Course Name: Software Engineering

Course Code: KCS601

UNIT 3: SOFTWARE DESIGN

Faculty Name: Mr. Ashish Jain

Faculty Email: ashish.jain@glbitm.ac.in

Vision and Mission

Vision

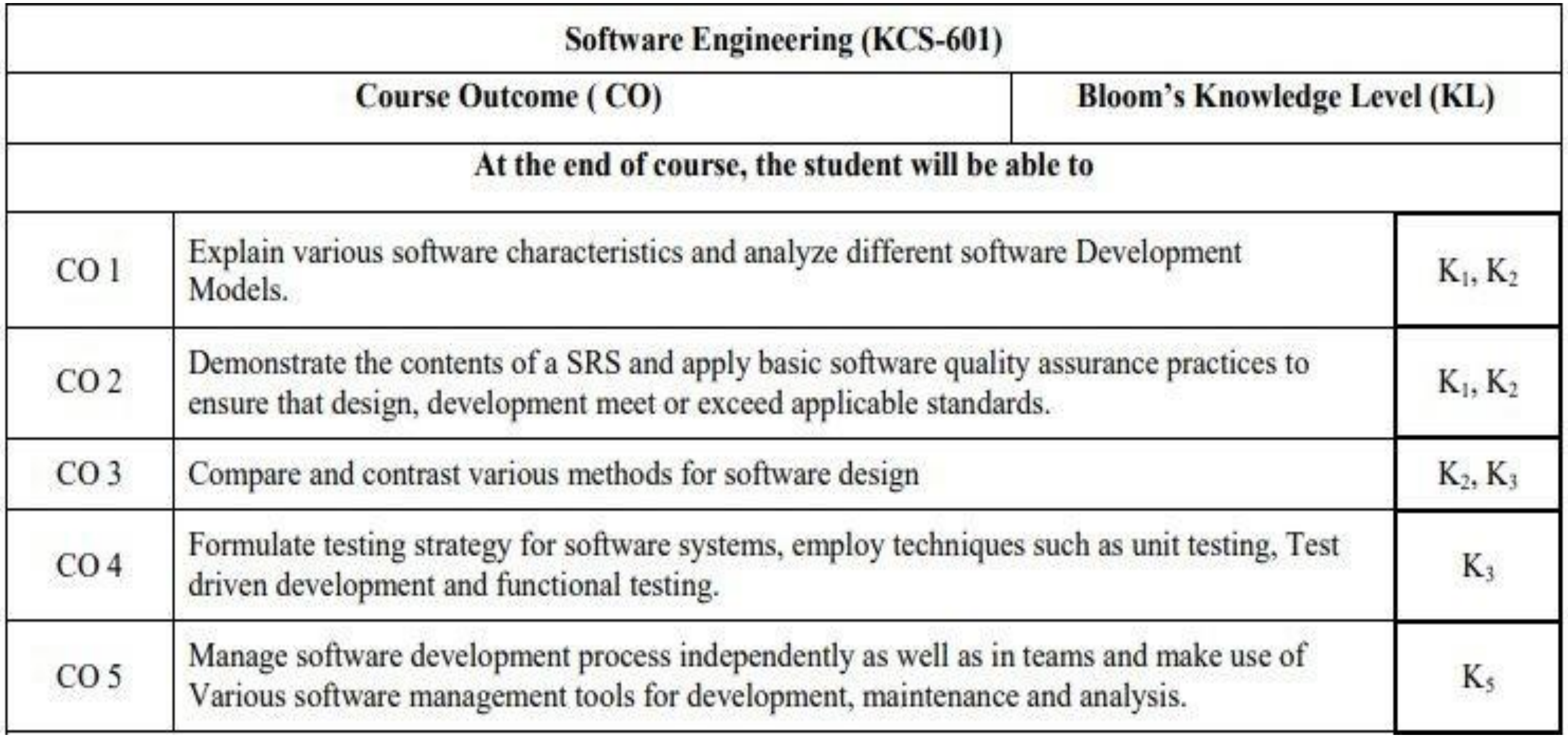
To build strong teaching environment that responds to the needs of industry and challenges of the society

Mission

- **M1 :** Developing strong mathematical & computing skill set among the students.
- **M2 :** Extending the role of computer science and engineering in diverse areas like Internet of Things (IoT), Artificial Intelligence & Machine Learning and Data Analytics.
- **M3 :** Imbibing the students with a deep understanding of professional ethics and high integrity to serve the Nation.
- **M4 :** Providing an environment to the students for their growth both as individuals and as globally competent Computer Science professional with encouragement for innovation & start-up culture.

Course Outcome

EDCFM



Software Engineering (KCS-601)		
Course Outcome (CO)		Bloom's Knowledge Level (KL)
At the end of course, the student will be able to		
CO 1	Explain various software characteristics and analyze different software Development Models.	K_1, K_2
CO 2	Demonstrate the contents of a SRS and apply basic software quality assurance practices to ensure that design, development meet or exceed applicable standards.	K_1, K_2
CO 3	Compare and contrast various methods for software design	K_2, K_3
CO 4	Formulate testing strategy for software systems, employ techniques such as unit testing, Test driven development and functional testing.	K_3
CO 5	Manage software development process independently as well as in teams and make use of Various software management tools for development, maintenance and analysis.	K_5

Unit-3

Software Design

1. Software Design
2. Software Design principles
3. Architectural design
4. Coupling and Cohesion Measures
5. Function Oriented Design
6. Object Oriented Design, Top Down and Bottom-Up Design
7. Software Measurement and Metrics
    1. Halestead's Software Science
    2. Function Point (FP)

Topic: Introduction to software Engineering

LECTURE -1

Topic: Introduction to software Design

DATE :

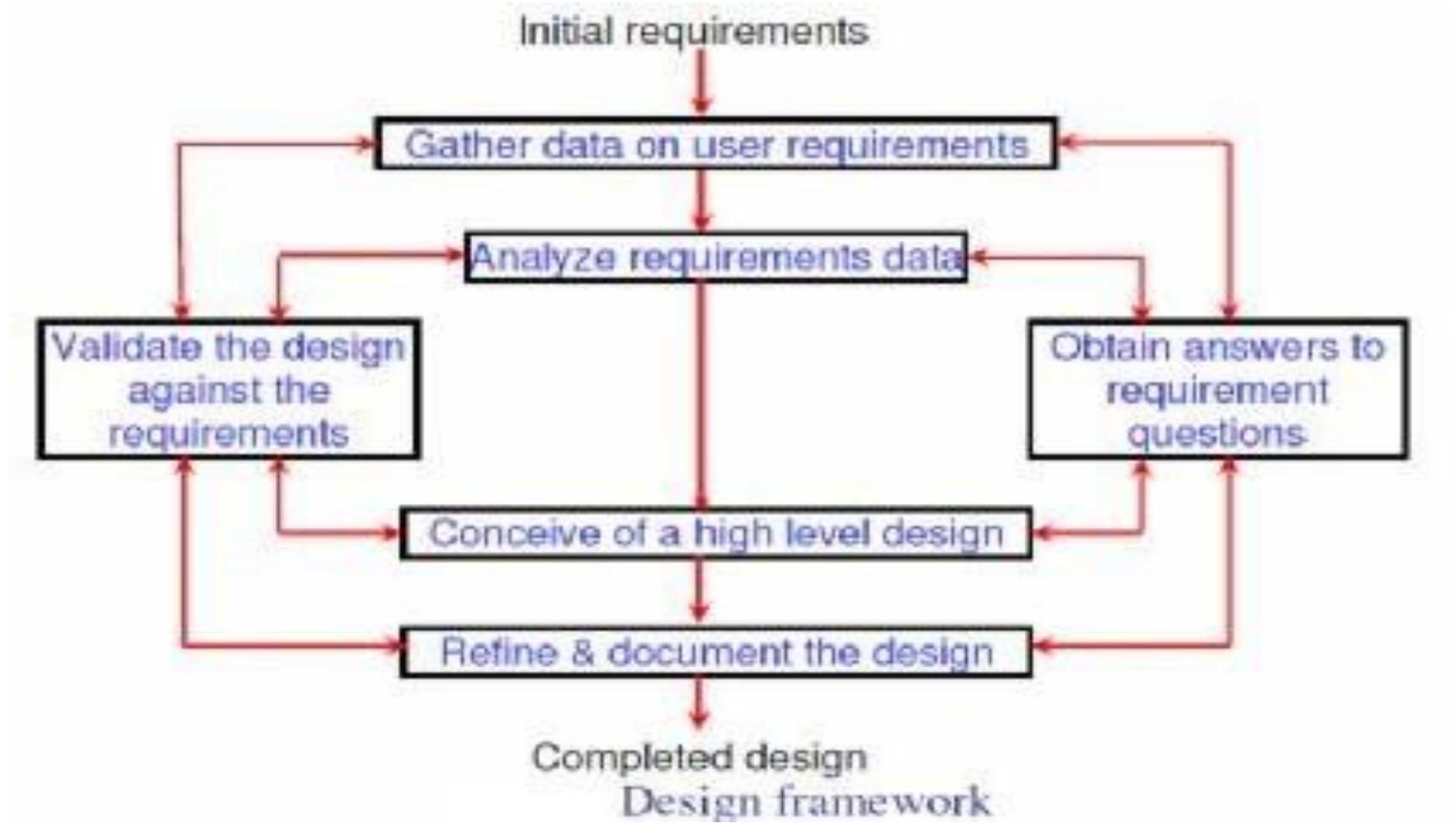
Presented By: Mr. Ashish Jain

Process and Project

Software design

- SRS tell us **what a system does** and becomes input to design process, which tells us “**how” a software system works.**
- During Design Phase designer plans **<u>“how” a software system should be produced in order to make it functional, reliable and reasonably easy to understand, modify and maintain.</u>**
- Software design involves identifying the component of software design, their inner workings, and their interface from the SRS.
- Software design deals with transforming the customer requirements, as described in the SRS document, into a form (a set of documents) called **software design document(SDD) that is suitable for implementation in a programming language.**

Program and Software

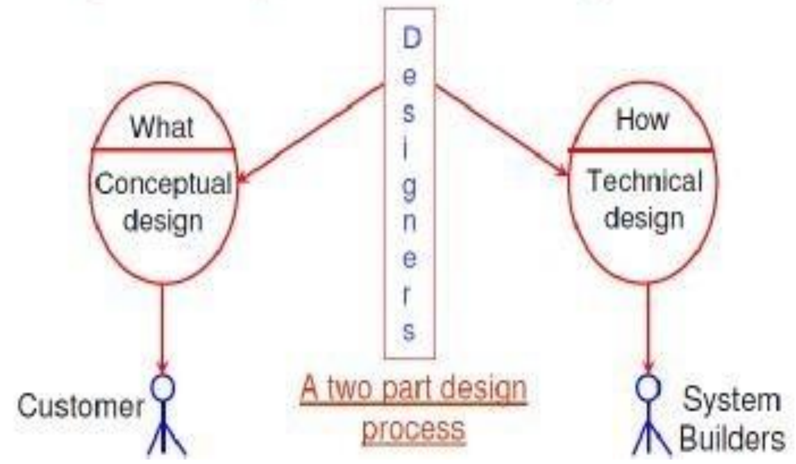

Initial requirements
Gather data on user requirements
Analyze requirements data
Validate the design against the requirements
Obtain answers to requirement questions
Conceive of a high level design
Refine & document the design
Completed design
Design framework

Program and Software

Difference between Conceptual and Technical design

- Conceptual design describe the system in language understandable to the customer. it does not contain any technical terms and is independent of implementation
- By contrast the technical design describe the hardware configuration, software needs, communication interface, input and output of system, network architecture that translate the requirement in to the solution to the customer's problem

Conceptual Design and Technical Design



Program and Software

Characteristics and objectives of a good software design

Good design is the key of successful product.

- **Correctness:** A good design should correctly implement all the functionalities identified in the SRS document.
- **Understandability:** A good design is easily understandable.
- **Efficiency:** It should be efficient.
- **Maintainability:** It should be easily amenable to change.
- Low Coupling and High Cohesion

Features of a design document

- It should use **consistent and meaningful names** for various design components.
- The design should be **modular**. The term modularity means that it should use a cleanly decomposed set of modules.
- It should neatly arrange the **modules in a hierarchy**, e.g. in a tree-like diagram.

Program and Software

Software Design principle

- **Abstraction**

It is a tool that permits a designer to consider a component at abstract level; without worrying about the detail of the implementation of the component.

- **Encapsulation/Information hiding**

the concept of information hiding is to hide the implementation details of shared information and processing items by specifying modules called hiding information modules. Design decisions that are likely to change in the future should be identified and modules should be designed in such a way that those design decisions are hidden from other modules

- **Coupling and cohesion**

Cohesion

It is a measure of the degree to which the elements of a module are functionally related. Cohesion is weak if elements are bundled simply because they perform similar

Program and Software

or related functions . Cohesion is strong if all parts are needed for the functioning of other parts (e..Important design objective is to maximize module cohesion and minimize module coupling.

Coupling

It is the measure of the degree of interdependence between modules. Coupling is highly between components if they depend heavily on one another, (e.g., there is a lot of communication between them).

- **Decomposition and modularization**

Decomposition and modularization large software in to small independent once, usually with the goal of placing different **functionality or responsibility** in different component

GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Program and Software

Modularization

Modularization is a technique to divide a software system into multiple discrete and independent modules, which are expected to be capable of carrying out task(s) independently. These modules may work as basic constructs for the entire software. Designers tend to design modules such that they can be executed and/or compiled separately and independently.

Modular design unintentionally **follows the rules of 'divide and conquer'** problem-solving strategy this is because there are many other benefits attached with the modular design of a software.

Advantage of modularization:

- Smaller components are easier to maintain
- Program can be divided based on functional aspects
- Desired level of abstraction can be brought in the program
- Components with high cohesion can be re-used again
- Concurrent execution can be made possible
- Desired from security aspect

Program and Software

Software Design Approaches

1. **Top-down** approach (is also known as **step-wise design**) is essentially the breaking down of a system to gain insight into its compositional sub-systems. In a top-down approach an **overview of the system is formulated, specifying but not detailing any first-level subsystems**. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. **A top-down model is often specified with the assistance of "black boxes",** these make it easier to manipulate. However, black boxes may fail to elucidate elementary mechanisms or be detailed enough to realistically validate the model.

2. **Bottom-up** approach is the piecing together of systems to give rise to grander systems, thus making the original systems sub-systems of the emergent system. In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed. This strategy often resembles a "seed" model, whereby the beginnings are small but eventually grow in complexity and completeness. However, "organic strategies" may result in a tangle of elements and subsystems, developed in isolation and subject to local optimization as opposed to meeting a global purpose. Mechanisms or be detailed enough to realistically validate the model.

Program and Software

Software Design process/Levels

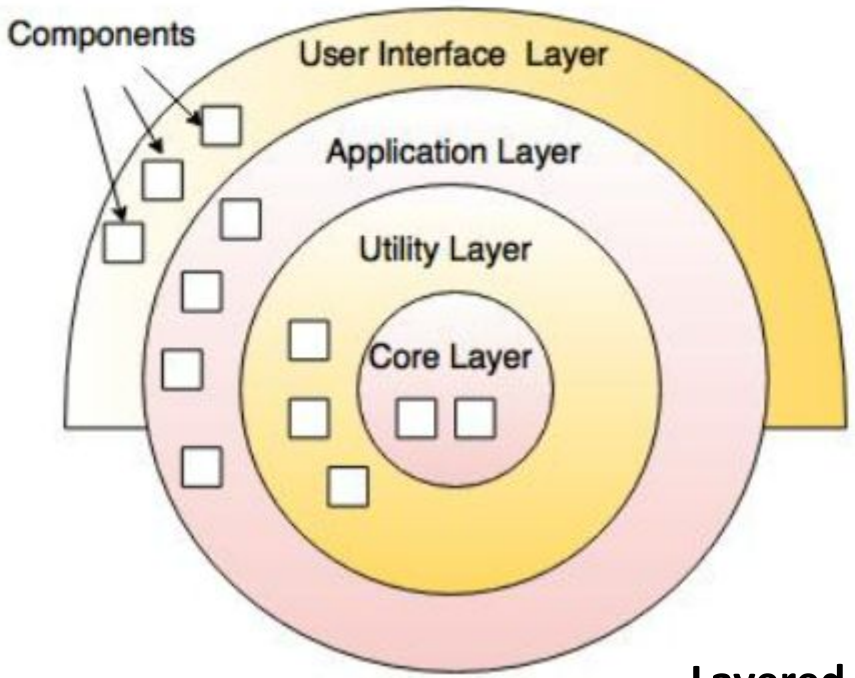
1. **Architectural Design (Top Level design):-**

- Describe how software is decomposed and organized into components
- Architectural design represents the structure of data and program components that are required to build a computer-based system.
- It considers the architectural style that the system will take, the structure and properties of the components that constitute the system, and the interrelationships that occur among all architectural components of a system.
- Architectural design begins with data design and then proceeds to the derivation of one or more representations of the architectural structure of the system.
- Alternative architectural styles or patterns are analyzed to derive the structure that is best suited to customer requirements and quality attributes.
- Once an alternative has been selected, the architecture is elaborated using an architectural design method.

Program and Software

Types of Architectural design

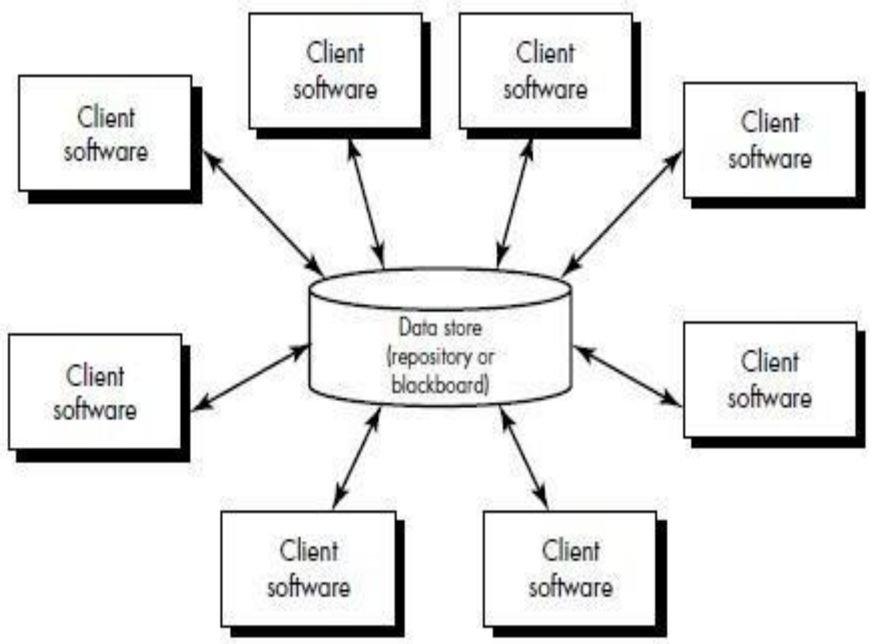
Object-oriented architectures: - The components of a system encapsulate data and the operations that must be applied to manipulate the data. Communication and coordination between components is accomplished via message passing.



Layered Architecture

Program and Software

Data-centered architectures: - A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Client software accesses a central repository. In some cases the data repository is *passive*. That is, client software accesses the data independent of any changes to the data or the actions of other client software. A variation on this approach transforms the repository into a "blackboard" that sends notifications to client software when data of interest to the client change.

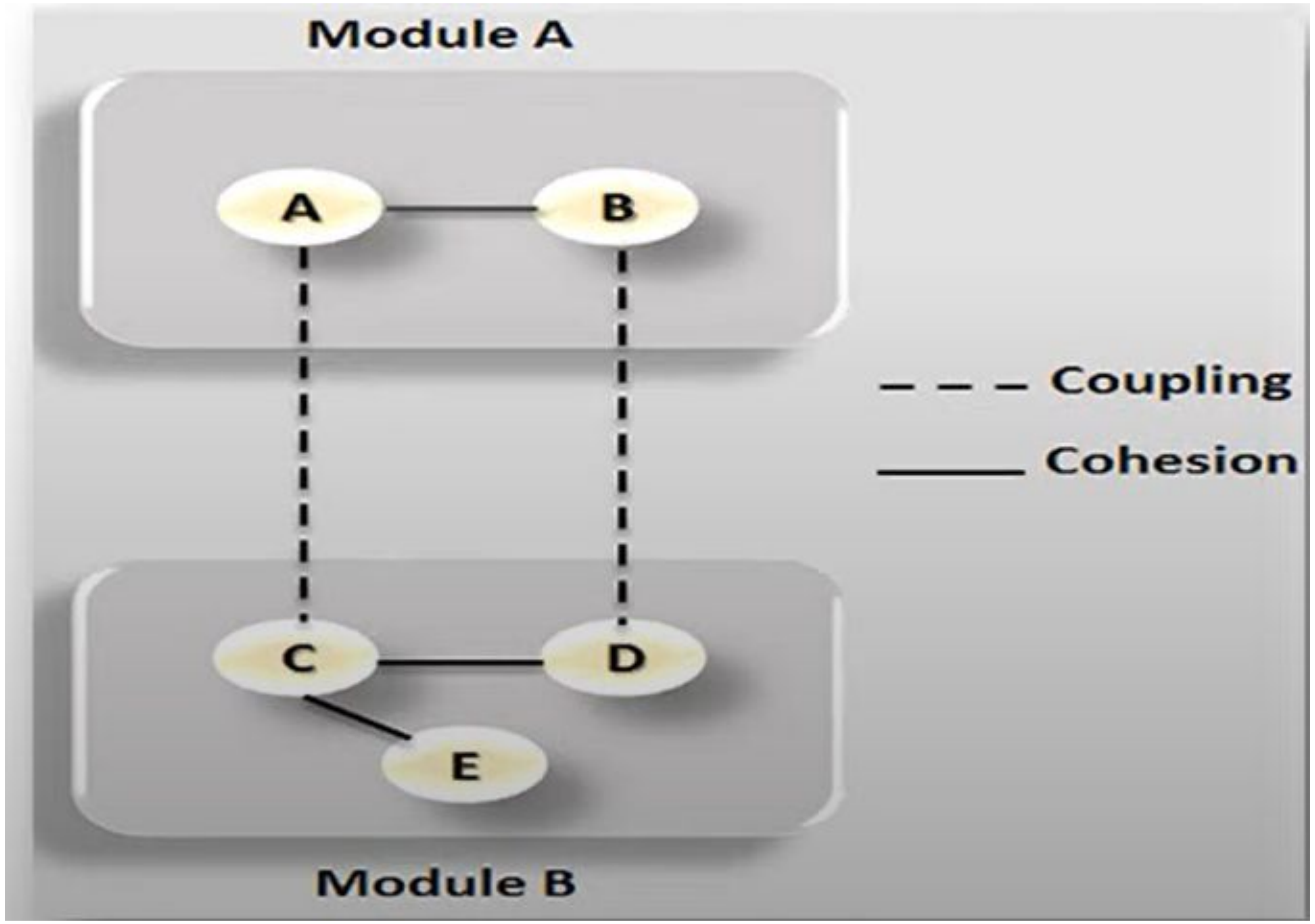


Program and Software

2. Detailed Design (Low level design):-describe the specific behavior of these components. The output of this process is a set of models that records the major decision that has been taken.

Program and Software

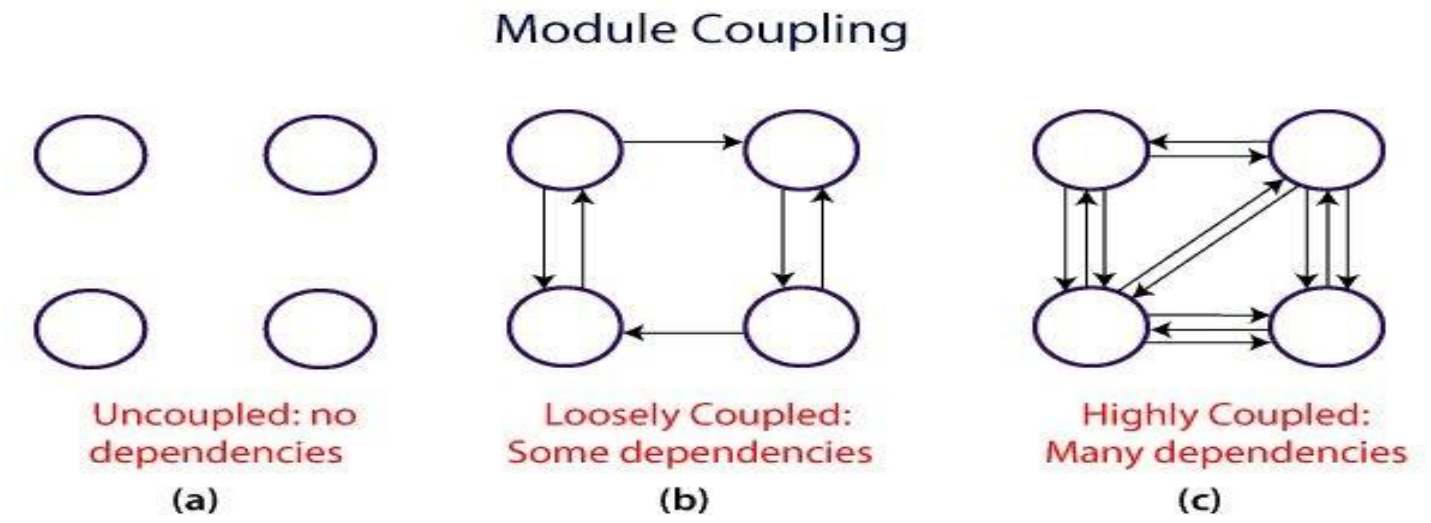
COUPLING AND COHESION



Program and Software

Module Coupling

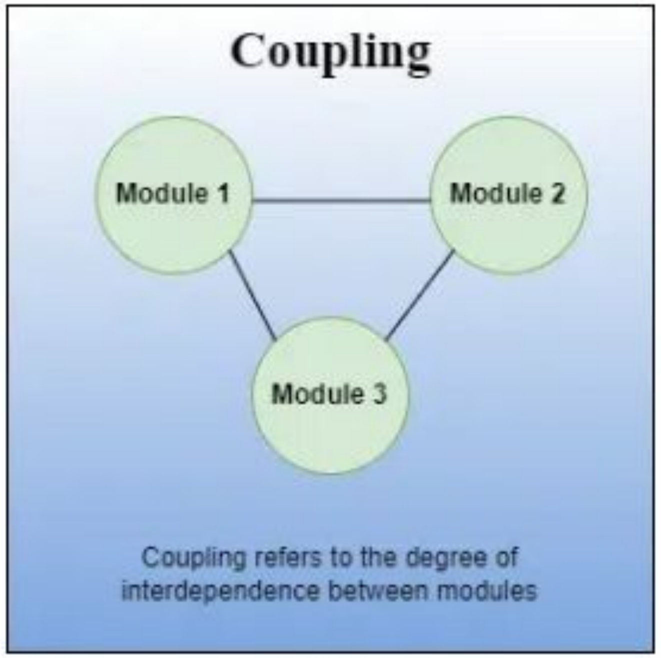
- The coupling is the **degree of interdependence between software modules.**
- **The various types of coupling techniques are shown in fig:**


Module Coupling
Uncoupled: no dependencies
(a)
Loosely Coupled: Some dependencies
(b)
Highly Coupled: Many dependencies
(c)

Program and Software

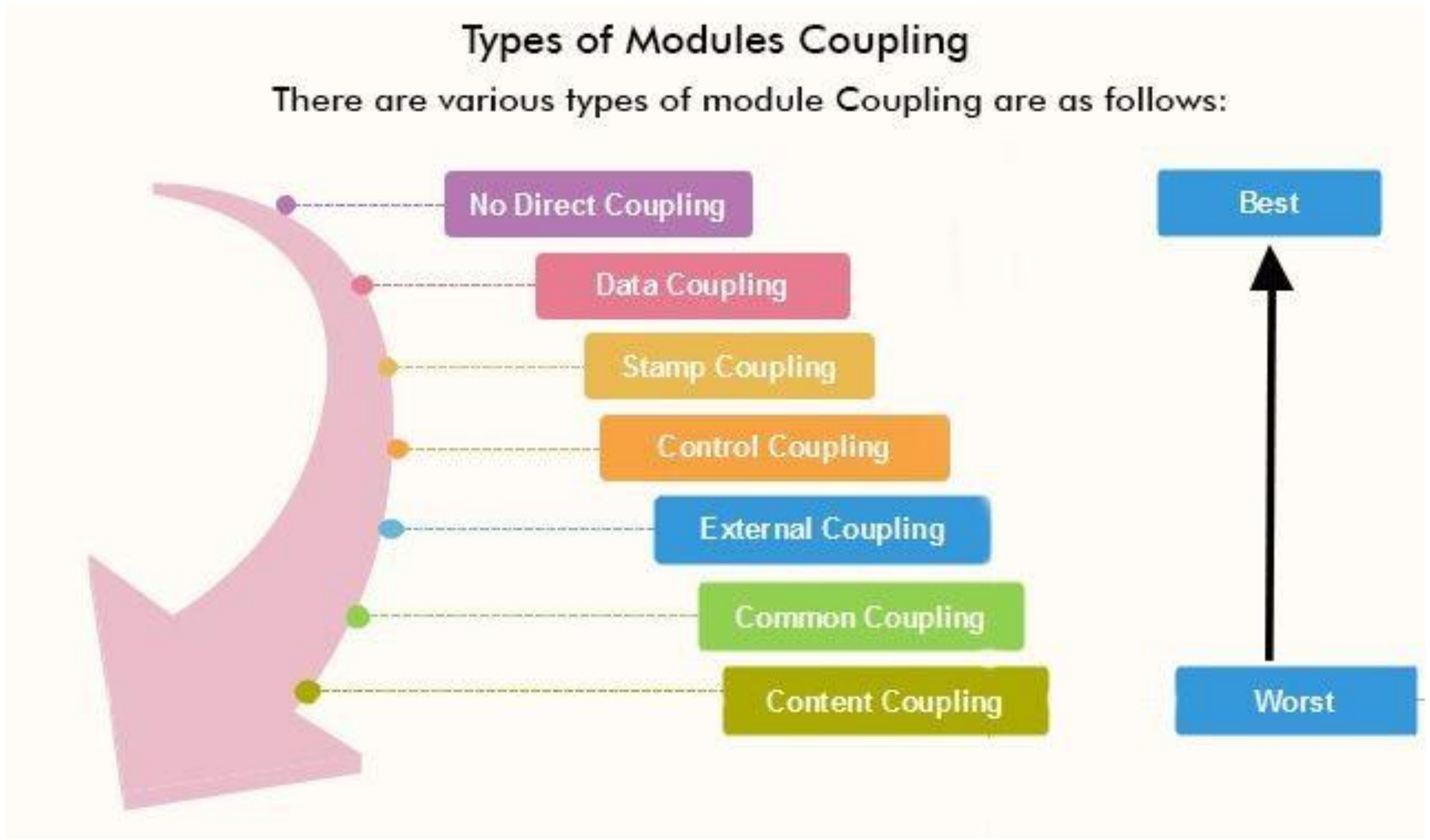
A good design is the one that has **low coupling**.

Coupling is measured by the **number of relations between the modules**. That is, the coupling increases as the number of calls between modules increase or the amount of shared data is large. Thus, it can be said that a design with high coupling will have more errors.


Coupling
Module 1
Module 2
Module 3
Coupling refers to the degree of interdependence between modules

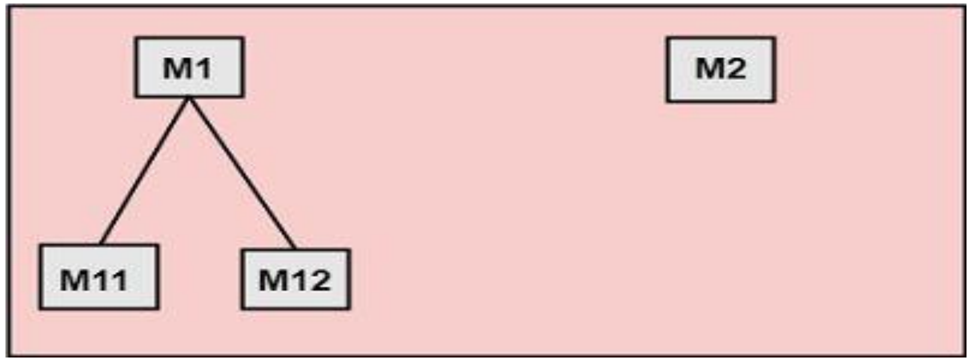
Coupling

Types of Module Coupling



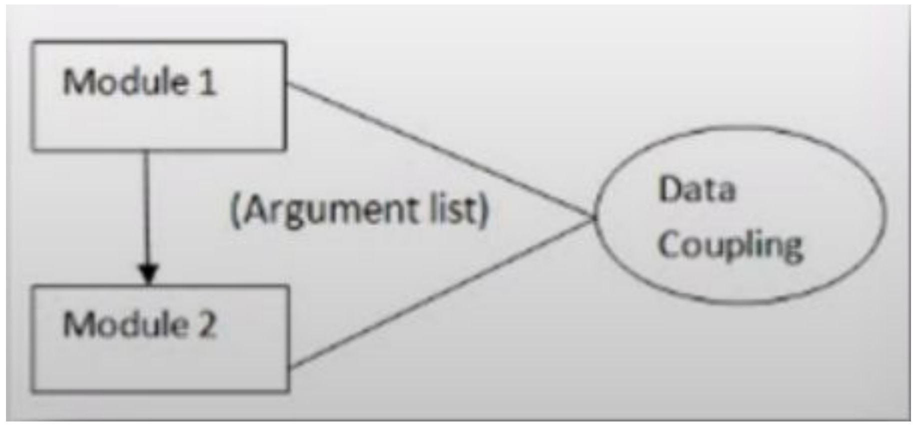
Program and Software

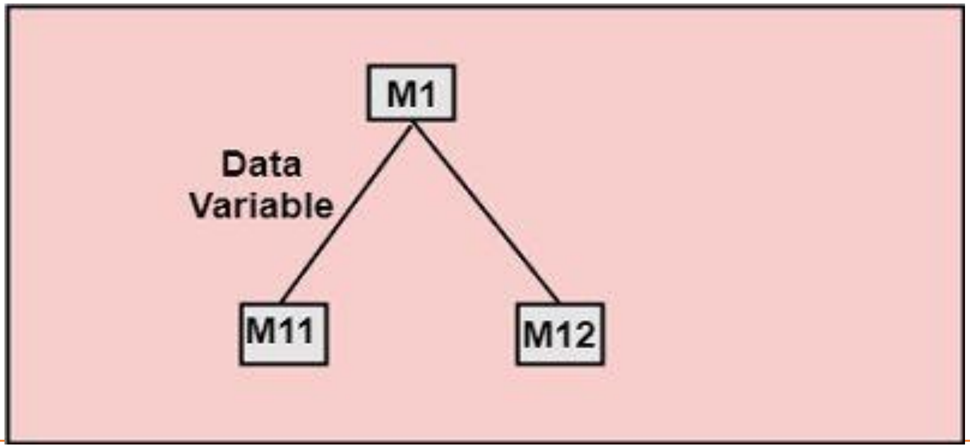
1. No Direct Coupling: There is no direct coupling between M1 and M2.



In this case, modules are subordinates to different modules. Therefore, no direct coupling.

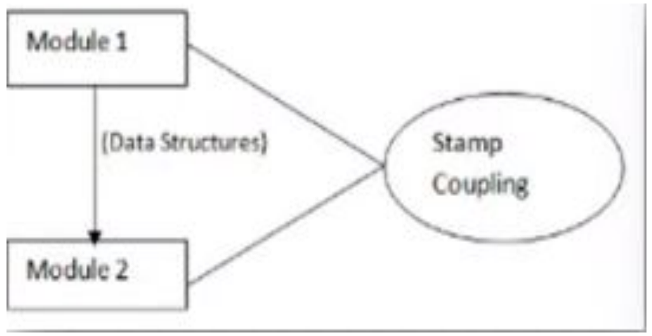
2. Data Coupling: When data of one module is passed to another module, this is called data coupling.





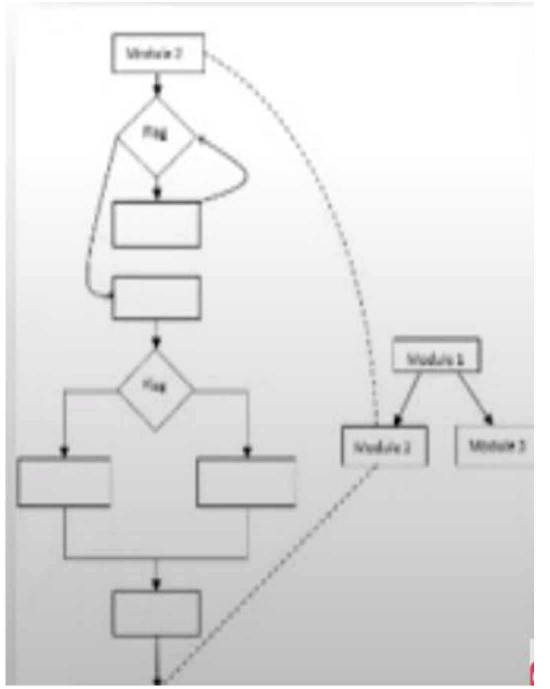
Program and Software

3.Stamp Coupling: Two modules are stamp coupled if they communicate using composite data items such as **structure, objects,** etc. When the module passes non-global data structure or entire structure to another module, they are said to be stamp coupled. For example, passing structure variable in C or object in C++ language to a module.



4.Control Coupling: Control Coupling exists among two modules if data from one module is used to direct the structure of instruction execution in another.

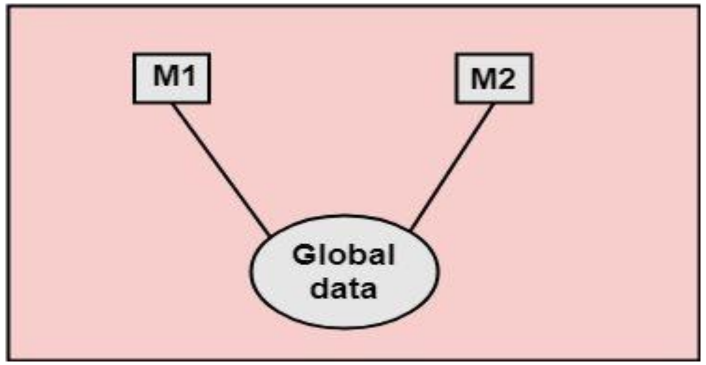
Example: Module 1- Set Flag = 1 then only Module 2 perform action.



Program and Software

5.External Coupling: External Coupling arises when two modules share an externally imposed data format, communication protocols, or device interface. This is related to communication to external tools and devices. **Ex. Libraries, path of resource**

6. Common Coupling: Two modules are common coupled if they share information through some global data items.

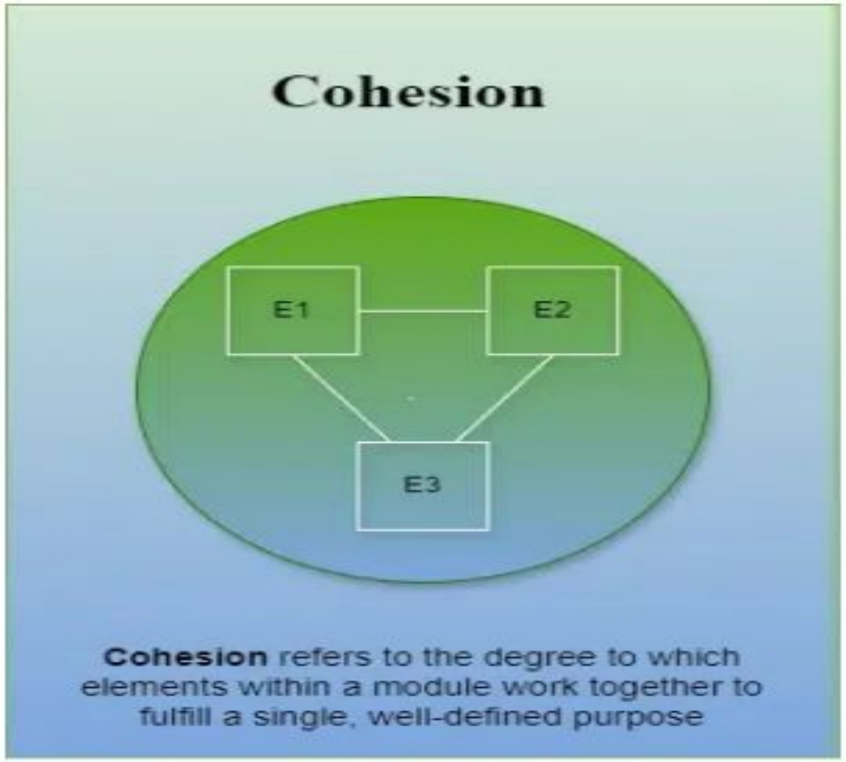


7. Content Coupling: Content Coupling exists among two modules if **<u>they share the same content e.g., functions, methods from one module into another module.</u>**

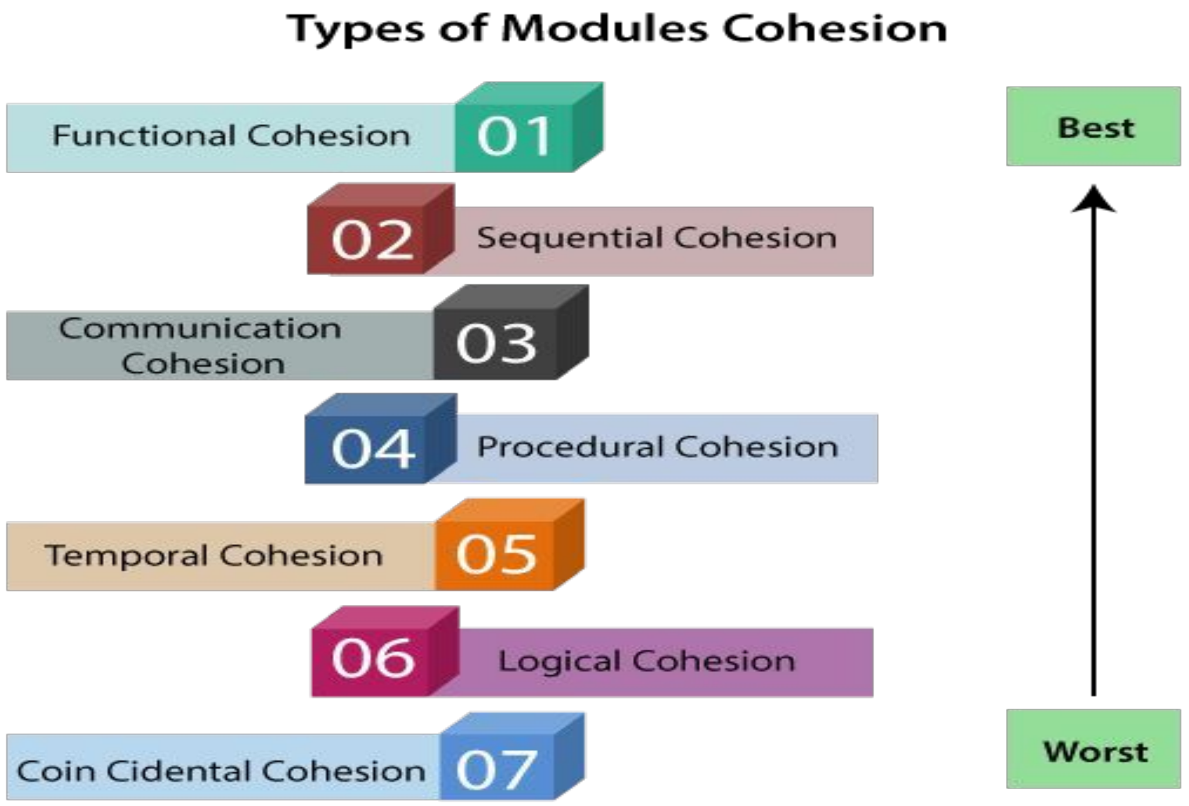
Program and Software

Module Cohesion

- Cohesion defines the degree to which the elements of a module belong together.
- cohesion measures the strength of relationships between pieces of functionality within a given module.
- A good design have high cohesion.

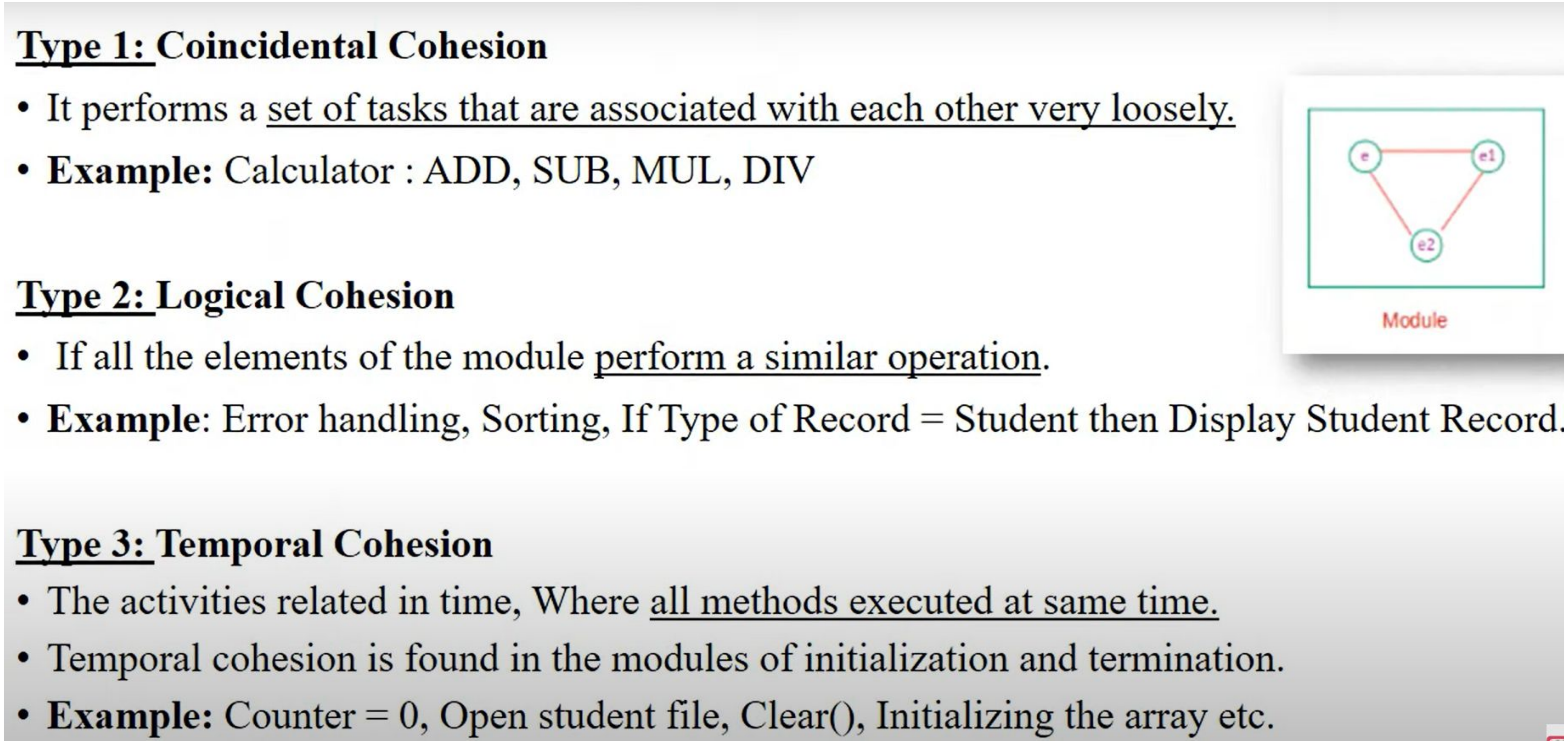

Cohesion
E1
E2
E3
Cohesion refers to the degree to which elements within a module work together to fulfill a single, well-defined purpose

Cohesion



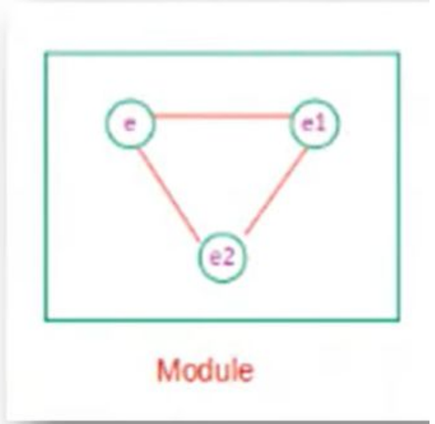
GL BAJAJ Institute of Technology & Management FIND YOUR SPARK

Program and Software



Type 1: Coincidental Cohesion

- It performs a set of tasks that are associated with each other very loosely.
- **Example:** Calculator : ADD, SUB, MUL, DIV



Type 2: Logical Cohesion

- If all the elements of the module perform a similar operation.
- **Example**: Error handling, Sorting, If Type of Record = Student then Display Student Record.

Type 3: Temporal Cohesion

- The activities related in time, Where all methods executed at same time.
- Temporal cohesion is found in the modules of initialization and termination.
- **Example:** Counter = 0, Open student file, Clear(), Initializing the array etc.

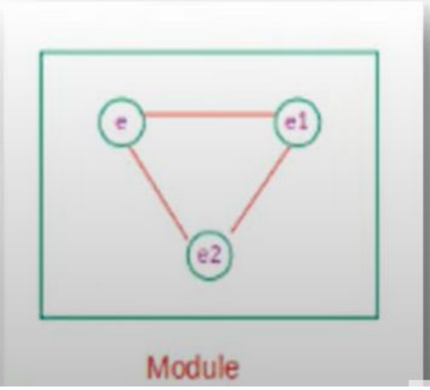
Program and Software

Type 4: Procedural Cohesion

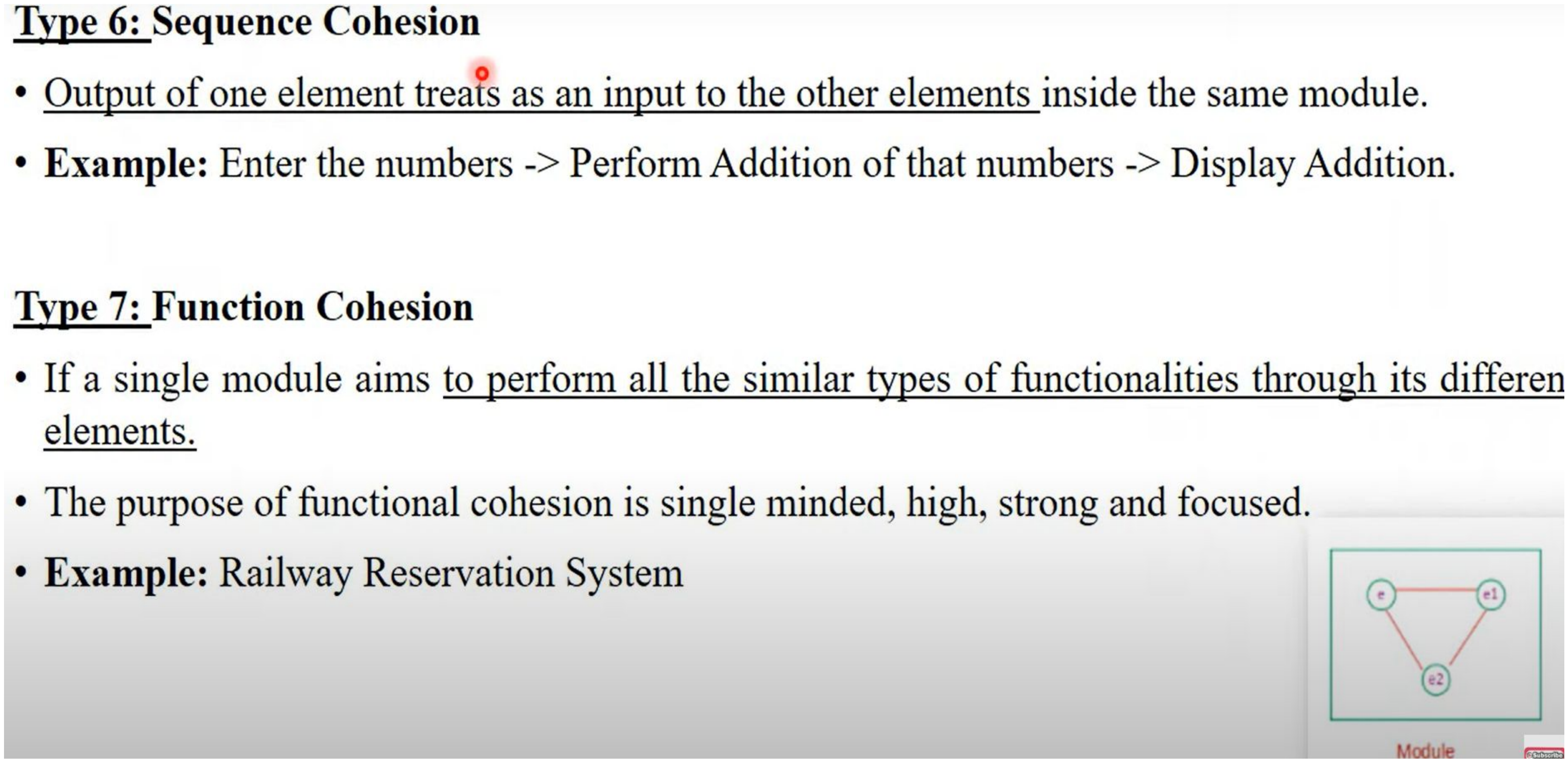
- All parts of a procedure execute in particular sequence of steps for achieving goal.
- **Example:** Calling one function to another function, Loop statements, Reading record etc.

Type 5: Communicational Cohesion

- If all the elements of a module are working on the same input & output data and are accessing that data through the same data structures.
- **Example:** Update record in the database and send it to the printer.



Program and Software



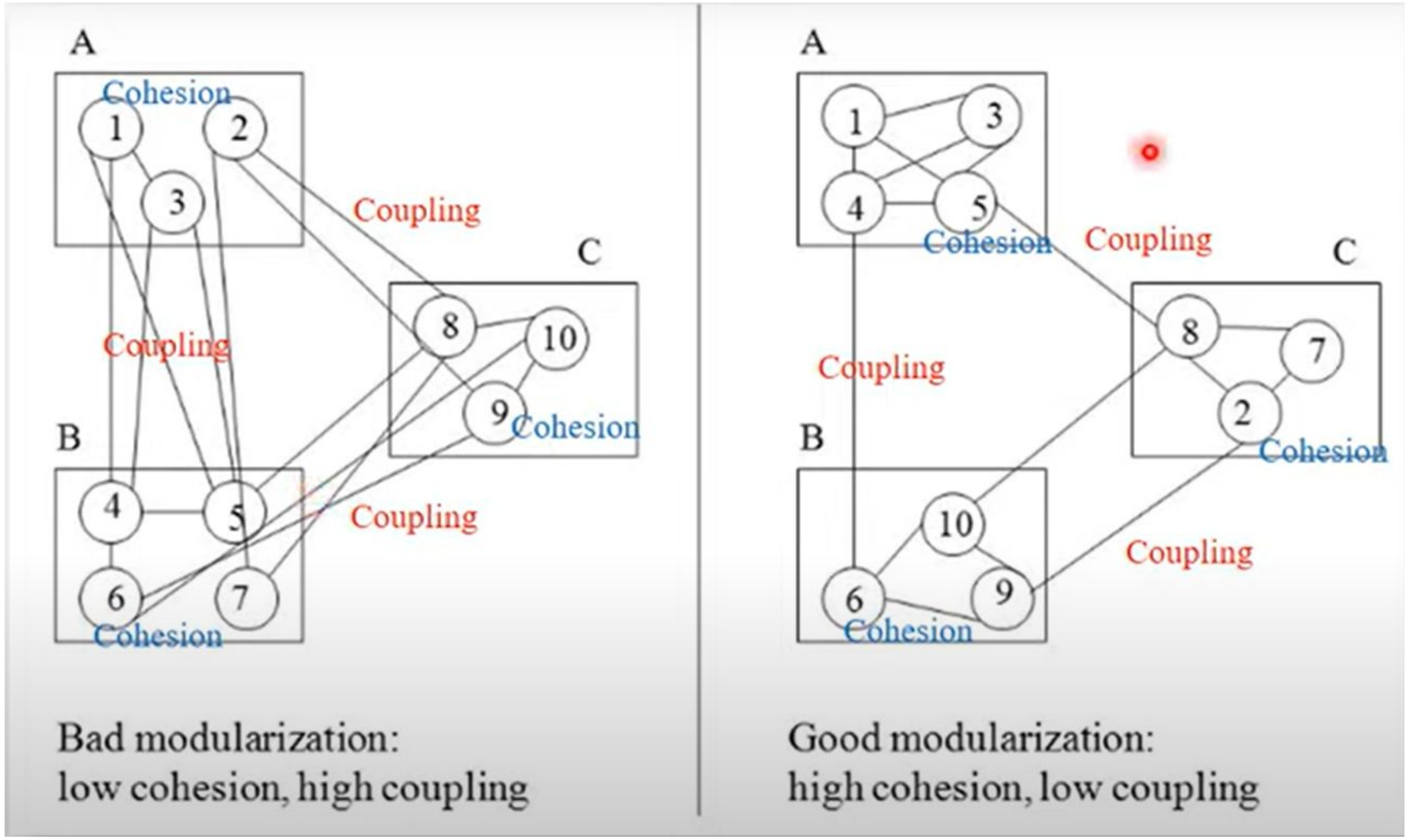
Type 6: Sequence Cohesion

- Output of one element treats as an input to the other elements inside the same module.
- **Example:** Enter the numbers -> Perform Addition of that numbers -> Display Addition.

Type 7: Function Cohesion

- If a single module aims to perform all the similar types of functionalities through its differen elements.
- The purpose of functional cohesion is single minded, high, strong and focused.
- **Example:** Railway Reservation System

Good & Bad Software Design



Program and Software

1. **Functional Cohesion:** Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.

2. **Sequential Cohesion:** A module is said to possess sequential cohesion if the element of a module form the components of the sequence, where the output from one component of the sequence is input to the next.

3. **Communicational Cohesion:** A module is said to have communicational cohesion, if all tasks of the module refer to or update the same data structure, e.g., **the set of functions defined on an array or a stack.**

4. **Procedural Cohesion:** A module is said to be procedural cohesion if the set of purpose of the module are all parts of a procedure in which particular sequence of steps has to be carried out for achieving a goal, e.g., the algorithm for decoding a message.

Program and Software

1. **Functional Cohesion:** Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.

2. **Sequential Cohesion:** A module is said to possess sequential cohesion if the element of a module form the components of the sequence, where the output from one component of the sequence is input to the next.

3. **Communicational Cohesion:** A module is said to have communicational cohesion, if all tasks of the module refer to or update the same data structure, e.g., **the set of functions defined on an array or a stack.**

4. **Procedural Cohesion:** A module is said to be procedural cohesion if the set of purpose of the module are all parts of a procedure in which particular sequence of steps has to be carried out for achieving a goal, e.g., the algorithm for decoding a message.

Program and Software

5.Temporal Cohesion: When a module includes functions that are associated by the fact that all the methods must be executed in the same time, the module is said to exhibit temporal cohesion.

6.Logical Cohesion: A module is said to be logically cohesive if all the elements of the module perform a similar operation. For example Error handling, data input and data output, etc.

7.Coincidental Cohesion: A module is said to have coincidental cohesion if it performs a set of tasks that are associated with each other very loosely, if at all

GL BAJAJ Institute of Technology & Management FIND YOUR SPARK

Program and Software

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 thing. modules.	In cohesion, the module focuses on a single

Program and Software

Software Design Strategies

Software design is a process to conceptualize the software requirements into software implementation. Software design takes the user requirements as challenges and tries to find optimum solution. While the software is being conceptualized, a plan is chalked out to find the best possible design for implementing the intended solution.

There are multiple variants of software design.

1. Structured Design (modular design)

Structured design is a conceptualization of problem into several well-organized elements of solution. It is basically concerned with the solution design. Benefit of structured design is, it gives better understanding of how the problem is being solved. Structured design also makes it simpler for designer to concentrate on the problem more accurately.

Structured design is mostly based on ‘divide and conquer’ strategy where a problem is broken into several small problems

Program and Software

and each small problem is individually solved until the whole problem is solved.

The small pieces of problem are solved by means of solution modules. Structured design emphasis that these modules be well organized in order to achieve precise solution.

These modules are arranged in hierarchy. They communicate with each other. A good structured design always follows some rules for communication among multiple modules, namely -

Cohesion - grouping of all functionally related elements.

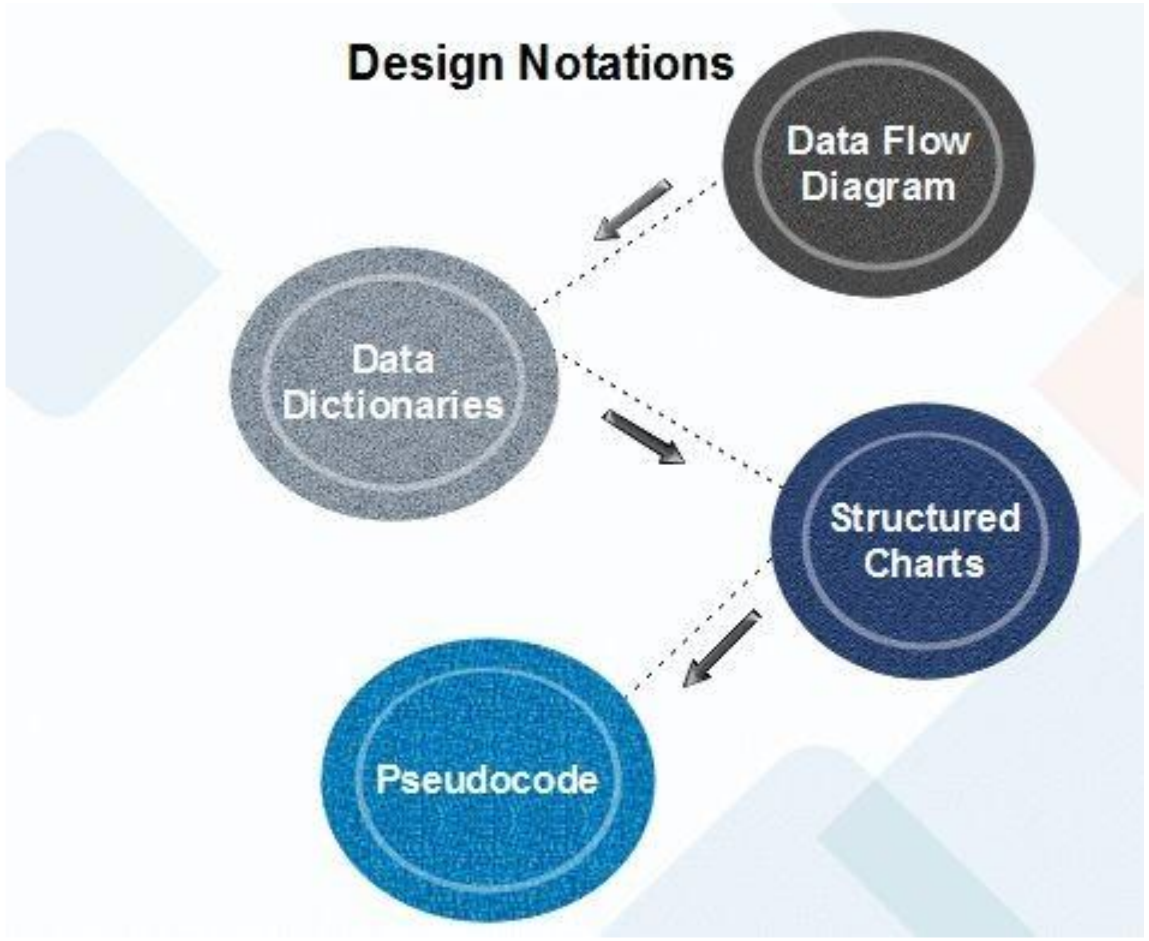
Coupling - communication between different modules.

A good structured design has high cohesion and low coupling arrangements.

Program and Software

2. Function oriented design

Function Oriented design is a method to software design where the model is decomposed into a set of interacting units or modules where each unit or module has a clearly defined function. Thus, the system is designed from a functional viewpoint.



Program and Software

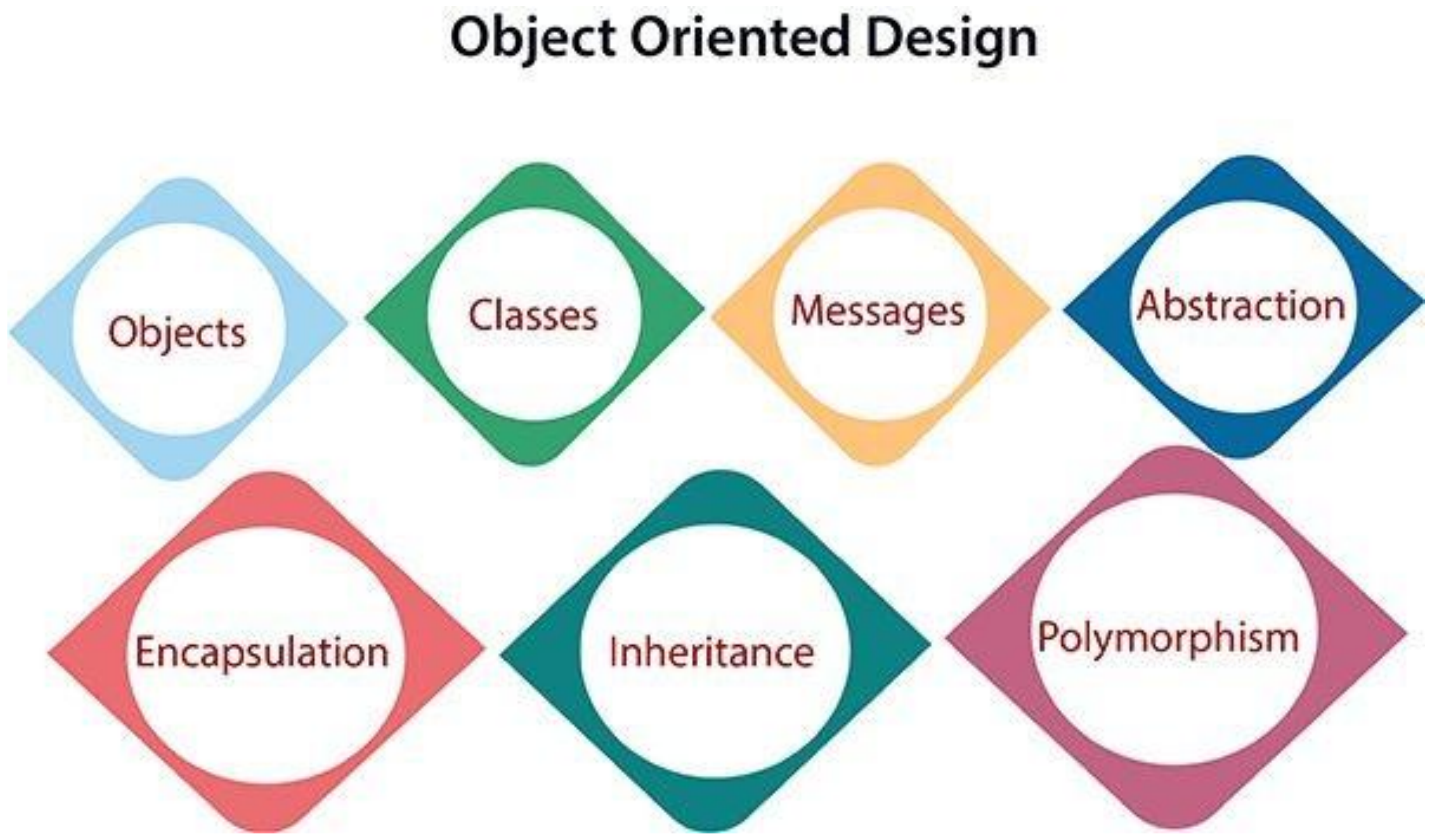
3. Object Oriented Design

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. Similar objects create a class. In other words, each object is a member of some class. Classes may inherit features from the superclass.

GL BAJAJ Institute of Technology & Management FIND YOUR SPARK

Program and Software

The different terms related to object design are:



Program and Software

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

Program and Software

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
   - maturity of the process

iii. **Project metrics:** describe the project characteristics and execution. Examples are:-
   - number of software developers
   - staffing pattern over the life cycle of the software
   - cost and schedule
   - productivity

Program and Software

Size Oriented Metrics

1. **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.

Program and Software

- 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.

While counting lines of code, simplest standard is:

- Don’t count blank lines
- Don’t count comments
- Count everything else

The size-oriented measure is not a universally accepted method.

Program and Software

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

Size = kilo LOC(KLOC)

Effort=person / month

Cost =$/ KLOC.

Documentation=Pages of documentation/KLOC.

Productivity = KLOC/PM = KLOC/effort

Quality= no of faults / KLOC

Program and Software

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.

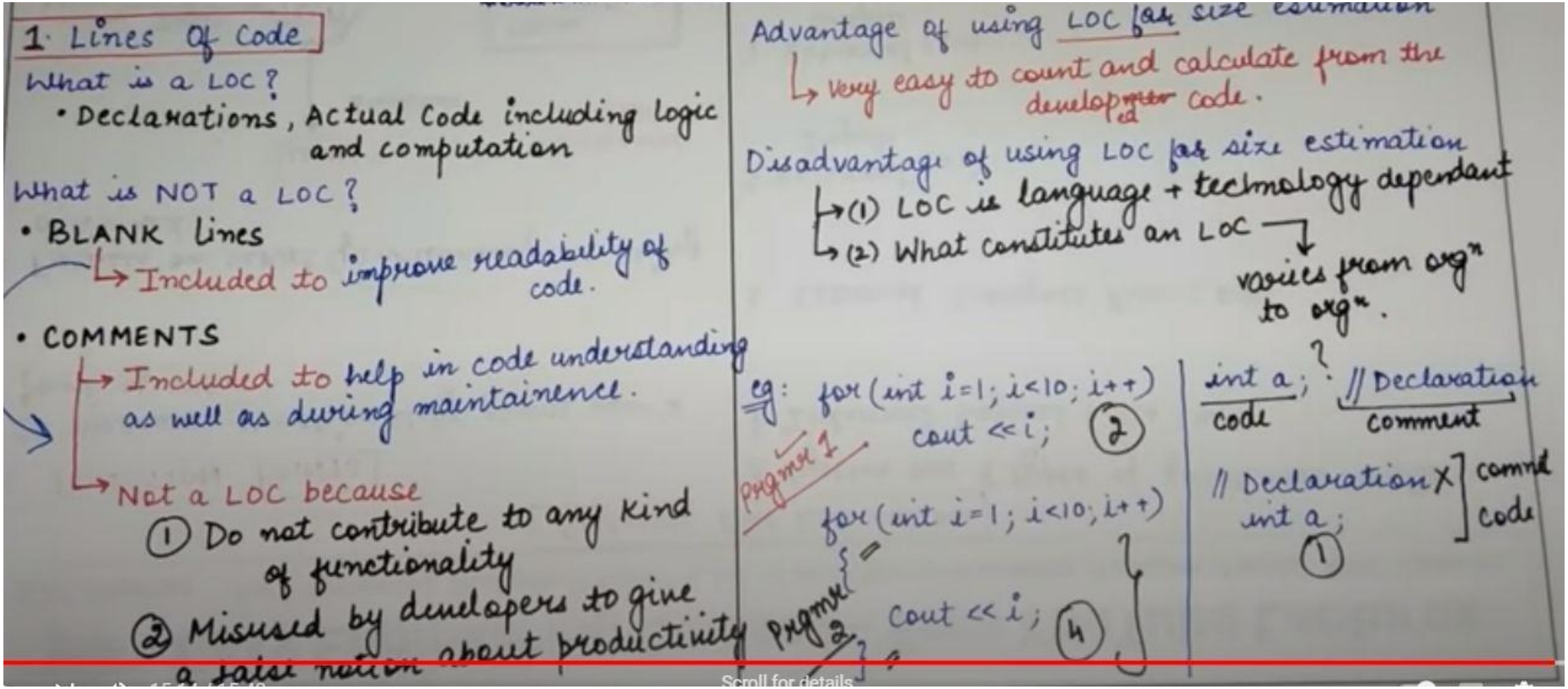
It characterizes only one specific view of size, namely length, it takes no account of functionality or complexity

Bad software design may cause an excessive line of code

It is language dependent

Users cannot easily understand it

Program and Software



1. Lines of Code

What is a LOC?

- Declarations, Actual Code including logic and computation

What is NOT a LOC?

- BLANK lines
  - ↳ Included to improve readability of code.
- COMMENTS
  - ↳ Included to help in code understanding as well as during maintainence.
  - ↳ Not a LOC because
    - ① Do not contribute to any kind of functionality
    - ② Misused by developers to give a false notion about productivity

Advantage of using LOC for size estimation

↳ very easy to count and calculate from the ~~developer~~ developed code.

Disadvantage of using LOC for size estimation

↳ (1) LOC is language + technology dependant

↳ (2) What constitutes an LOC → varies from orgⁿ to orgⁿ.

eg:

prgmr 1

```
for(int i=1; i<10; i++)
    cout << i;        ②
```

prgmr 2

```
for(int i=1; i<10; i++)
{
    cout << i;        ④
}
```

```
int a;   // Declaration
code        comment
```

```
// Declaration   comnt
int a;           code
   ①
```

Scroll for details

Program and Software

Halstead's Software Metrics

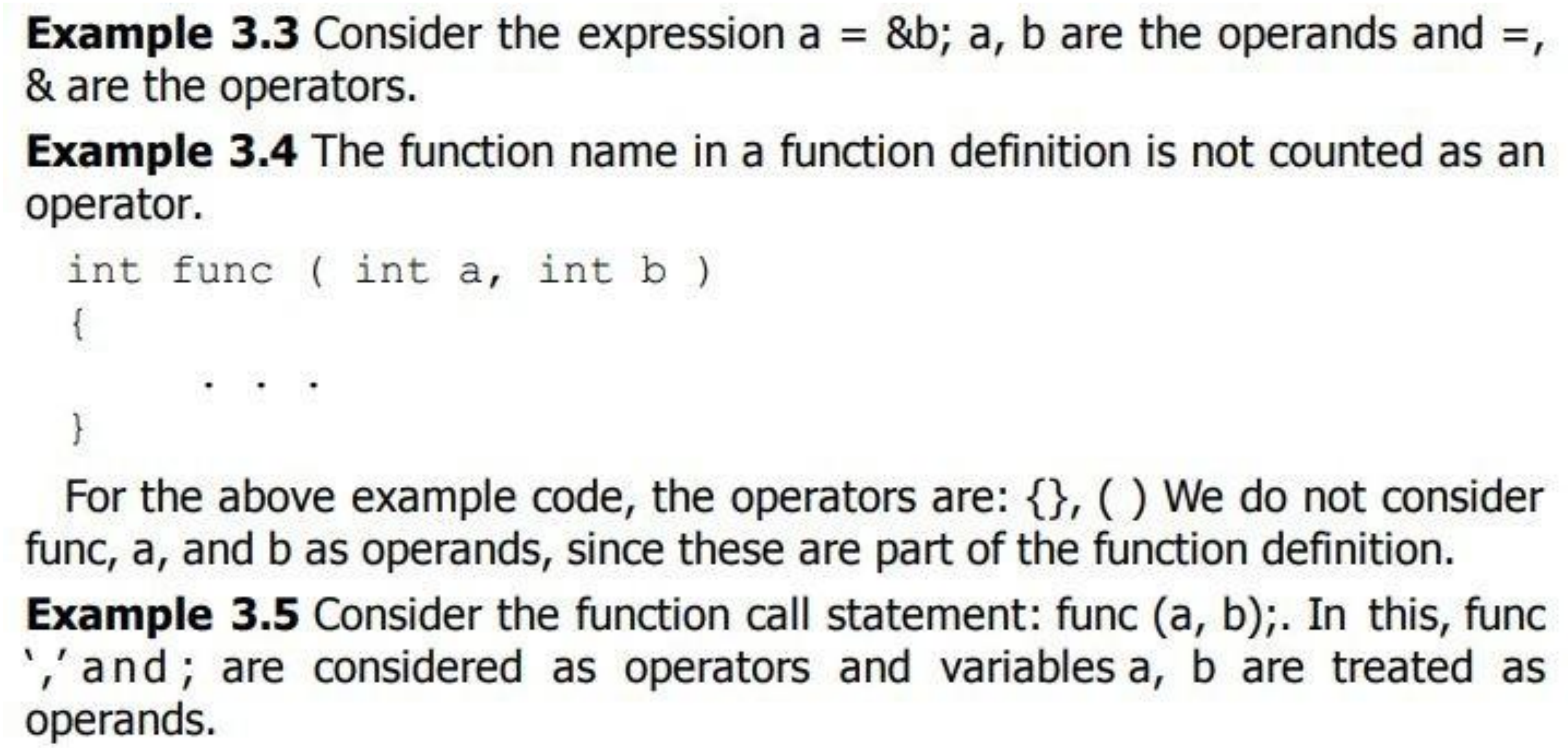
According to Halstead's "A computer program is an implementation of an algorithm considered to be a collection of tokens which can be classified as either operators or operand."
Token Count

In these metrics, a computer program is considered to be a collection of tokens, which may be classified as either operators or operands. All software science metrics can be defined in terms of these basic symbols. These symbols are called as a token.

The basic measures are
n1 = count of unique operators.
n2 = count of unique operands.
N1 = count of total occurrences of operators.
N2 = count of total occurrence of operands.

total tokens used, the size of the program can be expressed as N = N1 + N2.

Program and Software



Example 3.3 Consider the expression a = &b; a, b are the operands and =, & are the operators.

Example 3.4 The function name in a function definition is not counted as an operator.

```
int func ( int a, int b )
{
        . . .
}
```

For the above example code, the operators are: {}, ( ) We do not consider func, a, and b as operands, since these are part of the function definition.

Example 3.5 Consider the function call statement: func (a, b);. In this, func ',' and ; are considered as operators and variables a, b are treated as operands.

Program and Software

Length and Vocabulary

length N = N_1 + N_2.

program vocabulary $h = h_1 + h_2$.

Program Volume

$V = N \log_2 h$

Potential Minimum Volume

, $V^* = (2 + h_2) \log_2 (2 + h_2)$.

program level L is given by $L = V^*/V$.

Effort and Time

effort E = V /L, *= D * V = Difficulty * Volume*

effort E = V^2/V^* (since $L = V^*/V$)

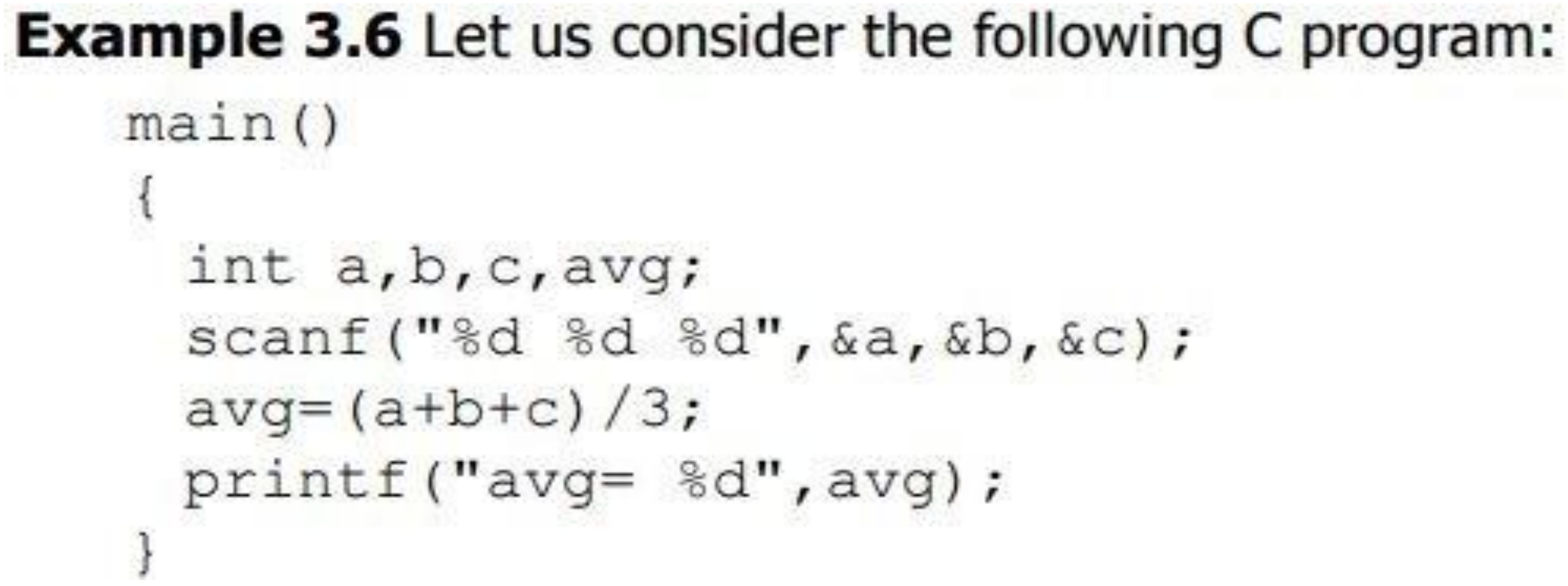
Program Difficulty: *D = (n1 / 2) * (N2 / n2) D = 1 / L*

Length Estimation

$$N = log_2 \eta_1^{\eta_1} + log_2 \eta_2^{\eta_2}$$

$$= \eta_1 log_2 \eta_1 + \eta_2 log_2 \eta_2$$

Program and Software



Example 3.6 Let us consider the following C program:

```
main()
{
  int a,b,c,avg;
  scanf("%d %d %d",&a,&b,&c);
  avg=(a+b+c)/3;
  printf("avg= %d",avg);
}
```

Program and Software

The unique operators are: `main, (), {}, int, scanf, &, “,”, “;”, =, +, /, printf`

The unique operands are: `a,b,c,&a,&b,&c,a+b+c,avg,3,”%d %d %d”, “avg=%d”`

Therefore,

$$\eta_1 = 12, \eta_2 = 11$$

$$\text{Estimated Length} = (12 * \log 12 + 11 * \log 11)$$

$$= (12 * 3.58 + 11 * 3.45) = (43 + 38) = 81$$

Program and Software

Example – List out the operators and operands and also calculate the values of software science measures.

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

Program and Software

Operators	Occurrences	Operands	Occurrences
int	4	sort	1
()	5	x	7
,	4	n	3
[]	7	i	8
if	2	j	7
<	2	save	3
;	11	im1	3
for	2	2	2
=	6	1	3
–	1	0	1
<=	2	–	–
++	2	–	–
return	2	–	–
{}	3	–	–
n1=14	N1=53	n2=10	N2=38

Program and Software

Therefore,

N = 91

n = 24

V = 417.23 bits

V* = 11.6

L = 0.027

D = 37.03

Program and Software

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.

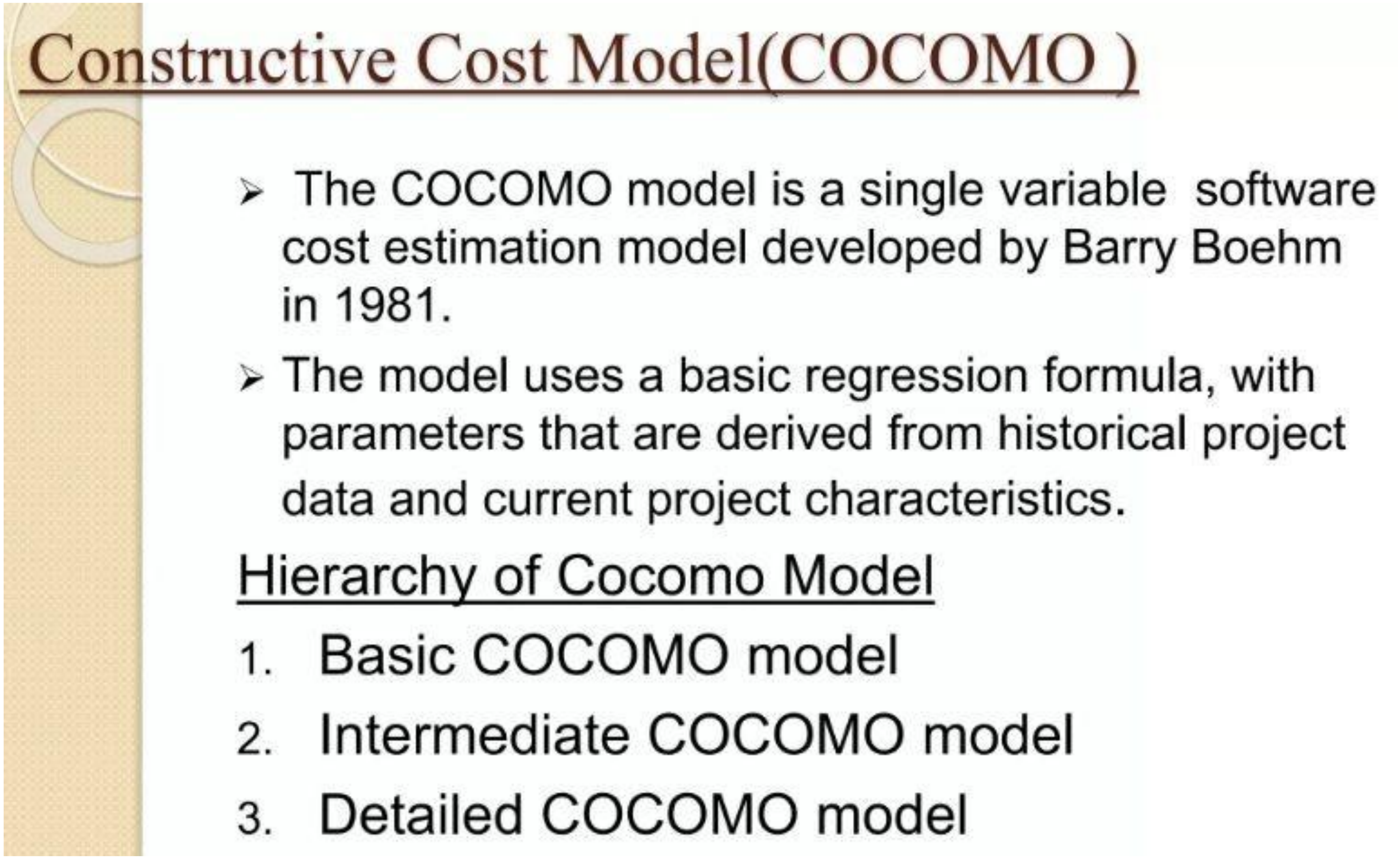
Program and Software

.

Dis-advantages 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

Program and Software



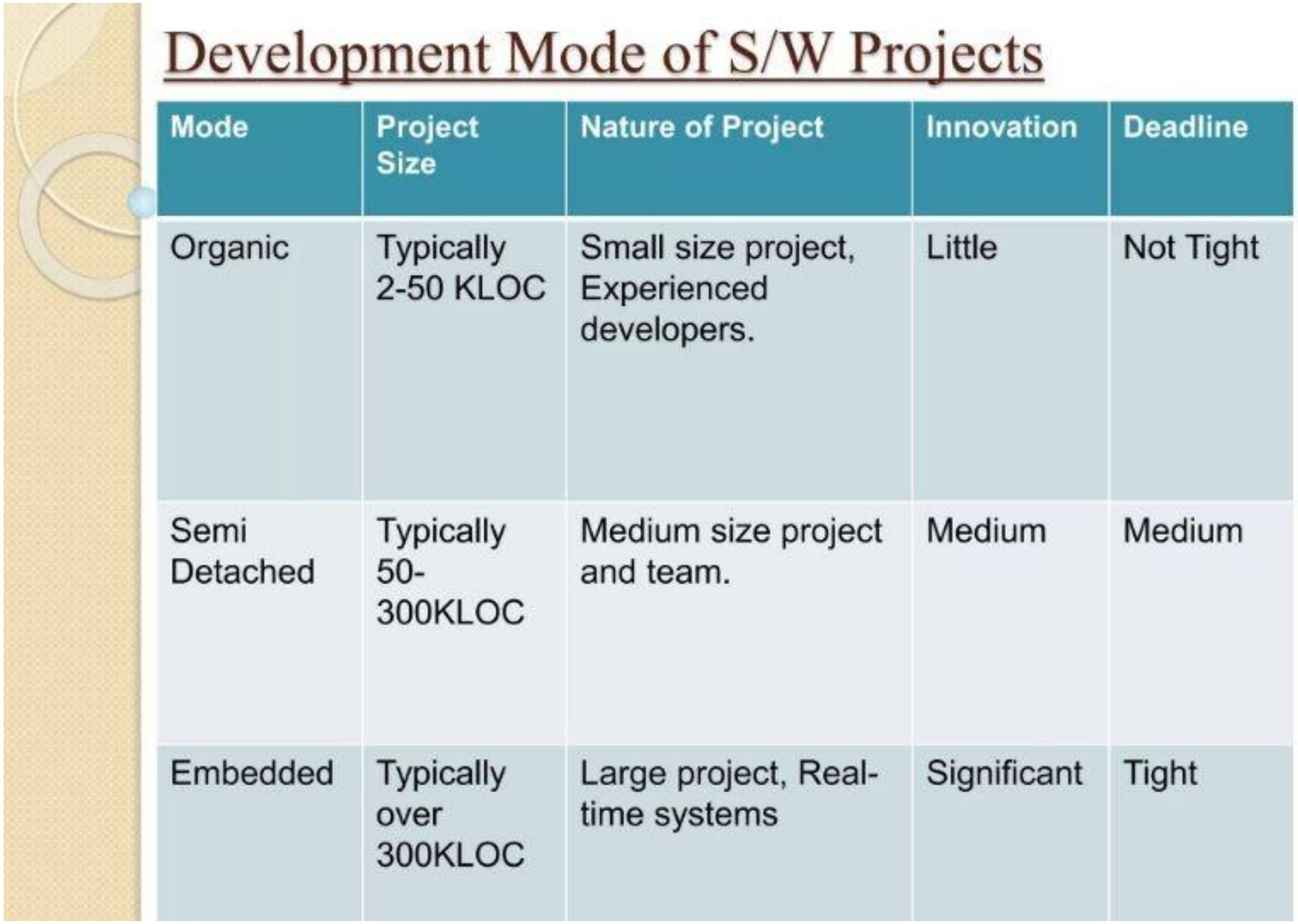
Constructive Cost Model(COCOMO)

- The COCOMO model is a single variable software cost estimation model developed by Barry Boehm in 1981.
- The model uses a basic regression formula, with parameters that are derived from historical project data and current project characteristics.

Hierarchy of Cocomo Model

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

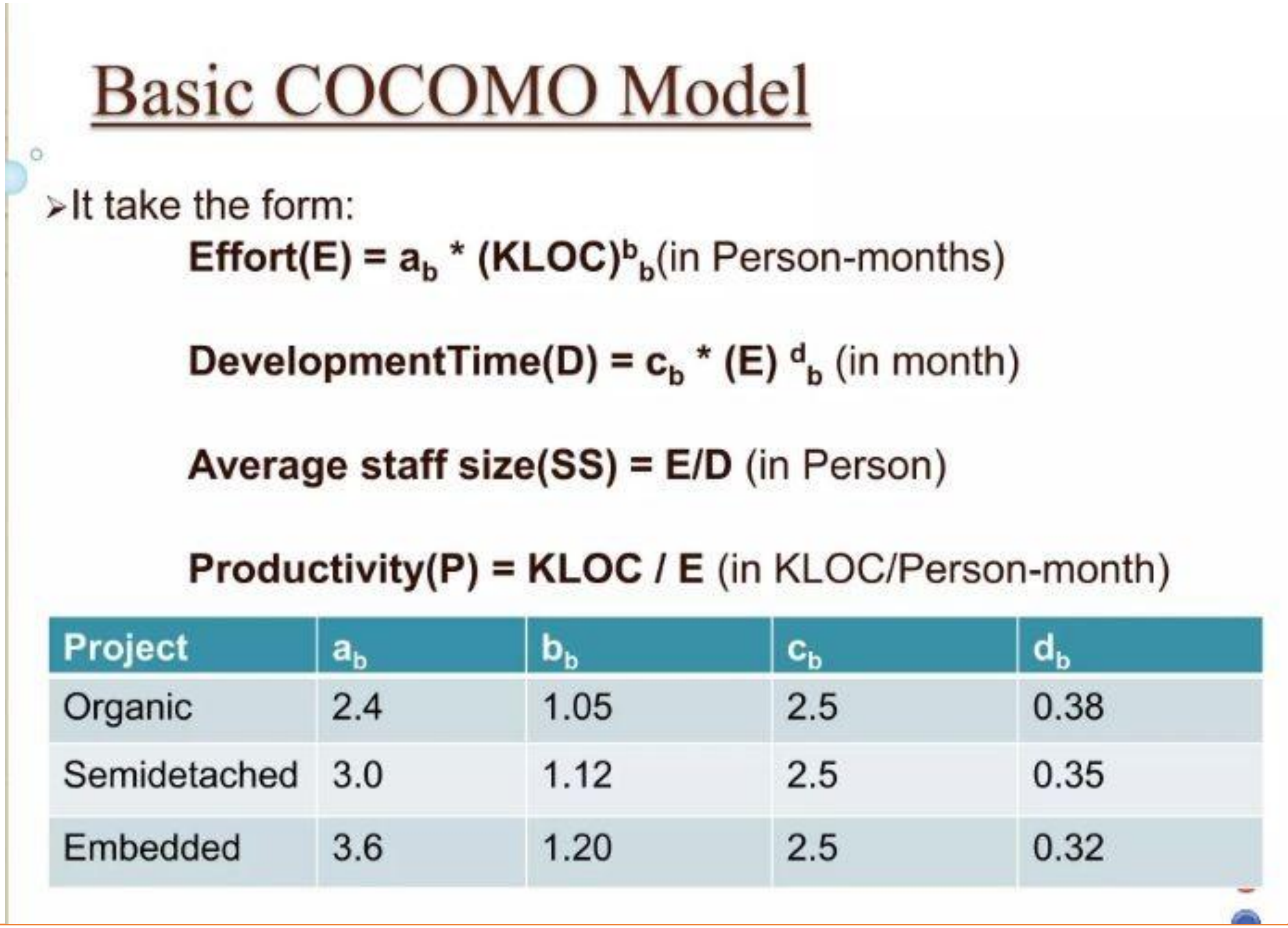
Program and Software



Development Mode of S/W Projects

Mode	Project Size	Nature of Project	Innovation	Deadline
Organic	Typically 2-50 KLOC	Small size project, Experienced developers.	Little	Not Tight
Semi Detached	Typically 50-300KLOC	Medium size project and team.	Medium	Medium
Embedded	Typically over 300KLOC	Large project, Real-time systems	Significant	Tight

Program and Software



Basic COCOMO Model

- It take the form:

  Effort(E) = a_b * $(KLOC)^{b_b}$ (in Person-months)

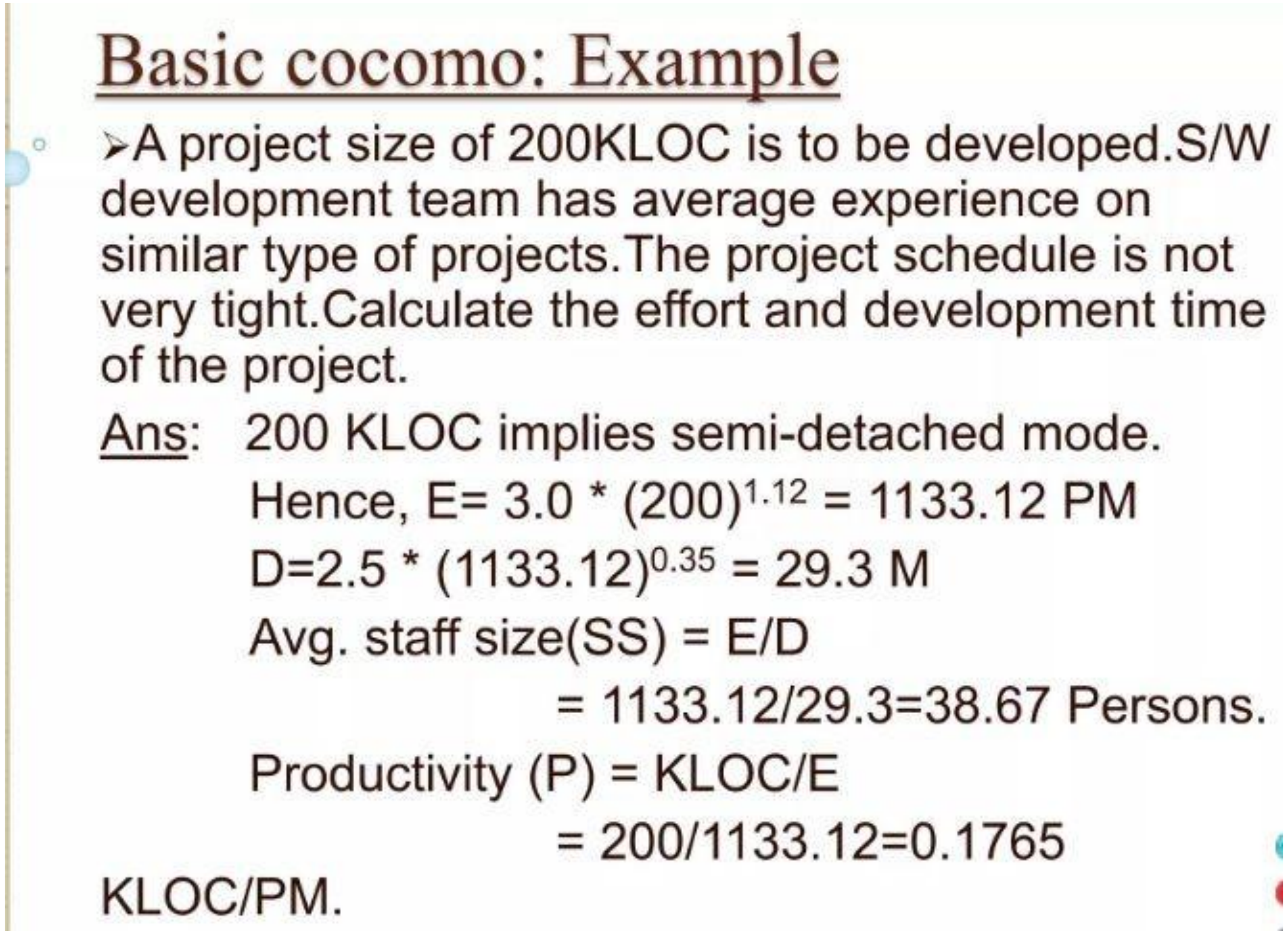
  DevelopmentTime(D) = c_b * $(E)^{d_b}$ (in month)

  Average staff size(SS) = E/D (in Person)

  Productivity(P) = KLOC / E (in KLOC/Person-month)

Project	a_b	b_b	c_b	d_b
Organic	2.4	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Program and Software



Basic cocomo: Example

➢A project size of 200KLOC is to be developed.S/W development team has average experience on similar type of projects.The project schedule is not very tight.Calculate the effort and development time of the project.

Ans: 200 KLOC implies semi-detached mode.

Hence, E= 3.0 * $(200)^{1.12}$ = 1133.12 PM

D=2.5 * $(1133.12)^{0.35}$ = 29.3 M

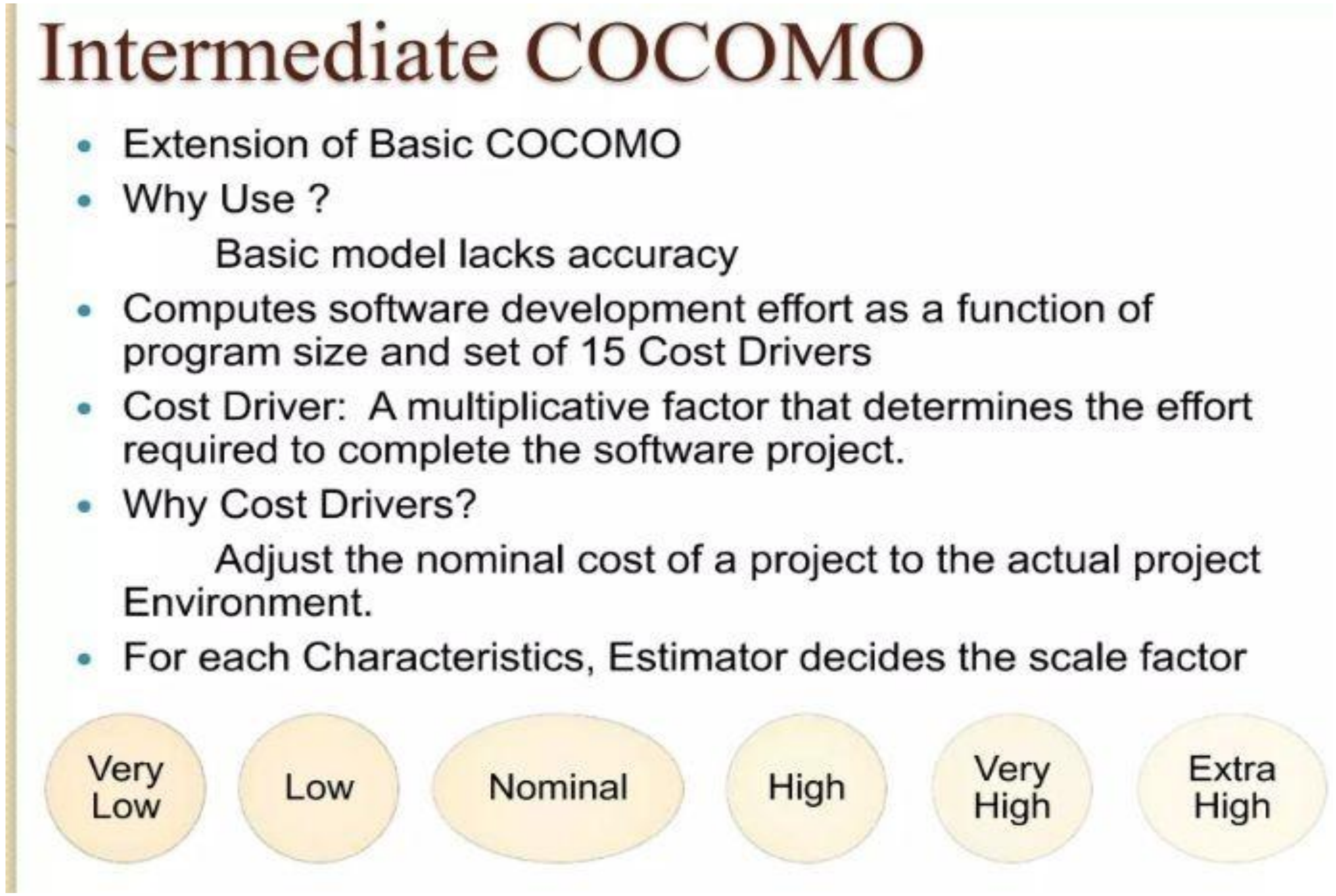
Avg. staff size(SS) = E/D

= 1133.12/29.3=38.67 Persons.

Productivity (P) = KLOC/E

= 200/1133.12=0.1765 KLOC/PM.

Program and Software



Intermediate COCOMO

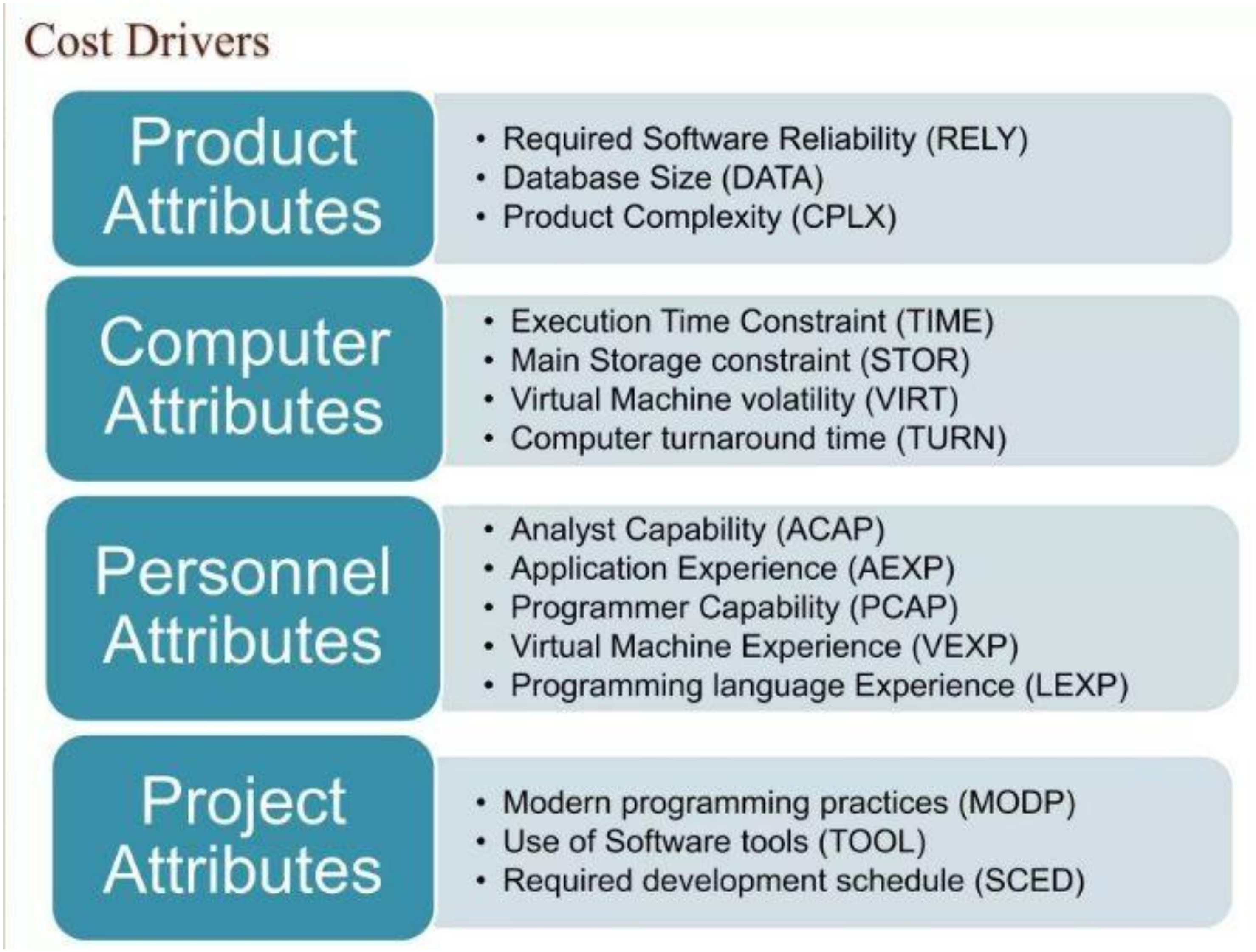
- Extension of Basic COCOMO
- Why Use ?

  Basic model lacks accuracy
- Computes software development effort as a function of program size and set of 15 Cost Drivers
- Cost Driver: A multiplicative factor that determines the effort required to complete the software project.
- Why Cost Drivers?

  Adjust the nominal cost of a project to the actual project Environment.
- For each Characteristics, Estimator decides the scale factor

Very Low | Low | Nominal | High | Very High | Extra High

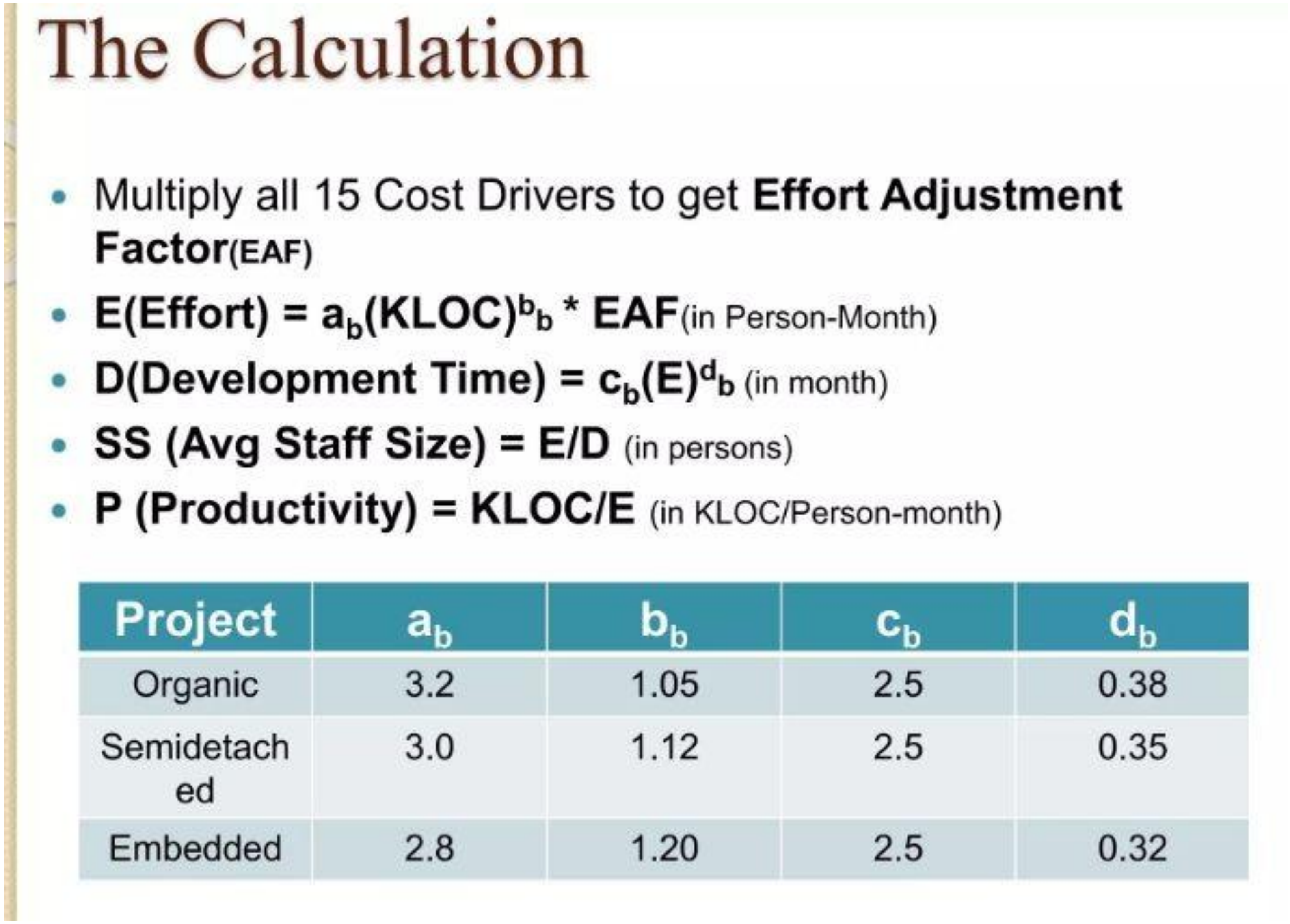
Program and Software



Cost Drivers

Product Attributes	• Required Software Reliability (RELY) • Database Size (DATA) • Product Complexity (CPLX)
Computer Attributes	• Execution Time Constraint (TIME) • Main Storage constraint (STOR) • Virtual Machine volatility (VIRT) • Computer turnaround time (TURN)
Personnel Attributes	• Analyst Capability (ACAP) • Application Experience (AEXP) • Programmer Capability (PCAP) • Virtual Machine Experience (VEXP) • Programming language Experience (LEXP)
Project Attributes	• Modern programming practices (MODP) • Use of Software tools (TOOL) • Required development schedule (SCED)

Program and Software



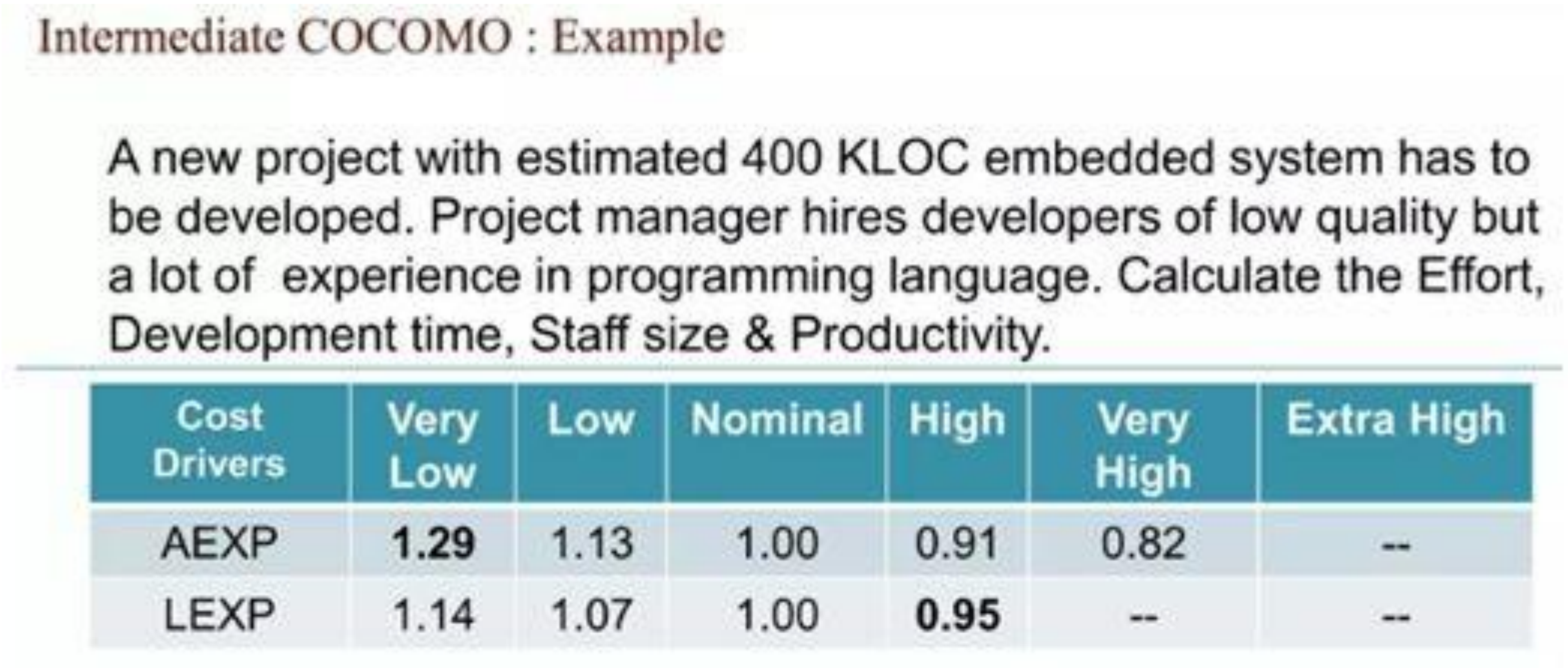
The Calculation

- Multiply all 15 Cost Drivers to get **Effort Adjustment Factor**(EAF)
- **E(Effort) = a_b(KLOC)^{b_b} * EAF**(in Person-Month)
- **D(Development Time) = c_b(E)^{d_b}** (in month)
- **SS (Avg Staff Size) = E/D** (in persons)
- **P (Productivity) = KLOC/E** (in KLOC/Person-month)

Project	a_b	b_b	c_b	d_b
Organic	3.2	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

GL BAJAJ Institute of Technology & Management FIND YOUR SPARK

Program and Software

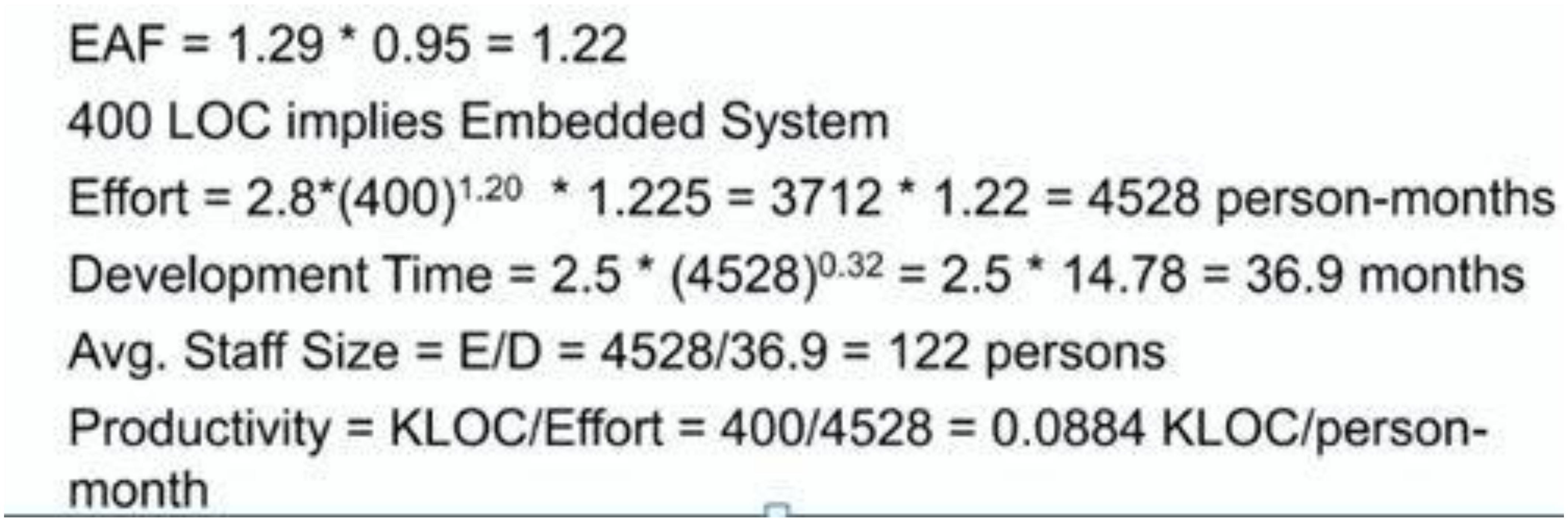


Intermediate COCOMO : Example

A new project with estimated 400 KLOC embedded system has to be developed. Project manager hires developers of low quality but a lot of experience in programming language. Calculate the Effort, Development time, Staff size & Productivity.

Cost Drivers	Very Low	Low	Nominal	High	Very High	Extra High
AEXP	**1.29**	1.13	1.00	0.91	0.82	--
LEXP	1.14	1.07	1.00	**0.95**	--	--

Program and Software



EAF = 1.29 * 0.95 = 1.22

400 LOC implies Embedded System

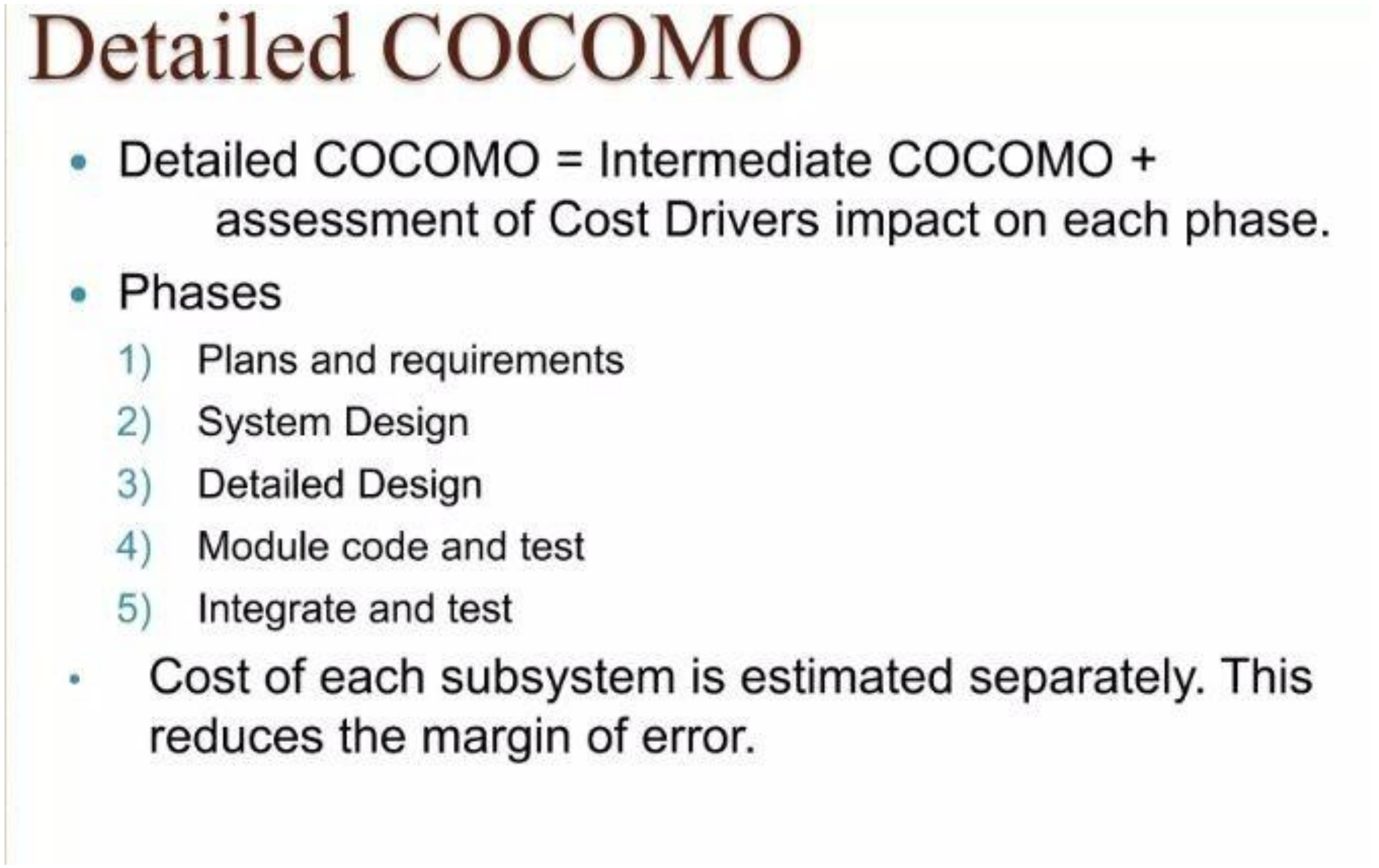
Effort = $2.8*(400)^{1.20}$ * 1.225 = 3712 * 1.22 = 4528 person-months

Development Time = $2.5 * (4528)^{0.32}$ = 2.5 * 14.78 = 36.9 months

Avg. Staff Size = E/D = 4528/36.9 = 122 persons

Productivity = KLOC/Effort = 400/4528 = 0.0884 KLOC/person-month

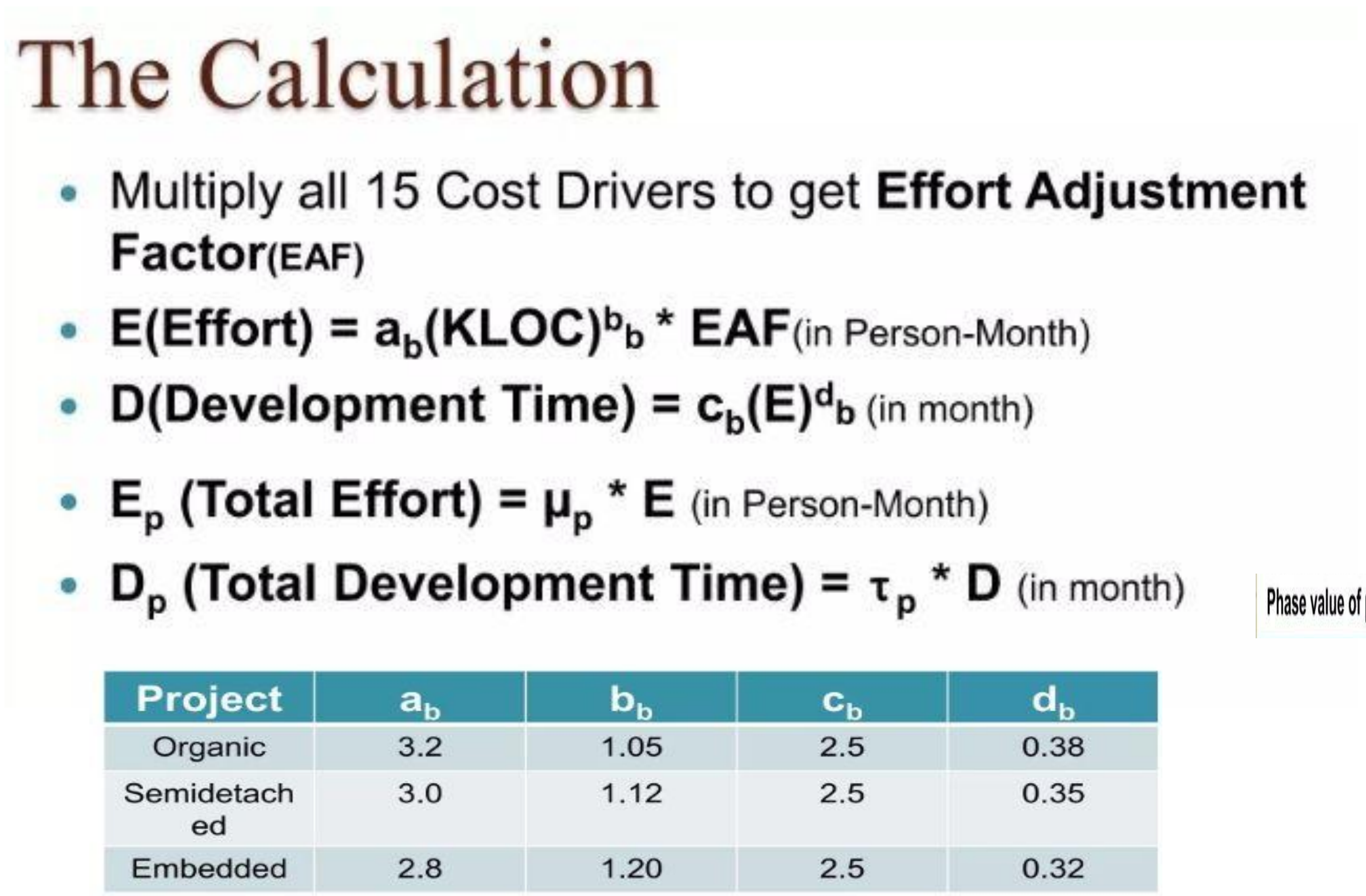
Program and Software



Detailed COCOMO

- Detailed COCOMO = Intermediate COCOMO + assessment of Cost Drivers impact on each phase.
- Phases
  1) Plans and requirements
  2) System Design
  3) Detailed Design
  4) Module code and test
  5) Integrate and test
- Cost of each subsystem is estimated separately. This reduces the margin of error.

Program and Software



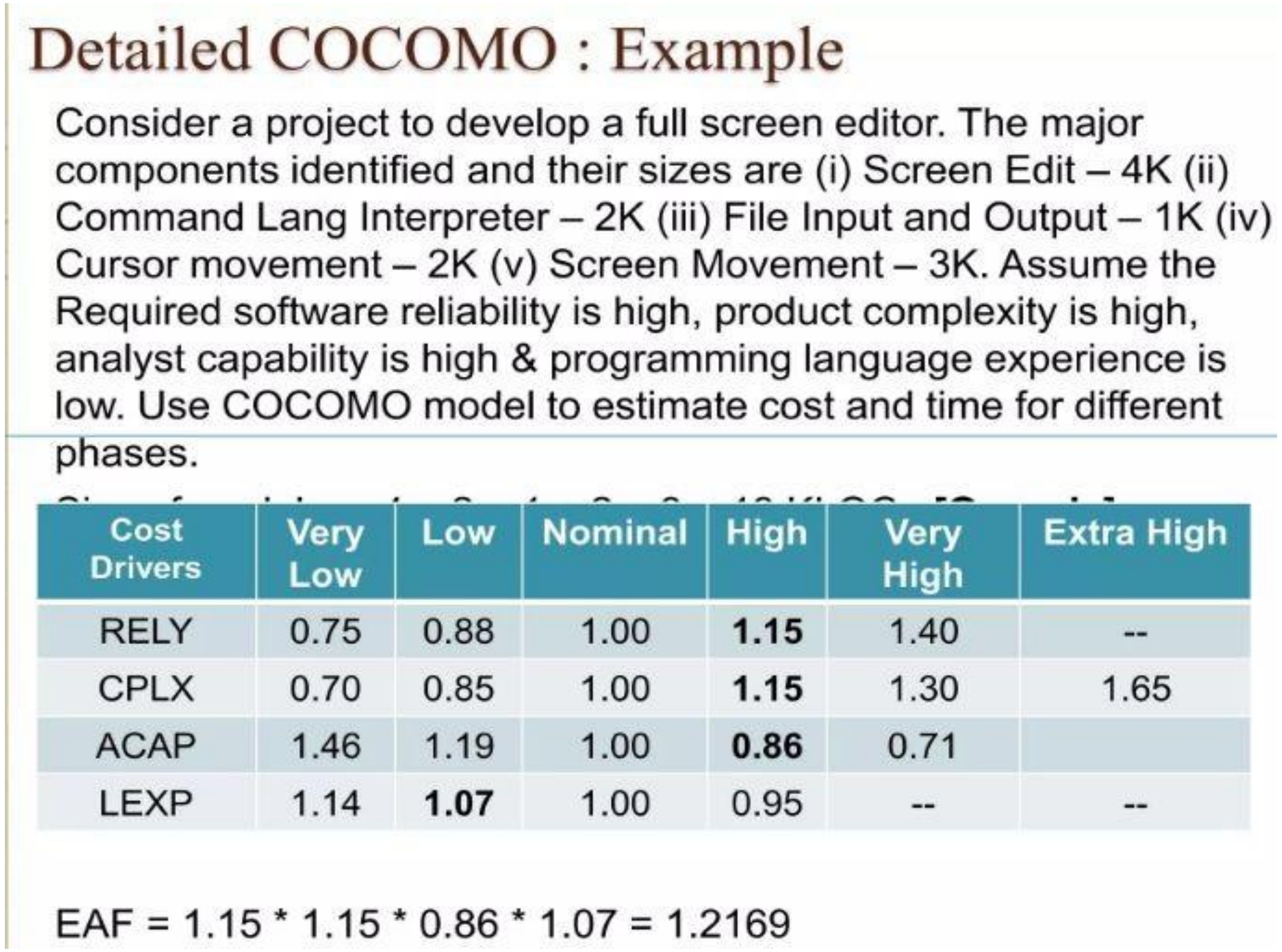
The Calculation

- Multiply all 15 Cost Drivers to get **Effort Adjustment Factor**(EAF)
- **E(Effort) = a_b(KLOC)^{b_b} * EAF**(in Person-Month)
- **D(Development Time) = c_b(E)^{d_b}** (in month)
- **E_p (Total Effort) = μ_p * E** (in Person-Month)
- **D_p (Total Development Time) = τ_p * D** (in month)

Project	a_b	b_b	c_b	d_b
Organic	3.2	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

Phase value of μ_p and τ_p

Program and Software



Detailed COCOMO : Example

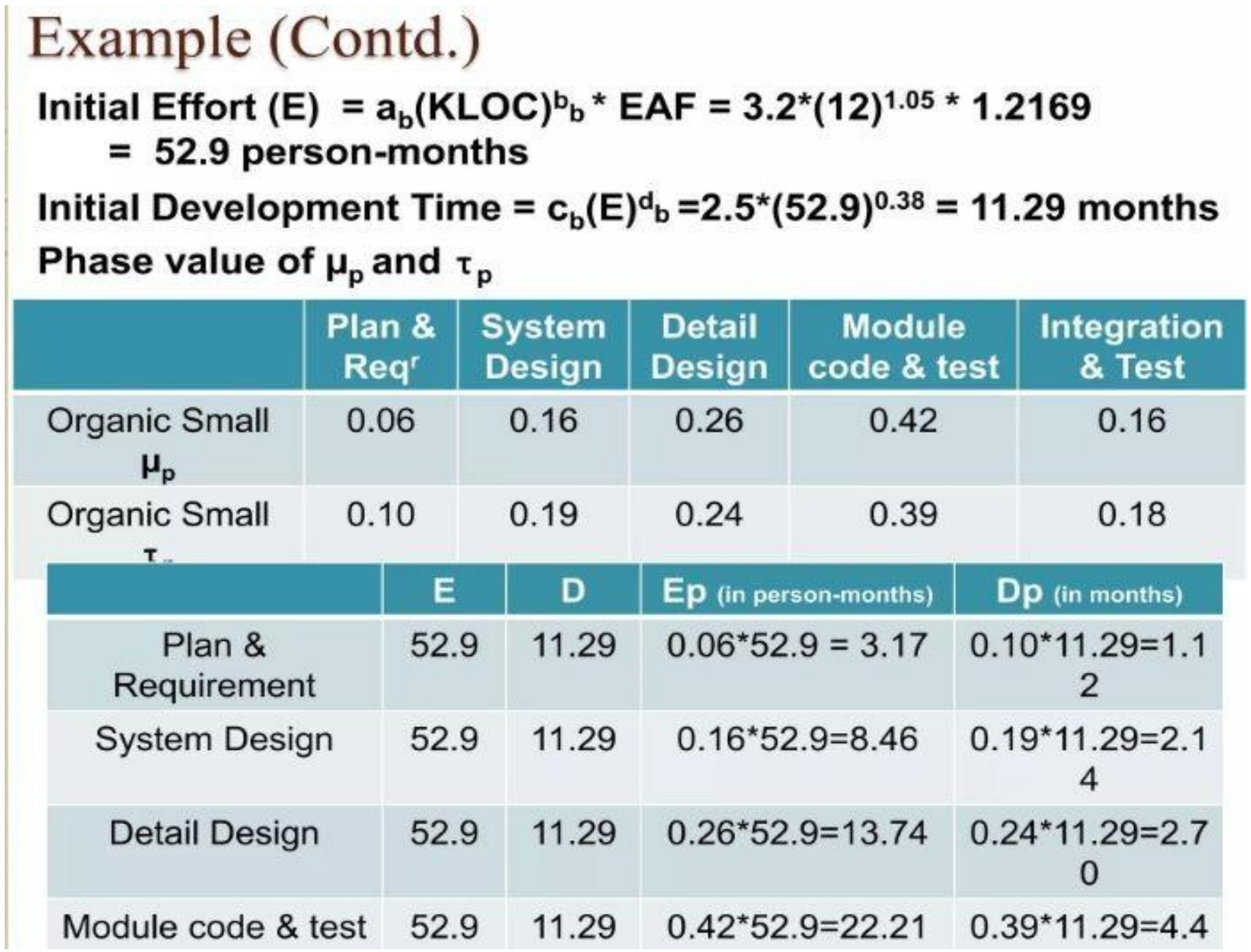
Consider a project to develop a full screen editor. The major components identified and their sizes are (i) Screen Edit – 4K (ii) Command Lang Interpreter – 2K (iii) File Input and Output – 1K (iv) Cursor movement – 2K (v) Screen Movement – 3K. Assume the Required software reliability is high, product complexity is high, analyst capability is high & programming language experience is low. Use COCOMO model to estimate cost and time for different phases.

[illegible]

Cost Drivers	Very Low	Low	Nominal	High	Very High	Extra High
RELY	0.75	0.88	1.00	**1.15**	1.40	--
CPLX	0.70	0.85	1.00	**1.15**	1.30	1.65
ACAP	1.46	1.19	1.00	**0.86**	0.71	
LEXP	1.14	**1.07**	1.00	0.95	--	--

EAF = 1.15 * 1.15 * 0.86 * 1.07 = 1.2169

Program and Software



Example (Contd.)

Initial Effort (E) = $a_b(KLOC)^{b_b}$ * EAF = $3.2*(12)^{1.05}$ * 1.2169
= 52.9 person-months

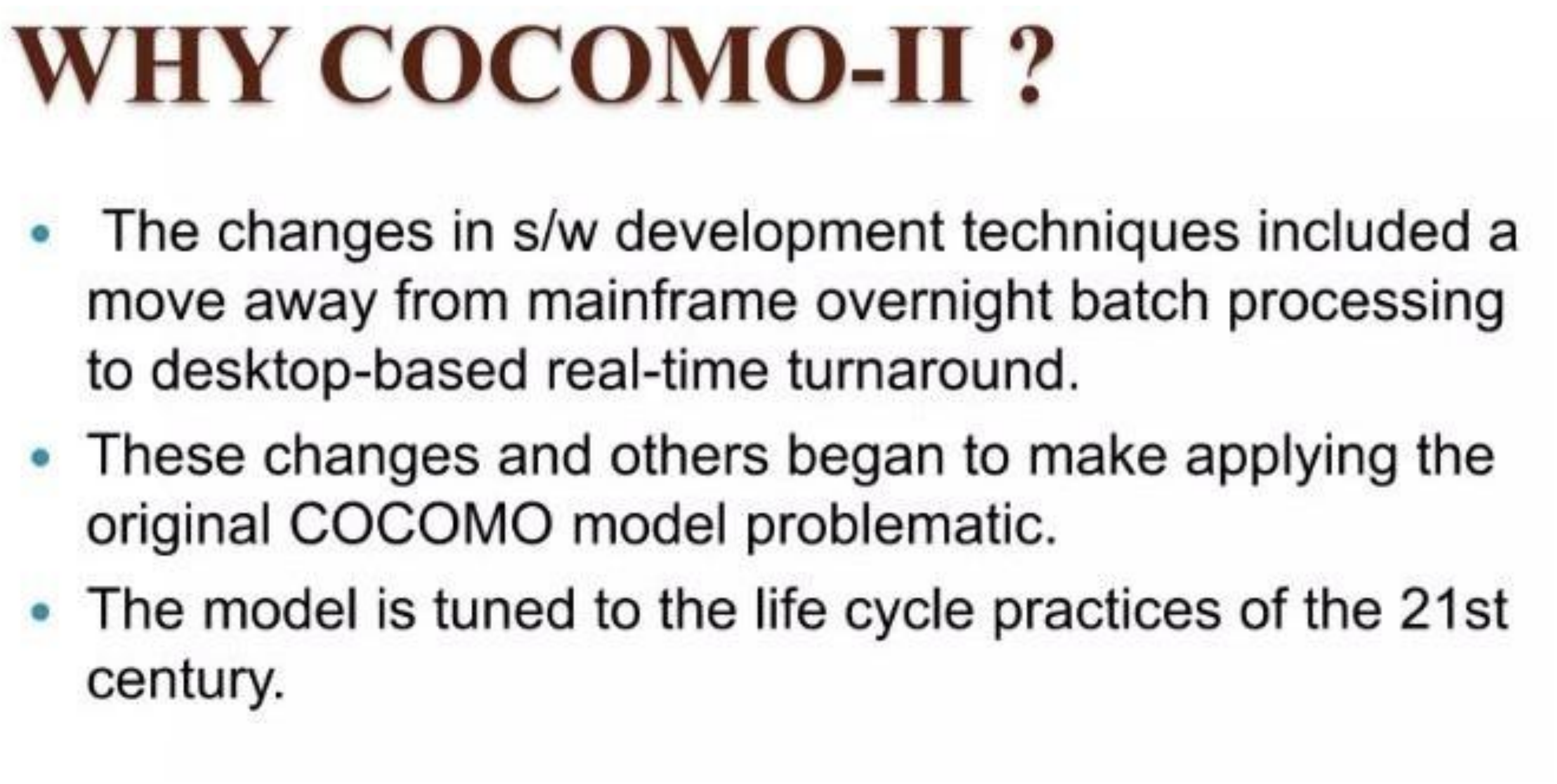
Initial Development Time = $c_b(E)^{d_b}$ =$2.5*(52.9)^{0.38}$ = 11.29 months

Phase value of μ_p and τ_p

	Plan & Req^r	System Design	Detail Design	Module code & test	Integration & Test
Organic Small μ_p	0.06	0.16	0.26	0.42	0.16
Organic Small τ	0.10	0.19	0.24	0.39	0.18

	E	D	Ep (in person-months)	Dp (in months)
Plan & Requirement	52.9	11.29	0.06*52.9 = 3.17	0.10*11.29=1.12
System Design	52.9	11.29	0.16*52.9=8.46	0.19*11.29=2.14
Detail Design	52.9	11.29	0.26*52.9=13.74	0.24*11.29=2.70
Module code & test	52.9	11.29	0.42*52.9=22.21	0.39*11.29=4.4

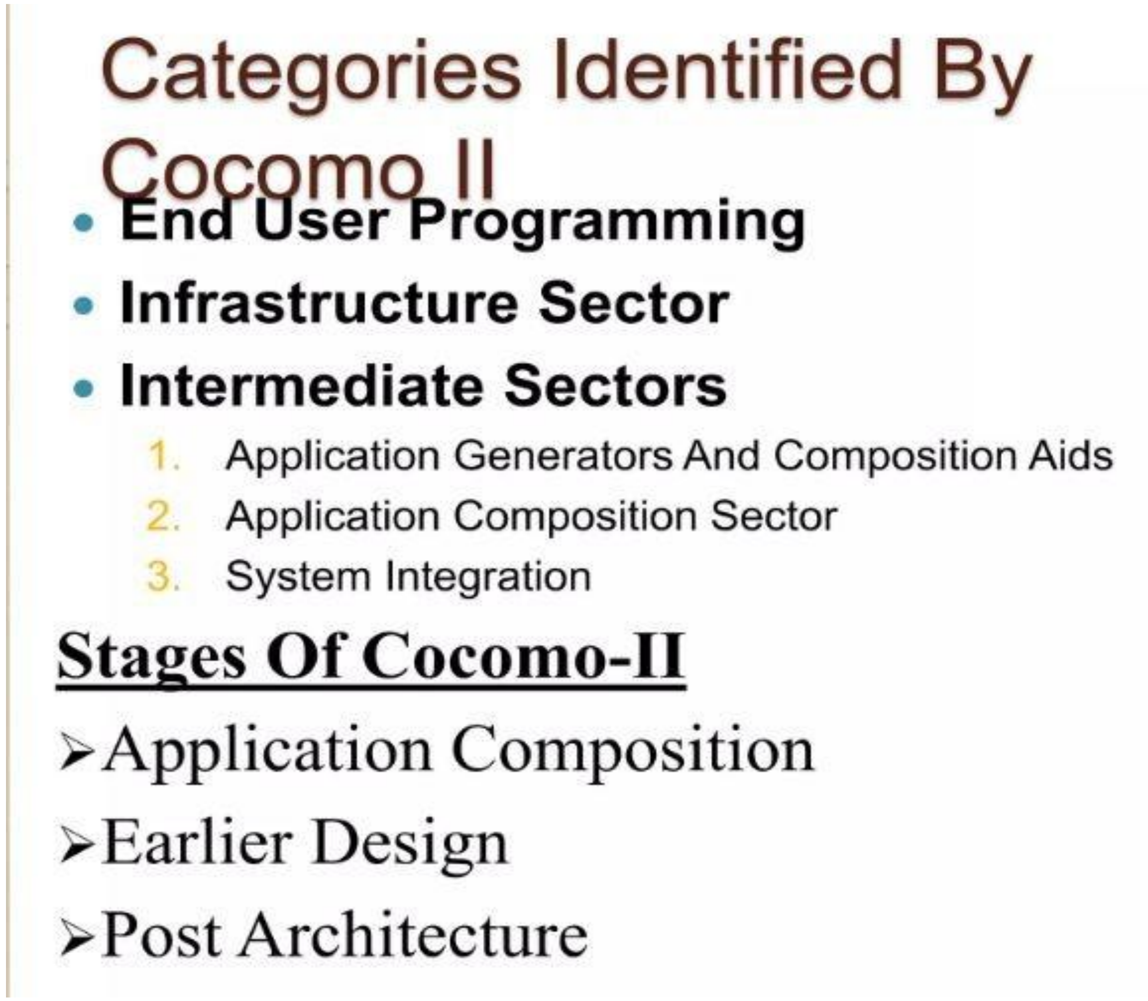
Program and Software



WHY COCOMO-II ?

- The changes in s/w development techniques included a move away from mainframe overnight batch processing to desktop-based real-time turnaround.
- These changes and others began to make applying the original COCOMO model problematic.
- The model is tuned to the life cycle practices of the 21st century.

Program and Software



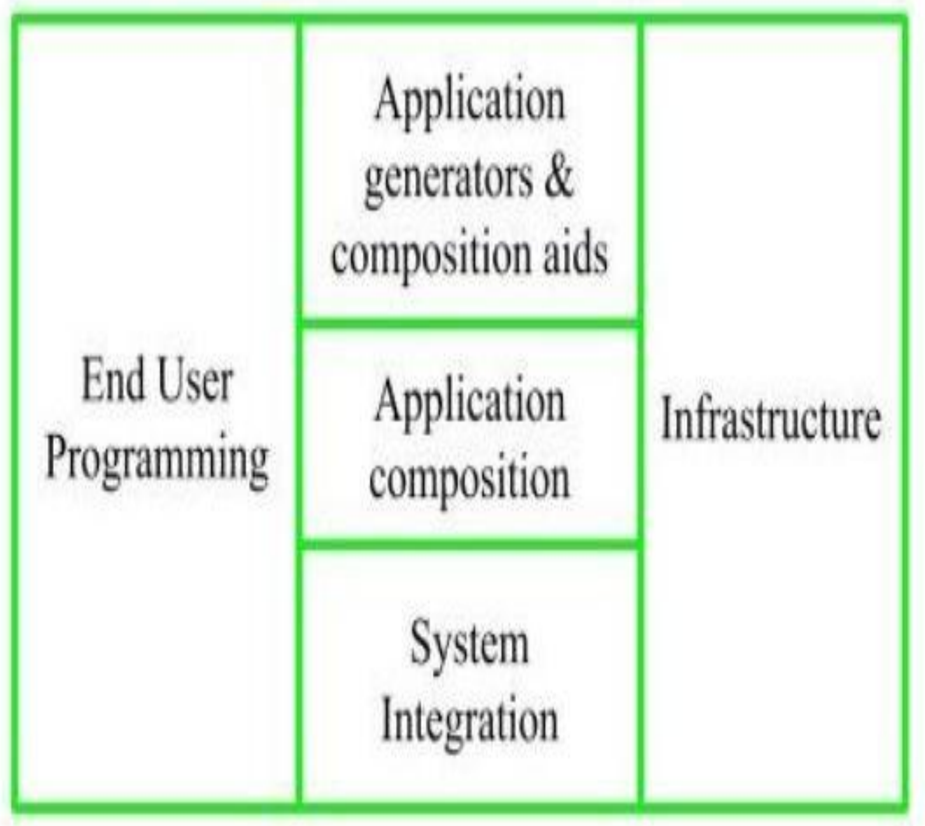
Categories Identified By Cocomo II

- **End User Programming**
- **Infrastructure Sector**
- **Intermediate Sectors**
  1. Application Generators And Composition Aids
  2. Application Composition Sector
  3. System Integration

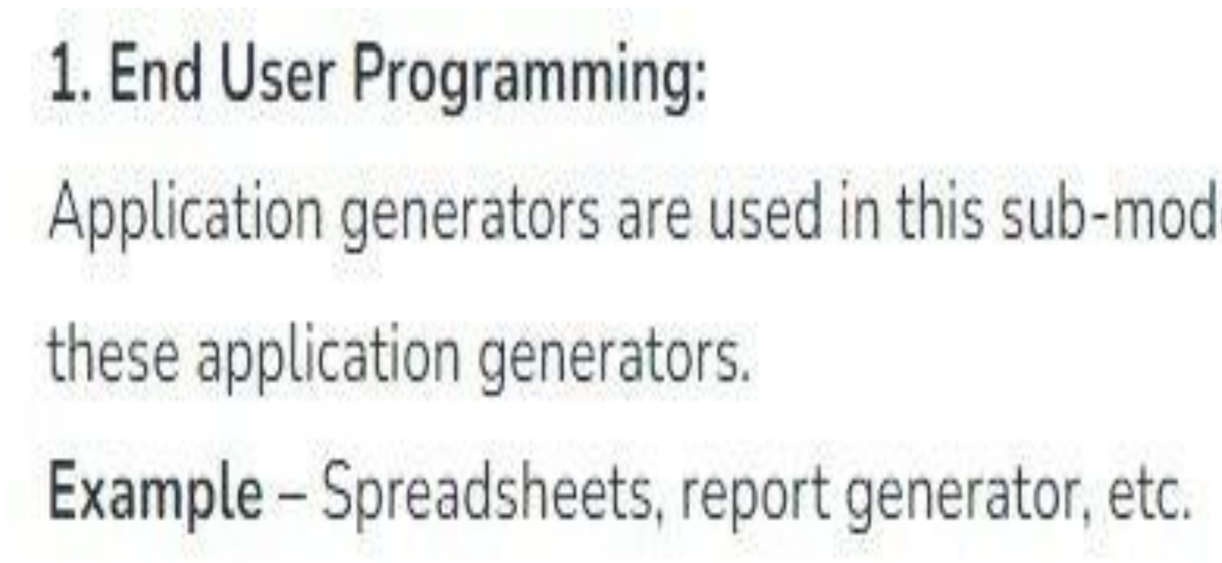
Stages Of Cocomo-II

- ➢Application Composition
- ➢Earlier Design
- ➢Post Architecture

It consists of three sub-models:



Program and Software



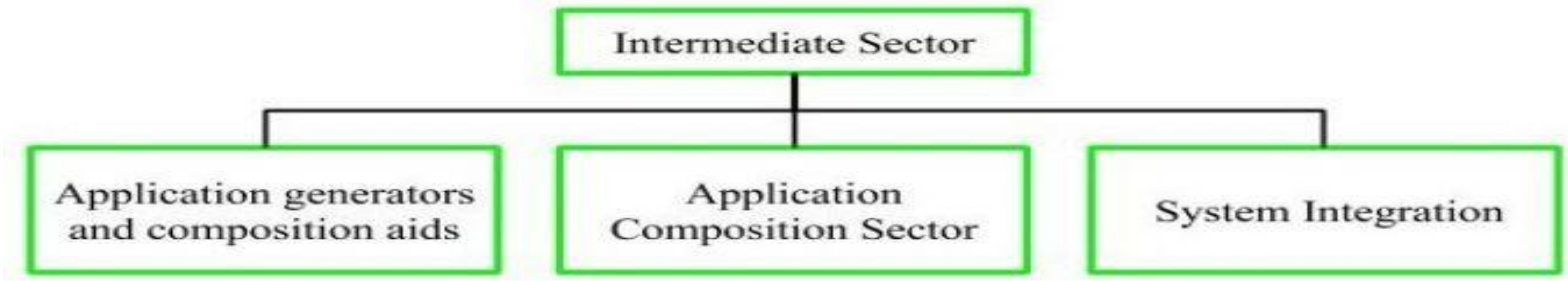
1. End User Programming:

Application generators are used in this sub-model. End user write the code by using these application generators.

Example – Spreadsheets, report generator, etc.

Program and Software

2. Intermediate Sector:

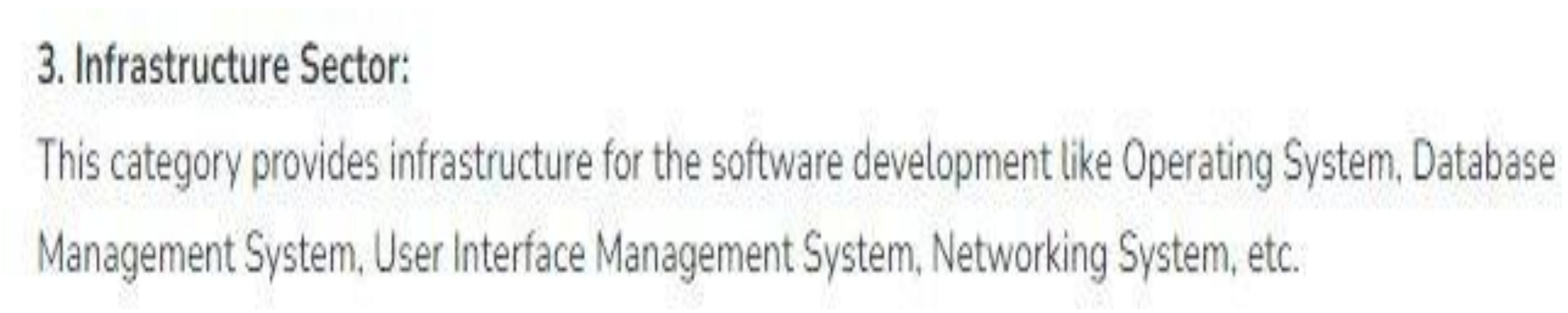

Intermediate Sector
Application generators and composition aids
Application Composition Sector
System Integration

- **(a). Application Generators and Composition Aids –**
  This category will create largely prepackaged capabilities for user programming. Their product will have many reusable components. Typical firms operating in this sector are Microsoft, Lotus, Oracle, IBM, Borland, Novell.
- **(b). Application Composition Sector –**
  This category is too diversified and to be handled by prepackaged solutions. It includes GUI, Databases, domain specific components such as financial, medical or industrial process control packages.
- **(c). System Integration –**
  This category deals with large scale and highly embedded systems.

3. Infrastructure Sector:

This category provides infrastructure for the software development like Operating System, Database Management System, User Interface Management System, Networking System, etc.

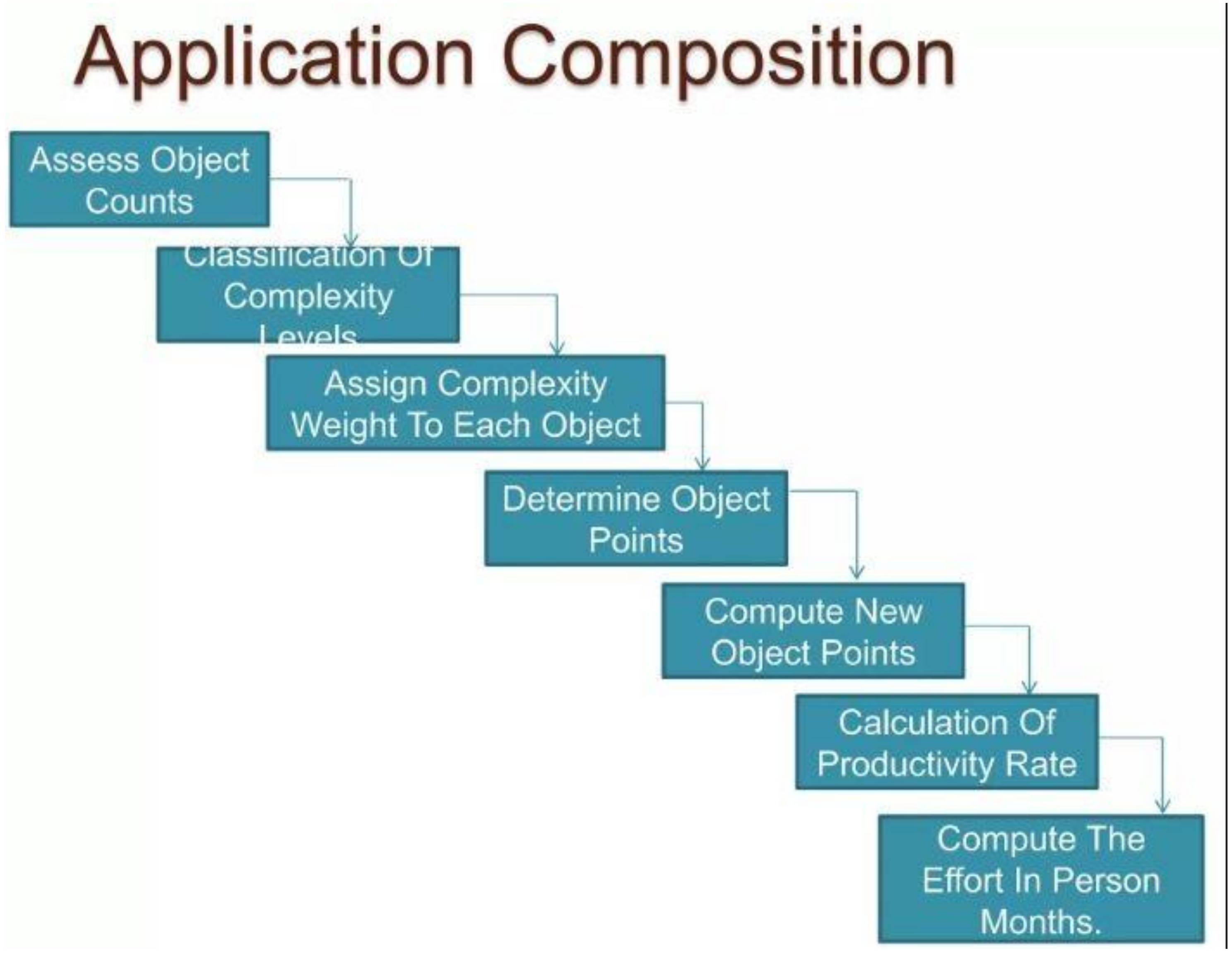
Program and Software



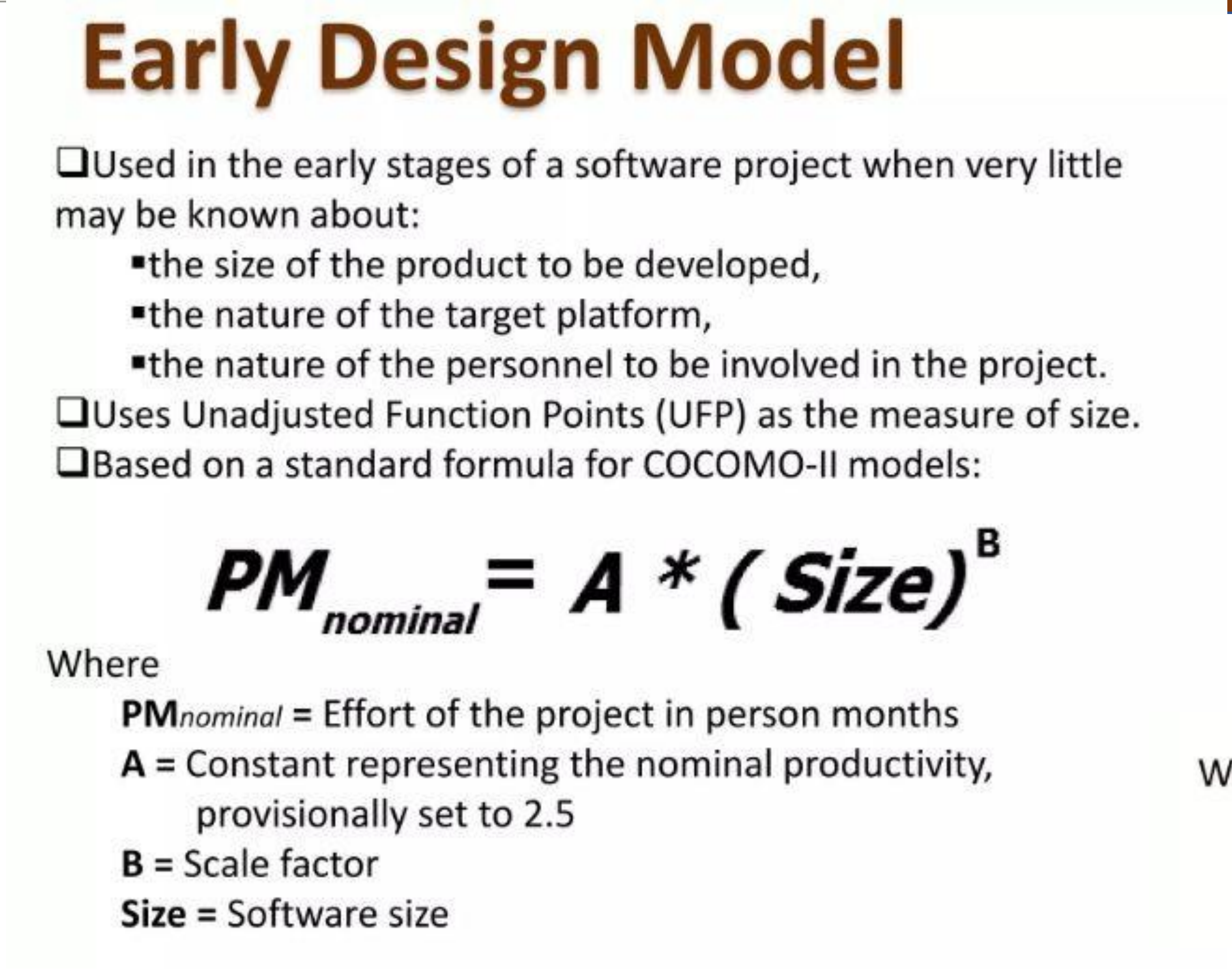
3. Infrastructure Sector:

This category provides infrastructure for the software development like Operating System, Database Management System, User Interface Management System, Networking System, etc.

Program and Software


Application Composition
Assess Object Counts
Classification Of Complexity Levels
Assign Complexity Weight To Each Object
Determine Object Points
Compute New Object Points
Calculation Of Productivity Rate
Compute The Effort In Person Months.

Program and Software



Early Design Model

- ❑Used in the early stages of a software project when very little may be known about:
  - ▪the size of the product to be developed,
  - ▪the nature of the target platform,
  - ▪the nature of the personnel to be involved in the project.
- ❑Uses Unadjusted Function Points (UFP) as the measure of size.
- ❑Based on a standard formula for COCOMO-II models:

$$PM_{nominal} = A * (Size)^{B}$$

Where

PM_nominal_ = Effort of the project in person months

A = Constant representing the nominal productivity, provisionally set to 2.5

B = Scale factor

Size = Software size

Based on 7 cost drivers

$$PM_{adjusted} = PM_{nominal} * \left[\prod_{i=1}^{7} EM_i\right]$$

Where

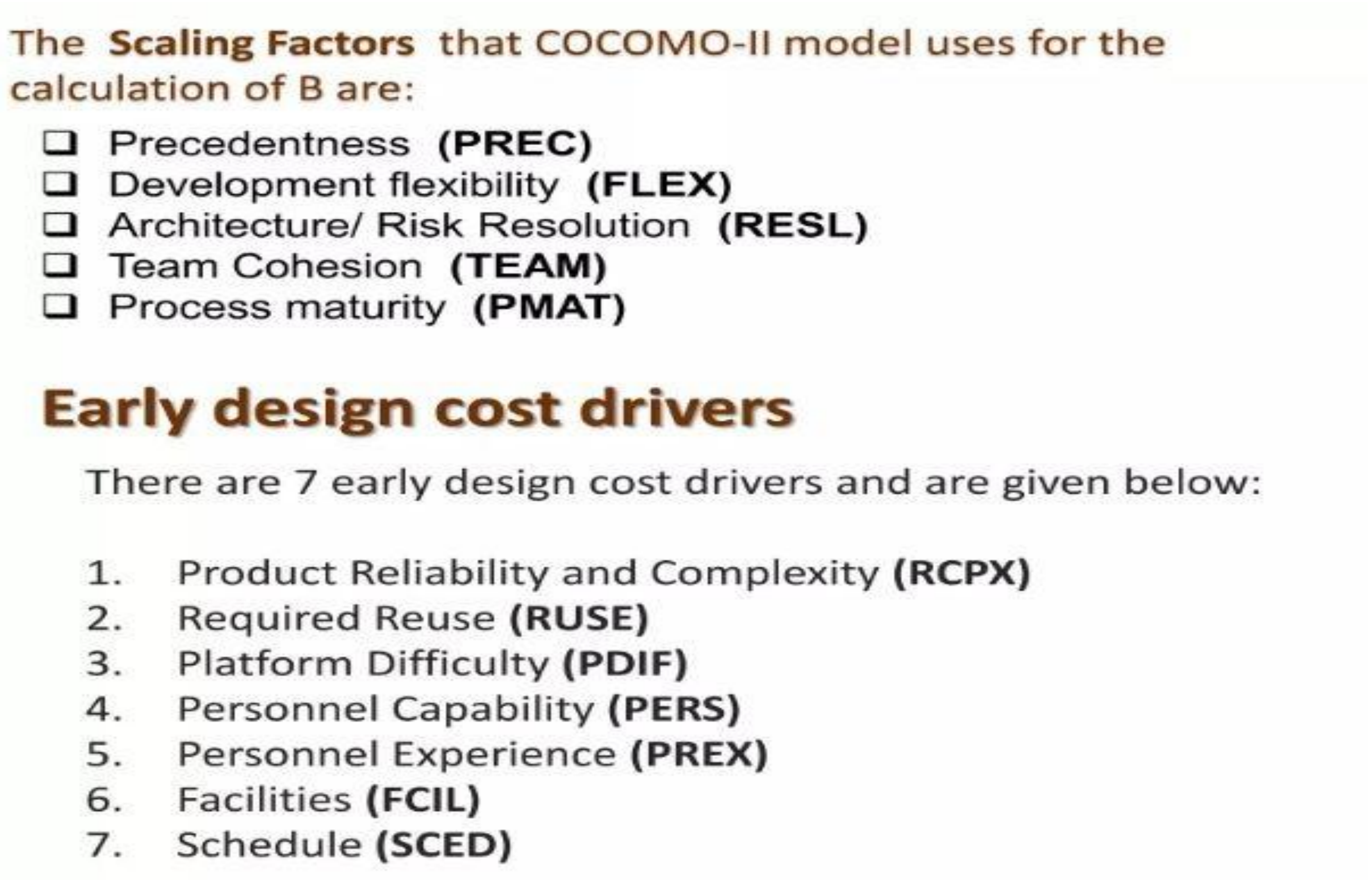
EM : Effort multiplier which is the product of 7 **cost drivers.**

PM$_{nominal}$ = Effort of the project in person months.

Where

B = 0.91 + 0.01 * (Sum of rating on scaling factors for the project)

Program and Software



The **Scaling Factors** that COCOMO-II model uses for the calculation of B are:

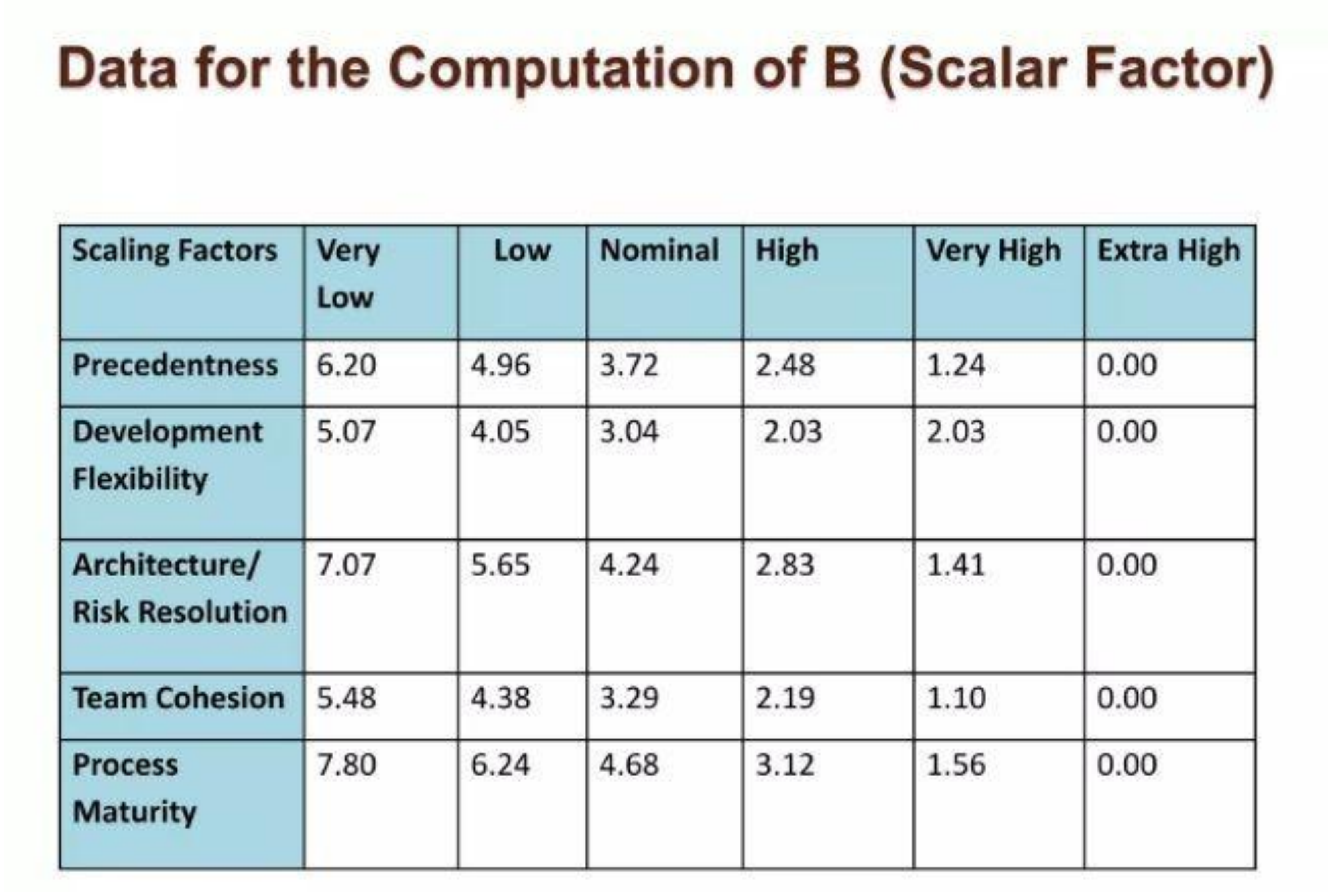
- ❑ Precedentness **(PREC)**
- ❑ Development flexibility **(FLEX)**
- ❑ Architecture/ Risk Resolution **(RESL)**
- ❑ Team Cohesion **(TEAM)**
- ❑ Process maturity **(PMAT)**

Early design cost drivers

There are 7 early design cost drivers and are given below:

1. Product Reliability and Complexity **(RCPX)**
2. Required Reuse **(RUSE)**
3. Platform Difficulty **(PDIF)**
4. Personnel Capability **(PERS)**
5. Personnel Experience **(PREX)**
6. Facilities **(FCIL)**
7. Schedule **(SCED)**

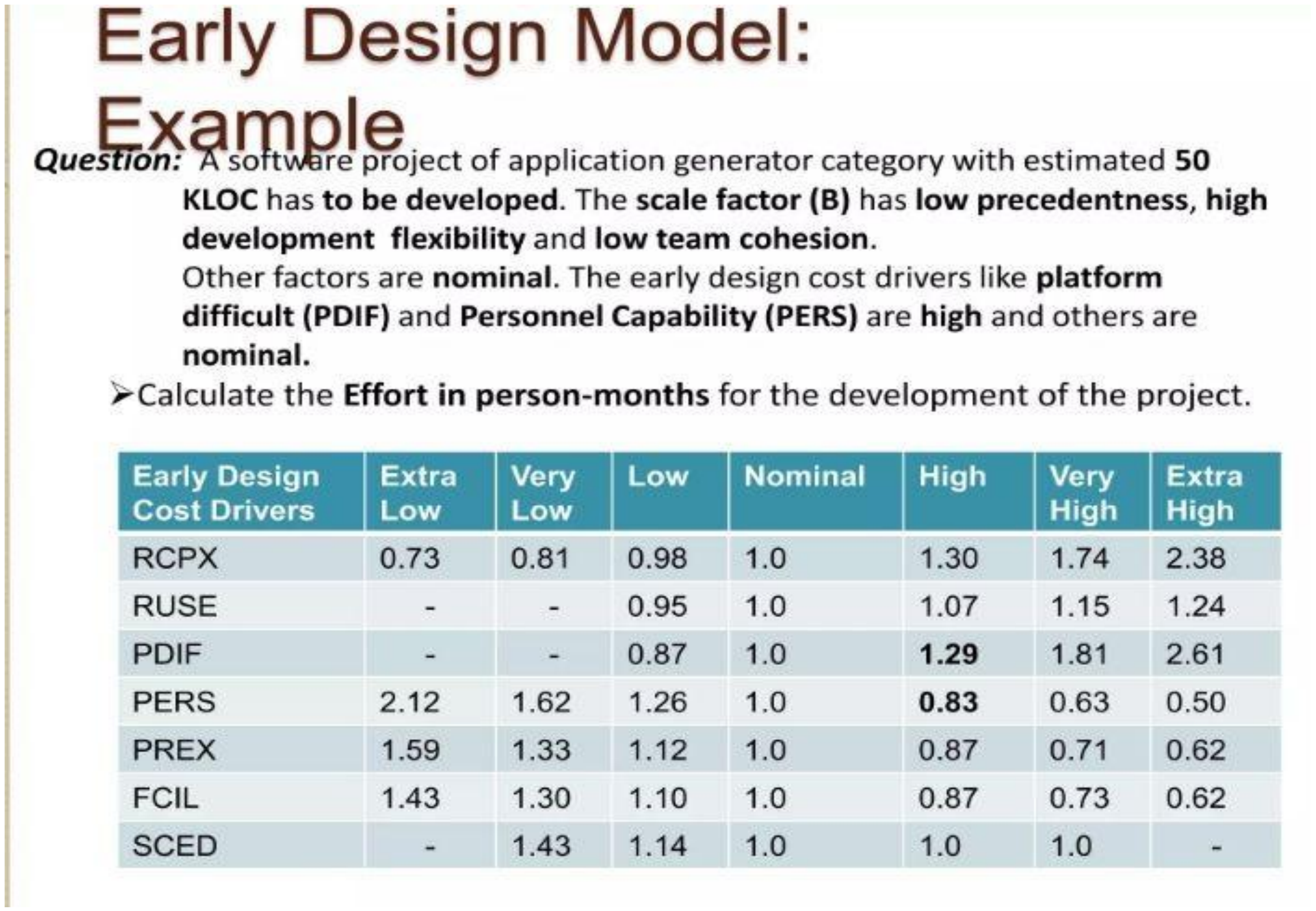
Program and Software



Data for the Computation of B (Scalar Factor)

Scaling Factors	Very Low	Low	Nominal	High	Very High	Extra High
Precedentness	6.20	4.96	3.72	2.48	1.24	0.00
Development Flexibility	5.07	4.05	3.04	2.03	2.03	0.00
Architecture/ Risk Resolution	7.07	5.65	4.24	2.83	1.41	0.00
Team Cohesion	5.48	4.38	3.29	2.19	1.10	0.00
Process Maturity	7.80	6.24	4.68	3.12	1.56	0.00

Program and Software



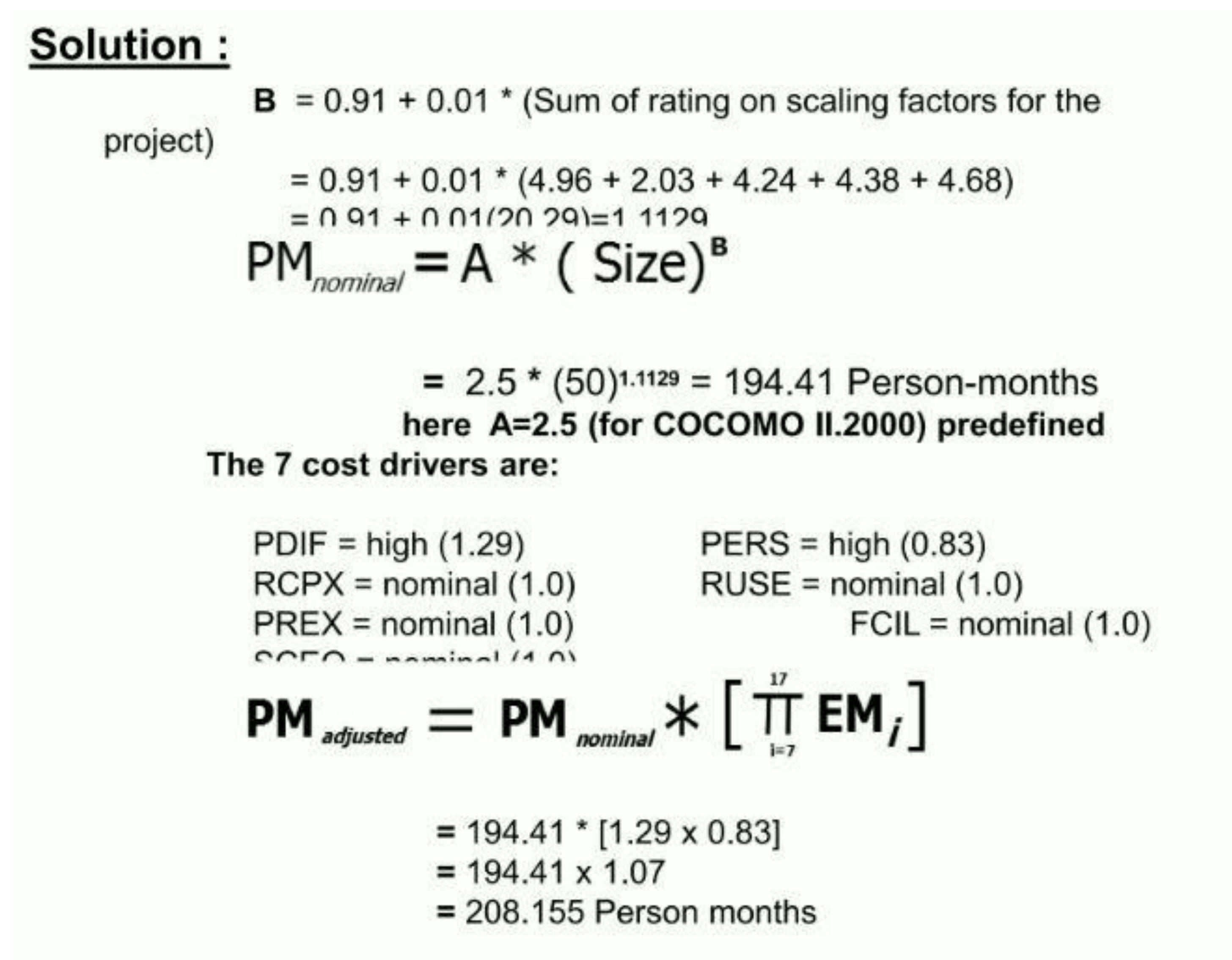
Early Design Model: Example

Question: A software project of application generator category with estimated **50 KLOC** has **to be developed**. The **scale factor (B)** has **low precedentness, high development flexibility** and **low team cohesion.**
Other factors are **nominal**. The early design cost drivers like **platform difficult (PDIF)** and **Personnel Capability (PERS)** are **high** and others are **nominal.**

- Calculate the **Effort in person-months** for the development of the project.

Early Design Cost Drivers	Extra Low	Very Low	Low	Nominal	High	Very High	Extra High
RCPX	0.73	0.81	0.98	1.0	1.30	1.74	2.38
RUSE	-	-	0.95	1.0	1.07	1.15	1.24
PDIF	-	-	0.87	1.0	**1.29**	1.81	2.61
PERS	2.12	1.62	1.26	1.0	**0.83**	0.63	0.50
PREX	1.59	1.33	1.12	1.0	0.87	0.71	0.62
FCIL	1.43	1.30	1.10	1.0	0.87	0.73	0.62
SCED	-	1.43	1.14	1.0	1.0	1.0	-

Program and Software



Solution :

B = 0.91 + 0.01 * (Sum of rating on scaling factors for the project)

= 0.91 + 0.01 * (4.96 + 2.03 + 4.24 + 4.38 + 4.68)

= 0.91 + 0.01(20.29)=1.1129

$$PM_{nominal} = A * (\,Size)^{B}$$

= 2.5 * $(50)^{1.1129}$ = 194.41 Person-months

here A=2.5 (for COCOMO II.2000) predefined

The 7 cost drivers are:

PDIF = high (1.29)	PERS = high (0.83)
RCPX = nominal (1.0)	RUSE = nominal (1.0)
PREX = nominal (1.0)	FCIL = nominal (1.0)
SCED = nominal (1.0)	

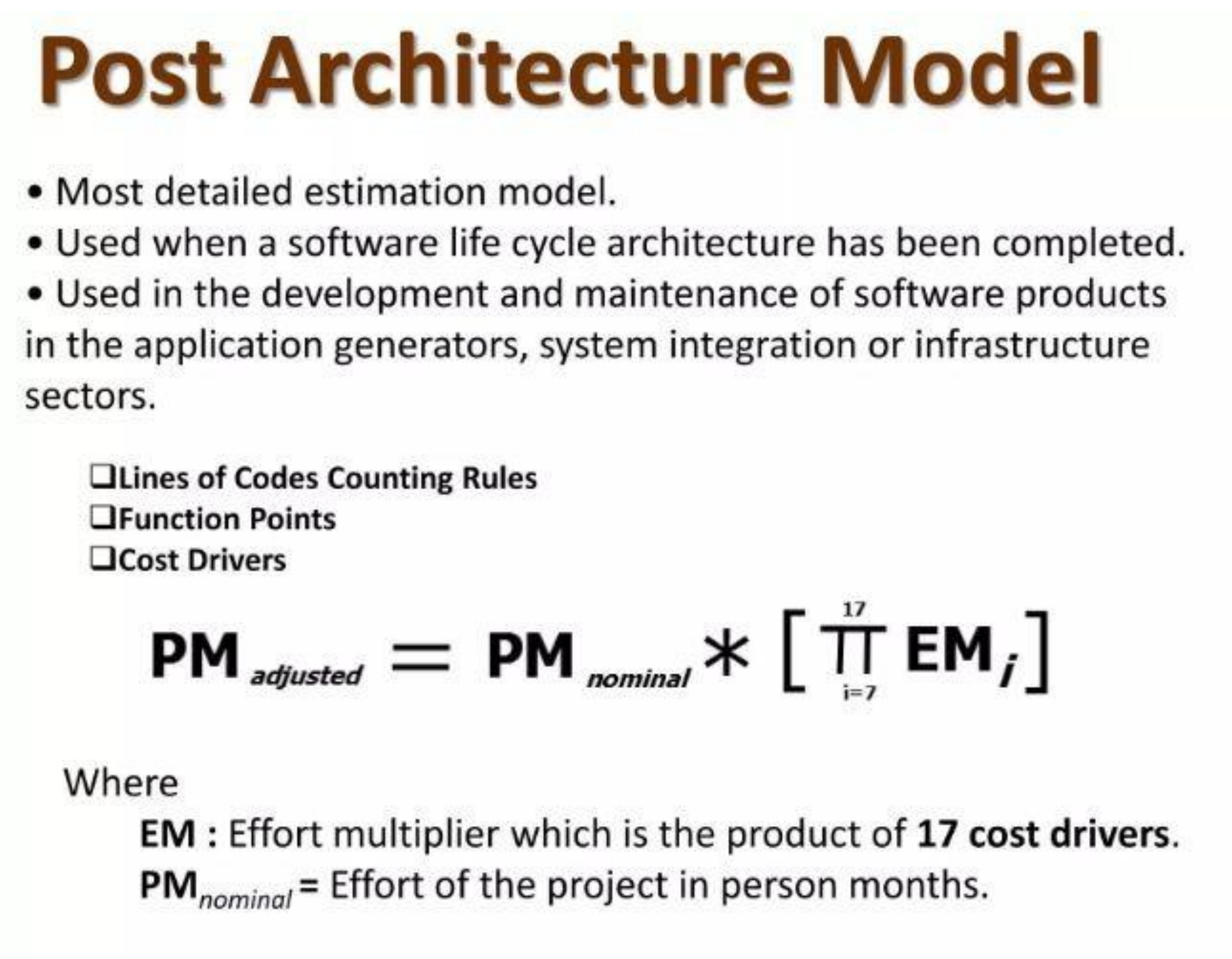
$$\mathbf{PM}_{adjusted} = \mathbf{PM}_{nominal} * \left[\prod_{i=7}^{17} \mathbf{EM}_i\right]$$

= 194.41 * [1.29 x 0.83]

= 194.41 x 1.07

= 208.155 Person months

Program and Software



Post Architecture Model

- Most detailed estimation model.
- Used when a software life cycle architecture has been completed.
- Used in the development and maintenance of software products in the application generators, system integration or infrastructure sectors.

  - ❑Lines of Codes Counting Rules
  - ❑Function Points
  - ❑Cost Drivers

$$PM_{adjusted} = PM_{nominal} * \left[\prod_{i=7}^{17} EM_{i} \right]$$

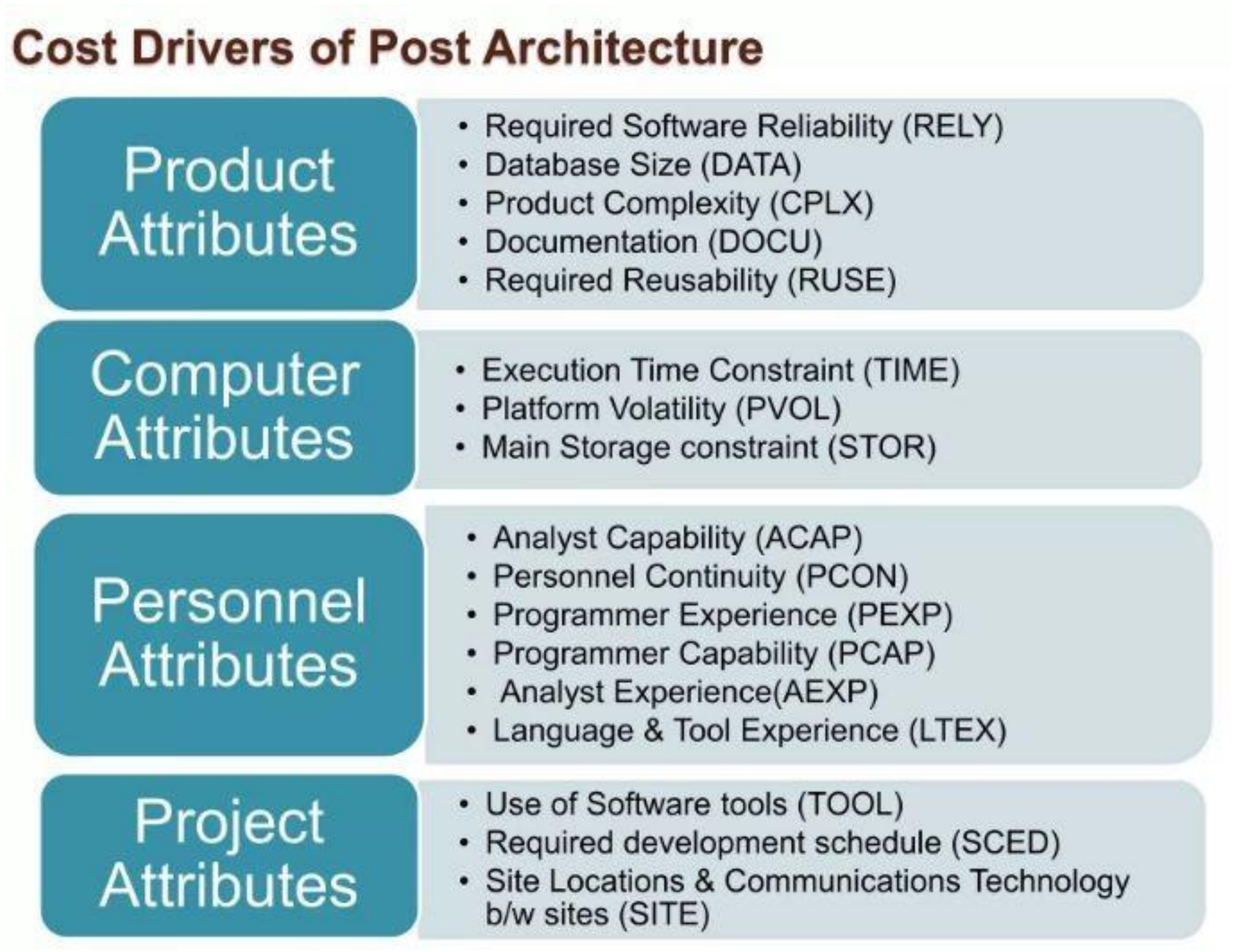
Where

EM : Effort multiplier which is the product of **17 cost drivers**.

PM$_{nominal}$ = Effort of the project in person months.

GL BAJAJ Institute of Technology & Management FIND YOUR SPARK

Program and Software

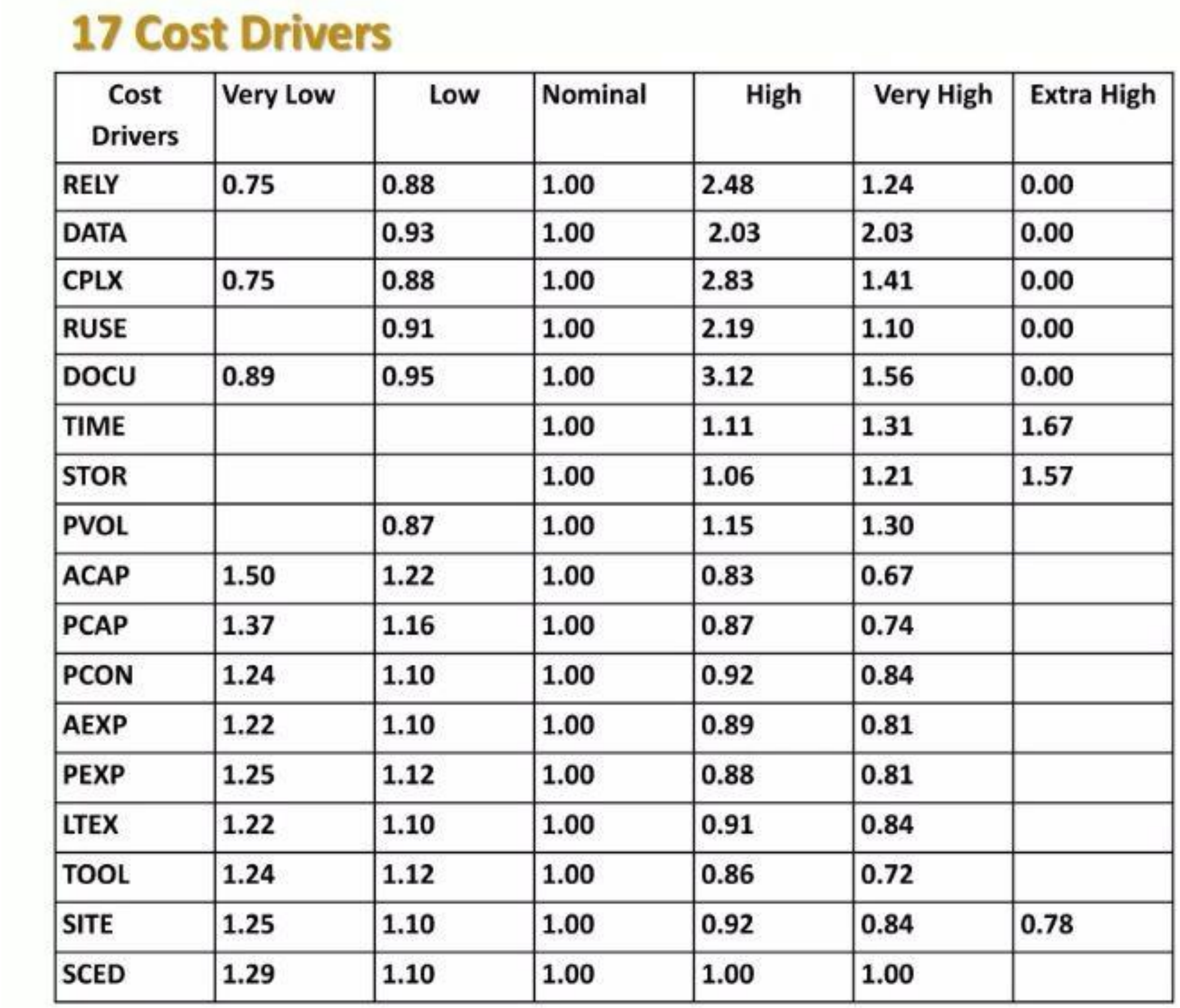


Cost Drivers of Post Architecture

Product Attributes	• Required Software Reliability (RELY)
	• Database Size (DATA)
	• Product Complexity (CPLX)
	• Documentation (DOCU)
	• Required Reusability (RUSE)
Computer Attributes	• Execution Time Constraint (TIME)
	• Platform Volatility (PVOL)
	• Main Storage constraint (STOR)
Personnel Attributes	• Analyst Capability (ACAP)
	• Personnel Continuity (PCON)
	• Programmer Experience (PEXP)
	• Programmer Capability (PCAP)
	• Analyst Experience(AEXP)
	• Language & Tool Experience (LTEX)
Project Attributes	• Use of Software tools (TOOL)
	• Required development schedule (SCED)
	• Site Locations & Communications Technology b/w sites (SITE)

GL BAJAJ Institute of Technology & Management FIND YOUR SPARK

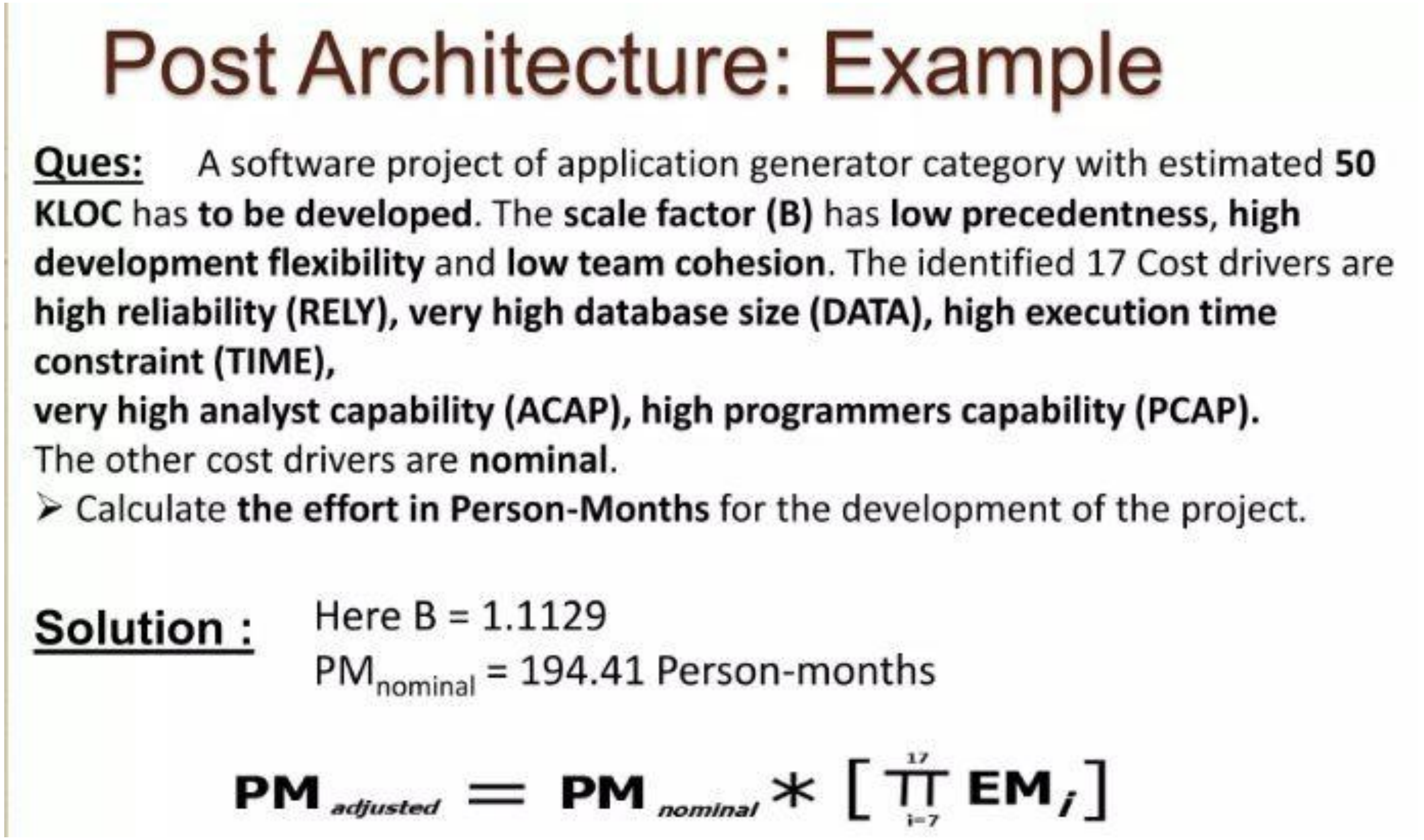
Program and Software



17 Cost Drivers

Cost Drivers	Very Low	Low	Nominal	High	Very High	Extra High
RELY	0.75	0.88	1.00	2.48	1.24	0.00
DATA		0.93	1.00	2.03	2.03	0.00
CPLX	0.75	0.88	1.00	2.83	1.41	0.00
RUSE		0.91	1.00	2.19	1.10	0.00
DOCU	0.89	0.95	1.00	3.12	1.56	0.00
TIME			1.00	1.11	1.31	1.67
STOR			1.00	1.06	1.21	1.57
PVOL		0.87	1.00	1.15	1.30	
ACAP	1.50	1.22	1.00	0.83	0.67	
PCAP	1.37	1.16	1.00	0.87	0.74	
PCON	1.24	1.10	1.00	0.92	0.84	
AEXP	1.22	1.10	1.00	0.89	0.81	
PEXP	1.25	1.12	1.00	0.88	0.81	
LTEX	1.22	1.10	1.00	0.91	0.84	
TOOL	1.24	1.12	1.00	0.86	0.72	
SITE	1.25	1.10	1.00	0.92	0.84	0.78
SCED	1.29	1.10	1.00	1.00	1.00	

Program and Software



Post Architecture: Example

Ques: A software project of application generator category with estimated **50 KLOC** has **to be developed**. The **scale factor (B)** has **low precedentness, high development flexibility** and **low team cohesion**. The identified 17 Cost drivers are **high reliability (RELY), very high database size (DATA), high execution time constraint (TIME),**
very high analyst capability (ACAP), high programmers capability (PCAP).
The other cost drivers are **nominal**.

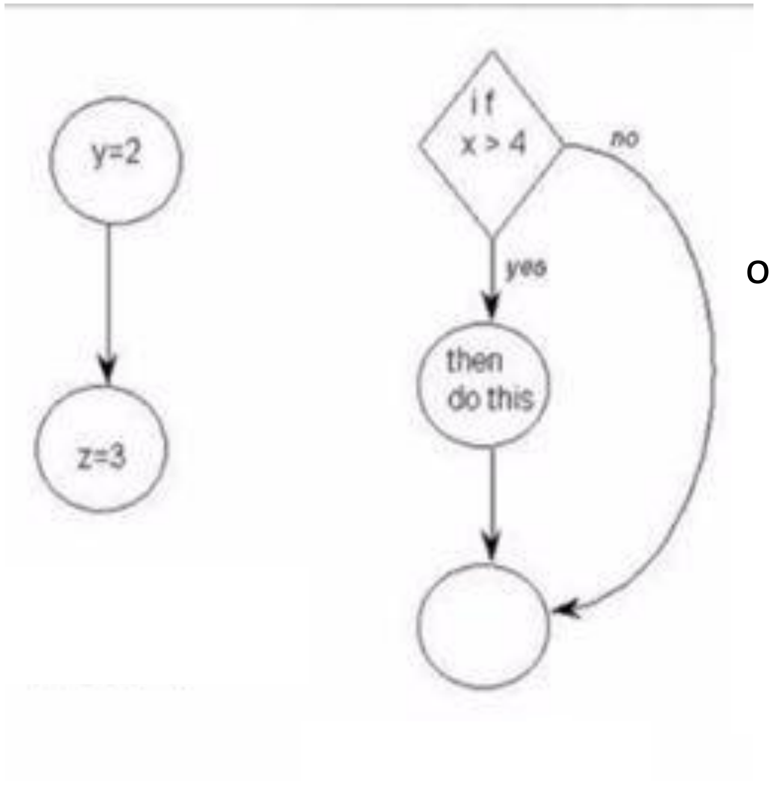
- Calculate **the effort in Person-Months** for the development of the project.

Solution : Here B = 1.1129
$PM_{nominal}$ = 194.41 Person-months

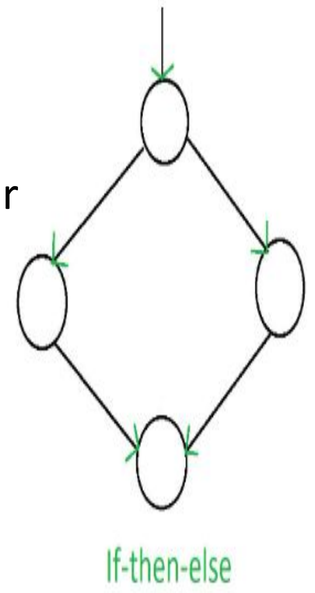
$$PM_{adjusted} = PM_{nominal} * \left[\prod_{i=7}^{17} EM_i\right]$$

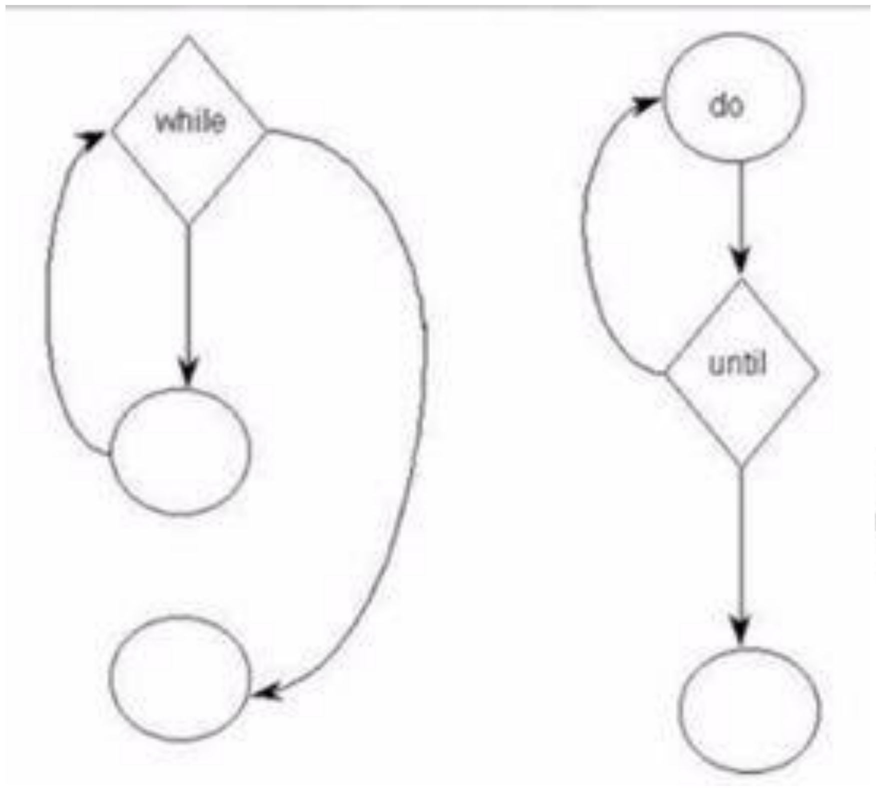
Control flow graph (CFG)

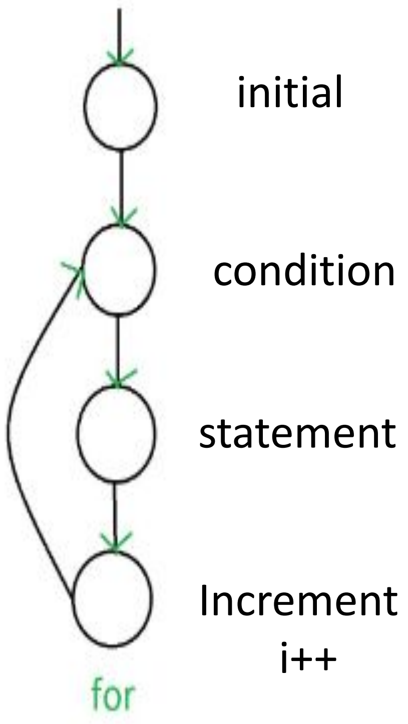
A control flow graph describes the sequence in which the different instructions of a program get executed.



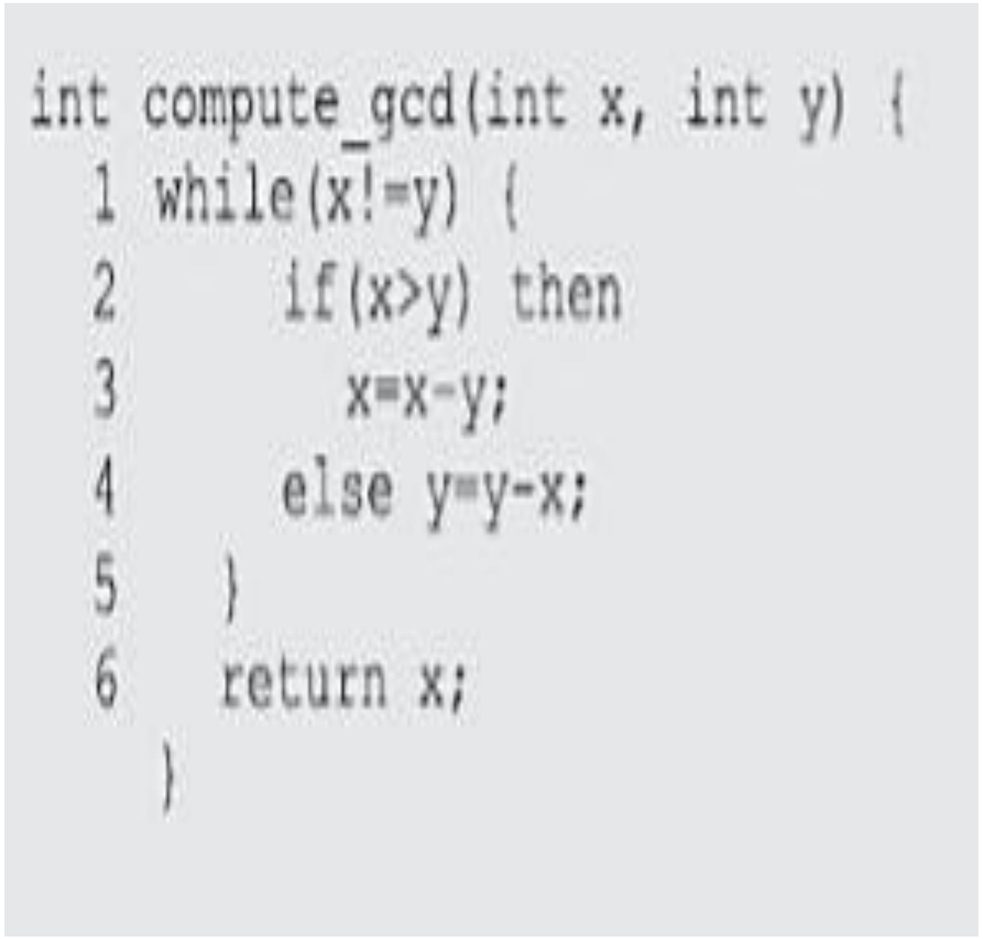
or





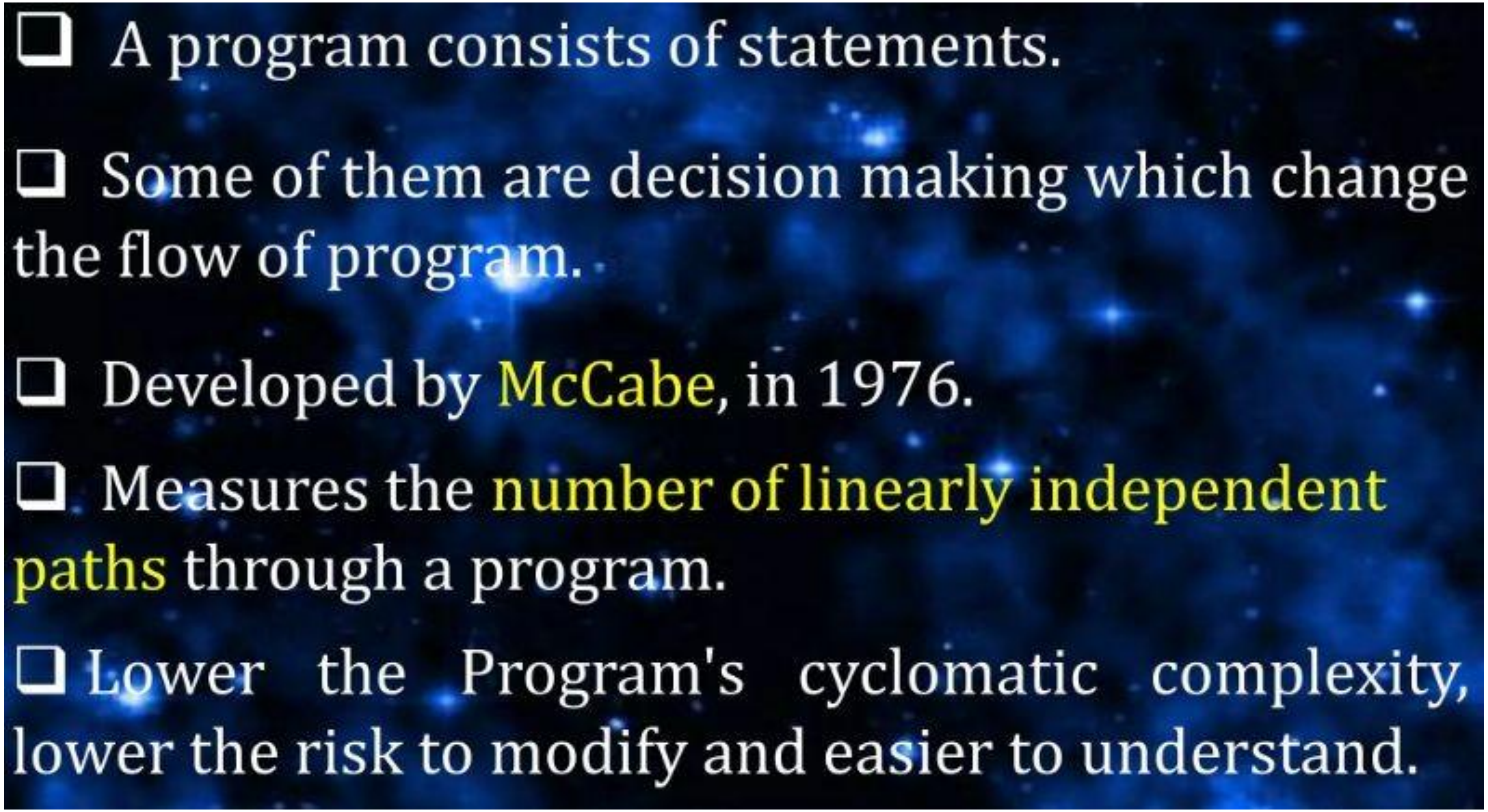


Draw CFG for the given Program



```
int compute_gcd(int x, int y) {
  1 while(x!=y) {
  2       if(x>y) then
  3          x=x-y;
  4       else y=y-x;
  5    }
  6    return x;
    }
```

Cyclometic Complexity..



❑ A program consists of statements.

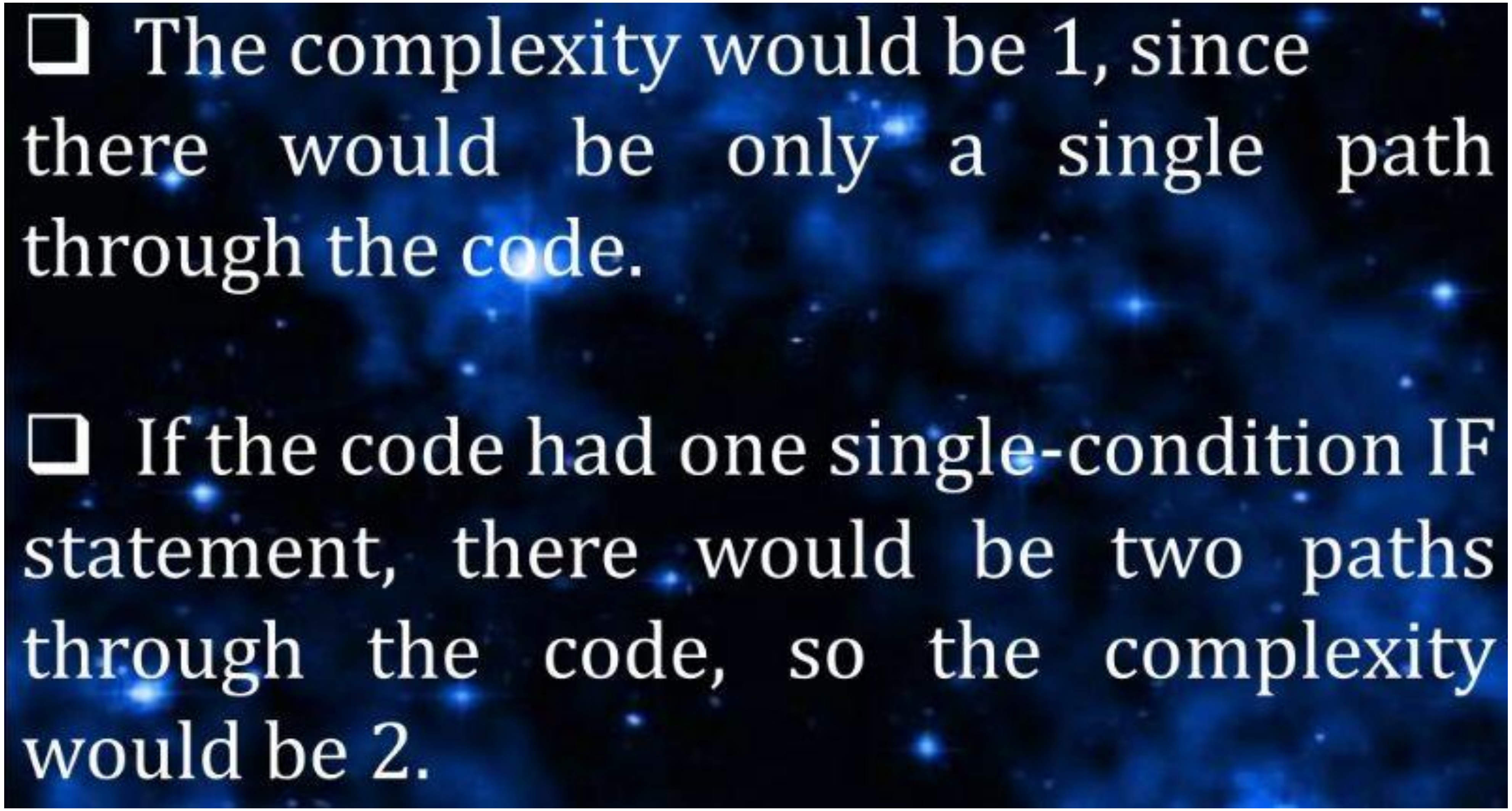
❑ Some of them are decision making which change the flow of program.

❑ Developed by McCabe, in 1976.

❑ Measures the number of linearly independent paths through a program.

❑ Lower the Program's cyclomatic complexity, lower the risk to modify and easier to understand.

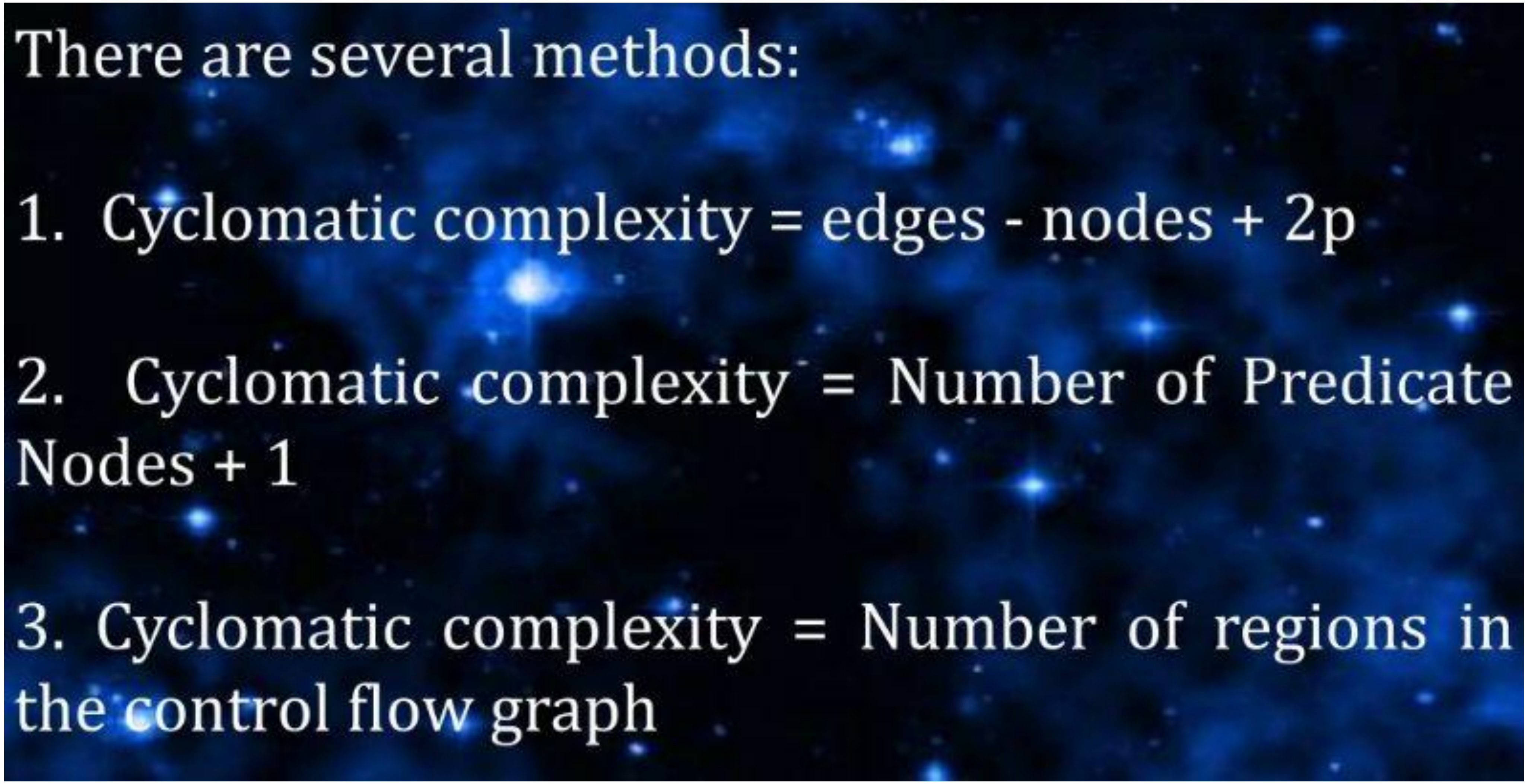
Program and Software



- ❑ The complexity would be 1, since there would be only a single path through the code.

- ❑ If the code had one single-condition IF statement, there would be two paths through the code, so the complexity would be 2.

Program and Software



There are several methods:

1. Cyclomatic complexity = edges - nodes + 2p

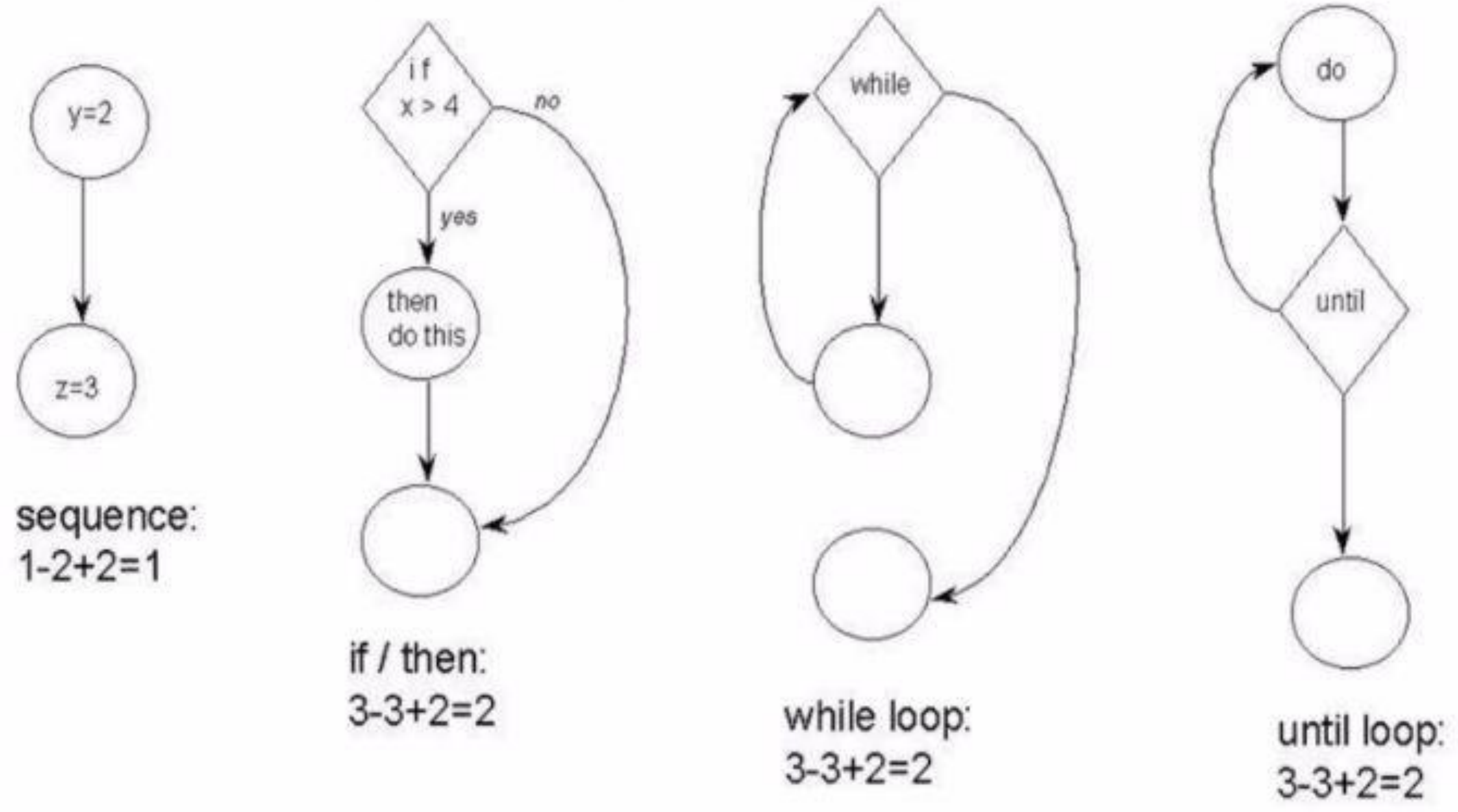
2. Cyclomatic complexity = Number of Predicate Nodes + 1

3. Cyclomatic complexity = Number of regions in the control flow graph

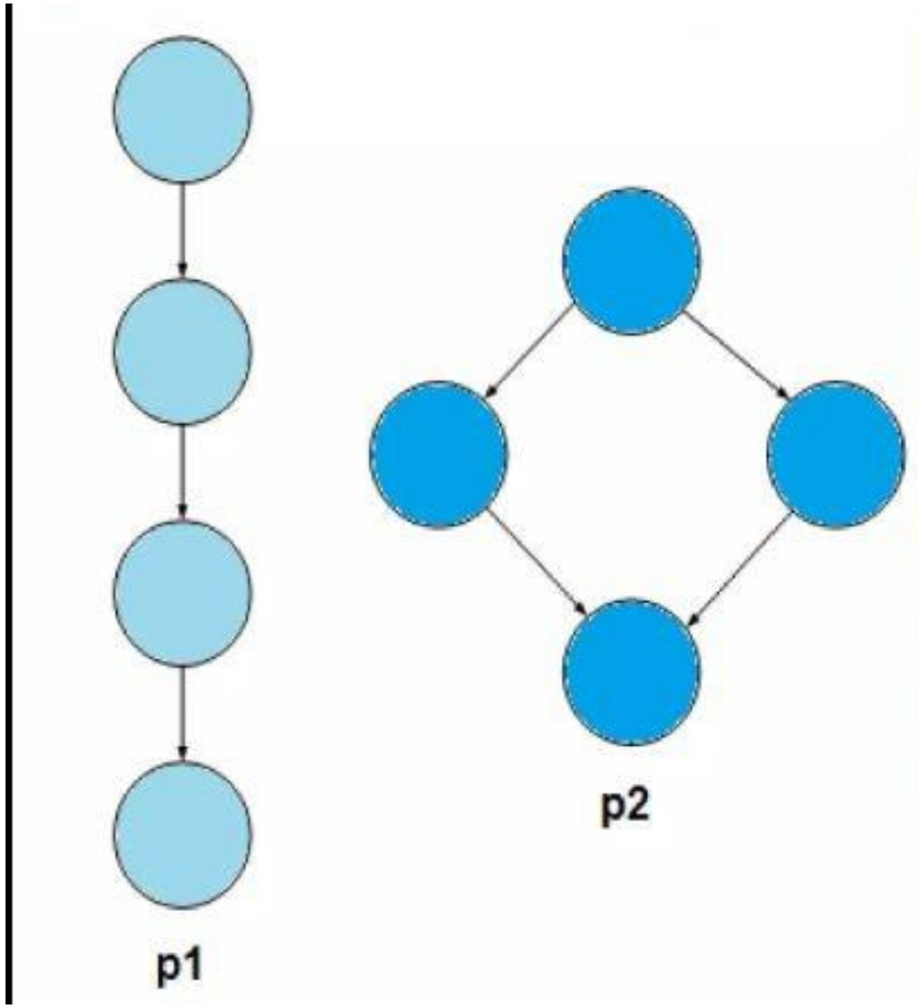
Program and Software

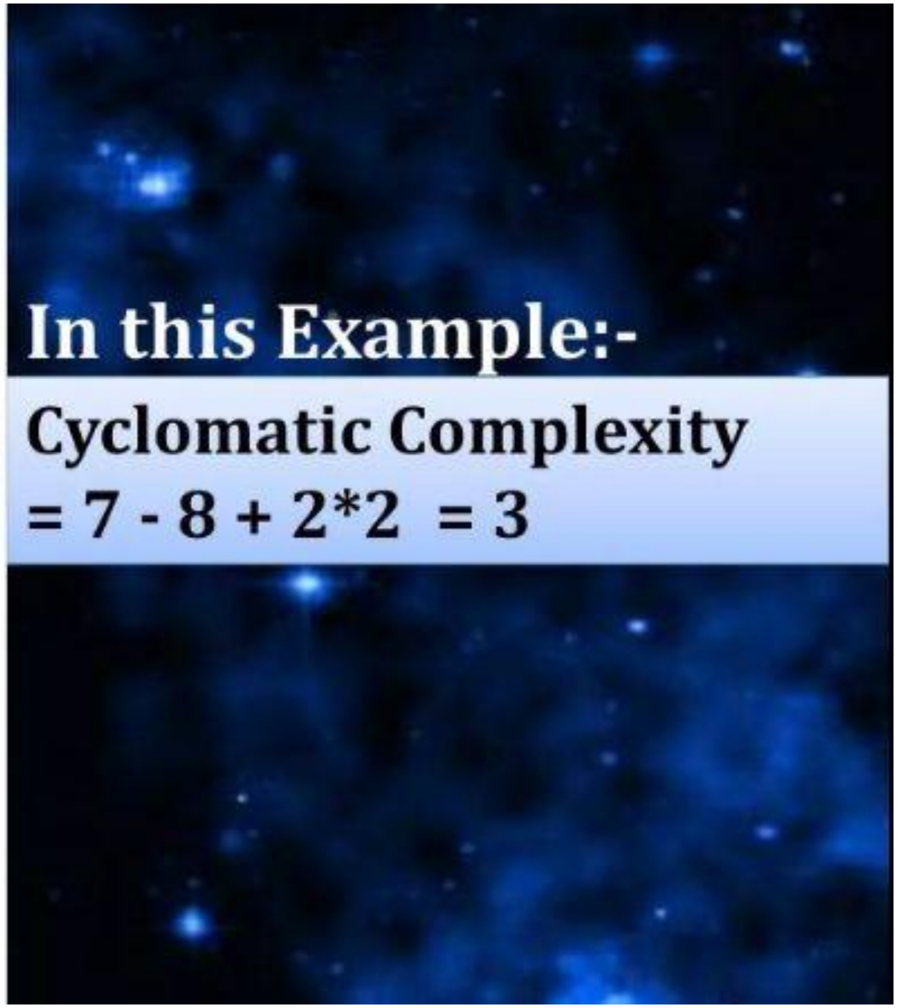
Cyclometic Complexity = Edges –Nodes + 2P

P = Number of unconnected parts of the graph.



Program and Software

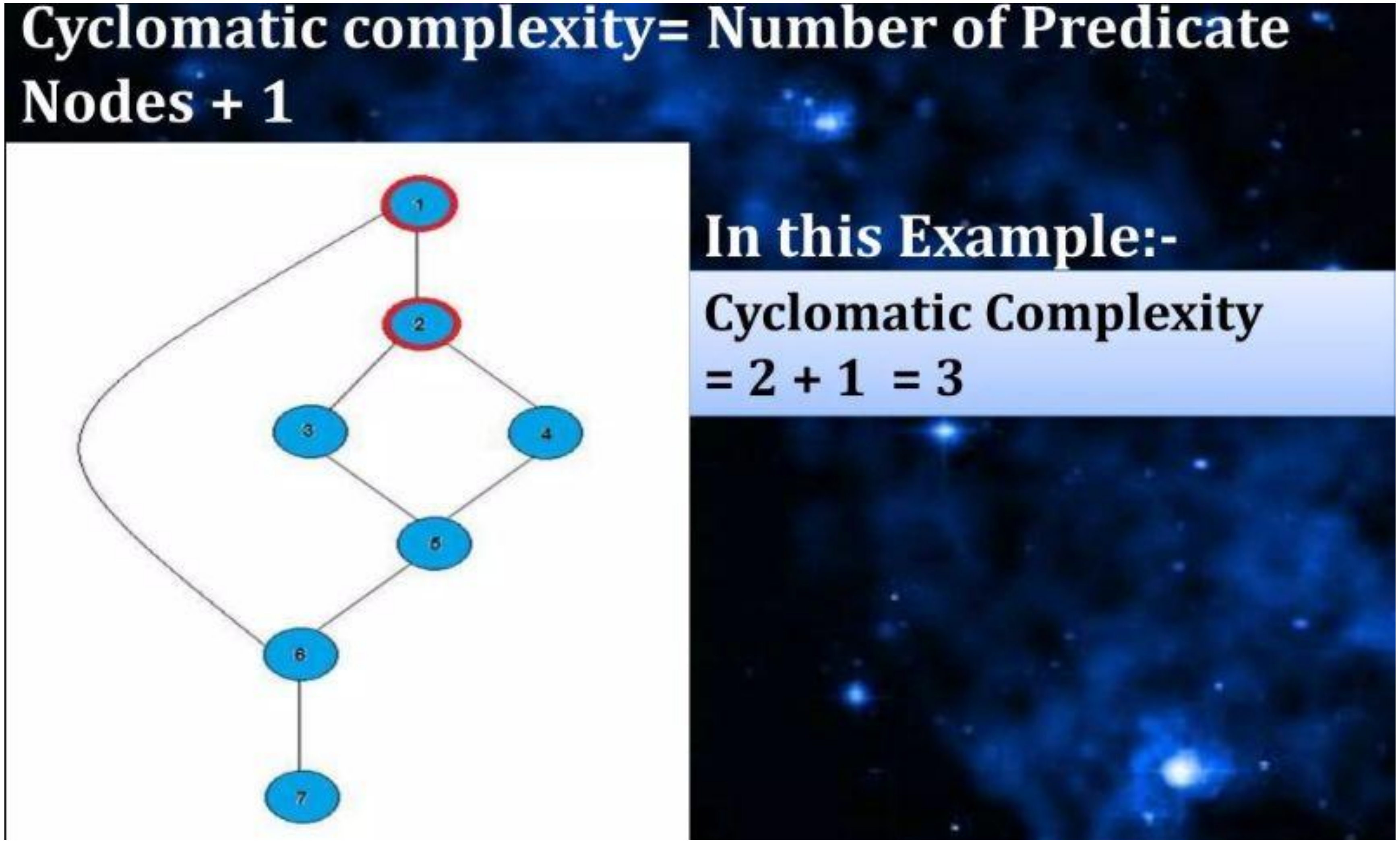



In this Example:-

Cyclomatic Complexity

= 7 - 8 + 2*2 = 3

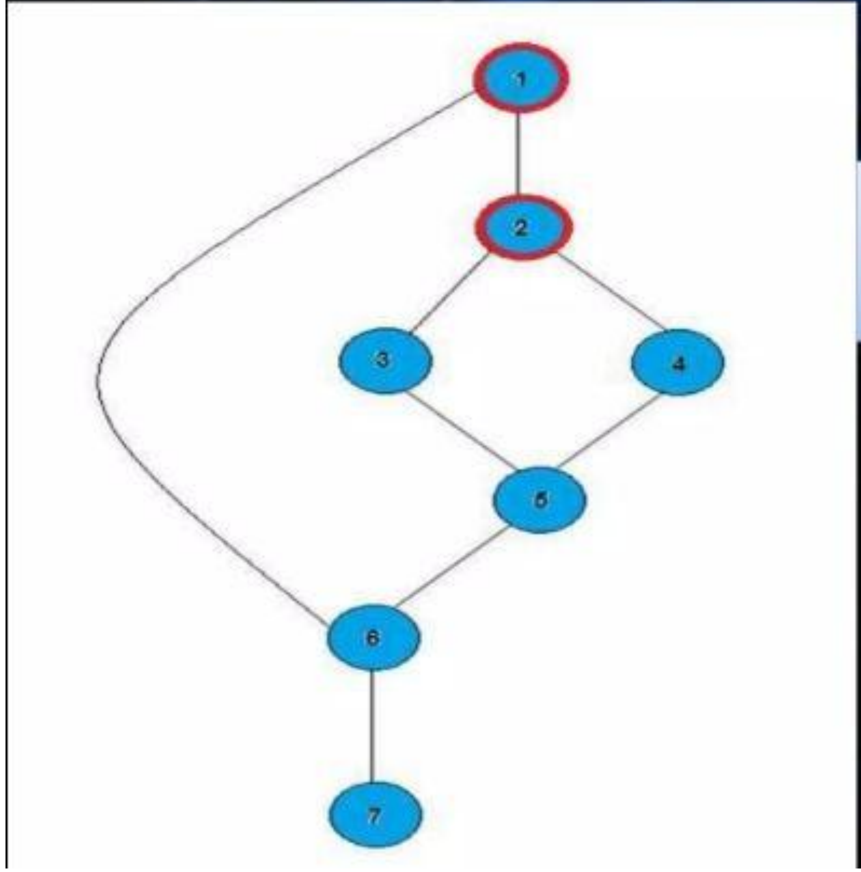
Program and Software



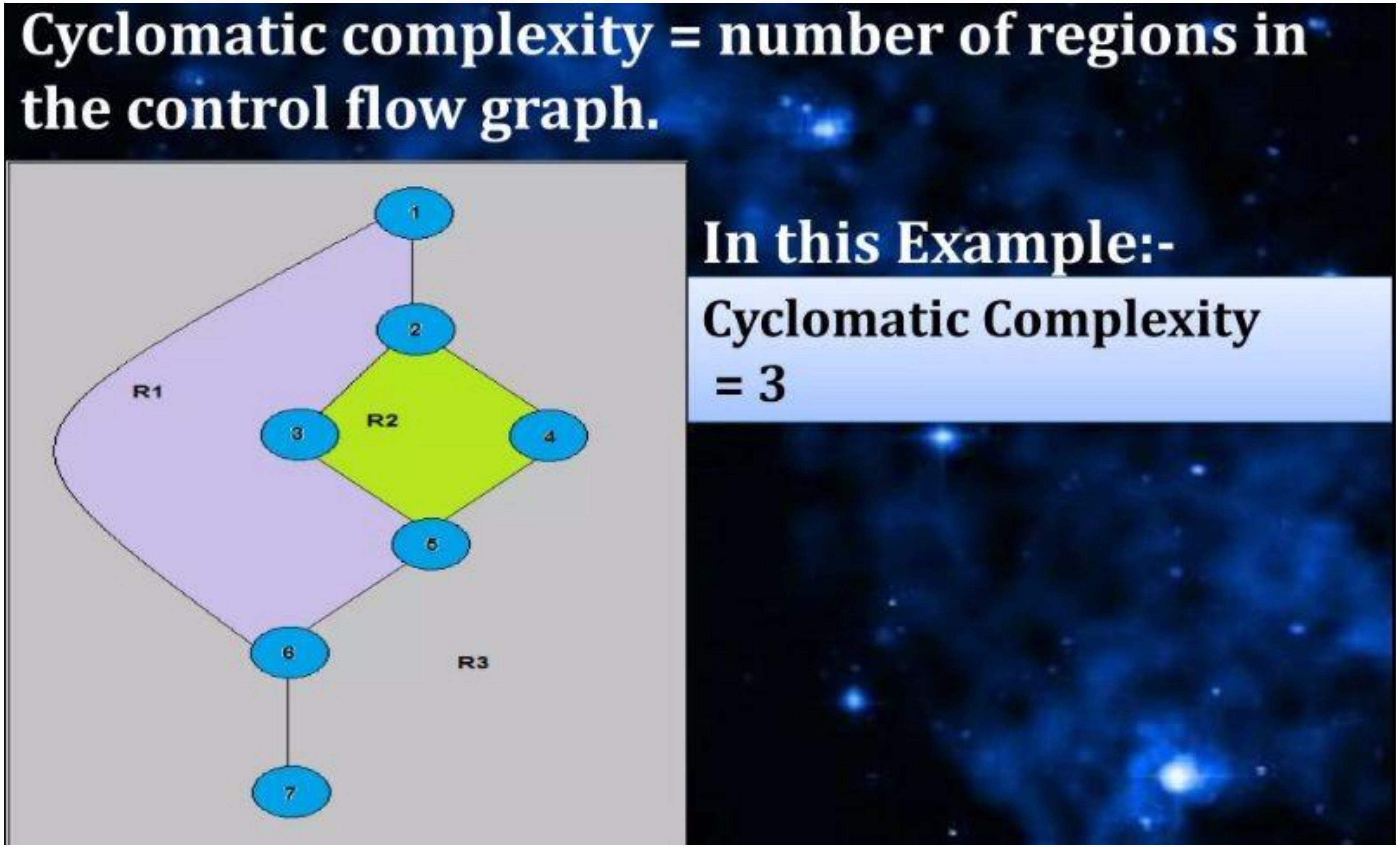
Cyclomatic complexity= Number of Predicate Nodes + 1

In this Example:-

Cyclomatic Complexity
= 2 + 1 = 3



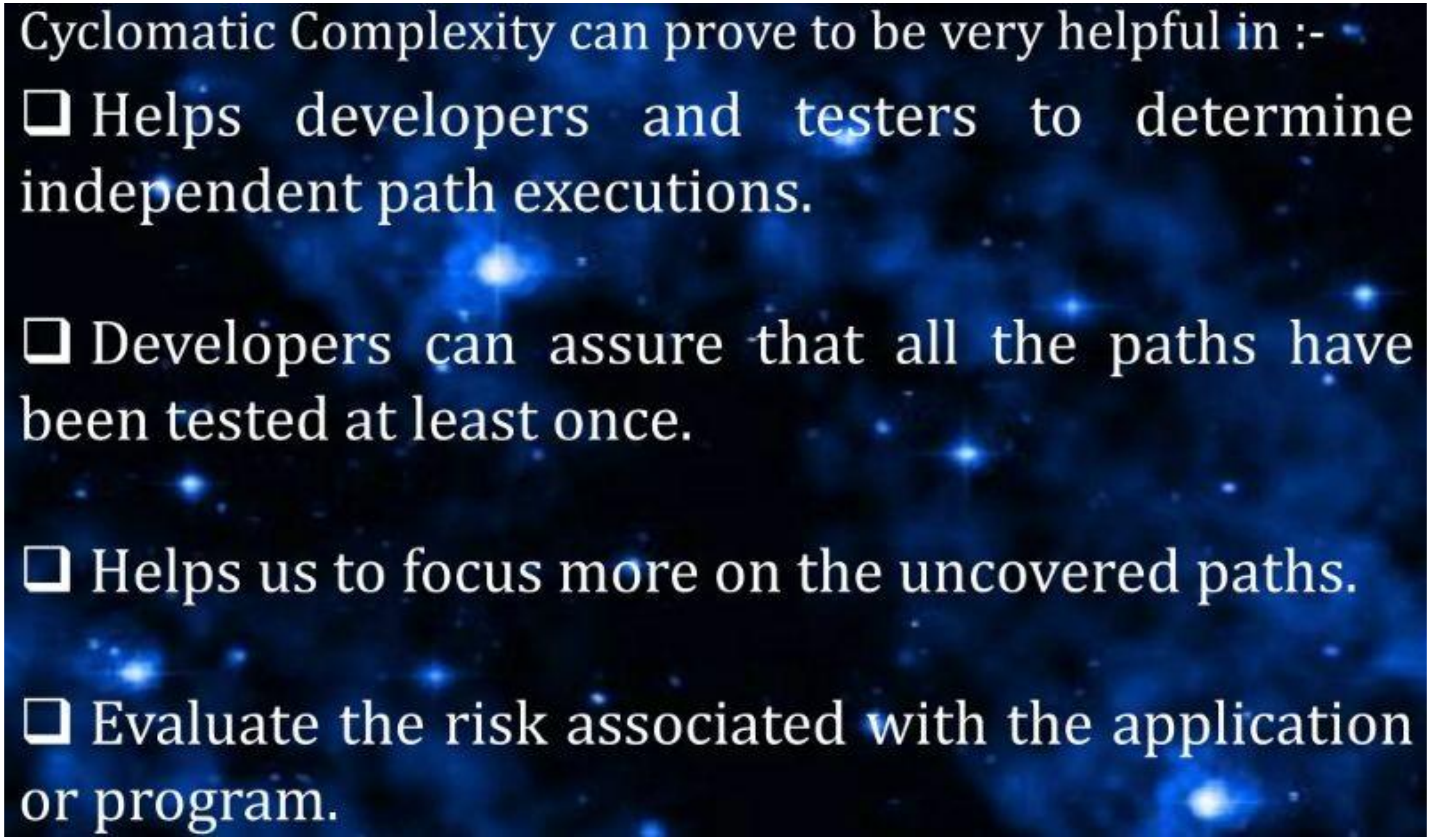
Program and Software


Cyclomatic complexity = number of regions in the control flow graph.

In this Example:-

Cyclomatic Complexity = 3

Program and Software



Cyclomatic Complexity can prove to be very helpful in :-

❑ Helps developers and testers to determine independent path executions.

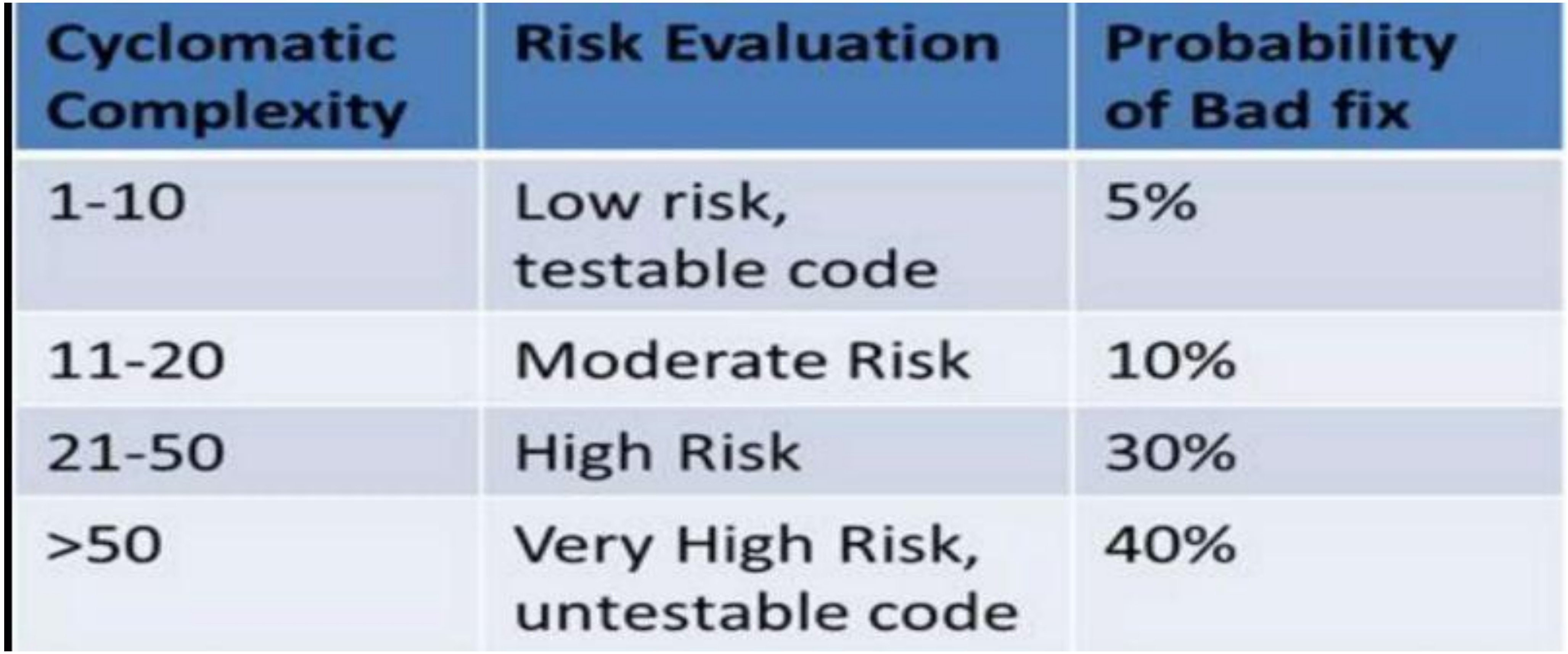
❑ Developers can assure that all the paths have been tested at least once.

❑ Helps us to focus more on the uncovered paths.

❑ Evaluate the risk associated with the application or program.

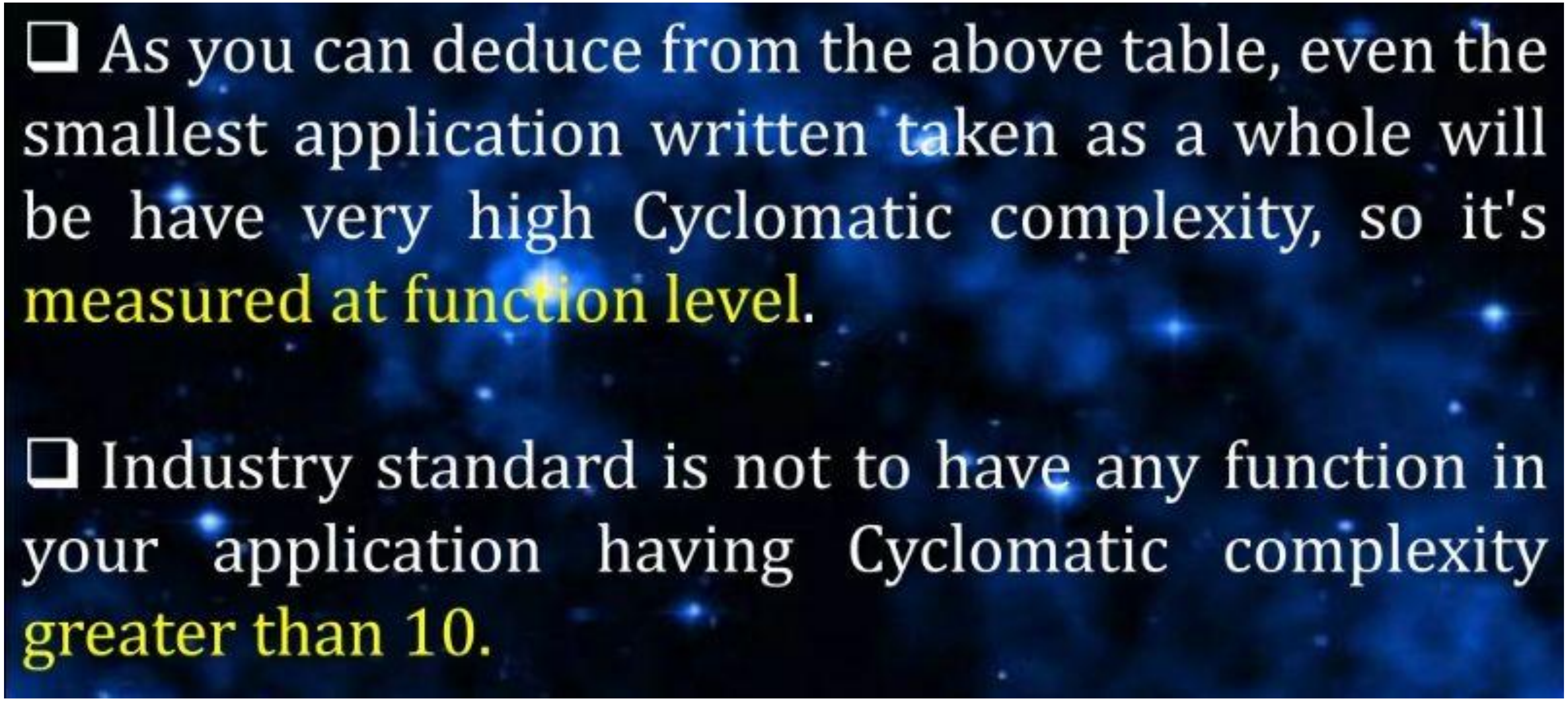
GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Program and Software



Cyclomatic Complexity	Risk Evaluation	Probability of Bad fix
1-10	Low risk, testable code	5%
11-20	Moderate Risk	10%
21-50	High Risk	30%
>50	Very High Risk, untestable code	40%

Program and Software

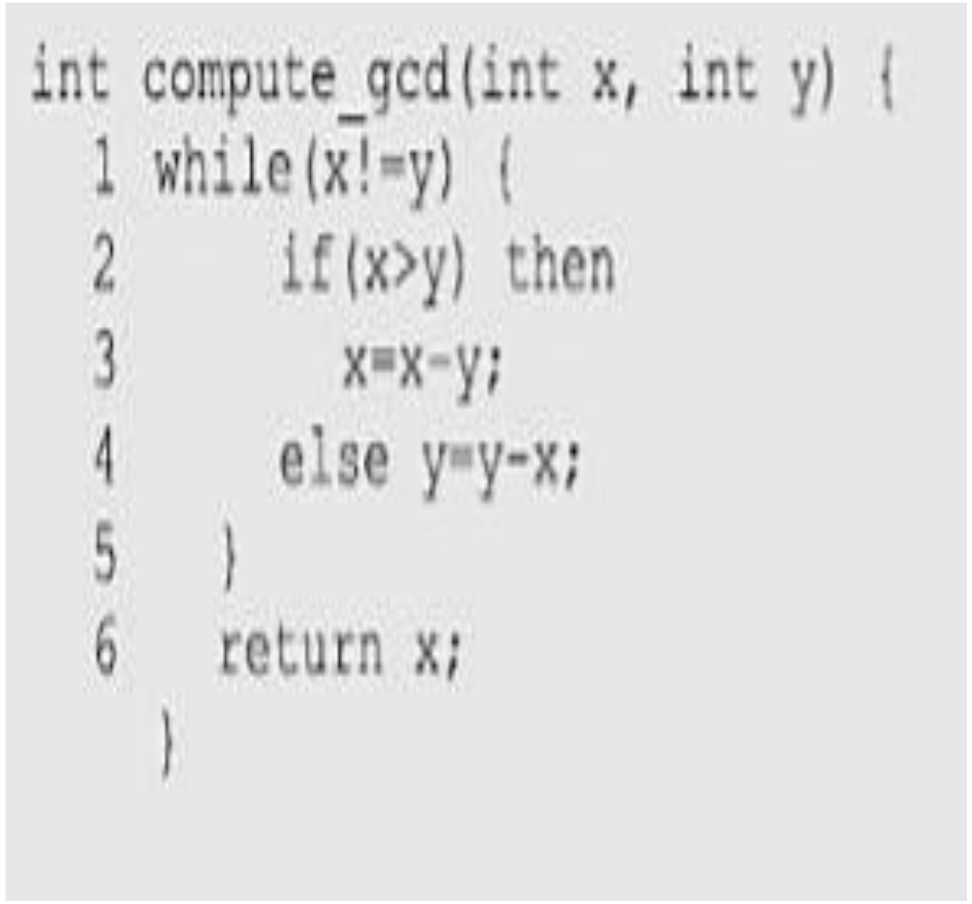


❑ As you can deduce from the above table, even the smallest application written taken as a whole will be have very high Cyclomatic complexity, so it's measured at function level.

❑ Industry standard is not to have any function in your application having Cyclomatic complexity greater than 10.

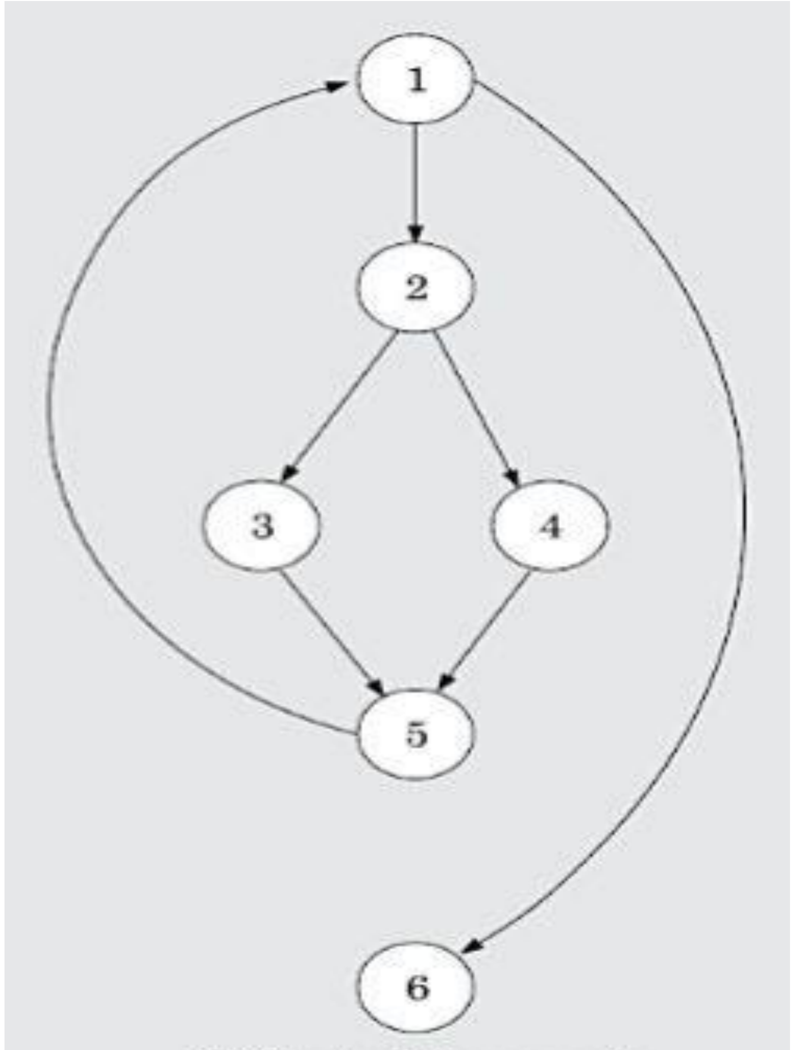
Program and Software

Draw the control flow graph for the following function named find-maximum. From the control flow graph, determine its cyclomatic complexity.



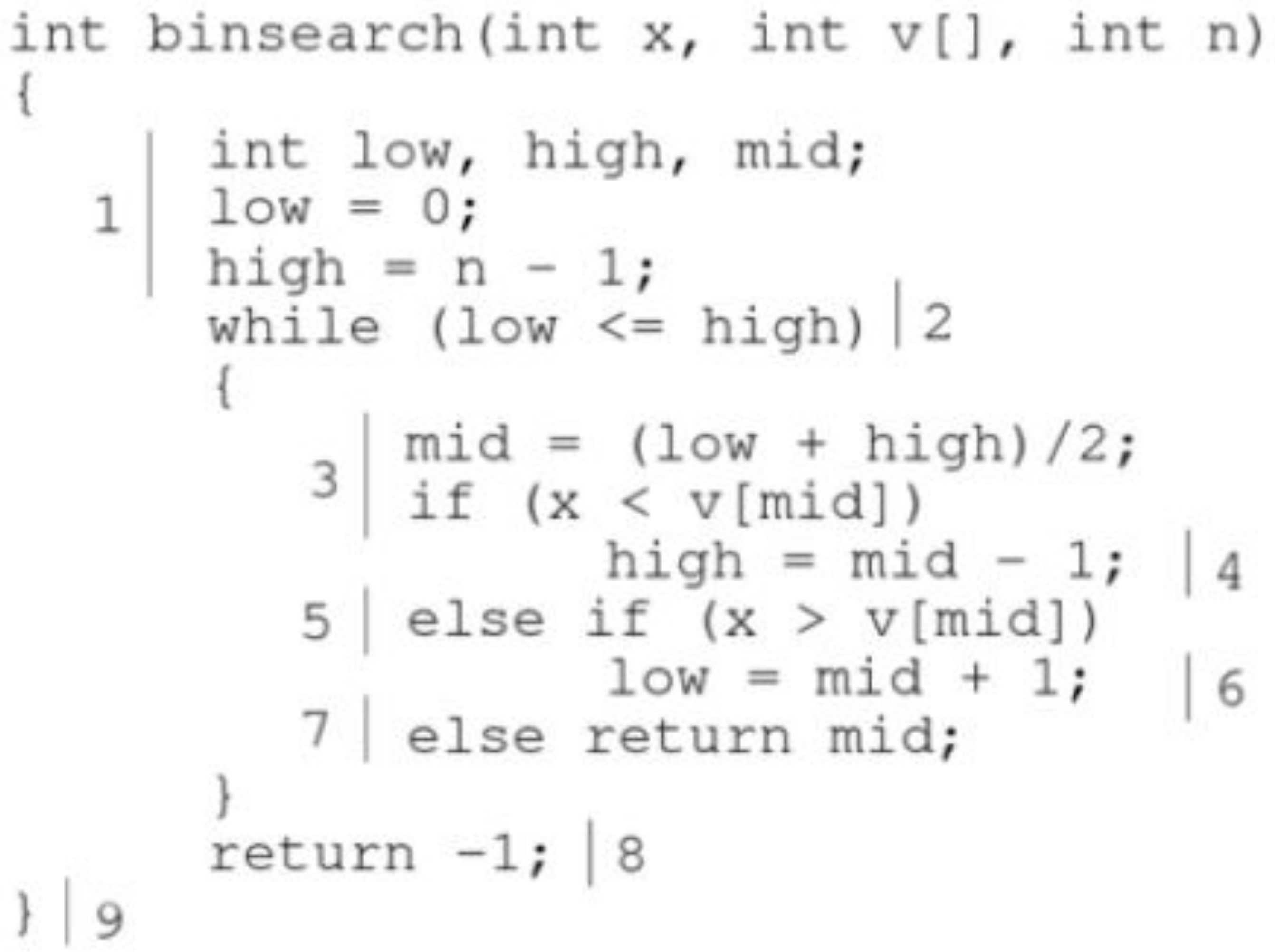
```
int compute_gcd(int x, int y) {
  1 while(x!=y) {
  2      if(x>y) then
  3         x=x-y;
  4      else y=y-x;
  5   }
  6   return x;
    }
```

Program and Software



(b) Control flow graph

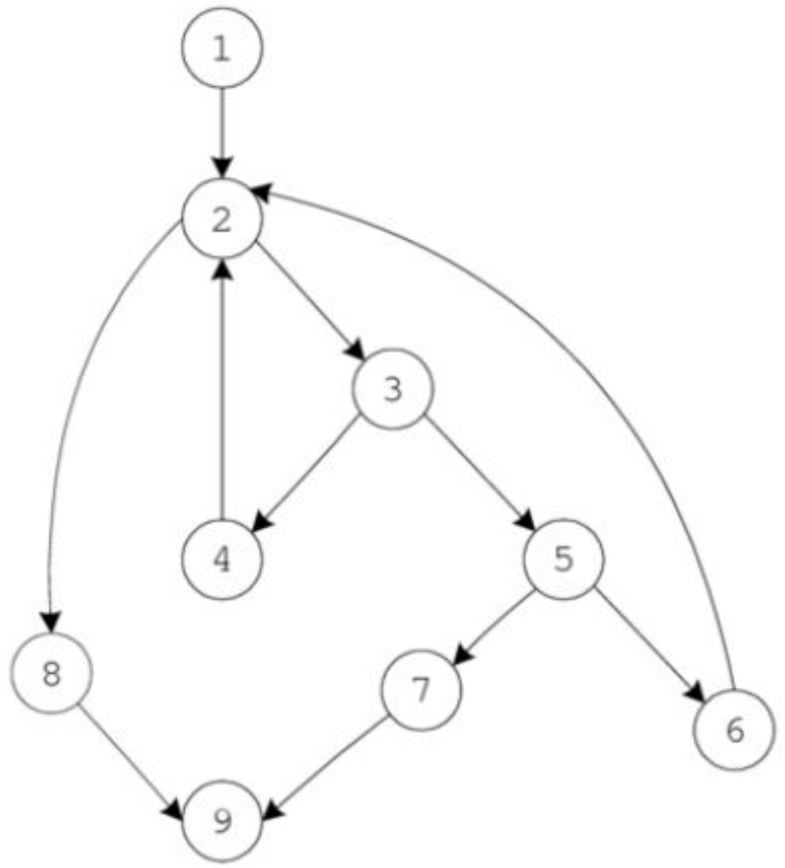
Program and Software



```
int binsearch(int x, int v[], int n)
{
     1 | int low, high, mid;
       | low = 0;
       | high = n - 1;
         while (low <= high) | 2
         {
             3 | mid = (low + high)/2;
               | if (x < v[mid])
                          high = mid - 1;   | 4
             5 | else if (x > v[mid])
                          low = mid + 1;    | 6
             7 | else return mid;
         }
         return -1; | 8
} | 9
```

Program and Software

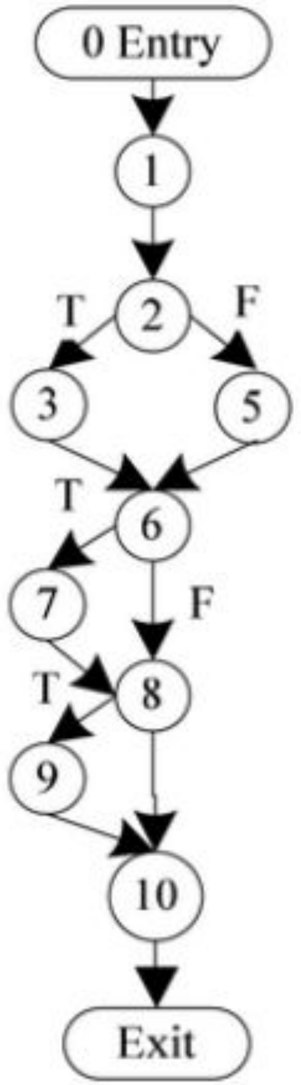
CFG:



Program and Software

```
0  Void maxMin(int a, int b, int c)
   {
1        int max = 0, min = 0;
2        if (a > b) {
3                  max = a; min = b;}
4        else {
5                  max = b; min = a;}
6        if (max< c) {
7                  max = c;}
8        if (min > c) {
9                  min = c;}
10     printf("max=%d\n", max);
       Printf("min=%d\n", min);
   }
```

Program and Software

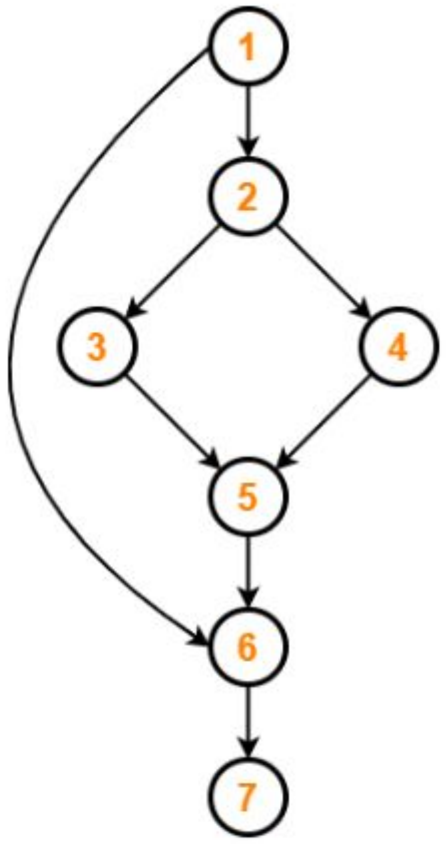


Program and Software

Calculate cyclomatic complexity for the given code-

```
IF A = 354
    THEN IF B > C
              THEN A = B
               ELSE A = C
     END IF
END IF
PRINT A
```

Program and Software



Control Flow Graph

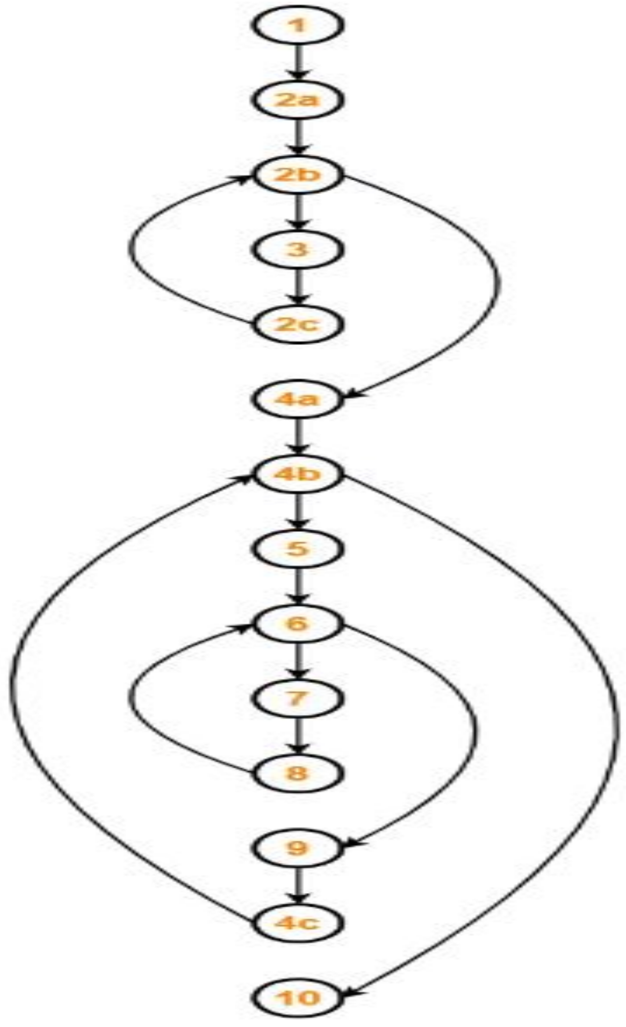
GL BAJAJ
Institute of Technology & Management
FIND YOUR SPARK

Program and Software

Calculate cyclomatic complexity for the given code-

```
 int i, j, k;
for (i=0 ; i<=N ; i++)
p[i] = 1;
for (i=2 ; i<=N ; i++)
{
k = p[i]; j=1;
while (a[p[j-1]] > a[k] {
p[j] = p[j-1];
j--;
}
p[j]=k;
}
```

Program and Software



Control Flow Graph

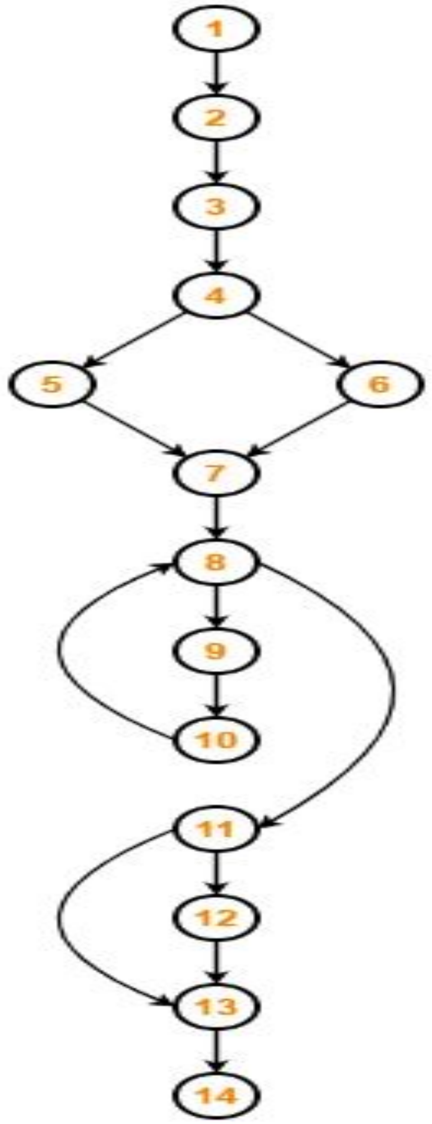
CC=4

Program and Software

Calculate cyclomatic complexity for the given code-

```
begin int x, y, power;
float z;
input(x, y);
if(y<0)
power = -y;
else power = y;
z=1;
while(power!=0)
{ z=z*x;
power=power-1;
} if(y<0)
z=1/z;
output(z);
end
```

Program and Software



CC=4