AMITY UNIVERSITY

MADHYA PRADESH

Module II

Software Measurement

- Software measurement is the quantitative process of determining the size, complexity, quality, or cost of software products and processes to support data-driven decision-making. Governed by ISO standards, it involves collecting metrics on projects to improve productivity, estimate resources, and manage development, using both direct (e.g., lines of code) and indirect (e.g., maintainability) methods.
- Software metrics provide quantitative, reproducible measurements for project planning, cost estimation, and quality control, covering size (LOC, tokens, function points), design, data structures, and information flow. Key estimation models include COCOMO and Putnam, while effective risk management minimizes project delays and defects.

Types of Metrics:

- **Product Metrics:** Measure characteristics of the software itself (e.g., size, complexity, functionality, quality).
- **Process Metrics:** Measure characteristics of the development process (e.g., defect removal efficiency, development speed).
- **Project Metrics:** Measure resources and project status (e.g., cost, effort, schedule).

Methods:

- Direct Measures: Countable, such as lines of code, execution speed, and cost.
- Indirect Measures: Evaluative, such as complexity, functionality, and maintainability.

Examples:

- Size: Lines of Code (LOC), Function Points (FP).
- Complexity: Cyclomatic complexity (logic flow), Halstead complexity (operators/operands).

Why Measure Software?

- **Control:** Monitoring if a project is on time and within budget.
- **Improvement:** Identifying bottlenecks to enhance productivity and quality.
- **Prediction:** Estimating effort, cost, and defect rates for future projects.

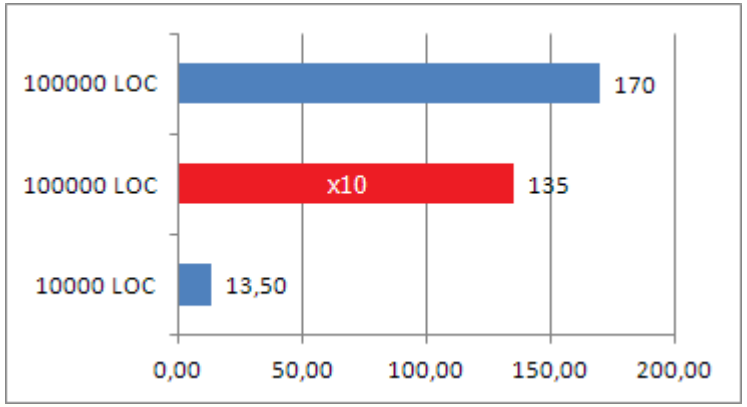
Software Size Metrics

These quantify the volume of software to estimate effort.

- LOC (Lines of Code): Simple, language-dependent measure of source instructions, excluding comments/blank lines.
- Token Count: Treats programs as a series of tokens (operators/operands) to measure software science complexity.
- Function Count (Function Points - FP): Measures functionality from a user's perspective, independent of programming language. It counts Inputs, Outputs, Inquiries, Files, and Interfaces, adjusted by complexity factors.

Lines of Code (LOC)

- Lines of Code (LOC) is a,size-oriented software metric that counts the total lines in source code to estimate project size, cost, and developer productivity.
- It generally excludes comments and blank lines, focusing on executable and non-executable statements. While easy to use, it is language-dependent and can encourage poor code quality (bloat).



Lines of Code (LOC) Metrics

LOC measures the physical or logical size of a program, often used for productivity, cost, and maintenance estimation.

- Physical Lines (LOC): Counts every line in the source file, including comments and blank lines.
- Logical Lines of Code (LLOC/NCLOC): Counts the number of executable statements or declarations, often language-dependent.
- **Common Units:** Often measured in KLOC (Thousands of Lines of Code)

LOC typically fall into three categories for cost estimation: **Three-Point Estimation**, **Productivity/Cost Calculations**, and **Conversion from Function Points**.

Three-Point Estimation (PERT)

Since it is hard to guess the exact LOC early in a project, planners often use a weighted average of three values: **Optimistic (S_{opt})**, **Most Likely (S_m)**, and **Pessimistic (S_{pess})**.

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

Numerical Example:

- **Optimistic:** 10,000 LOC
- **Most Likely:** 12,000 LOC
- **Pessimistic:** 18,000 LOC

Calculation:

$$S = \frac{10,000 + 4(12,000) + 18,000}{6} = \frac{76,000}{6} = \mathbf{12,667}\ \text{LOC}$$

Productivity and Cost

These problems ask you to calculate the total cost or effort based on historical productivity rates.

Typical Metrics:

- **Productivity** = $\frac{LOC}{\text{Person-Month}}$
- **Cost per Line** = $\frac{\text{Total Cost}}{LOC}$

Numerical Example:

A project is estimated at **33,200 LOC**. Historical data shows the team produces **620 LOC/pm** (person-month) and the labor rate is **$8,000 per month**.

1. **Effort** = $33,200/620 \approx \mathbf{53.5}\ \text{Person-Months}$
2. **Total Cost** = $53.5\ \text{PM} \times \$8,000 = \mathbf{\$428,000}$

Disadvantages:

- **Language Dependency:** Different languages require varying line counts to achieve the same functionality.
- **Quality Issues:** High LOC can promote inefficient "code bloat" rather than concise, maintainable code.
- **Refactoring:** Code optimization often reduces LOC, which can inaccurately imply lower productivity.

Token Count

- Token count in software engineering is a size-oriented metric measuring code volume by counting individual lexical units—operators and operands—rather than lines of code (LOC).
- Halstead's Software metrics are a set of measures proposed by Maurice Halstead to evaluate the complexity of a software program. These metrics are based on the number of distinct operators and operands in the program and are used to estimate the effort required to develop and maintain the program.

Token Count (Halstead Metrics)

It is a key component of Halstead's Software Metrics used to estimate program length, complexity, vocabulary and volume, often providing a more precise, language-independent measure of code size than LOC.

- Tokens: Unique operators (symbols, keywords) and operands (variables, constants).
- Halstead Volume (V): Measures the size of the implementation (V = N X log(n)), where N is total tokens and n is unique tokens.
- Application: Used to estimate complexity, effort, and error rates.

Key Aspects of Token Count Metric:

Definition: Tokens are the smallest functional units in code, classified into operands (variables, constants, labels) and operators (arithmetic symbols, keywords like if/for, delimiters).

Formula Components:

- n1: Number of unique operators(eg, +, if, float, ()).
- n2: Number of unique operands (e.g, x, temp, 100).
- N1: Total occurrences of operators.
- N2: Total occurrences of operands.

Measurements Derived:

- Vocabulary (n): n1+n2.
- Program Length (N) : N1 +N2
- Volume (V): N X log (n)

Numerical Example

Problem: Calculate the Vocabulary and Token Count for the following C++ snippet:

```
int main() {
    int a, b, sum;
    a = 10;
    b = 20;
    sum = a + b;
}
```

Step 1: Identify Operators (n1 and N1)

Operator	Occurrences
int	2
main	1
()	1
{}	1
,	2
;	4
=	3
+	1
Totals	**$n_1 = 8$ (Distinct)**

Cont.

- Step 2: Identify Operands (n2 and N2)

Operand	Occurrences
a	3
b	3
sum	2
10	1
20	1
Totals	**n_2 = 5 (Distinct)**

Cont.

- Step 3: Final Results

  Vocabulary (n): 8 + 5 = 13

  Total Token Count (N): 15 + 10 = 25

3. Advanced Halstead Metrics

Once you have the token counts, you are often asked to calculate the "Estimated" length and Volume:

1. **Estimated Length ($\hat{N}$):**

$$\hat{N} = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

*(This predicts what the token count **should** be based on your vocabulary).*

2. **Volume (V):**

$$V = N \times \log_2 n$$

(This represents the size of the program in bits).

- **Estimated Program Level / Difficulty**

Halstead offered an alternate formula that estimate the program level.

$$\hat{L} = 2\eta_2 / (\eta_1 N_2)$$

where

$$\hat{D} = \frac{1}{\hat{L}} = \frac{\eta_1 N_2}{2\eta_2}$$

- **Effort and Time**

$$E = V / \hat{L} = V * \hat{D}$$

$$= (n_1 N_2 N \log_2 \eta) / 2\eta_2$$

$$T = E / \beta$$

β is normally set to 18 since this seemed to give best results in Halstead's earliest experiments, which compared the predicted times with observed programming times, including the time for design, coding, and testing.

Function Count & Functional Metrics

The term **"function count size metric"** refers primarily to the **Function Point (FP) metric**, a standardized unit of measurement in software engineering used to quantify the amount of business functionality a software system provides from a user's perspective. It serves as a language-independent and technology-neutral measure of software size, often used for project estimation and productivity comparison.

- **Function Point Analysis (FP):** Counts user inputs, outputs, inquiries, internal files, and external interfaces, weighted by complexity.
- **Feature Point Analysis:** A subset of Function Points, often used for systems where algorithm complexity is high.
- **Function Count:** A simpler metric estimating the total number of distinct modules or functions in the program.

Key Concepts

- **Function Point (FP):** A metric that measures the size of the software based on the number and complexity of the functions or transactions it provides to the user.
- **User-Oriented:** Unlike Lines of Code (LOC), which is design-oriented and dependent on the programming language, the FP metric is user-oriented, focusing on what the user requests and receives.
- **Purpose:** FPs are used to estimate the effort, cost, and duration required for a software project at an early stage in the software development lifecycle, even before coding begins.

Components of Function Point Analysis

- Function point analysis involves counting and categorizing the software's functional requirements into five information domain characteristics, which are then weighted based on their complexity (simple, average, or complex).

The five components are:

- External Inputs (EI): The number of unique data or control information input from external sources that enter the application boundary.
- External Outputs (EO): The number of unique data or control information outputs sent from the application to external sources (e.g., reports, screen messages, error messages).
- External Inquiries (EQ): Interactive queries that retrieve data from internal logical files and present it as an output, without changing the system's data.
- Internal Logical Files (ILF): A logically related group of data or control information maintained within the boundary of the application (e.g., a database table or a major data file).
- External Interface Files (EIF): A logically related group of data or control information referenced by the application but maintained by another application's internal logical file.

The Standard Weighting Table: UFP

Measurement Parameter	Low	Average	High
Number of external inputs (EI)	3	4	6
Number of external outputs (EO)	4	5	7
Number of external inquiries (EQ)	3	4	6
Number of internal files (ILF)	7	10	15
Number of External Interfaces (EIF)	5	7	10

FP = Unadjusted Function Points (UFP) X Value Adjustment Factor (VAF)

- ISO/IEC 20926
- We categorize the system's components into five types and weigh them based on complexity (Low, Average, High).

Step B: Calculate Value Adjustment Factor (VAF)

This accounts for 14 General System Characteristics (GSCs) like data communications, performance, and reusability. Each is rated from 0 (no influence) to 5 (strong influence).

$$VAF = 0.65 + (0.01 \times \sum F_i)$$

(Where $\sum F_i$ is the sum of the 14 ratings, ranging from 0 to 70).

Numerical Example

You are building a small Library Management System with the following specs:

- **External Inputs:** 2 (Simple)
- **External Outputs:** 1 (Average)
- **External Inquiries:** 2 (Simple)
- **Internal Logical Files:** 1 (High complexity)
- **External Interface Files:** 0
- **Sum of 14 Factors ($\sum F_i$):** 35 (Average complexity environment)

Function Point Calculation Explained Numerically

Calculation:

1. Calculate UFP:

- $EI = 2 \times 3 = 6$
- $EO = 1 \times 5 = 5$
- $EQ = 2 \times 3 = 6$
- $ILF = 1 \times 15 = 15$
- $EIF = 0 \times 7 = 0$
- **Total UFP** $= 6 + 5 + 6 + 15 + 0 = \mathbf{32}$

2. Calculate VAF:

- $VAF = 0.65 + (0.01 \times 35)$
- $VAF = 0.65 + 0.35 = \mathbf{1.0}$

3. Final Function Point (FP):

- $FP = 32 \times 1.0 = \mathbf{32}$

If you know that your team's productivity is, for example, 5 person-hours per FP, you can now estimate the project timeline: 32 FP x 5 hours/FP = 160 hours of effort.

standard conversion ratios for different languages

- Function Point to LOC

Language	Average LOC/FP
C	128
C++	64
Java	53
Python	40
SQL	12

Numerical Example: If your software has 15.4 FP and you are writing it in C++:
Estimated LOC = 15.4 X 64 = 985.6 LOC

Design Metrics

- Design metrics measure the complexity, structure, and quality of the system architecture rather than just the code volume.
- **Structural Coupling:** Analyzes the interaction between modules (e.g., control, data, content coupling) via static analysis.
- **Information Flow Metrics:** Measure the complexity of connections and information transfer between different components.
- **Cyclomatic Complexity** (V(G)): Measures the number of linearly independent paths through a program's source code, assessing logical complexity.

Data Structure Metrics

Data structure metrics provide insight into the amount of data a software system processes (input, internal, and output). These metrics help estimate the effort and time needed for project completion and reveal potential weaknesses in a program's structure.

Key data structure metrics and concepts include:

- Amount of Data: Measures the volume of data in a module, often quantified by the number of variables (VARS), unique operands (η2), and the total occurrences of variables (N2).
- Usage of Data within a Module: Evaluates how data is used internally, typically by calculating the average number of live variables (variables that hold a value which may be needed in the future) within a procedure.
- Sharing of Data Among Modules: Assesses data coupling between modules. Increased data sharing (higher coupling) often means more parameters need to be passed, increasing complexity and the potential for errors.
- Program Weakness: Relates to the lack of cohesion in modules. Weak modules require more effort and time to complete the project.

Information flow metrics

Information flow metrics, also known as Henry and Kafura's Metrics, measure code complexity by analyzing the passage of information among system components or modules. These metrics focus on the interconnections and dependencies between different parts of the system and are effective for identifying potential design flaws and predicting changes.

Key information flow metrics and concepts include:

- FAN-IN: A count of the number of other components that can call or pass information to a specific component.
- FAN-OUT: The number of components that are called by or receive information from a specific component.
- Information Flow Index (IF(A)): A measure of a component's complexity derived from its fan-in and fan-out.
- **Coupling and Cohesion:** These metrics are directly related to the principles of coupling (degree of linkage between modules) and cohesion (degree to which elements within a module are functionally related). High fan-in and fan-out often indicate weak cohesion and excessive coupling, which can lead to maintenance difficulties.

Cost Estimation Model

- Software cost estimation models are mathematical algorithms or empirical methods used in software engineering to predict the effort (person-hours/months), duration, and cost required to develop a project.
- In starting a new **software project**, it is important to know how much will it cost to develop and how much development time will it take to complete.
- These evaluations are needed before development is started and conveyed to the team. The software industry has inconsistently defined and explained metrics or atomic units of measure, data from real and actual projects are largely and highly suspect in terms of consistency and comparability.

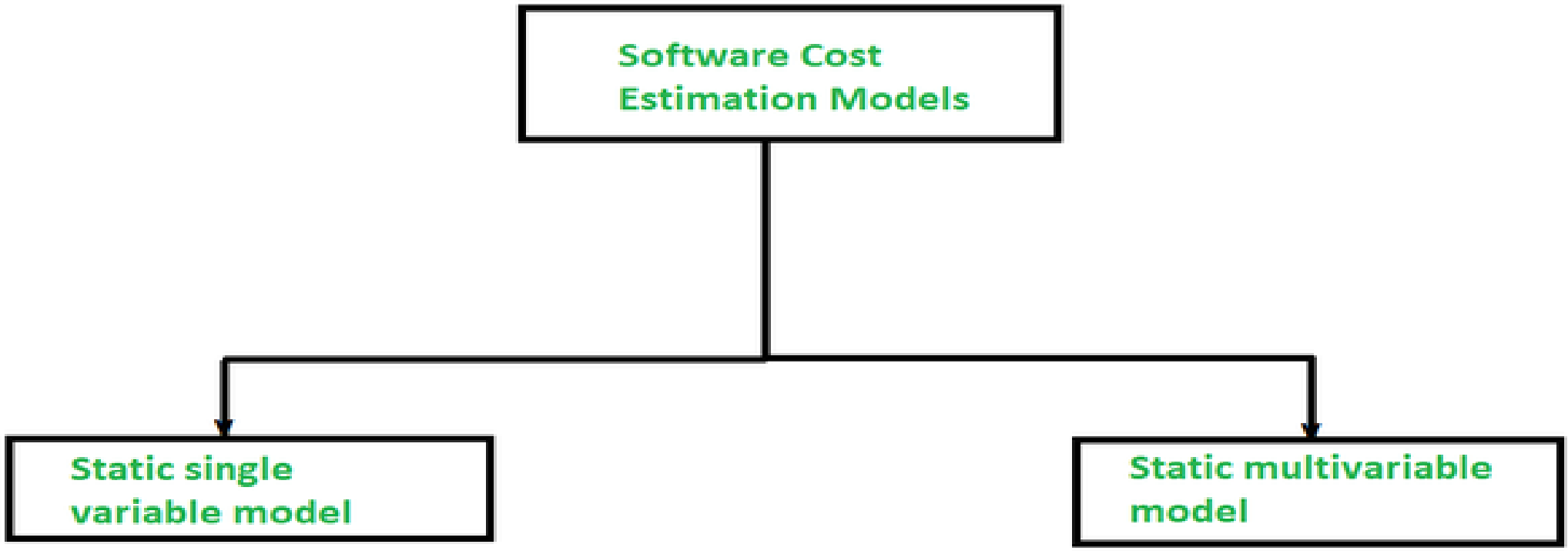
Common characteristics:

- The scope of the software project must be created before starting to develop the software.
- Metrics like FP or LOC are used for assessing the software.
- To achieve the targeted cost & schedule estimate, several things arise.
- Obtain one or more cost and effort of a project.

Static single variable model

- Methods using this model utilize an equation to estimate the desired value such as time, cost and effort and so on they all depend on same variable used as predictor (say, size). An illustration of the most common equation is **C= a Lb**
- Where C is the cost effort expressed in the unit of manpower for example persons months and L is the size generally given in the line of code (LOC). The constants a as well as b are derived from the historical data of the organization. Since a as well as b depends on the local development environment these models are not transportable to different organizations.

Types


Software Cost Estimation Models
Static single variable model
Static multivariable model

Static Single Variable model

- The Methods using this model utilize an equation to get the desired values such as cost, time, effort, etc. And these all depend on the same variable used as a predictor like, size. Below is an example of the most common equation:

$$C = aL^b$$

- Where C is cost, L is size and a, b are constants.
- We have an example of the static single-variable mode. e. i.e. **SEL model** which is used for estimating software production. The equation of this model is given below:

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

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

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

- Where E is in Person-months, DOC i.e, documentation is in the number of pages and is duration which is months.

Static Multivariable model

- These models are also known as **multivariable models.** This model is often based on the first equation and depends on several variables representing different aspects of the software development environment.
- Equations are:

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

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

- Where E is in Person-months, D is duration which is months.

COCOMO (Constructive Cost Model)

- The COCOMO (Constructive Cost Model), developed by Barry Boehm in 1981.
- It is an algorithmic, regression-based model used to estimate software project effort, cost, and schedule based on the size of the code (typically in KLOC - Thousands of Lines of Code).
- It classifies projects into organic, semi-detached, or embedded types and exists in three levels of accuracy: Basic, Intermediate, and Detailed.
- The key parameters that define the quality of any **Software Product**, which are also an outcome of COCOMO, are primarily effort and schedule

Phases Of COCOMO Model

Cont.

- **Planning and requirements:** This initial phase involves defining the scope, objectives, and constraints of the project. It includes developing a project plan that outlines the schedule, resources, and milestones
- **System design:** : In this phase, the high-level architecture of the software system is created. This includes defining the system's overall structure, including major components, their interactions, and the data flow between them**.**
- **Detailed design:** This phase involves creating detailed specifications for each component of the system. It breaks down the system design into detailed descriptions of each module, including data structures, algorithms, and interfaces.

Cont.

- **Module code and test:** This involves writing the actual source code for each module or component as defined in the detailed design. It includes coding the functionalities, implementing algorithms, and developing interfaces.
- **Integration and test:** This phase involves combining individual modules into a complete system and ensuring that they work together as intended.
- **Cost Constructive model:** The Constructive Cost Model (COCOMO) is a widely used method for estimating the cost and effort required for software development projects.

Types of Projects in COCOMO Model

- In the COCOMO model, software projects are categorized into three types based on their complexity, size, and the development environment. These types are:

1. Organic

- A software project is said to be an organic type if the team size required is adequately small, the problem is well understood and has been solved in the past and also the team members have a nominal experience regarding the problem.

2. Semi-detached

- A software project is said to be a Semi-detached type if the vital characteristics such as team size, experience, and knowledge of the various programming environments lie in between organic and embedded.
- The projects classified as Semi-Detached are comparatively less familiar and difficult to develop compared to the organic ones and require more experience better guidance and creativity. Eg: Compilers or different Embedded Systems can be considered Semi-Detached types.

3. Embedded

- A software project requiring the highest level of complexity, creativity, and experience requirement falls under this category. Such software requires a larger team size than the other two models and also the developers need to be sufficiently experienced and creative to develop such complex models.

Comparision

Aspects	Organic	Semidetached	Embedded
Project Size	2 to 50 KLOC	50-300 KLOC	300 and above KLOC
Complexity	Low	Medium	High
Team Experience	Highly experienced	Some experienced as well as inexperienced staff	Mixed experience, includes experts
Environment	Flexible, fewer constraints	Somewhat flexible, moderate constraints	Highly rigorous, strict requirements
Effort Equation	$E = 2.4(400)^{1.05}$	$E = 3.0(400)^{1.12}$	$E = 3.6(400)^{1.20}$
Example	Simple payroll system	New system interfacing with existing systems	Flight control software

Numerical

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:

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

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

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

- Sample question

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.

Solutions

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

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

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

(ii) Semidetached mode

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

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

(iii) Embedded mode

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

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

SOLVE

- 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.

Advantages

- **Cost Estimation:** To help with resource planning and project budgeting, COCOMO offers a methodical approach to software development cost estimation.
- **Resource Management:** By taking team experience, project size, and complexity into account, the model helps with efficient resource allocation.
- **Project Planning**: COCOMO assists in developing practical project plans that include attainable objectives, due dates, and benchmarks.
- **Risk management**: Early in the development process, COCOMO assists in identifying and mitigating potential hazards by including risk elements.
- **Support for Decisions**: During project planning, the model provides a quantitative foundation for choices about scope, priorities, and resource allocation.
- **Benchmarking**: To compare and assess various software development projects to industry standards, COCOMO offers a benchmark.
- **Resource Optimization:** The model helps to maximize the use of resources, which raises productivity and lowers costs.

Putnam Resource Allocation Model

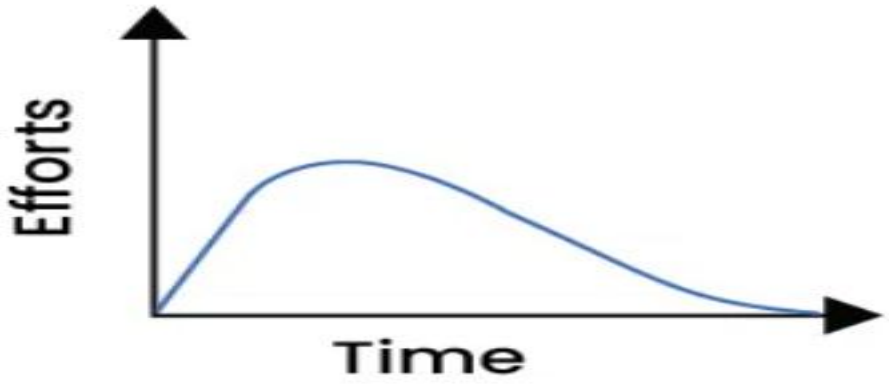
- The Putnam Resource Allocation Model also known as the Putnam Model is one of the models that are helpful in forecasting and resource distribution.
- Created by Lawrence H. Putnam at the end of the 1970s, this model is used to define the rough order of magnitude that has to be set as resources and schedule for a software development project.
- It is founded on the dynamics of software project on the similar name, and it is especially useful for large and intricate software projects.

Cont.

- The Putnam Model is a formal tool used to quantify the cost, time, and effort required for software project development.
- It operates on the principle that there is a trade-off between the time available and the resources (effort) needed to complete a project. The model uses the Rayleigh curve to illustrate how effort should be distributed over the project timeline.
- The Rayleigh curve represents the distribution of effort over time, showing that effort typically increases to a peak and then decreases as the project progresses.
- The core concept of the Putnam Model is that the rate at which resources, including manpower, are applied to the project impacts both the project's costs and its delivery time.

Rayleigh Curve

- Here, the X-axis takes time and the Y-axis takes effort in man hours for instance, months. Effort thus ranges upwards as the project advances and then may drop down as the project approaches its final stages, according to the curve. The shape of the curve implies the total work content necessary to complete the undertaking.



Software Life Cycle Model (SLIM)

- The SLIM model uses the following key equation, known as the Putnam-Norden-Rayleigh (PNR) equation, to estimate effort:

$$S = Ck^{1/3}t^{4/3}$$

- ***Where:***
- *S is the software size (in LOC, function points, etc.).*
- *C is the technology factor, reflecting the capabilities of the development environment and team.*
- *k is the effort coefficient, a constant depending on the project environment.*
- *t is the time to complete the project (in months or years).*

Efforts

- *E(t) is the effort distribution over time.*
- *Ex:* Let's assume the initial project plan estimates a duration t_1 of 12 months. However, due to market demands, the project schedule is shortened to t_2 of 8 months.
- Using the PNR equation, the initial effort E_1 and the revised effort E_2 can be calculated:

$$E_1 = ck^{1/3}(t_1)^{4/3}$$

$$E_1 = ck^{1/3}(t_2)^{4/3}$$

Cont.

Assume:

- $C = 1$
- $k = 1$
- *Initial duration* $t_1 = 12$
- *New duration* $t_2 = 8$
- ***Initial Effort*** E_1***:*** $E_1 = 1 \cdot 1^{1/3} \cdot 12^{4/3} \approx 22.63$ *person-months*
- ***Revised Effort*** E_2***:*** $E_2 = 1.1^{1/3} \cdot 8^{4/3} \approx 11.31$ *person-months*
- By shortening the schedule from 12 months to 8 months, the effort has increased significantly, illustrating the model's principle that compressing the schedule results in higher costs.

Risk management

- Risk management in software engineering is the proactive process of identifying, analyzing, prioritizing, and mitigating potential threats (technical, project, or business) to ensure project success. It involves anticipating risks like scope creep or developer turnover, establishing contingency plans, and monitoring them to minimize negative impacts on cost, quality, and schedule.

Components of Risk Management

- Risk Identification: Uncovering potential risks through checklists, brainstorming, or analyzing previous projects.
- Risk Analysis & Assessment: Evaluating the probability, impact, and severity of each risk.
- Risk Mitigation & Planning: Developing strategies to reduce the likelihood or impact of risks (e.g., prototyping, training).
- Risk Monitoring & Control: Tracking identified risks and implementing contingency plans if they occur.

Common Types of Software Risks

- **Project Risks:** Include budget overruns, schedule delays, and resource shortages.
- **Technical Risks:** Involve, for example, using unproven technology, complex code, or poor performance.
- **Business Risks:** Risks that threaten the product's viability, such as building the wrong product or losing management support.
- **Requirements Risks:**

The END