 3 nodes)

Basis Set - A set of possible execution path of a program

or

There will be four independent paths for the flow graph.

Path1: 1, 7

Path2: 1, 2, 6, 1, 7

Path3: 1, 2, 3, 4, 5, 2, 6, 1, 7

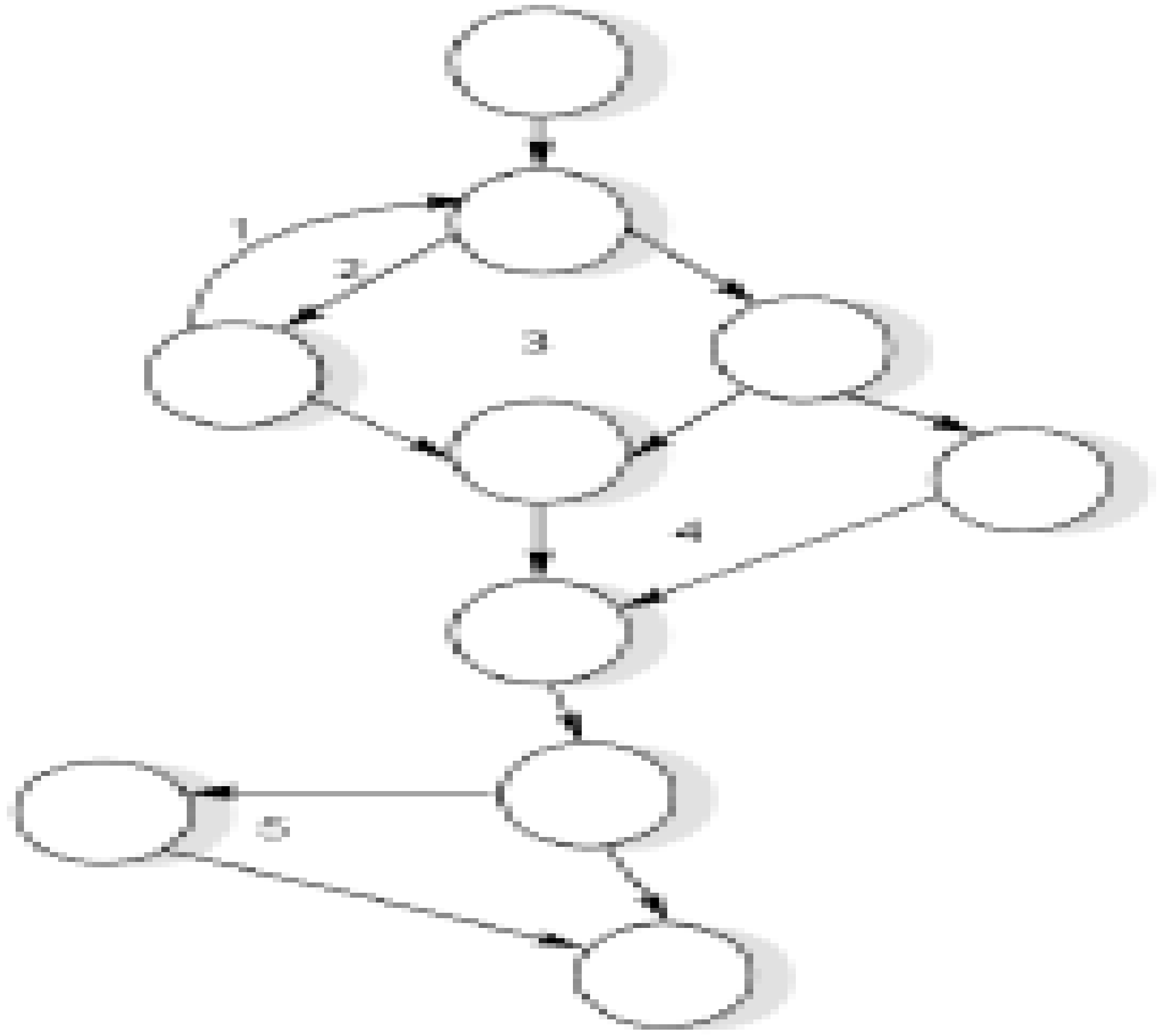
Path4: 1, 2, 3, 5, 2, 6, 1, 7

Properties of Cyclomatic complexity:

Following are the properties of Cyclomatic complexity:

1. V (G) >=1
2. V (G) is the maximum number of independent paths in the graph
3. Inserting and deleting functional statements to G does not affect V(G)
4. G will have only one path if and only if V (G) = 1
5. Inserting a new row in G increases V(G) by unity.

Example



SOLUTION

It can be calculated by any of the three methods

1. V(G) = E – N + 2P
   = 13 – 10 + 2
   = 5

2. V(G) = **Π + 1**
   = 4 + 1 = 5

3. V(G) = no. of regions + 1;
   = 4 + 1 = 5

HALSTEAD’s SOFTWARE SCIENCE METRICS

It is an analytical technique to measure **size, development effort and development cost of software products**.

Halstead used a few primitive program parameters to develop the expressions for overall program length, potential minimum value, actual volume, effort and development time.

Token Count

The **size of the vocabulary** of a program, which consists of the number of unique tokens used to build a program is defined as:

$$\eta = \eta_1 + \eta_2$$

where

η : vocabulary of a program

η_1 : number of unique operators

η_2 : number of unique operands

The **length of the program** in the terms of the total number of tokens used is

$$N = N_1 + N_2$$

where

N : program length

N_1 : total occurrences of operators

N_2 : total occurrences of operands

Program Volume

$$V = N * \log_2 \eta$$

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.

Program Length Equation

$$N` = \eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$$

` on N means it is calculated rather than counted

Potential Volume

$V^* = (\eta_1^* + \eta_2^*) \log_2 (\eta_1^* + \eta_2^*)$ where $\eta_1^* = 2$

η_2^* represents the no. of conceptually unique input and output parameters.

Program Level

$L = V^* / V$

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).

Program Difficulty

$$D = 1 / L$$

As the volume of an implementation of a program increases, the program level decreases and the difficulty increases. Thus, programming practices such as redundant usage of operands, or the failure to use higher-level control constructs will tend to increase the volume as well as the difficulty.

Effort Equation

$$E = V / L = D * V$$

The unit of measurement of E is elementary mental discriminations.

Time equation

$T' = E/S$

John Stroud defined a moment as the time required by the human brain to perform the most elementary discrimination. The Stroud number S is 18.

.

Advantages of Halstead Metrics

- Simple to calculate
- Do not require in-depth analysis of programming structure.
- Measure overall quality of programs.
- Predicts maintenance efforts.
- Useful in scheduling the projects.

Counting rules for C language

1. Comments are not considered.

2. All the variables and constants are considered operands.

3. Global variables used in different modules of the same program are counted as multiple occurrences of the same variable.

4. Local variables with the same name in different functions are counted as unique operands.

5. Functions calls are considered as operators.

6. All looping statements e.g., do {…} while ( ), while ( ) {…}, for ( ) {…}, all control statements e.g., if ( ) {…}, if ( ) {…} else {…}, etc. are considered as operators.

7. In control construct switch ( ) {case:…}, switch as well as all the case statements are considered as operators.

8. The reserve words like return, default, continue, break, sizeof, etc., are considered as operators.

9. All the brackets, commas, and terminators are considered as operators.

10. GOTO is counted as an operator and the label is counted as an operand.

11. The unary and binary occurrence of “+” and “-” are dealt separately. Similarly “*” (multiplication operator) are dealt with separately.

12. In the array variables such as “array-name [index]” “array- name” and “index” are considered as operands and [ ] is considered as operator.

13. In the structure variables such as “struct-name, member-name” or “struct-name -> member-name”, struct-name, member-name are taken as operands and ‘.’, ‘->’ are taken as operators. Some names of member elements in different structure variables are counted as unique operands.

14. All the hash directive are ignored.

Example

```
z=0;
while x>0
        z=z+y;
        x=x-1;
end while;
print (z);
```

Operators	Count	Operands	Count
=	3	z	4
;	5	x	3
while-end while	1	y	1
>	1	0	2
+	1	1	1
-	1	-	-
print	1	-	-
()	1	-	-

- Here, $\eta_1 = 8$, $\eta_2 = 5$, $N_1 = 14$, $N_2 = 11$

- **Halstead's Software Metrics**
- Program Length (N) = N1+N2 = 25
- Program volume (V) = N * $\log_2 \eta$ = 92.51
- Program Length Equation N` = $\eta_1 \log_2 \eta_1 + \eta_2 \log_2 \eta_2$ = 35.61
- Potential Volume(V*) = $(2 + \eta_2^*) \log_2 (2 + \eta_2^*)$ = 8
- Program length(N) = N Program Level (L) = V*/V = 0.086
- Program Difficulty (D) = 1/L = 11.56
- Programming Effort (E) = V/L = D*V = 1069.76
- Time (T') = E/18 = 59 sec

Example

```
int f=1, n=7;
for (int i=1; i<=n; i+=1)
f*=i;
```

Example

```
int f=1, n=7; for (int i=1; i<=n; i+=1)   f*=i;
```

Operators	Count	Operands	Count
int	2	f	2
=	3	1	3
,	1	n	2
;	4	7	1
for	1	i	4
()	1	-	-
<=	1	-	-
+=	1	-	-
*=	1	-	-
9	15	5	12

Solution

N, the length of the program to be N1+N2 = 27

V, the volume of the program, to be N × $\log_2(\eta_1 + \eta_2)$ = 102.8

Halstead's Software Metrics

- Program Length (N) = N1+N2 = 27
- Program volume (V) = N * $\log_2 \eta$ = 1 0 2 . 8
- Potential Volume(V*) = $(2 + \eta_2^*) \log_2 (2 + \eta_2^*)$ = 2
- Program length(N) = N Program Level (L) = V*/V = 0.019
- Program Difficulty (D) = 1/L = 51.4
- Programming Effort (E) = V/L = D*V = 5283.92
- Time (T') = E/18 = 293 sec

Functional Point (FP) Analysis

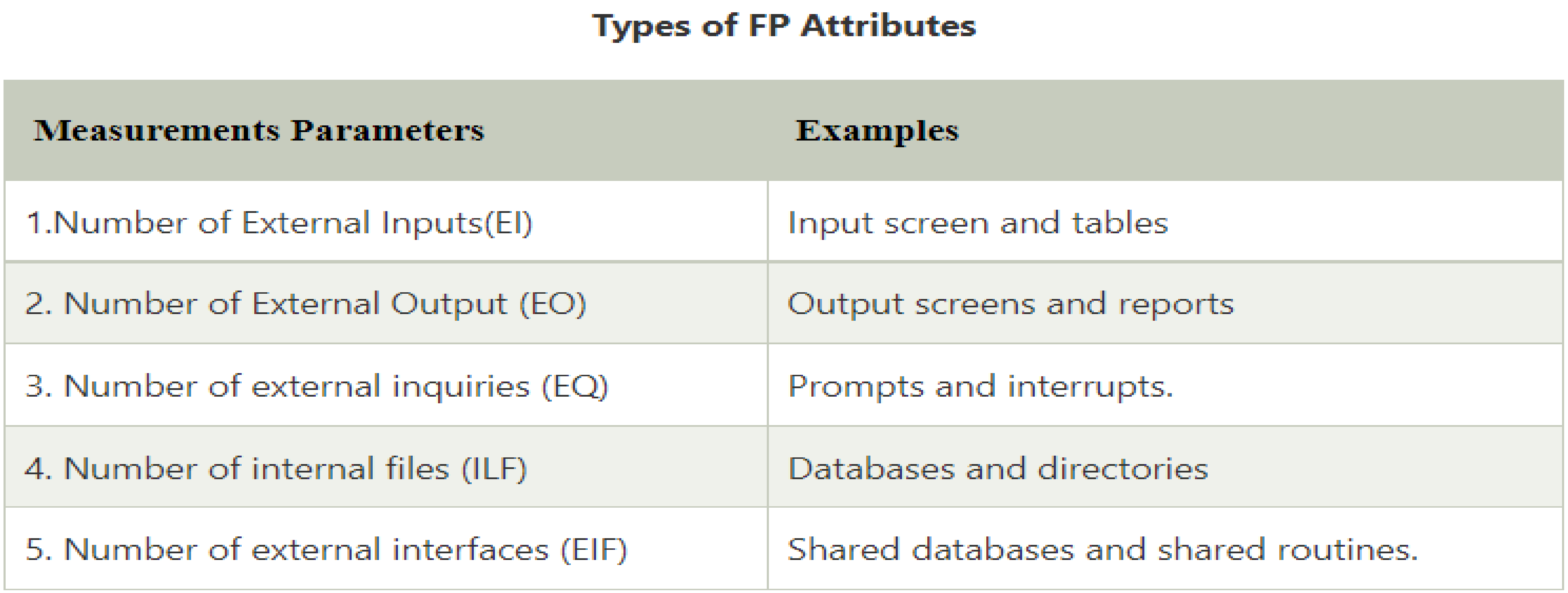
- Allan J. Albrecht initially developed function Point Analysis in 1979 at IBM and it has been further modified by the International Function Point Users Group (IFPUG).
- FPA is used to make estimate of the software project, including its testing in terms of functionality or function size of the software product.
- 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.
- It is used to measure the software project development along with its maintenance, consistently throughout the project irrespective of the tools and the technologies.

Following are the points regarding FPs

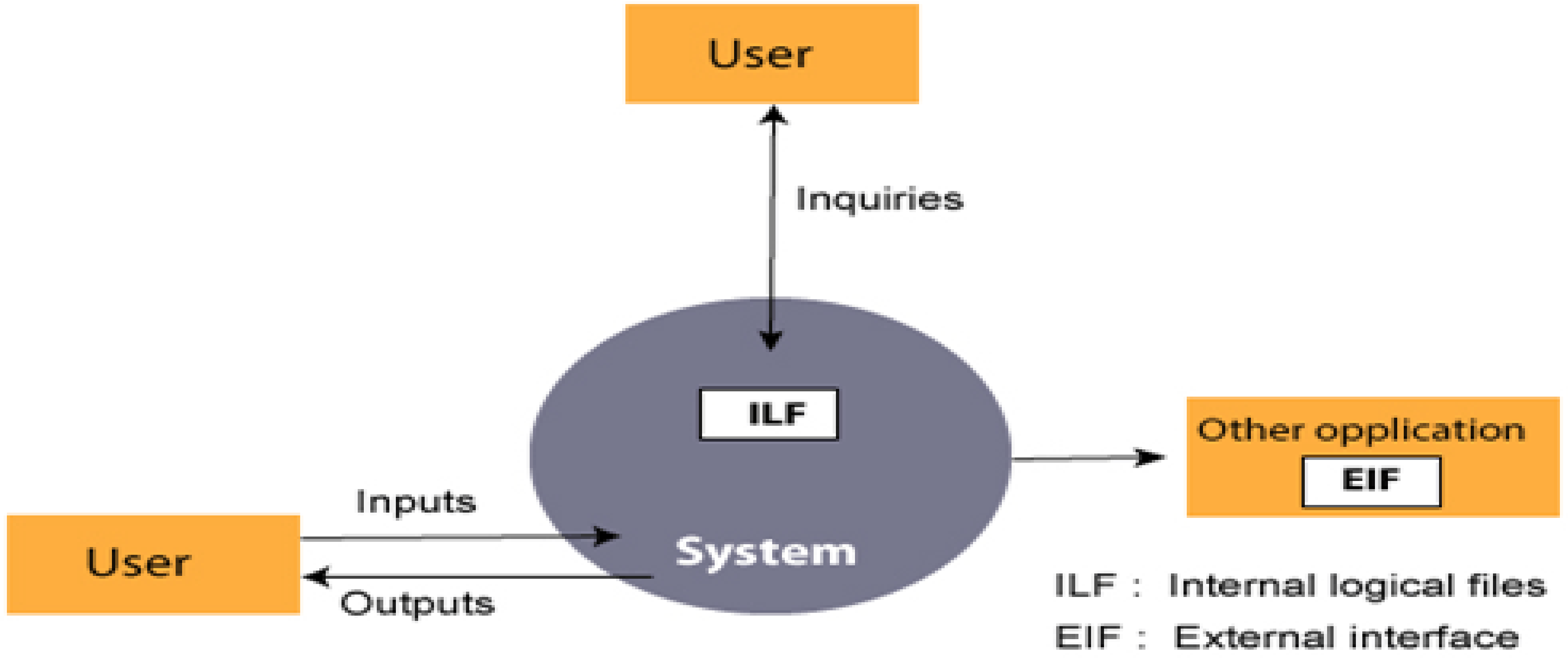
Various functions used in an application can be put under five types, as shown in Table:



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.

The FPA functional units are shown in Fig:

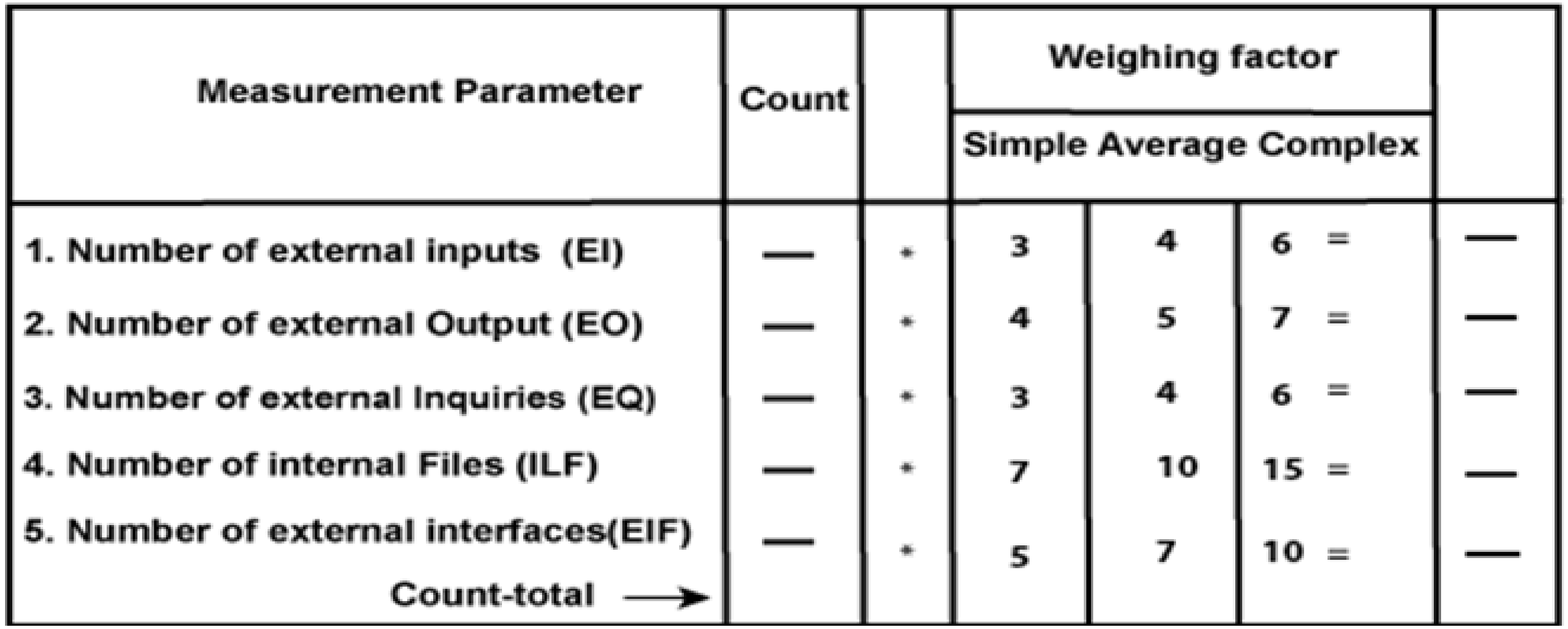


FPAs Functional Units System

- The effort required to develop the project depends on what the software does.
- FP is programming language independent.
- FP method is used for data processing systems, business systems like information systems.
- The five parameters mentioned above are also known as information domain characteristics.

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 →						

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

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

$$= \text{Count-total} * CAF$$

where Count-total is obtained from the previous Table.

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

$\sum(f_i)$ 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

1. When **$\sum(f_i)$** = 0 then CAF = 0.65
2. 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
- Defects/FP
- Pages of documentation/FP
- Errors/PM.
- Productivity = FP/PM (effort is measured in person-months).

Compute the function point, productivity, documentation, cost per function for the following data:

Number of user inputs = 24

Number of user outputs = 46

Number of inquiries = 8

Number of files = 4

Number of external interfaces = 2

Effort = 36.9 p-m

Technical documents = 265 pages

User documents = 122 pages

Cost = $7744/ month

Various processing complexity factors are:

4, 1, 0, 3, 3, 5, 4, 4, 3, 3, 2, 2, 4, 5.