Software Engineering

Dr. Mritunjay Shall Peelam
Assistant Professor-SG, UPES

UPES
UNIVERSITY OF TOMORROW

Course Introduction and Objectives

Course Code	Course name	L	T	P	C
CSEG2064	Software Engineering	3	0	0	3
Total Units to be Covered: 5	Total Contact Hours: 45				
Prerequisite(s):		Syllabus version: 1.0			

Course Objectives

1. To explore software development methodologies (waterfall, agile, DevOps) and their integration of testing, quality assurance, reliability, and risk management.
2. To comprehend software requirements engineering and develop skills in creating well-structured Software Requirements Specifications (SRS).
3. To acquire understanding of planning a software project, its cost estimation models and to understand the software quality models.

Course Introduction and Objectives

Course Outcomes

CO 1. Understand the fundamental concepts and importance of Software Engineering in modern software development.

CO 2. Learn various software development methodologies, including Agile, Waterfall, and iterative approaches.

CO 3. Explore software design principles and architectural patterns for creating robust and maintainable software systems.

CO 4. Apply project management principles to effectively plan, monitor, and control software projects.

Course Introduction and Objectives

Syllabus

Unit I: Introduction to Software Engineering **7 Lecture Hours**

Definition of Software Engineering, S/W characteristics, applications, Software development life cycle ; Life Cycle Models – Waterfall (classical and iterative), Spiral, Prototyping & RAD Models, Software processes, Process Models – overview Agile Model and Various Agile methodologies - Scrum, XP, Lean, and Kanban. Scope of each model and their comparison in real-world case studies.

Course Introduction and Objectives

Unit II: Requirements Modelling and Design **9 Lecture Hours**

System and software requirements; Requirements Engineering-Crucial steps; types of requirements, Functional and non-functional requirements; Domain requirements; User requirements; Elicitation and analysis of requirements; Requirements documentation – Nature of Software, Software requirements specification, Use case diagrams with guidelines, DFD, SRS Structure, SRS Case study, Design concepts and principles - Abstraction - Refinement - Modularity Cohesion coupling, Architectural design, Detailed Design Transaction Transformation, Refactoring of designs, Object-oriented Design User-Interface Design.

UPES
UNIVERSITY OF TOMORROW

Course Introduction and Objectives

Unit III: Software Reliability **9 Lecture Hours**

Introduction to Software Reliability; Hardware reliability vs. Software reliability; Reliability metrics; Failure and Faults – Prevention, Removal, Tolerance, Forecast; Dependability Concept – Failure Behavior, Characteristics, Maintenance Policy; Reliability and Availability Modeling; Reliability Evaluation Testing methods, Limits, Starvation, Coverage, Filtering; Microscopic Model of Software Risk; Classes of software reliability Models; Statistical reliability models; Reliability growth models; Defining and interpreting reliability metrics; Fault Detection and Prevention; Techniques for detecting and mitigating software faults; Static analysis tools and techniques; Dynamic analysis methods; Software Fault Tolerance; Software Maintenance and Reliability; Reliability Assessment and Evaluation; Methods for assessing and quantifying software reliability; Case Studies and Real-world Applications.

Course Introduction and Objectives

Unit IV: Software Testing, metrics and Quality Assurance **10 Lecture Hours**

Testing types and techniques such as black box, white box, and gray box testing, functional and structural testing; Test-driven development, code coverage, and quality metrics; Testing process, design of Test cases, testing techniques - boundary value analysis - equivalence class testing - decision table testing, cause-effect graphing, path testing, data flow testing, and mutation testing. Unit, integration, system, alpha, and beta testing, debugging techniques; verification and validation techniques, levels of testing, regression testing, quality management activities, product and process quality standards (ISO9000, CMM), metrics understanding (process, product, project metrics), size metrics (LOC, Function Count, Albrecht FPA), product metrics, metrics for software maintenance, cost estimation techniques (static, single variable, multivariable models), cost-benefit evaluation techniques, Testing tools and standards such as Jira and Selenium, test automation frameworks and tools (Selenium, Appium, JUnit), performance testing and load testing, and defect management and root cause analysis.

Course Introduction and Objectives

Unit V: Software Quality and Risk Management **10 Lecture Hours**

McCall quality factors, ISO and CMM Model, Tools and Techniques for Quality Control, Pareto Analysis, Statistical Sampling, Quality Control Charts and the seven Run Rule. Modern Quality Management, Risk Management – importance, types, process and phases, qualitative and quantitative risk analysis, Risk Analysis and Assessment, Risk Strategies, Risk Monitoring and Control, Risk Response and Evaluation. Software Reliability: Reliability Metrics, Reliability Growth Modeling. Use Case: Defect Tracking and Management. Test Automation Tools: Jira, Selenium:, Appium; JUnit.

Text Books and References

Textbooks	1. Roger S. Pressman, "Software Engineering: A practitioner's approach", 7th Edition, McGraw Hill, 2009. 2. Pankaj Jalote, "An integrated approach to Software Engineering", 3rd Edition, Springer/Narosa, 2005.
Reference books	1. James F. Peters, and Witold Pedrycz, "Software Engineering: an Engineering approach", John Wiley, 2007. 2. Waman S Jawadekar, "Software Engineering principles and practice", McGraw Hill, 2004.
Web Resources **Journals**	
MOOCs, online courses	

Modes of Evaluation

Modes of Evaluation: Quiz/Assignment/ presentation/ extempore/ Written Examination etc.

Examination Scheme

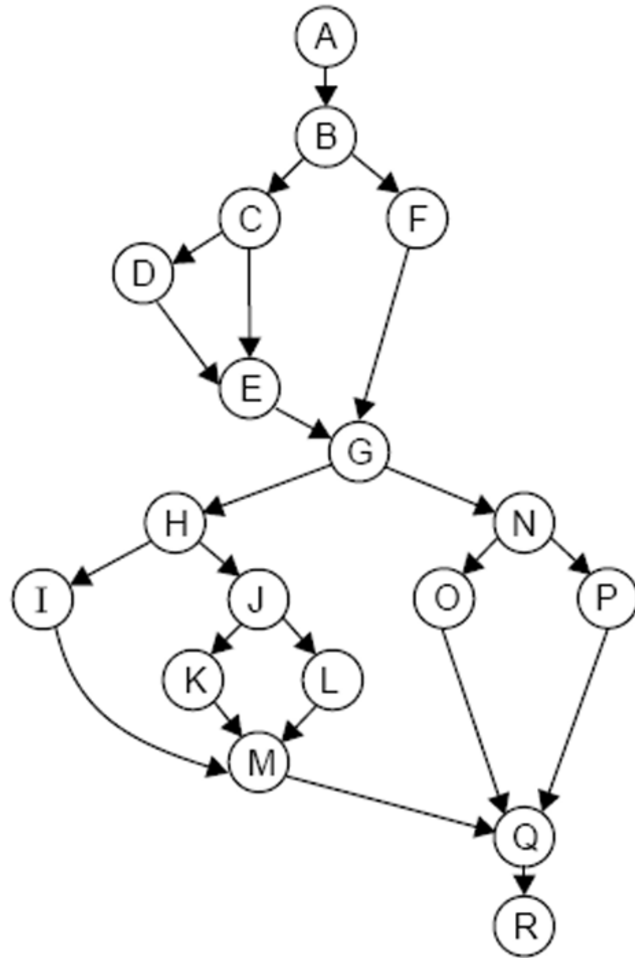
Components	IA	MID SEM	End Sem	Total
Weightage (%)	50	20	30	100

Unit-4

Unit IV: Software Testing, metrics and Quality Assurance **10 Lecture Hours**

Testing types and techniques such as black box, white box, and gray box testing, functional and structural testing; Test-driven development, code coverage, and quality metrics; Testing process, design of Test cases, testing techniques - boundary value analysis - equivalence class testing - decision table testing, cause-effect graphing, path testing, data flow testing, and mutation testing. Unit, integration, system, alpha, and beta testing, debugging techniques; verification and validation techniques, levels of testing, regression testing, quality management activities, product and process quality standards (ISO9000, CMM), metrics understanding (process, product, project metrics), size metrics (LOC, Function Count, Albrecht FPA), product metrics, metrics for software maintenance, cost estimation techniques (static, single variable, multivariable models), cost-benefit evaluation techniques, Testing tools and standards such as Jira and Selenium, test automation frameworks and tools (Selenium, Appium, JUnit), performance testing and load testing, and defect management and root cause analysis.

DD Path graph is given in Fig. 20 (b)



Independent paths are:

(i) ABFGNPQR
(ii) ABFGNOQR
(iii) ABCEGNPQR
(iv) ABCDEGNOQR
(v) ABFGHIMQR
(vi) ABFGHJKMQR
(vii) ABFGHJMQR

Fig. 20 (b): DD Path graph

Cyclomatic Complexity

McCabe's cyclomatic metric V(G) = $e - n + 2P$.

For example, a flow graph shown in in Fig. 21 with entry node '*a*' and exit node '*f*'.

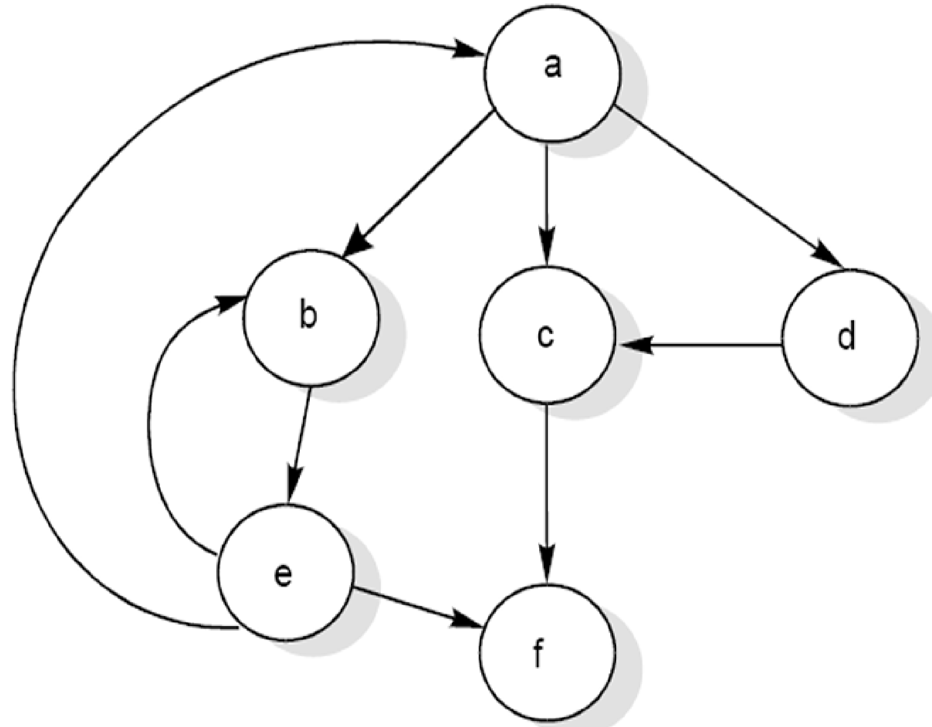


Fig. 21: Flow graph

The value of cyclomatic complexity can be calculated as :

$$V(G) = 9 - 6 + 2 = 5$$

Here $e = 9$, $n = 6$ and $P = 1$

There will be five independent paths for the flow graph illustrated in Fig. 21.

Path 1 : *a c f*

Path 2 : *a b e f*

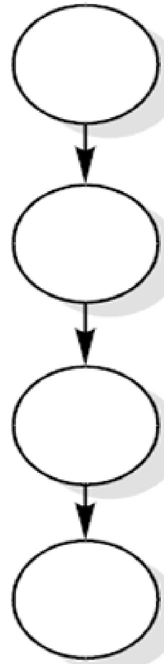
Path 3 : *a d c f*

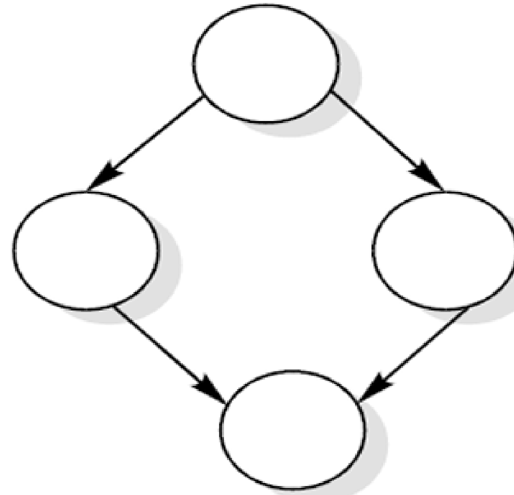
Path 4 : *a b e a c f* **or** *a b e a b e f*

Path 5 : *a b e b e f*

Several properties of cyclomatic complexity are stated below:

1. $V(G) \geq 1$
2. V (G) is the maximum number of independent paths in graph G.
3. Inserting & deleting functional statements to G does not affect V(G).
4. G has only one path if and only if V(G)=1.
5. Inserting a new row in G increases V(G) by unity.
6. V(G) depends only on the decision structure of G.





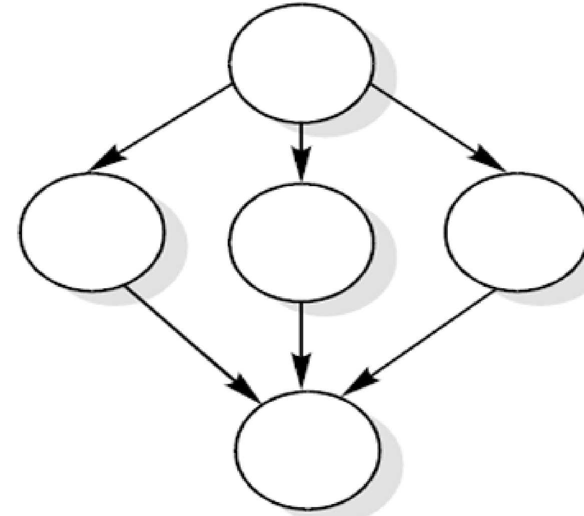


Fig. 22

Let us denote the total graph above with 3 connected components as

$$V(M \cup A \cup B) = e - n + 2P$$

$$= 13-13+2*3$$

$$= 6$$

This method with P ≠ 1 can be used to calculate the complexity of a collection of programs, particularly a hierarchical nest of subroutines.

UPES
UNIVERSITY OF TOMORROW

Two alternate methods are available for the complexity calculations.

1. Cyclomatic complexity V(G) of a flow graph G is equal to the number of predicate (decision) nodes plus one.

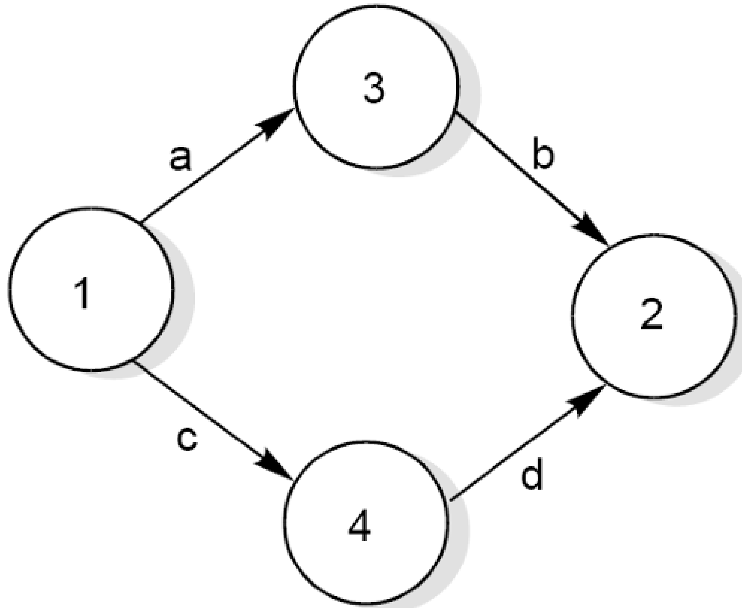
   $V(G) = \prod + 1$

   Where $\prod$ is the number of predicate nodes contained in the flow graph G.

2. Cyclomatic complexity is equal to the number of regions of the flow graph.

Graph Matrices

A graph matrix is a square matrix with one row and one column for every node in the graph. The size of the matrix (i.e., the number of rows and columns) is equal to the number of nodes in the flow graph. Some examples of graphs and associated matrices are shown in fig. 24.



Flow graph

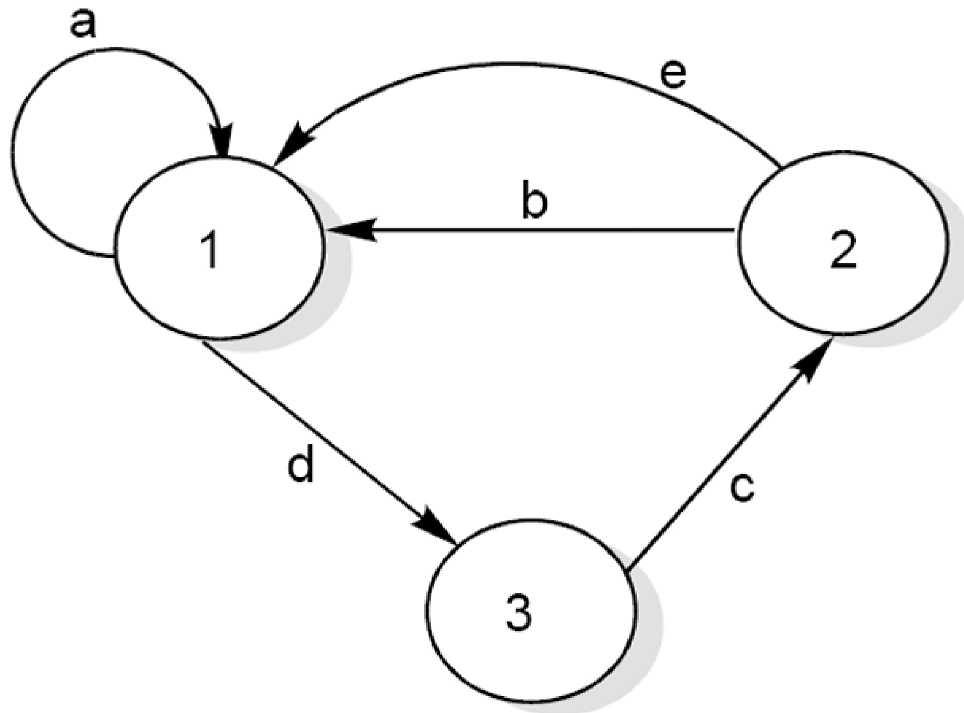
(a)

	1	2	3	4
1			a	c
2				
3		b		
4		d		

Graph Matrix

Fig. 24 (a): Flow graph and graph matrices

(Contd.)...



Flow graph

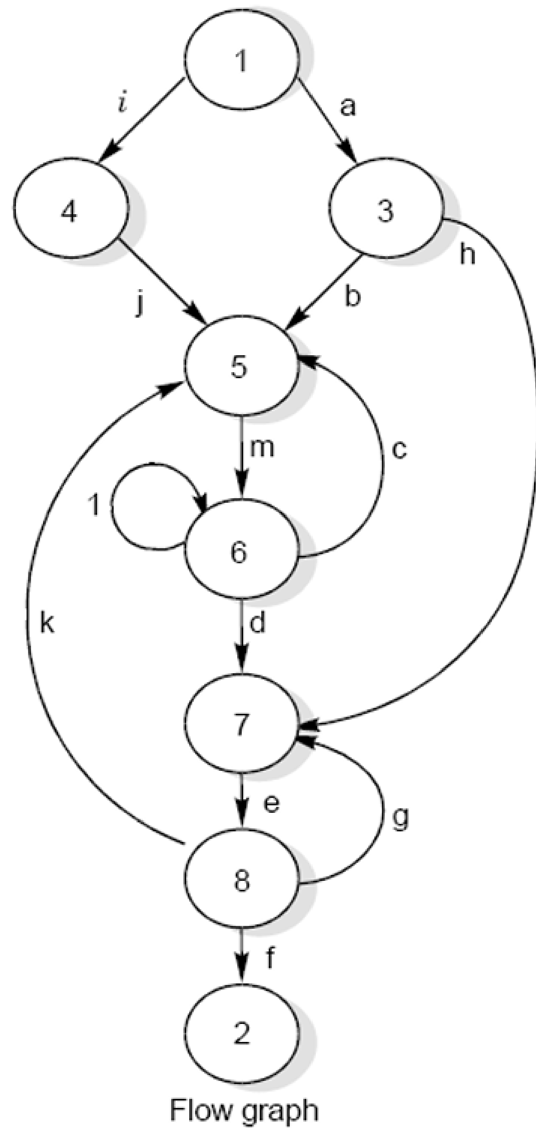
(b)

	1	2	3
1	a		d
2	b + e		
3		c	

Graph Matrix

Fig. 24 (b): Flow graph and graph matrices

(Contd.)...



	1	2	3	4	5	6	7	8
1			a	i				
2								
3					b		h	
4					j			
5						m		
6					c	l	d	
7								e
8		f			k		g	

Graph Matrix

(c)

Fig. 24 (c): Flow graph and graph matrices

UPES
UNIVERSITY OF TOMORROW

	1	2	3	4	5	6	7	8	Connections
1			1	1					2 – 1 = 1
2									
3					1		1		2 – 1 = 1
4					1				1 – 1 = 0
5						1			1 – 1 = 0
6					1	1	1		3 – 1 = 2
7								1	1 – 1 = 0
8		1			1		1		3 – 1 = 2
									6 + 1 = 7

Fig. 25 : Connection matrix of flow graph shown in Fig. 24 (c)

UPES
UNIVERSITY OF TOMORROW

Data Flow Testing

Data flow testing is another from of structural testing. It has nothing to do with data flow diagrams.

i. Statements where variables receive values.

ii. Statements where these values are used or referenced.

As we know, variables are defined and referenced throughout the program. We may have few define/ reference anomalies:

i. A variable is defined but not used/ referenced.

ii. A variable is used but never defined.

iii. A variable is defined twice before it is used.

Definitions

The definitions refer to a program P that has a program graph G(P) and a set of program variables V. The G(P) has a single entry node and a single exit node. The set of all paths in P is PATHS(P)

(i) **Defining Node:** Node n ϵ G(P) is a defining node of the variable *v* ϵ V, written as DEF (*v*, *n*), if the value of the variable *v* is defined at the statement fragment corresponding to node *n*.

(ii) **Usage Node:** Node n ϵ G(P) is a usage node of the variable *v* ϵ V, written as USE (*v*, *n*), if the value of the variable *v* is used at statement fragment corresponding to node *n*. A usage node USE (*v*, n) is a predicate use (denote as p) if statement n is a predicate statement otherwise USE (*v*, n) is a computation use (denoted as *c*).

(iii) Definition use: A definition use path with respect to a variable v (denoted du-path) is a path in PATHS(P) such that, for some *v* ϵ V, there are define and usage nodes DEF(v, m) and USE(v, n) such that m and n are initial and final nodes of the path.

(iv) Definition clear : A definition clear path with respect to a variable v (denoted dc-path) is a definition use path in PATHS(P) with initial and final nodes DEF (*v*, m) and USE (*v*, n), such that no other node in the path is a defining node of *v*.

The du-paths and dc-paths describe the flow of data across source statements from points at which the values are defined to points at which the values are used. The du-paths that are not definition clear are potential trouble spots.

Hence, our objective is to find all du-paths and then identity those du-paths which are not dc-paths. The steps are given in Fig. 27. We may like to generate specific test cases for du-paths that are not dc-paths.

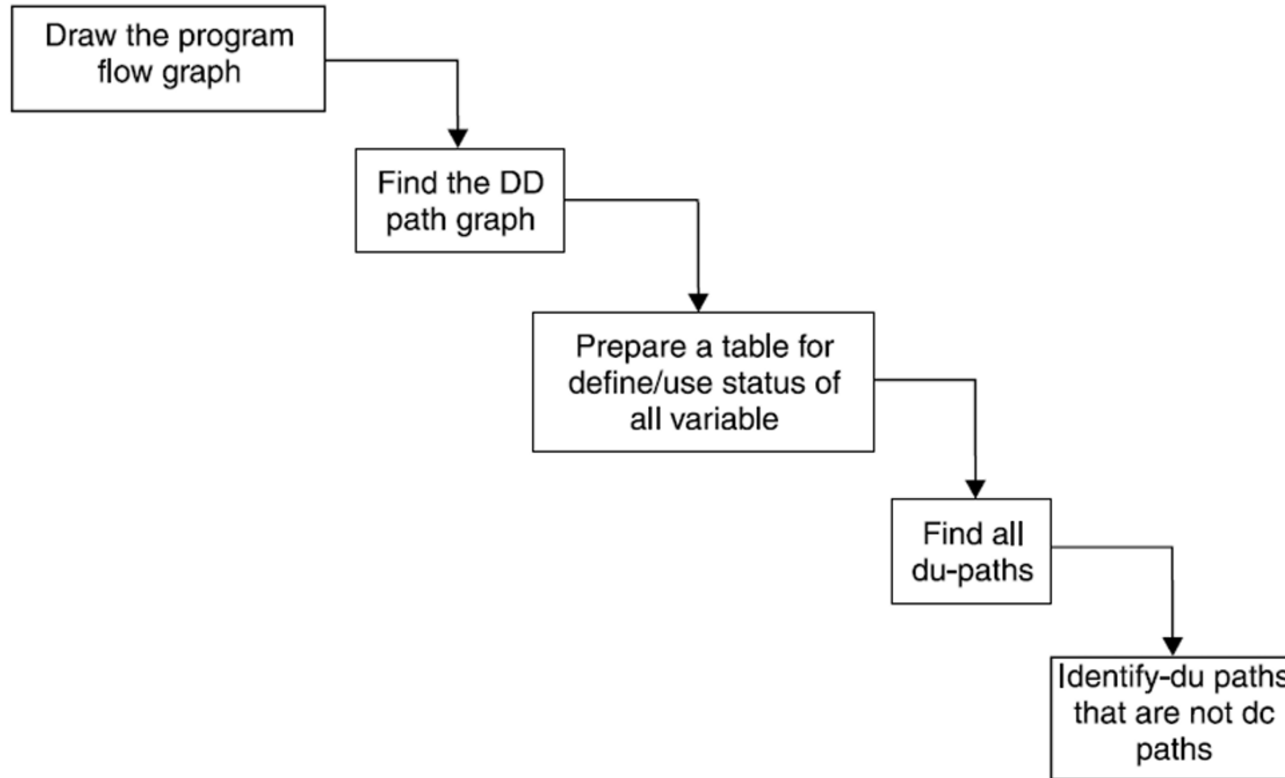


Fig. 27 : Steps for data flow testing

Mutation Testing

Mutation testing is a fault based technique that is similar to fault seeding, except that mutations to program statements are made in order to determine properties about test cases. it is basically a fault simulation technique.

Multiple copies of a program are made, and each copy is altered; this altered copy is called a mutant. Mutants are executed with test data to determine whether the test data are capable of detecting the change between the original program and the mutated program.

A mutant that is detected by a test case is termed "killed" and the goal of mutation procedure is to find a set of test cases that are able to kill groups of mutant programs.

When we mutate code there needs to be a way of measuring the degree to which the code has been modified. For example, if the original expression is x+1 and the mutant for that expression is x+2, that is a lesser change to the original code than a mutant such as (c*22), where both the operand and the operator are changed. We may have a ranking scheme, where a first order mutant is a single change to an expression, a second order mutant is a mutation to a first order mutant, and so on. High order mutants becomes intractable and thus in practice only low order mutants are used.

One difficulty associated with whether mutants will be killed is the problem of reaching the location; if a mutant is not executed, it cannot be killed. Special test cases are to be designed to reach a mutant. For example, suppose, we have the code.

```
Read (a,b,c);
If(a>b) and (b=c) then
x:=a*b*c; (make mutants; m1, m2, m3 .......)
```

To execute this, input domain must contain a value such that a is greater than b and b equals c. If input domain does not contain such a value, then all mutants made at this location should be considered equivalent to the original program, because the statement x:=a*b*c is dead code (code that cannot be reached during execution). If we make the mutant x+y for x+1, then we should take care about the value of y which should not be equal to 1 for designing a test case.

The manner by which a test suite is evaluated (scored) via mutation testing is as follows: for a specified test suite and a specific set of mutants, there will be three types of mutants in the code i.e., killed or dead, live, equivalent. The sum of the number of live, killed, and equivalent mutants will be the total number of mutants created. The score associated with a test suite T and mutants M is simply.

$$\frac{\#killed}{\#total - \#equivalent} \times 100\%$$

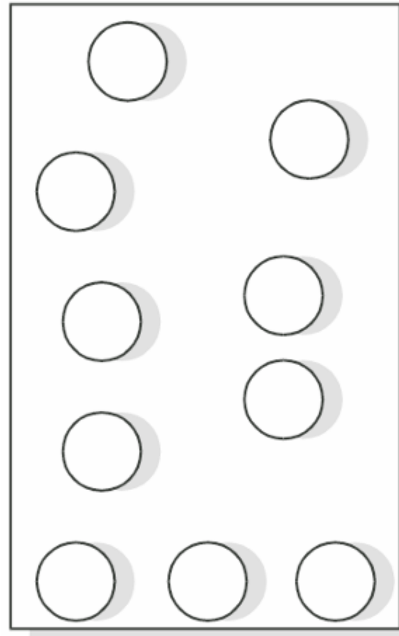
Levels of Testing

There are 3 levels of testing:

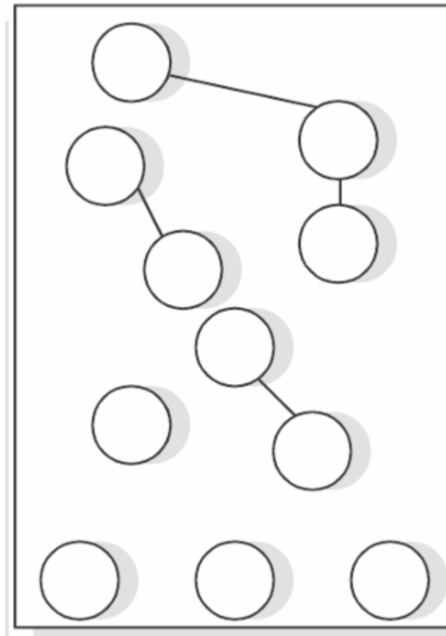
i. Unit Testing

ii. Integration Testing

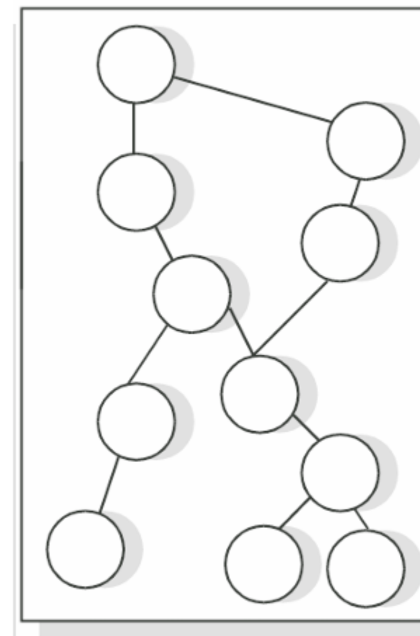
iii. System Testing



UNIT TESTING



INTEGRATION TESTING

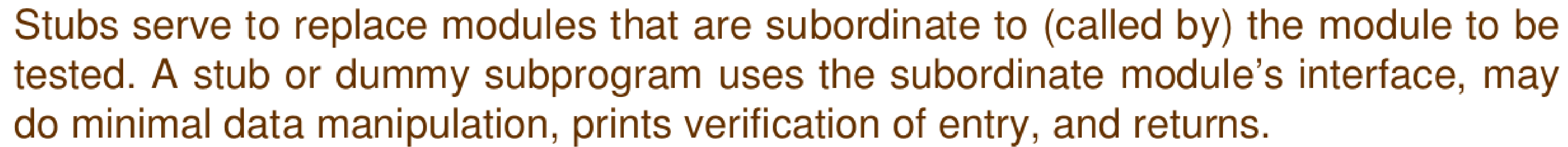


SYSTEM TESTING

Unit Testing

There are number of reasons in support of unit testing than testing the entire product.

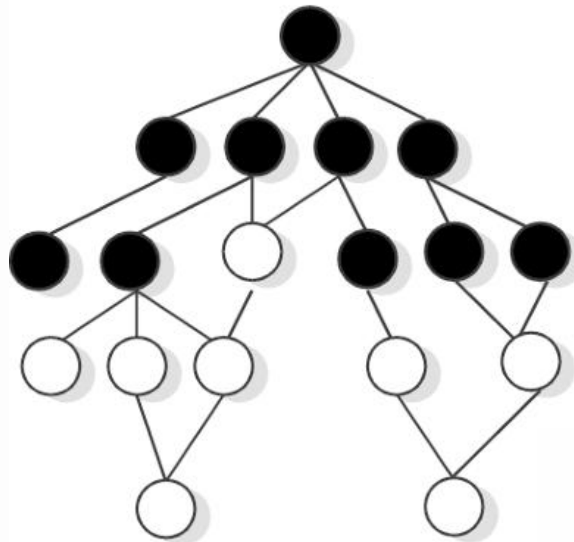
1. The size of a single module is small enough that we can locate an error fairly easily.
2. The module is small enough that we can attempt to test it in some demonstrably exhaustive fashion.
3. Confusing interactions of multiple errors in widely different parts of the software are eliminated.

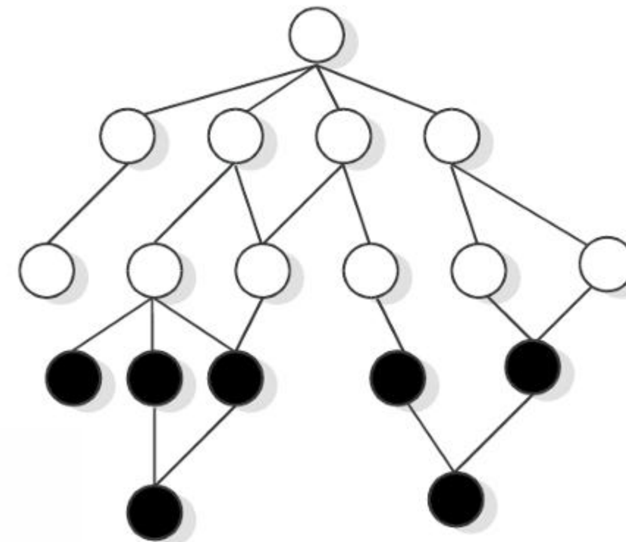

Stubs serve to replace modules that are subordinate to (called by) the module to be tested. A stub or dummy subprogram uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns.

This overhead code, called scaffolding represents effort that is import to testing, but does not appear in the delivered product as shown in Fig. 29.

Integration Testing

The purpose of unit testing is to determine that each independent module is correctly implemented. This gives little chance to determine that the interface between modules is also correct, and for this reason integration testing must be performed. One specific target of integration testing is the interface: whether parameters match on both sides as to type, permissible ranges, meaning and utilization.





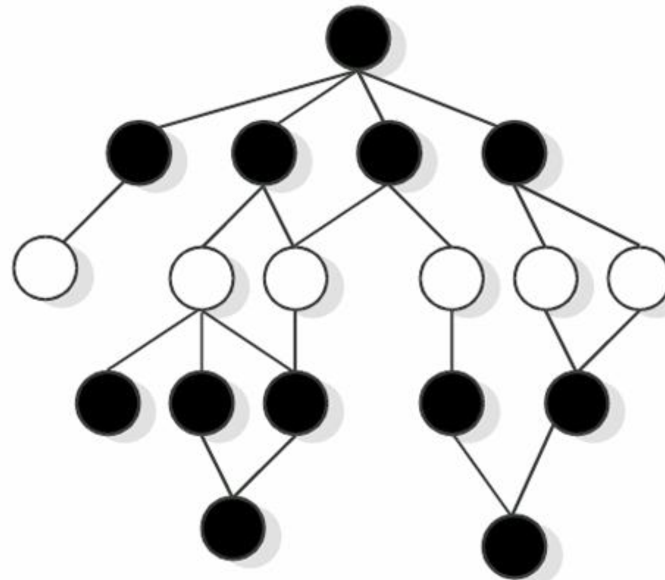


Fig. 30 : Three different integration approaches

System Testing

Of the three levels of testing, the system level is closet to everyday experiences. We test many things; a used car before we buy it, an on-line cable network service before we subscribe, and so on. A common pattern in these familiar forms is that we evaluate a product in terms of our expectations; not with respect to a specification or a standard. Consequently, goal is not to find faults, but to demonstrate performance. Because of this we tend to approach system testing from a functional standpoint rather than from a structural one. Since it is so intuitively familiar, system testing in practice tends to be less formal than it might be, and is compounded by the reduced testing interval that usually remains before a delivery deadline.

Petschenik gives some guidelines for choosing test cases during system testing.

During system testing, we should evaluate a number of attributes of the software that are vital to the user and are listed in Fig. 31. These represent the operational correctness of the product and may be part of the software specifications.

Usable	Is the product convenient, clear, and predictable?
Secure	Is access to sensitive data restricted to those with authorization?
Compatible	Will the product work correctly in conjunction with existing data, software, and procedures?
Dependable	Do adequate safeguards against failure and methods for recovery exist in the product?
Documented	Are manuals complete, correct, and understandable?

Fig. 31 : Attributes of software to be tested during system testing

Validation Testing

- It refers to test the software as a complete product.
- This should be done after unit & integration testing.
- Alpha, beta & acceptance testing are nothing but the various ways of involving customer during testing.

The Art of Debugging

The goal of testing is to identify errors (bugs) in the program. The process of testing generates symptoms, and a program's failure is a clear symptom of the presence of an error. After getting a symptom, we begin to investigate the cause and place of that error. After identification of place, we examine that portion to identify the cause of the problem. This process is called debugging.

Debugging Techniques

Pressman explained few characteristics of bugs that provide some clues.

1. "The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located in other part. Highly coupled program structures may complicate this situation.
2. The symptom may disappear (temporarily) when another error is corrected.

UPES
UNIVERSITY OF TOMORROW

3. The symptom may actually be caused by non errors (e.g. round off inaccuracies).

4. The symptom may be caused by a human error that is not easily traced.

5. The symptom may be a result of timing problems rather than processing problems.

6. It may be difficult to accurately reproduce input conditions (e.g. a real time application in which input ordering is indeterminate).

7. The symptom may be intermittent. This is particularly common in embedded system that couple hardware with software inextricably.

8. The symptom may be due to causes that are distributed across a number of tasks running on different processors".

Testing Tools

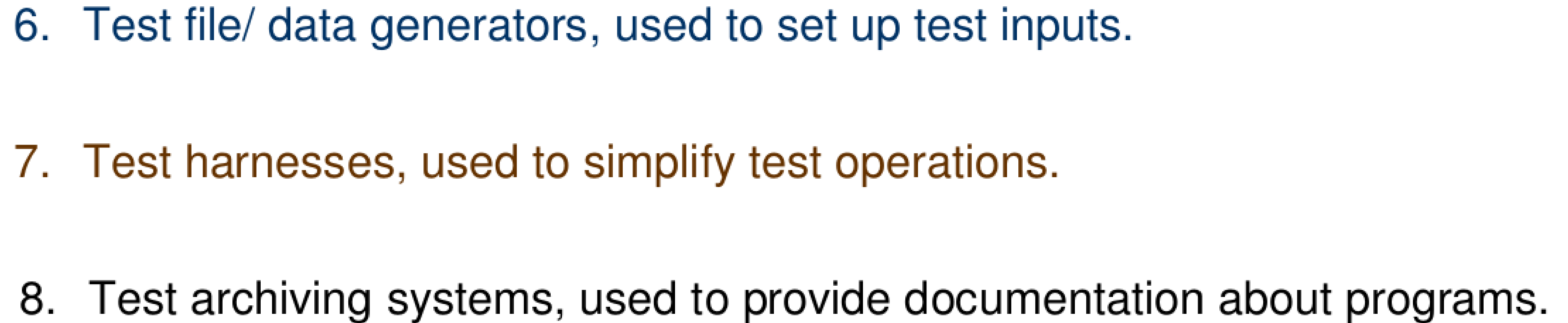
One way to improve the quality & quantity of testing is to make the process as pleasant as possible for the tester. This means that tools should be as concise, powerful & natural as possible.

The two broad categories of software testing tools are :

- Static
- Dynamic

There are different types of tools available and some are listed below:

1. Static analyzers, which examine programs systematically and automatically.
2. Code inspectors, who inspect programs automatically to make sure they adhere to minimum quality standards.
3. standards enforcers, which impose simple rules on the developer.
4. Coverage analysers, which measure the extent of coverage.
5. Output comparators, used to determine whether the output in a program is appropriate or not.


6. Test file/ data generators, used to set up test inputs.
7. Test harnesses, used to simplify test operations.
8. Test archiving systems, used to provide documentation about programs.

Software Project Planning

After the finalization of SRS, we would like to estimate size, cost and development time of the project. Also, in many cases, customer may like to know the cost and development time even prior to finalization of the SRS.

In order to conduct a successful software project, we must understand:

- Scope of work to be done
- The risk to be incurred
- The resources required
- The task to be accomplished
- The cost to be expended
- The schedule to be followed

Software planning begins before technical work starts, continues as the software evolves from concept to reality, and culminates only when the software is retired.

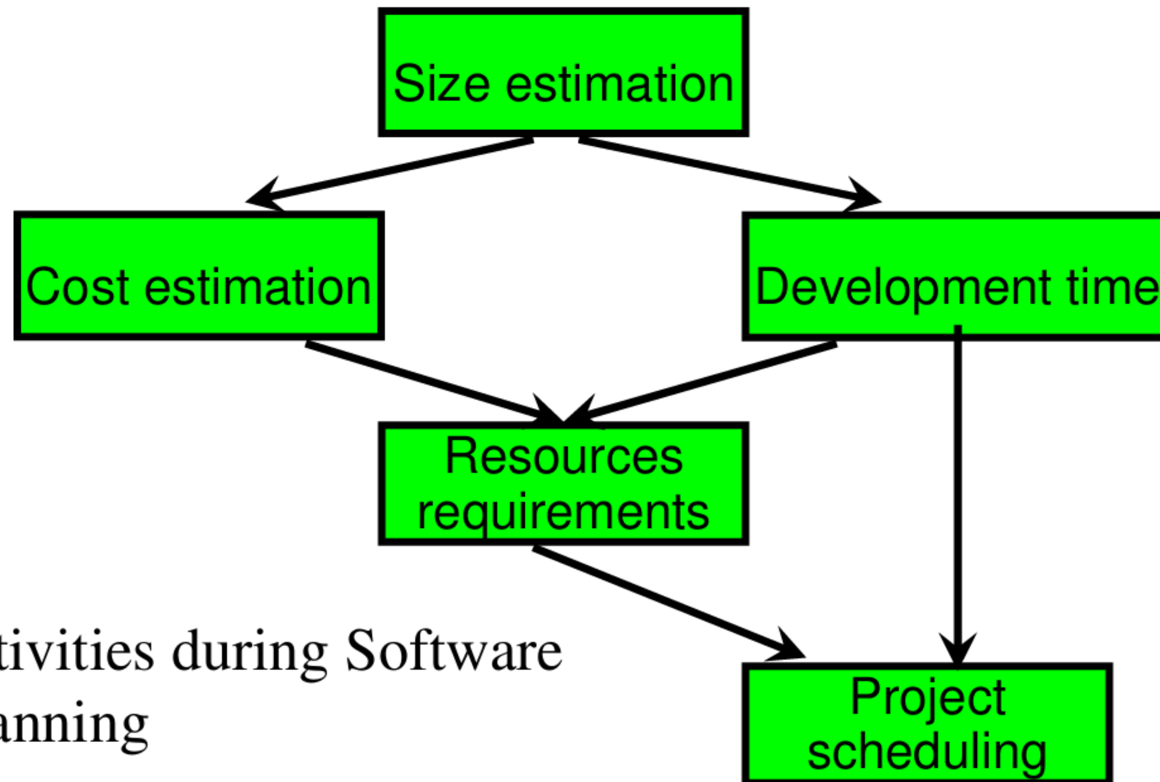


Fig. 1: Activities during Software Project Planning

"A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program header, declaration, and executable and non-executable statements".

This is the predominant definition for lines of code used by researchers. By this definition, figure shown above has 17 LOC.

UPES
UNIVERSITY OF TOMORROW

Function Count

Alan Albrecht while working for IBM, recognized the problem in size measurement in the 1970s, and developed a technique (which he called Function Point Analysis), which appeared to be a solution to the size measurement problem.

The principle of Albrecht's function point analysis (FPA) is that a system is decomposed into functional units.

▪ Inputs	:	information entering the system
▪ Outputs	:	information leaving the system
▪ Enquiries	:	requests for instant access to information
▪ Internal logical files	:	information held within the system
▪ External interface files	:	information held by other system that is used by the system being analyzed.

The FPA functional units are shown in figure given below:

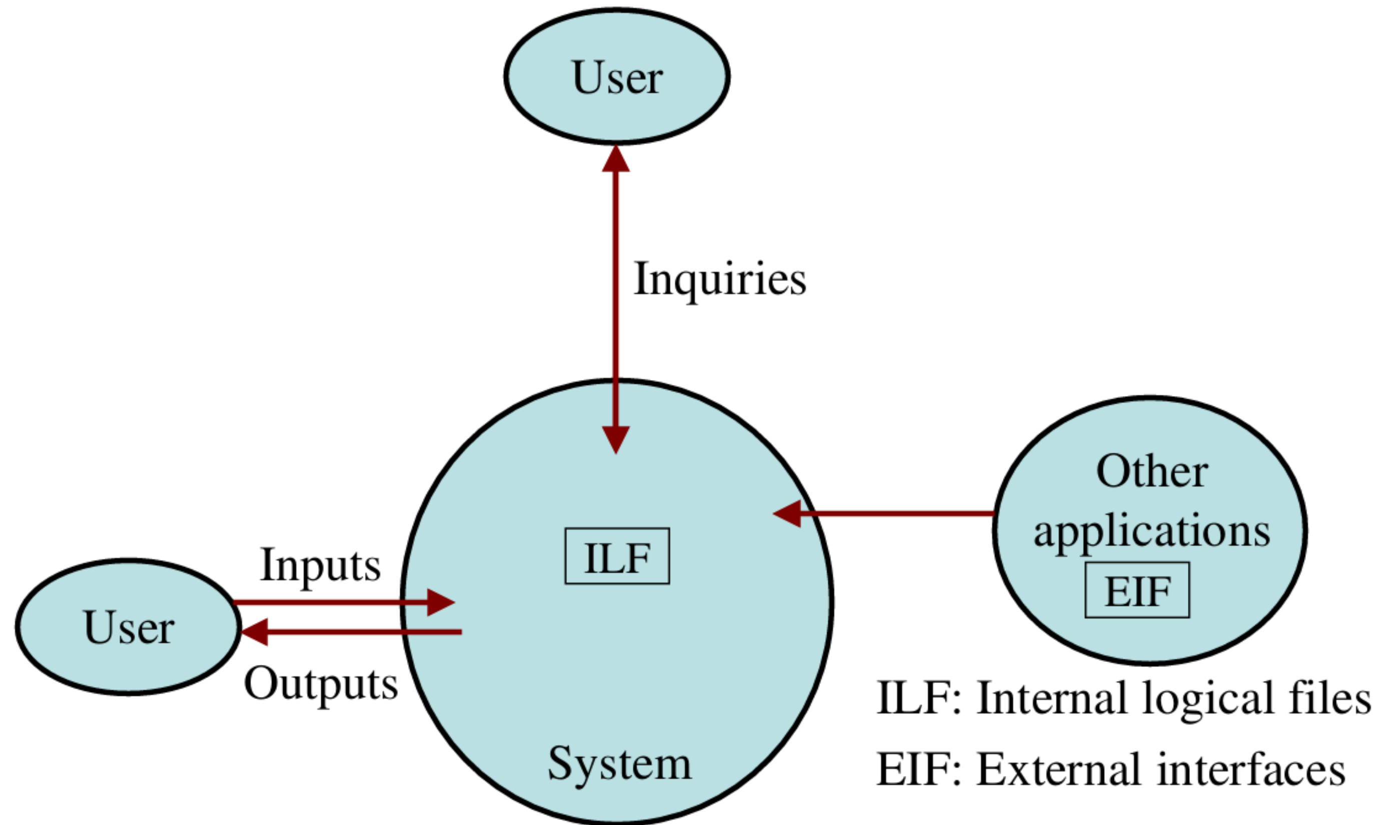


Fig. 3: FPAs functional units System

DELTA Page No.
Date / /

6. (*) Estimation Model. (*)

In the s/w engineering practices different estimation models are available to estimate various attributes of the s/w.

Empirical models are available to estimate various attribute. The model which is derived based on the experiance of the past project without proof is called as empirical model.

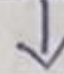
The structure of the empirical model = $E = A + B \cdot (ev)^C$

where E = effort (man-month)
A, B, C are the empirical constant.
ev = estimated value.

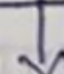
s/w attribute estimated sequence is -

S/w attribute estimated sequence is -

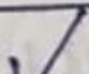
SRS.

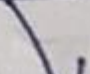


Size Estimation.



Effor Estimation.





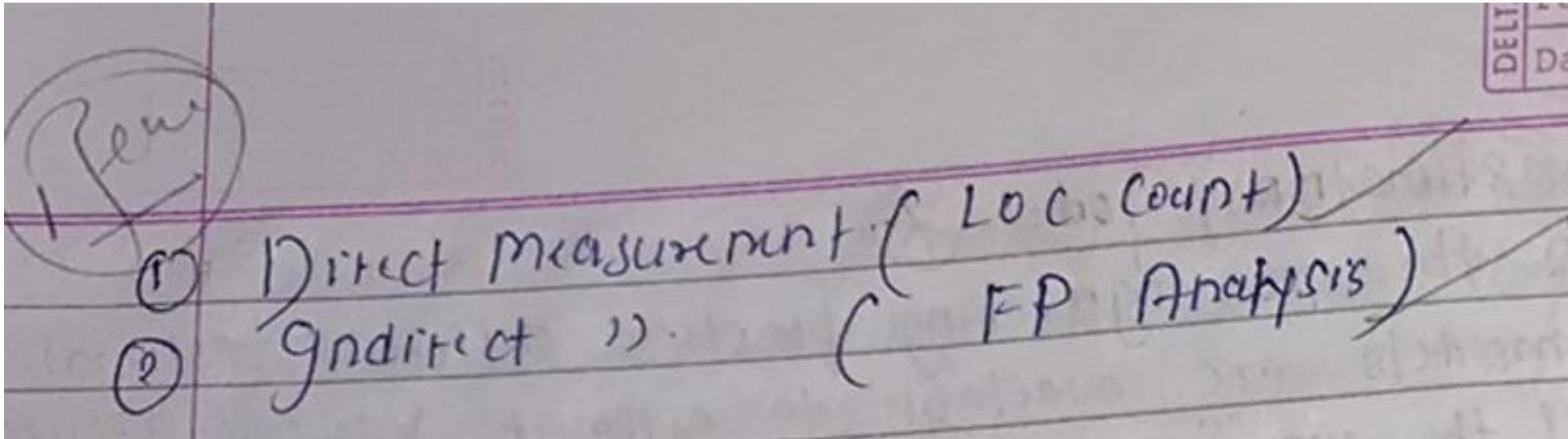
Cost Estimation

Scheddule Estimation

⊛ Size Estimation ⊛

Based on the SRS document we can estimate the size of the s/w. Size of the s/w is estimate

10+100 days -


Rem
① Direct measurement (LOC Count)
② Indirect " (FP Analysis)

① LOC (Line of Code) Count Estimation. ⊛

LOC count means physically count how many lines are present inthe developed program.
It gives the best result when the size is estimated after the developement.

If we want to estimate the size in terms of LOC before the developement there is need of concidearing programmar skill level.
A/c to the skill level developed program size is depend i.e. skill level developed code.

Sill level → Developed Code.

UPES UNIVERSITY OF TOMORROW

Sill Level — Developed Code.

1. efficient → Optistimistic (Sopt)
2. Average → most likely (Sm)
3. poor. → Pessimistic (Spess)

After conCedering the developer skill level we can estimate the size of the slco by using following formula:

$$S = \frac{S_{opt} + 4 \cdot S_m + S_{pess}}{6} \text{ loc.}$$

UPES
UNIVERSITY OF TOMORROW

2011 Gate.

Page No.
Date / /

Q. (1) A Company needs to develope the s/w for the web app. s/w is expected to have 4600 optestimic LOC, 6900 most likely LOC and 8600 pessimestic LOC what is the expected size of the s/w.

(a)

$$S = \frac{4 S_m + S_{pess} + S_{opt}}{6}$$

$$S = \frac{4 \times 6900 + 8600 + 4600}{6}$$

$$S = 6800 \text{ LOC}$$

(b) If the s/w developement is expected to involve 400 man years of the effort then what is the productivity.

(b) If the SLOC developement is expected to involve two man years of the effort then what is the productivity.

2 man-years. effort = 24 man-monts. effor — 6800 LOC proj.

Productivity means # of line of code developed / man-month.

Now. 1 man-month LOC (productivity) = $\frac{6800}{24 \text{ man-month}}$

= 283.3 LOC / man-month

(c) If the s/w developement is expectd, developent involve 5 month duration what is avg manning (man-power)

Avg (man-power) = ?

$$\text{manpower} = \frac{\text{effort}}{\text{Duration}} = \frac{24 \text{ m-month}}{5 \text{ month}} = 4.\sim = 5 \text{ devlo per}$$

UPES
UNIVERSITY OF TOMORROW

Page No.
Date / /

LOC. line of code.

KLOC kilo "

KDLOC kilo delevery LOC

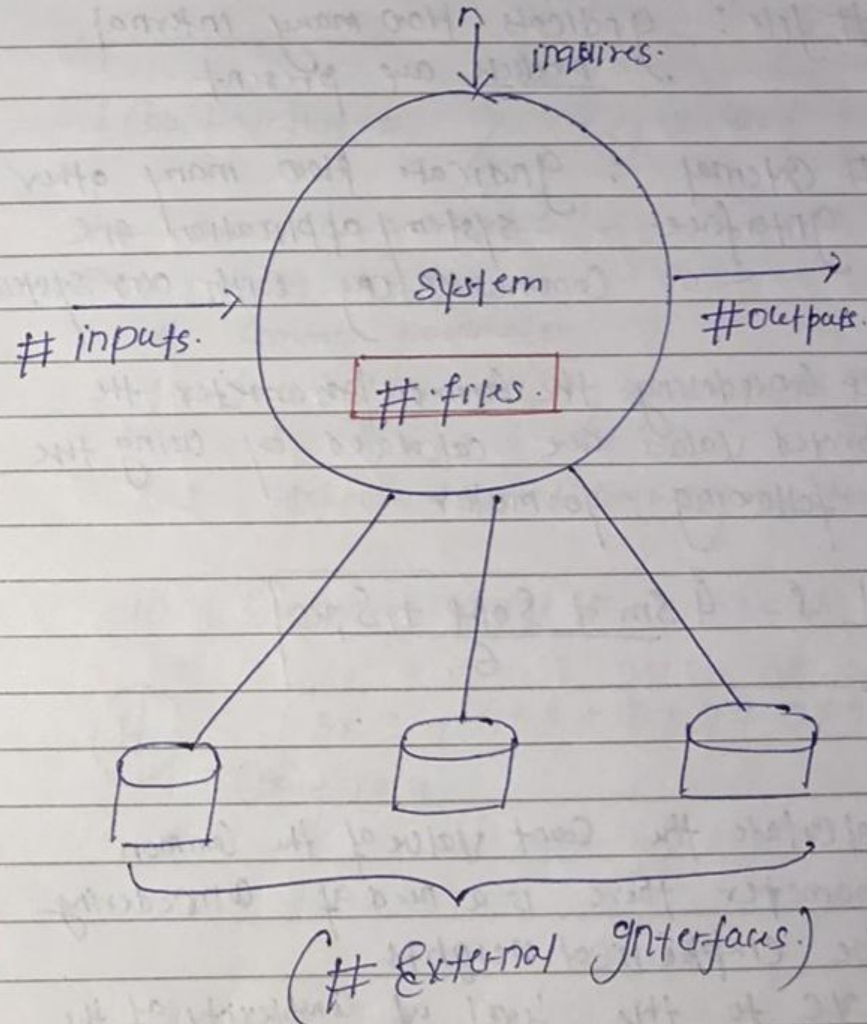
DSI. Delevery of source instruction

KDSI. kilo delevery of source instruction.

FP (Function Point) Analysis

- FP analysis is a indirect majored parameter.
- FP analysis is best suit to estimate the size of the s/w before the developement.
- To estimate the function point for the s/w there is a need of consider functional & Non-functional parameter.

The functional parameter s/w system is -


The functional parameter s/w system is -
inquires.
System
inputs.
outputs.
files.
(# External Interfaces)

* Any s/w system must contain the following sets of the parameters.

1. # inputs : Indicates How many i/p's are accepted into the system
2. # outputs : Indicates How many o/p's are generated by the system.
3. # enquiries : How many ways are present to access the quick information
4. # file : Indicates How many internal module are present.
5. # External Interface : Indicate How many other system/application are communicating with our system.

After considering the common parameter the expected values are calculated by using the following formular.

$$S = \frac{4 S_m + S_{opt} + S_{pe}}{6}$$

UPES UNIVERSITY OF TOMORROW

To calculate the count value of the common parameter there is a need of considering the empirical weights.

a/c to the level of complexity of the project different emprical const are derived based on the fast project.

For calculating the function point -

F.P = UFP x CAF

Unadjested function point

Complexity adjestment factor.

$$UFP = \sum_{i=1}^{5} \sum_{j=1}^{3} Z_{ij} \cdot W_{ij}$$

Number of I/P (Z_{ij}) — W_{ij} → Avg value.

$i=1$ → decomposed into five part

$j=1$ → loating factor

	Functional Unit Z_{ij}	Weighting factor (W_{ij}) Low	Avg	High
1	EI	3	4	6
2	EO	4	5	7
3	Enquires (ENQ)	3	4	6
4	ILF (5 7 10 15)	7	10	15
5	EIF (5 7 10)	5	7	10

Date:

Z_{ij} represents the number of functional unit and W_{ij} represents the waiting factor for that functional unit.

UPES
UNIVERSITY OF TOMORROW

The procedure for the calculation of UFP in mathematical form is given below:

$$UFP = \sum_{i=1}^{5} \sum_{J=1}^{3} Z_{ij} w_{ij}$$

Where i indicate the row and j indicates the column of Table 1

W_{ij} : It is the entry of the ith row and j^{th} column of the table 1

Zij : It is the count of the number of functional units of Type *i* that have been classified as having the complexity corresponding to column *j*.

Organizations that use function point methods develop a criterion for determining whether a particular entry is Low, Average or High. Nonetheless, the determination of complexity is somewhat subjective.

$$FP = UFP * CAF$$

Where CAF is complexity adjustment factor and is equal to [0.65 + 0.01 x ΣF_i]. The F_i (i=1 to 14) are the degree of influence and are based on responses to questions noted in table 3.

Table 3 : Computing function points.

Rate each factor on a scale of 0 to 5.

Number of factors considered (F_i)

1. Does the system require reliable backup and recovery ?
2. Is data communication required ?
3. Are there distributed processing functions ?
4. Is performance critical ?
5. Will the system run in an existing heavily utilized operational environment ?
6. Does the system require on line data entry ?
7. Does the on line data entry require the input transaction to be built over multiple screens or operations ?
8. Are the master files updated on line ?
9. Is the inputs, outputs, files, or inquiries complex ?
10. Is the internal processing complex ?
11. Is the code designed to be reusable ?
12. Are conversion and installation included in the design ?
13. Is the system designed for multiple installations in different organizations ?
14. Is the application designed to facilitate change and ease of use by the user ?

Functions points may compute the following important metrics:

Productivity	=	FP / persons-months
Quality	=	Defects / FP
Cost	=	Rupees / FP
Documentation	=	Pages of documentation per FP

These metrics are controversial and are not universally acceptable. There are standards issued by the International Functions Point User Group (IFPUG, covering the Albrecht method) and the United Kingdom Function Point User Group (UFPGU, covering the MK11 method). An ISO standard for function point method is also being developed.

Page No.
Date / /

Empirical weights.

Simple	Avg.	Complex.
3	4	6
4	5	7
3	4	6
7	10	15
5	7	10

After considering the empirical weights we can estimate the Count value by using the following formula-

$$\text{Count value} = \sum_{i=1,\, j=1}^{5,\,3} Z_{ij} * w_{ij}$$

i = Common parameter.

j = level of complexity of project. ① ② ③ (Simple, Avg, Complex)

Z = expected value of common parameter.

w: Empirical weights.

(H) $(3 \times 3 + 4 \times 4 + 8 \times 3 + 3 \times 7 + 2 \times 5)$ — terms numbered ① ② ③ ④ ⑤

Count value.

VAF Value.

* CAF * (Complexity Adjustment Factor)

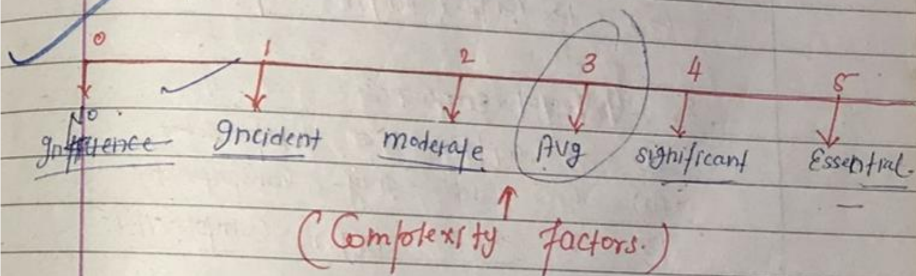
Complexity adjustment factor depends on 14 answers of the 14 question and its value — is —

$$CAF = [.65 + .01 * \sum F_i]$$

where i ranges from 1 to 14

i.e. ($F_1 + F_2 + F_3$ ---- F_{14})

These functional values depends on the following table —



(Complexity factors.)

Day 3. 31 December 2018 Day. Monday.

- In the function point analysis, 14 additional parameters are introduced to estimate the function points for the s/w.
- Addition parameter, level of involvement and associated impact is rated as follows –

Level of involvement	Impact
① If no influence	0
② If incidental	1
③ If moderate	2
④ If Avg	3
⑤ If significant	4
⑥ If essential	5

VAF (Value adjustment Factor) is calculated based on the additional parameter impact i.e.

$$VAF = 0.65 + 0.01 \times \sum_{i=1}^{14} F_i$$

The function points required for the s/w project is estimated as –

$$FP = Count\ Value * VAF$$

Count value = no * weight

Q. IMP Consider a project with the following functional unit.

1. no of user I/P = 50
2. no of user O/P = 40
3. no of user enquires = 35
4. no of user files = 6. ILF.
5. no of external interfaces = 4 EIF

Assume all complexity adjustment factor and weighting factor are average then compute function point for this project.

∴ FP = UFP × CAF.

UFP = 50 × 4 + 40 × 5 + 35 × 4 + 6 × 10 + 4 × 7.

UFP = 200 + 200 + 140 + 60 + 28

= 628

CAF = .65 + .01 * (3 + 3 ... 14 times)

CAF = .65 + .42.

CAF = 1.07.

Hence FP = UFP × CAF

= 628 × 1.07

F.P. = 672.

(always integer) Answer.

Cost Estimation

Effort Estimation (*)

Effort requires to develope the program is depending on the size of the S/W.
There are different emperical model present to estimate the effort and duration of the project.

The structure of emperical model is –

UPES
UNIVERSITY OF TOMORROW

Cost Estimation

$$\text{Effort} = A + B \cdot (\text{size})^{C}$$

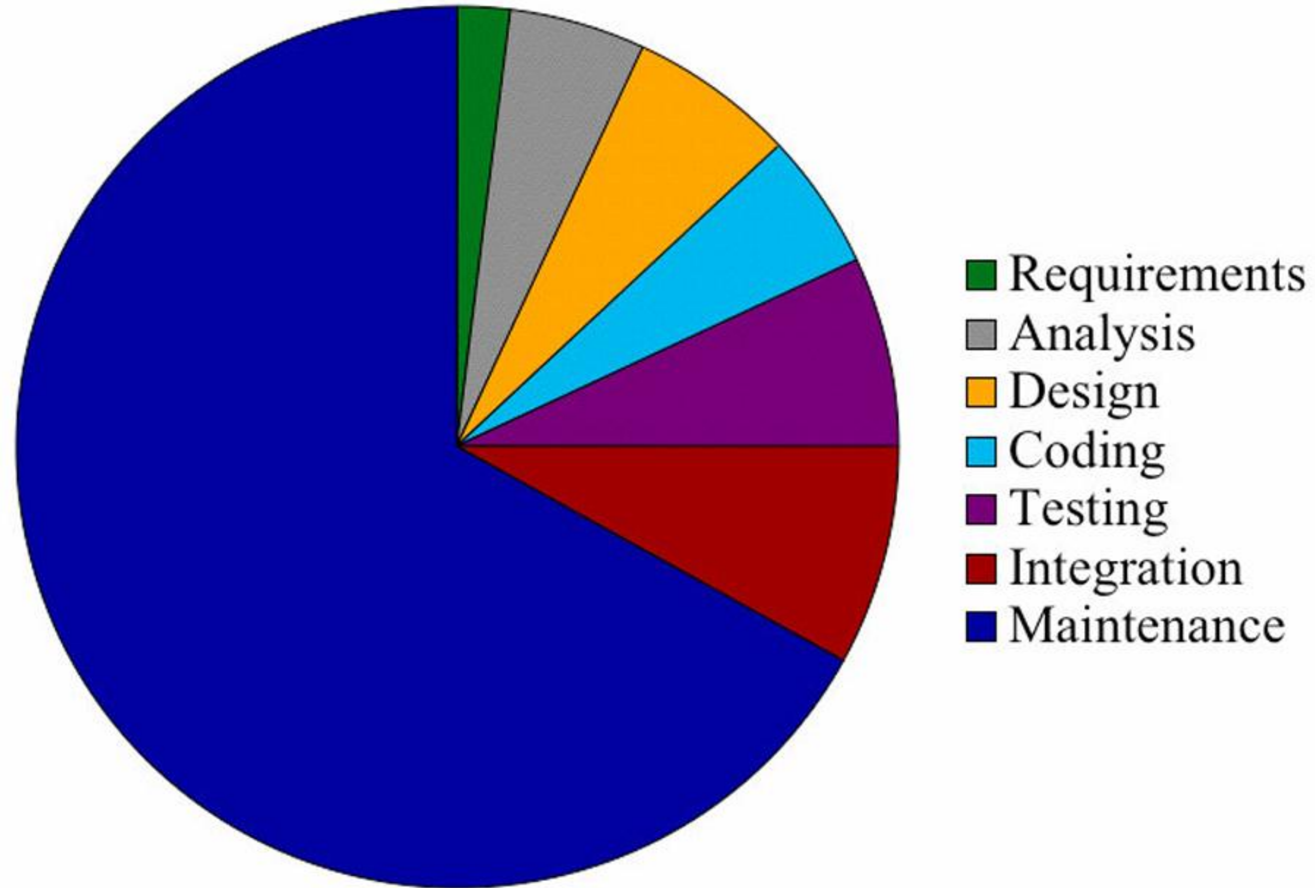
$$\text{size} = \text{KLOC} \mid \text{FP}$$

In the s/w engineering practices different emperical model are present the frequently used emperical model to estimate the effort & duration is

1. SEL Model (system deciplined structure i.e given by model)
2. WF Model (Defined by IBM).
3. COCOMO | Boehm's Model.

NOTE. Only emperical model's structure is same but emperical constant (A,B,C) are differ because person to person analysis function is varries.

Cost Estimation



Cost Estimation

- For any new software project, it is necessary to know how much will it cost to develop and how much development time will it take. These estimates are needed before development is initiated.

- But how is this done? In many cases estimates are made using past experience as the only guide. However, in most of the cases projects are different and hence past experience alone may not be enough.

- A number of estimation techniques have been developed and are having following attributes in common.

Cost Estimation

- Project scope must be established in advance
- Software metrics are used as a basis from which estimates are made
- The project is broken into small pieces which are estimated individually

To achieve reliable cost and schedule estimates, a number of options arise:

- Delay estimation until late in project
- Use simple decomposition techniques to generate project cost and schedule estimates
- Develop empirical models for estimation
- Acquire one or more automated estimation tools

UPES
UNIVERSITY OF TOMORROW

Models

- The model is concerned with the representation of the process to be estimated.

- A model may be static or dynamic. In a static model, a unique variable (say, size) is taken as a key element for calculating all others (say, cost, time). The form of equation used is the same for all calculations.

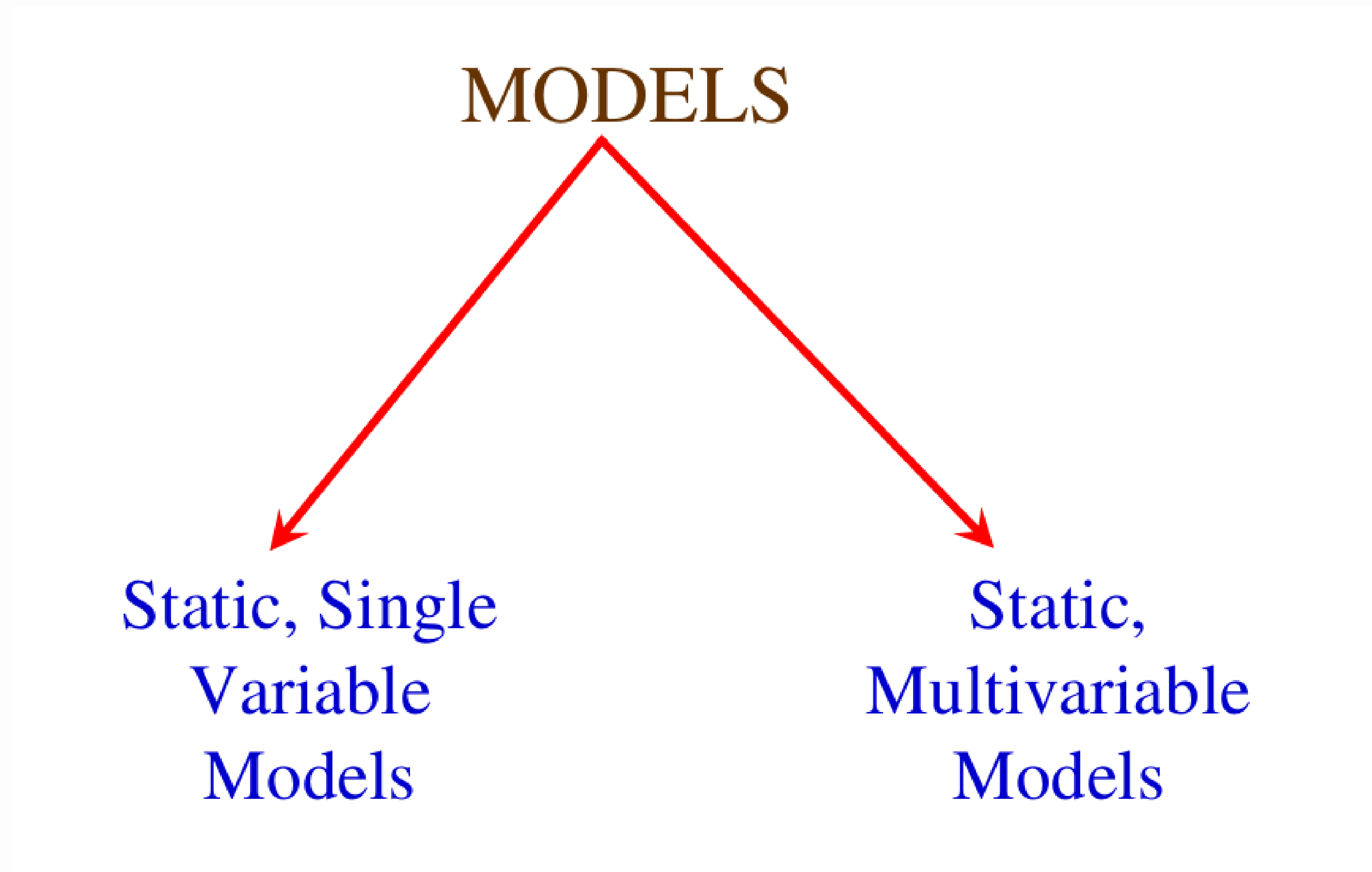
- In a dynamic model, all variables are interdependent and there is no basic variable as in the static model.

- When a model makes use of a single basic variable to calculate all others it is said to be a single-variable model.

Models

- In some models, several variables are needed to describe the software development process, and selected equations combine these variables to give the estimate of time and cost. These models are called multivariable.
- The variables, single or multiple, that are input to the model to predict the behaviour of a software development are called predictors.
- The choice and handling of these predictors are most crucial activity in estimating methodology.

Models



Static, Single Variable Models

- Methods using this model use an equation to estimate the desired values such as cost, time, effort, etc.

- They all depend on the same variable used as predictor (say, size). An example of the most common equation is:

$$C = aL^b \text{...............4.1}$$

- where C is the cost (effort expressed in any unit of manpower, for example, person-months) and L is the size generally given in the number of lines of code.

- The constants a and b are derived from the historical data of the organisation. Since a and b depend on the local development environment, these models are not transportable to different organisations.

UPES
UNIVERSITY OF TOMORROW

Static, Single Variable Models

- The Software Engineering Laboratory (SEL Model) of the University of Maryland has established a model, the SEL model, for estimating its own software productions.

- This model [BASL80] is a typical example of a static single-variable model:

$$E = 1.4L^{0.93}$$

$$DOC = 30.4L^{0.90}$$

$$D = 4.6L^{0.26}$$

- Effort (E in person-months), documentation (DOC, in number of pages) and duration (D, in months) are calculated from the number of lines of code (L, in thousands of lines) used as a predictor.

UPES
UNIVERSITY OF TOMORROW

Static, Multivariable Models

- Although these models are often based on equation (4.1), they actually depend on several variables representing various aspects of the software development environment, for example, methods used, user participation, customer-oriented changes, memory constraints, etc.

- The model developed by Walston and Felix at IBM [WALS77] provides a relationship between delivered lines of source code (L in thousands of lines) and effort E(E in person-months) and is given by the following equation:

$$E = 5.2L^{0.91}$$

- In the same fashion, the duration of the development (D in months) is given by:

$$D = 4.1L^{0.36}$$

Static, Multivariable Models

- Data collected on 60 software projects, representing a wide variety of applications and size (ranging from 4000 to 467000 lines of code), shows a relationship between productivity (expressed in number of lines of source code per person-months) and a productivity index I.

- The productivity index uses 29 variables which are found to be highly correlated to productivity as follows:

$$I = \sum_{i=1}^{29} W_i X_i \quad \text{........4.7}$$

- where W_i is a factor weight for the ith variable and $X_i = \{-, 1, 0 + 1\}$.

- The estimator gives X_i one of the values −1, 0 or +1 depending on whether the variable decreases, has no effect, or increases the productivity respectively.

- The terms of equation (4.7) are then added up to give the productivity index.

UPES
UNIVERSITY OF TOMORROW

Static, Multivariable Models

Example: 4.4

Compare the Walston-Felix model with the SEL model on a software development expected to involve 8 person-years of effort.

(a) Calculate the number of lines of source code that can be produced.

(b) Calculate the duration of the development.

(c) Calculate the productivity in LOC/PY

(d) Calculate the average manning

Static, Multivariable Models

Solution

The amount of manpower involved = 8 PY = 96 person-months

(a) Number of lines of source code can be obtained by reversing equation to give:

$$L = (E/a)^{1/b}$$

Then

$$L(SEL) = (96/1.4)^{1/0.93} = 94264 \text{ LOC}$$

$$L(W\text{-}F) = (96/5.2)^{1/0.91} = 24632 \text{ LOC}.$$

Static, Multivariable Models

(b) Duration in months can be calculated by means of equation

$$D(SEL) = 4.6\,(L)^{0.26}$$

$$= 4.6\,(94.264)^{0.26} = 15 \text{ months}$$

$$D(W\text{-}F) = 4.1\,L^{0.36}$$

$$= 4.1(24.632)^{0.36} = 13 \text{ months}$$

(c) Productivity is the lines of code produced per person/month (year)

$$P(SEL) = \frac{94264}{8} = 11783\, LOC / Person - Years$$

$$P(W - F) = \frac{24632}{8} = 3079\, LOC / Person - Years$$

UPES
UNIVERSITY OF TOMORROW

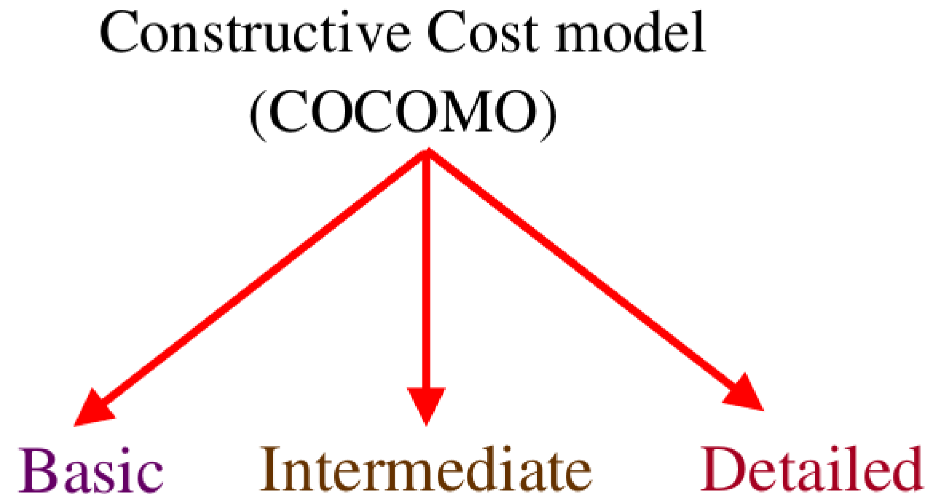
Static, Multivariable Models

(d) Average manning is the average number of persons required per month in the project.

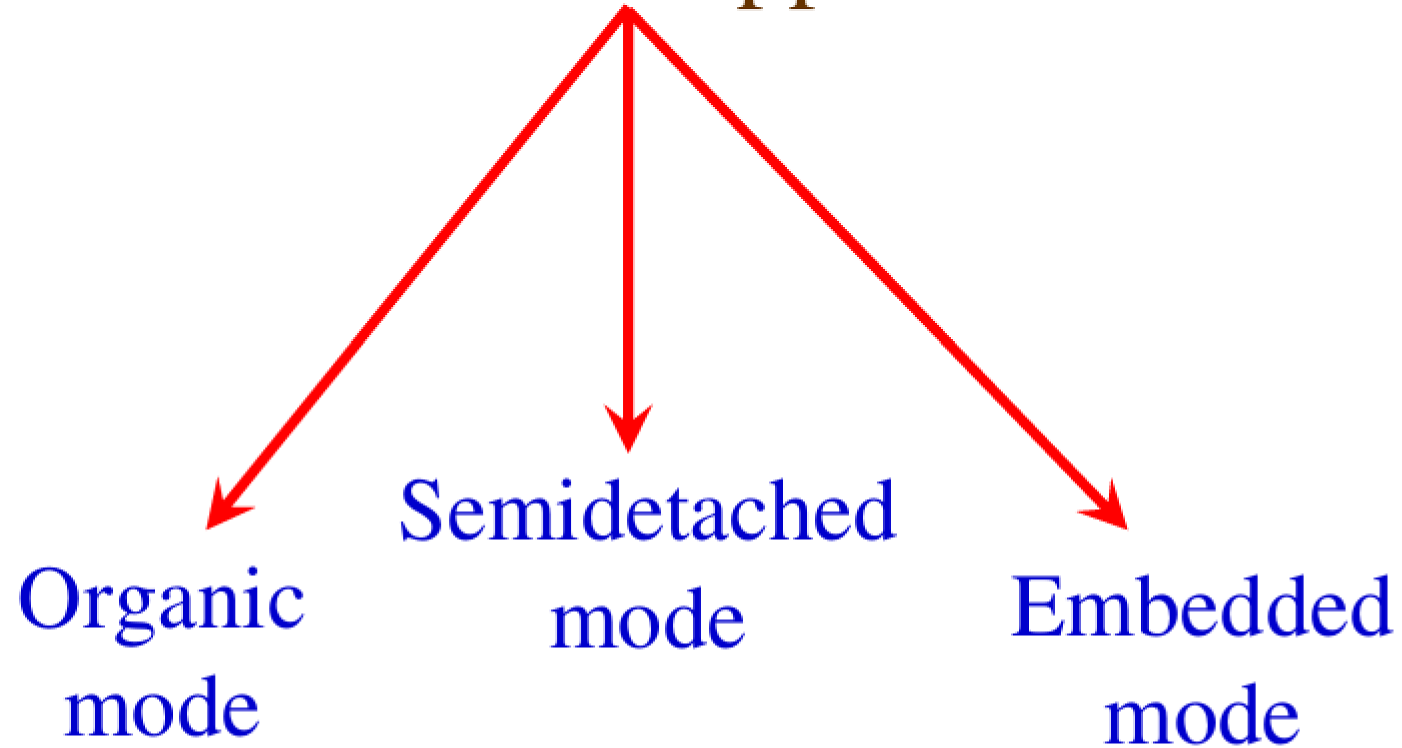
$$M(SEL) = \frac{96P - M}{15M} = 6.4 Persons$$

$$M(W - F) = \frac{96P - M}{13M} = 7.4 Persons$$

The Constructive Cost Model (COCOMO)



Model proposed by
B. W. Boehm's
through his book
Software Engineering Economics in 1981


COCOMO applied to
Organic mode
Semidetached mode
Embedded mode

Mode	Project size	Nature of Project	Innovation	Deadline of the project	Development Environment
Organic	Typically 2-50 KLOC	Small size project, experienced developers in the familiar environment. For example, pay roll, inventory projects etc.	Little	Not tight	Familiar & In house
Semi detached	Typically 50-300 KLOC	Medium size project, Medium size team, Average previous experience on similar project. For example: Utility systems like compilers, database systems, editors etc.	Medium	Medium	Medium
Embedded	Typically over 300 KLOC	Large project, Real time systems, Complex interfaces, Very little previous experience. For example: ATMs, Air Traffic Control etc.	Significant	Tight	Complex Hardware/ customer Interfaces required

(1) Basic Co-Co. Model. (*)

This model is used to take the quick and rough estimation.

Structure of the Co-co model is -

Basic Model

Basic COCOMO model takes the form

$$E = a_b (KLOC)^{b_b}$$

$$D = c_b (E)^{d_b}$$

where E is effort applied in Person-Months, and D is the development time in months. The coefficients a_b, b_b, c_b and d_b are given in table 4 (a).

Software 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

Table 4(a): Basic COCOMO coefficients

When effort and development time are known, the average staff size to complete the project may be calculated as:

$$\text{Average staff size } (SS) = \frac{E}{D} \; Persons$$

When project size is known, the productivity level may be calculated as:

$$\text{Productivity } (P) = \frac{KLOC}{E} \; KLOC\,/\,PM$$

UPES
UNIVERSITY OF TOMORROW

Example: 4.5

Suppose that a project was estimated to be 400 KLOC. Calculate the effort and development time for each of the three modes i.e., organic, semidetached and embedded.

UPES
UNIVERSITY OF TOMORROW

Solution

The basic COCOMO equation take the form:

$$E = a_b (KLOC)^{b_b}$$

$$D = c_b (KLOC)^{d_b}$$

Estimated size of the project = 400 KLOC

(i) Organic mode

$$E = 2.4(400)^{1.05} = 1295.31 \text{ PM}$$

$$D = 2.5(1295.31)^{0.38} = 38.07 \text{ PM}$$

(ii) Semidetached mode

$$E = 3.0(400)^{1.12} = 2462.79 \text{ PM}$$

$$D = 2.5(2462.79)^{0.35} = 38.45 \text{ PM}$$

(iii) Embedded mode

$$E = 3.6(400)^{1.20} = 4772.81 \text{ PM}$$

$$D = 2.5(4772.8)^{0.32} = 38 \text{ PM}$$

Example: 4.6

A project size of 200 KLOC is to be developed. Software development team has average experience on similar type of projects. The project schedule is not very tight. Calculate the effort, development time, average staff size and productivity of the project.

Solution

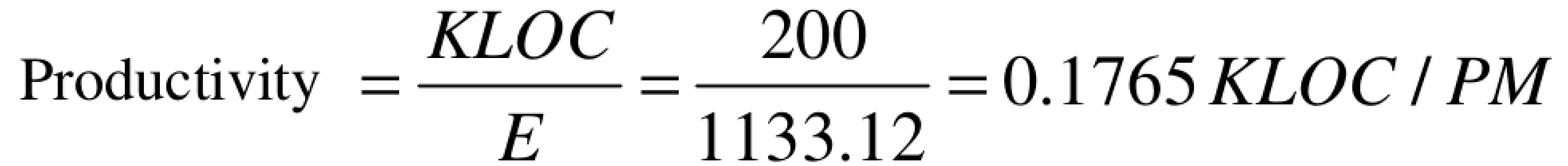
The semi-detached mode is the most appropriate mode; keeping in view the size, schedule and experience of the development team.

Hence $E = 3.0(200)^{1.12} = 1133.12 \text{ PM}$

$D = 2.5(1133.12)^{0.35} = 29.3 \text{ PM}$

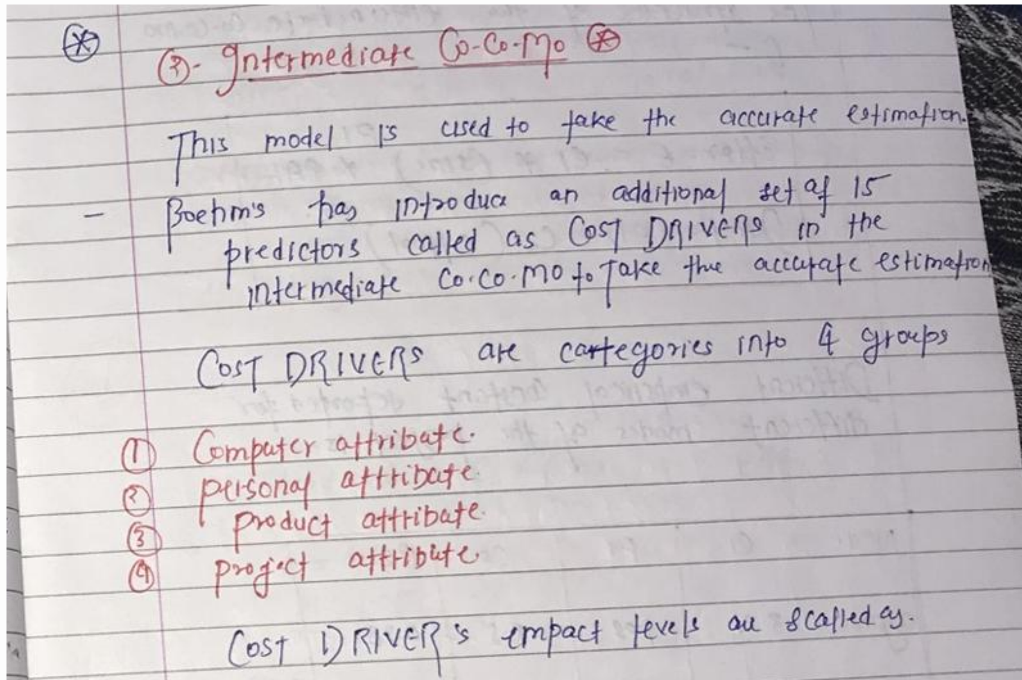
$$\text{Average staff size } (SS) = \frac{E}{D} \, Persons$$

$$= \frac{1133.12}{29.3} = 38.67 \, Persons$$



$$\text{Productivity} = \frac{KLOC}{E} = \frac{200}{1133.12} = 0.1765\, KLOC / PM$$

$$P = 176\, LOC / PM$$



Intermediate Model

Cost drivers

(*i*) Product Attributes

- Required s/w reliability
- Size of application database
- Complexity of the product

(*ii*) Hardware Attributes

- Run time performance constraints
- Memory constraints
- Virtual machine volatility
- Turnaround time

(*iii*) Personal Attributes

- Analyst capability
- Programmer capability
- Application experience
- Virtual m/c experience
- Programming language experience

(*iv*) Project Attributes

- Modern programming practices
- Use of software tools
- Required development Schedule

(2) Intermediate Co-Co-Mo Model

"When we would like to estimate the efforts and Development time more accurately then we choose the intermediate co-co-mo model."

The intermediate co-co-mo model is based on 15 cost drivers. These cost drivers may have 6 possible values—

The each cost drivers may be rated as very low, low, nominal, high, very high, extra high.

Very low	low	nominal	high	very high	Extra high
0.70 to ·75	·75 to ·95	1.	1.08 to 1.15	1.16 to 1.40	1.65

Multipliers of different cost drivers

Cost Drivers	RATINGS					
	Very low	Low	Nominal	High	Very high	Extra high
Product Attributes						
RELY	0.75	0.88	1.00	1.15	1.40	--
DATA	--	0.94	1.00	1.08	1.16	--
CPLX	0.70	0.85	1.00	1.15	1.30	1.65
Computer Attributes						
TIME	--	--	1.00	1.11	1.30	1.66
STOR	--	--	1.00	1.06	1.21	1.56
VIRT	--	0.87	1.00	1.15	1.30	--
TURN	--	0.87	1.00	1.07	1.15	--

Cost Drivers	RATINGS					
	Very low	Low	Nominal	High	Very high	Extra high
Personnel Attributes						
ACAP	1.46	1.19	1.00	0.86	0.71	--
AEXP	1.29	1.13	1.00	0.91	0.82	--
PCAP	1.42	1.17	1.00	0.86	0.70	--
VEXP	1.21	1.10	1.00	0.90	--	--
LEXP	1.14	1.07	1.00	0.95	--	--
Project Attributes						
MODP	1.24	1.10	1.00	0.91	0.82	--
TOOL	1.24	1.10	1.00	0.91	0.83	--
SCED	1.23	1.08	1.00	1.04	1.10	--

Intermediate COCOMO equations

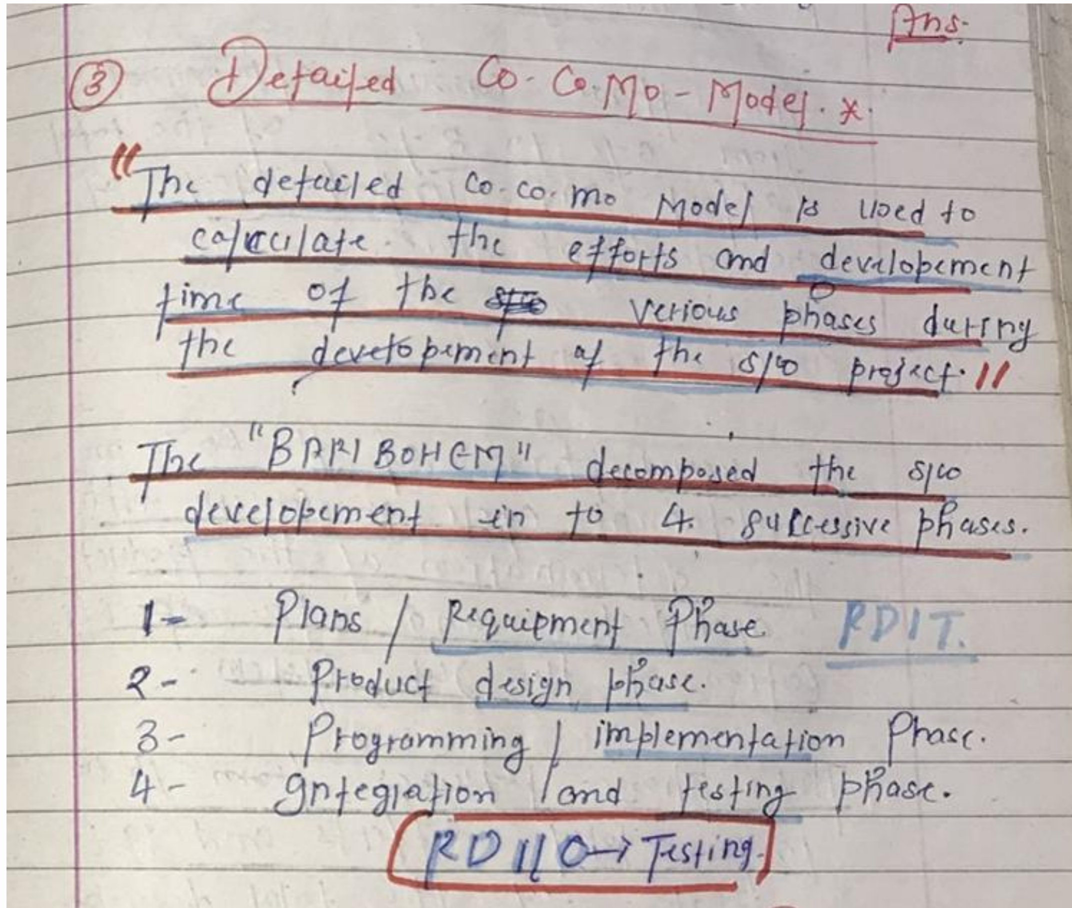
$$E = a_i (KLOC)^{b_i} * EAF$$

$$D = c_i (E)^{d_i}$$

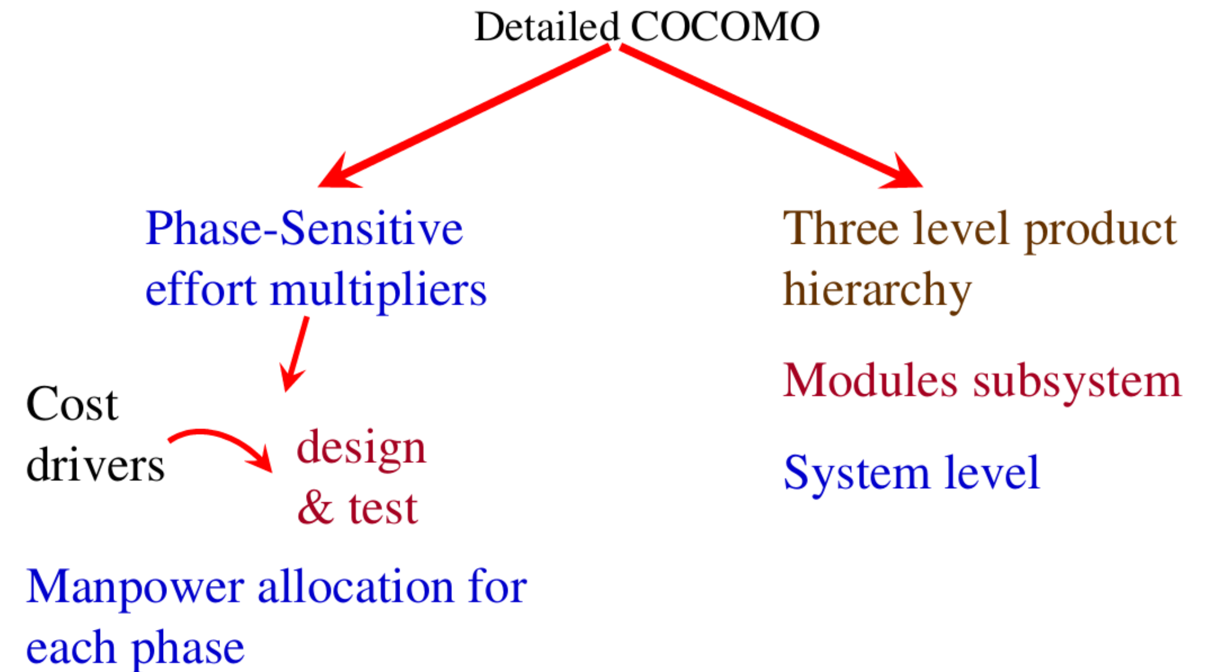
Project	a_i	b_i	c_i	d_i
Organic	3.2	1.05	2.5	0.38
Semidetached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

Table 6: Coefficients for intermediate COCOMO

UPES UNIVERSITY OF TOMORROW



Detailed COCOMO Model



① Plans / Requirement Phase.

This is the first phase of the SDLC development cycle the requirement is gathered then analyise and finally the

Date:

product plan is created and setup for developing the project and finally full product specification is generating.

This phase consume approximately from 6% to 8% of the total effort and 10% to 40% of the development time.

(2) Product Design *

The second phase of the Co-co mo development cycle is concern with the determination of the product architecture and the specification of the sub system.

This phase requires from 16 to 18 % total efforts and 19 to 38 % of the total development time.

UPES
UNIVERSITY OF TOMORROW

③ Programming / Implementation phase —

The third phase of the Co.Co.Mo development cycle is divided in to two sub phases

① detailed design
② Coding and unit testing.

This phase requires 48 to 68% of the efforts and 28% to 64% of the development time.

(4) Integration / testing phase *.

This the 4th [illegible] the development cycle occurs before of delivering the product in this phase we integrate all the modules together and then testing the final product.

This phase requires 16% to 34% of the efforts and 18 to 34% of the development time.

Principle of the effort estimate

Size equivalent

As the software might be partly developed from software already existing (that is, re-usable code), a full development is not always required. In such cases, the parts of design document (DD%), code (C%) and integration (I%) to be modified are estimated. Then, an adjustment factor, A, is calculated by means of the following equation.

$$A = 0.4\ DD + 0.3\ C + 0.3\ I$$

The size equivalent is obtained by

$$S\ (\text{equivalent}) = (S \times A) / 100$$

$$E_p = \mu_p E$$

$$D_p = \tau_p D$$

Lifecycle Phase Values of μ_p

Mode & Code Size	Plan & Requirements	System Design	Detailed Design	Module Code & Test	Integration & Test
Organic Small S≈2	0.06	0.16	0.26	0.42	0.16
Organic medium S≈32	0.06	0.16	0.24	0.38	0.22
Semidetached medium S≈32	0.07	0.17	0.25	0.33	0.25
Semidetached large S≈128	0.07	0.17	0.24	0.31	0.28
Embedded large S≈128	0.08	0.18	0.25	0.26	0.31
Embedded extra large S≈320	0.08	0.18	0.24	0.24	0.34

Table 7 : Effort and schedule fractions occurring in each phase of the lifecycle

Lifecycle Phase Values of τ_p

Mode & Code Size	Plan & Requirements	System Design	Detailed Design	Module Code & Test	Integration & Test
Organic Small S≈2	0.10	0.19	0.24	0.39	0.18
Organic medium S≈32	0.12	0.19	0.21	0.34	0.26
Semidetached medium S≈32	0.20	0.26	0.21	0.27	0.26
Semidetached large S≈128	0.22	0.27	0.19	0.25	0.29
Embedded large S≈128	0.36	0.36	0.18	0.18	0.28
Embedded extra large S≈320	0.40	0.38	0.16	0.16	0.30

Table 7 : Effort and schedule fractions occurring in each phase of the lifecycle

Date:

Question: Calculate the cost and scheddaule for different phases —

Rember

$E_p = \mu_p \cdot E$

$D_p = \tau_p \cdot D$

$E = 52.48$ PM

$D = 11.26$

where μ_p and τ_p are the constants in each phase of the life cycle.

	Plan & Req.	System Design	Programing	Integration
μ_p	0.06	0.16	0.42	0.16
τ_p	0.10	0.19	0.39	0.18

Plan & Req.

$\mu_p = 0.06 \times 52.48$

$\tau_p = 0.10 \times 11.2$

System Design

$\mu_p = 0.16 \times 52.48$

$\tau_p = 0.19 \times 11.26$

Programing

$\mu_p = 0.42 \times 52.48$

$\tau_p = 0.39 \times 11.26$

Integration / Test

$\mu_p = 0.16 \times 52.48$

$\tau_p = 0.18 \times 11.26$

COCOMO-II

- COCOMO-II is the revised version of the original COCOMO (discussed in article 4.4) and is developed at University of Southern California under the leadership of Dr. Barry Boehm.

- The model is tuned to the life cycle practices of the 21st century.

- It also provides a quantitative analytic framework, and set of tools and techniques for evaluating the effects of software technology improvements on software life cycle costs and schedules.

- The following categories of applications/projects are identified by COCOMO-II for the estimation [UCSD01] and are shown in Fig. 4.4.

COCOMO-II

COCOMO-II

The following categories of applications / projects are identified by COCOMO-II and are shown in fig. 4 shown below:

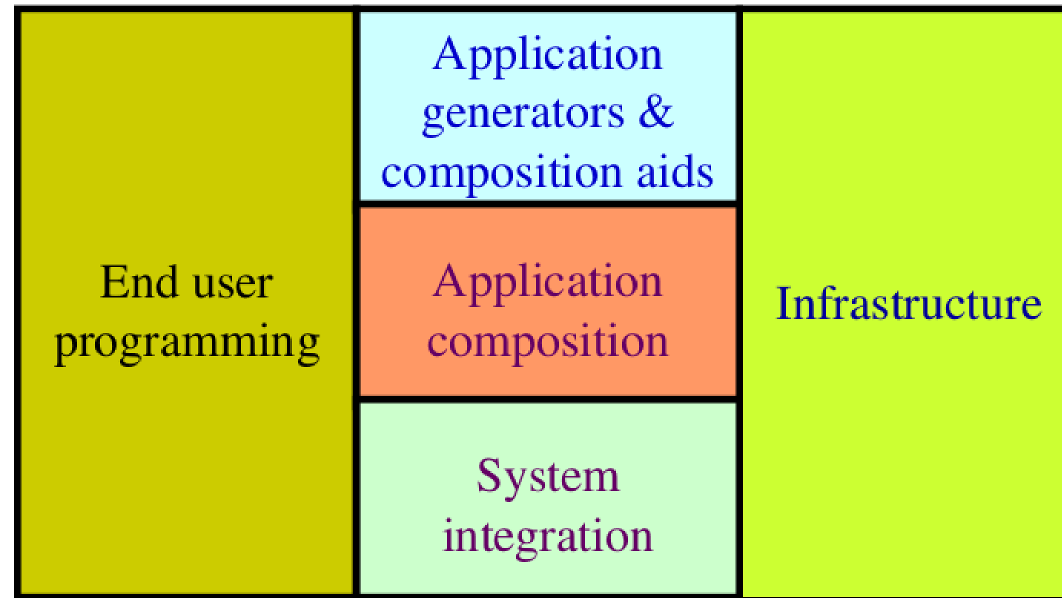


Fig. 4 : Categories of applications / projects

COCOMO-II

Stage No	Model Name	Application for the types of projects	Applications
Stage I	Application composition estimation model	Application composition	In addition to application composition type of projects, this model is also used for prototyping (if any) stage of application generators, infrastructure & system integration.
Stage II	Early design estimation model	Application generators, infrastructure & system integration	Used in early design stage of a project, when less is known about the project.
Stage III	Post architecture estimation model	Application generators, infrastructure & system integration	Used after the completion of the detailed architecture of the project.

Table 8: Stages of COCOMO-II

thank
you