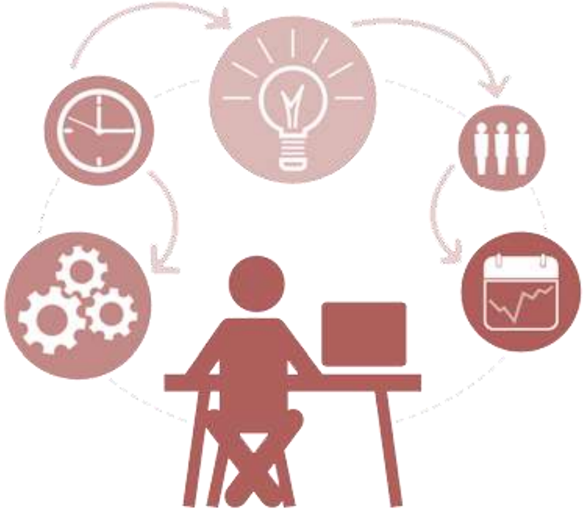
Managing Software Projects

Outline

- W^5 of Project Management
- Software Metrics
  - Process, Product and Project Metrics
- Software Project Estimations
- Software Project Planning (MS Project Tool)
- Project Scheduling and Tracking

Software Project Management



W^5HH of Project Management

Boehm suggests **an approach (W^5HH)** that addresses project objectives, milestones and schedules, responsibilities, management and technical approaches, and required resources

Why is the system being developed?
Enables all parties to assess the validity of business reasons for the software work. In another words - does the business purpose justify the expenditure of people, time, and money?

What will be done?
The **answers** to **these questions** help the team to **establish** a **project schedule** by **identifying key** project **tasks** and the **milestones** that are required by the customer

When will it be accomplished?
Project schedule to achieve milestone

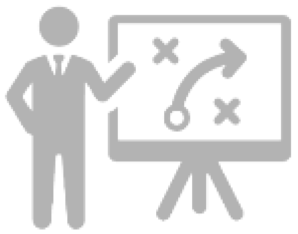
W^{5}HH of Project Management Cont.

Who is responsible?
Role and responsibility of each member

Where are they organizationally located?
Customer, end user and other stakeholders also have responsibility

How will the job be done technically and managerially?
Management and technical strategy must be defined

How much of each resource is needed?
Develop estimation



W^{5}HH

It is applicable **regardless** of **size** or **complexity** of software **project**

Terminologies

Measure

- It provides a **quantitative indication** of the extent (range), amount, dimension, capacity or size of some attributes of a product or process
- Ex., the number of uncovered errors

Metrics

- It is a **quantitative measure** of the degree (limit) to which a system, component or process possesses (obtain) a given attribute
- It **relates individual measures** in some way
- Ex., number of errors found per review

Direct Metrics

- **Immediately measurable attributes**
- Ex., Line of Code (LOC), Execution Speed, Defects Reported

Indirect Metrics

- **Aspects that are not immediately quantifiable**
- Ex., Functionality, Quantity, Reliability

Indicators

- It is a **metric or combination of metrics** that provides insight into the software process, project or the product itself
- It **enables** the **project manager** or software engineers **to adjust** the process, the project or the product to make things better
- Ex., Product Size (analysis and specification metrics) is an indicator of increased coding, integration and testing effort

Faults

- **Errors** - Faults found by the practitioners during software development
- **Defects** - Faults found by the customers after release

Why Measure Software?

1. To **determine** (to define) **quality** of a product or process.
2. To **predict qualities** of a product or process.
3. To **improve quality** of a product or process.

Metric Classification Base

- **Process**
  - Specifies **activities related to production** of software.
  - Specifies the abstract set of activities that should be performed to go from user needs to final product.
- **Project**
  - **Software development work** in which a software process is used
  - The actual act of executing the activities for some specific user needs
- **Product**
  - **The outcomes of** a software **project**
  - All the outputs that are produced while the activities are being executed

Process Metrics

- Process Metrics are an invaluable **tool** for companies to monitor, **evaluate** and **improve** their **operational performance** across the enterprise
- They are **used** for making **strategic decisions**
- Process **Metrics** are **collected across all projects** and over **long periods of time**
- Their **intent** is to **provide a set of process indicators** that lead to long-term software **process improvement**

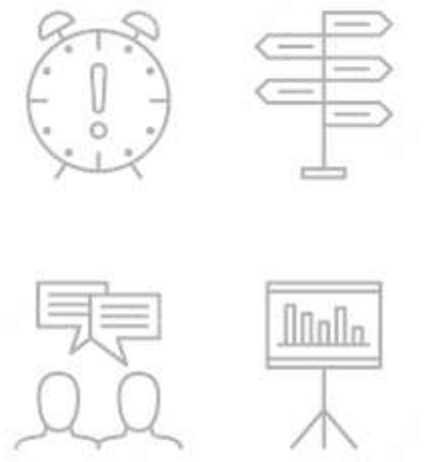
Ex., **Defect Removal Efficiency (DRE) metric**
*Relationship between **errors (E)** and **defects (D)***
The **ideal** is a **DRE** of **1**
DRE = E / (E + D)

We measure the effectiveness of a process by deriving a set of metrics based on outcomes of the process such as,

- **Errors uncovered** before release of the software
- **Defects delivered** to and reported by the end users
- **Work products** delivered
- **Human effort** expended
- **Calendar time** expended
- **Conformance** to the **schedule**
- **Time and effort** to **complete** each generic **activity**

Project Metrics

- Project **metrics enable** a software project **manager** to,
  - **Assess the status** of an ongoing project
  - **Track** potential **risks**
  - **Uncover problem areas** before their status becomes critical
  - **Adjust work flow** or tasks
  - **Evaluate** the project **team's ability** to **control quality** of software work products
- Many of the same metrics are used in both the process and project domain
- **Project metrics** are **used** for making **tactical (smart) decisions**
- They are **used** to adapt **project workflow** and **technical activities**
- Project metrics are used to
- **Minimize** the **development schedule** by making the **adjustments** necessary to avoid delays and **mitigate (to reduce)** potential (probable) problems and risks
- **Assess (evaluates) product quality** on an **ongoing basis** and guides to modify the technical approach to improve quality



Product Metrics

- Product metrics **help software engineers** to gain **insight into** the **design and construction** of the **software** they build
  - By **focusing** on specific, **measurable attributes** of software engineering work products
- Product metrics **provide** a **basis** from which **analysis, design, coding and testing can be conducted** more **objectively** and **assessed** more **quantitatively**
  - Ex., Code Complexity Metric

Types of Measures

Categories of Software Measurement

Direct measures of the	Indirect measures of the
Software process Ex., cost, effort, etc.	**Software product** Ex. functionality, quality, complexity, efficiency, reliability, etc.
Software product Ex., lines of code produced, execution speed, defects reported, etc.	

Software Measurement

Metrics for Software **Cost** and **Effort estimations**
Size Oriented Metrics
Function Oriented Metrics
Object Oriented Metrics
Use Case Oriented Metrics

Size-Oriented Metrics

- **Derived** by **normalizing** (standardizing) **quality** and/or **productivity** measures by **considering** the **size of the software** produced
- **Thousand lines of code (KLOC)** are often chosen as the normalization value

A set of simple size-oriented metrics can be developed for each project

- Errors per KLOC (thousand lines of code)
- Defects per KLOC
- $ per KLOC
- Pages of documentation per KLOC

In addition, other interesting metrics can be computed, like

- Errors per person-month
- KLOC per person-month
- $ per page of documentation

- **Size-oriented** metrics **are not universally accepted** as the best way **to measure the software** process

Opponents argue that KLOC measurements

- Are **dependent on the programming language**
- **Penalize well-designed** but **short programs**
- **Cannot** easily **accommodate nonprocedural** languages
- **Require** a **level of detail** that may be **difficult to achieve**

Function Oriented Metrics

- Function-oriented metrics use a measure of the **functionality delivered** by the application as a normalization value
- Most **widely used metric** of this type is the **Function Point**
  - **FP** = Count Total * [0.65 + 0.01 * Sum (Value Adjustment Factors)]
- Function Point **values on past projects** can be **used** to compute,
  - for example, the **average number of lines of code** per function point
- **Advantages**
  - FP is **programming language independent**
  - FP is based on **data** that are **more likely to be known in the early stages** of a project, making it more attractive as an estimation approach
- **Disadvantages**
  - FP **requires** some “**sleight of hand**” because the **computation** is based on **subjective data**
  - **Counts of the information** domain can be **difficult to collect**
  - FP has **no direct physical meaning**, it’s just a number

Object-Oriented Metrics

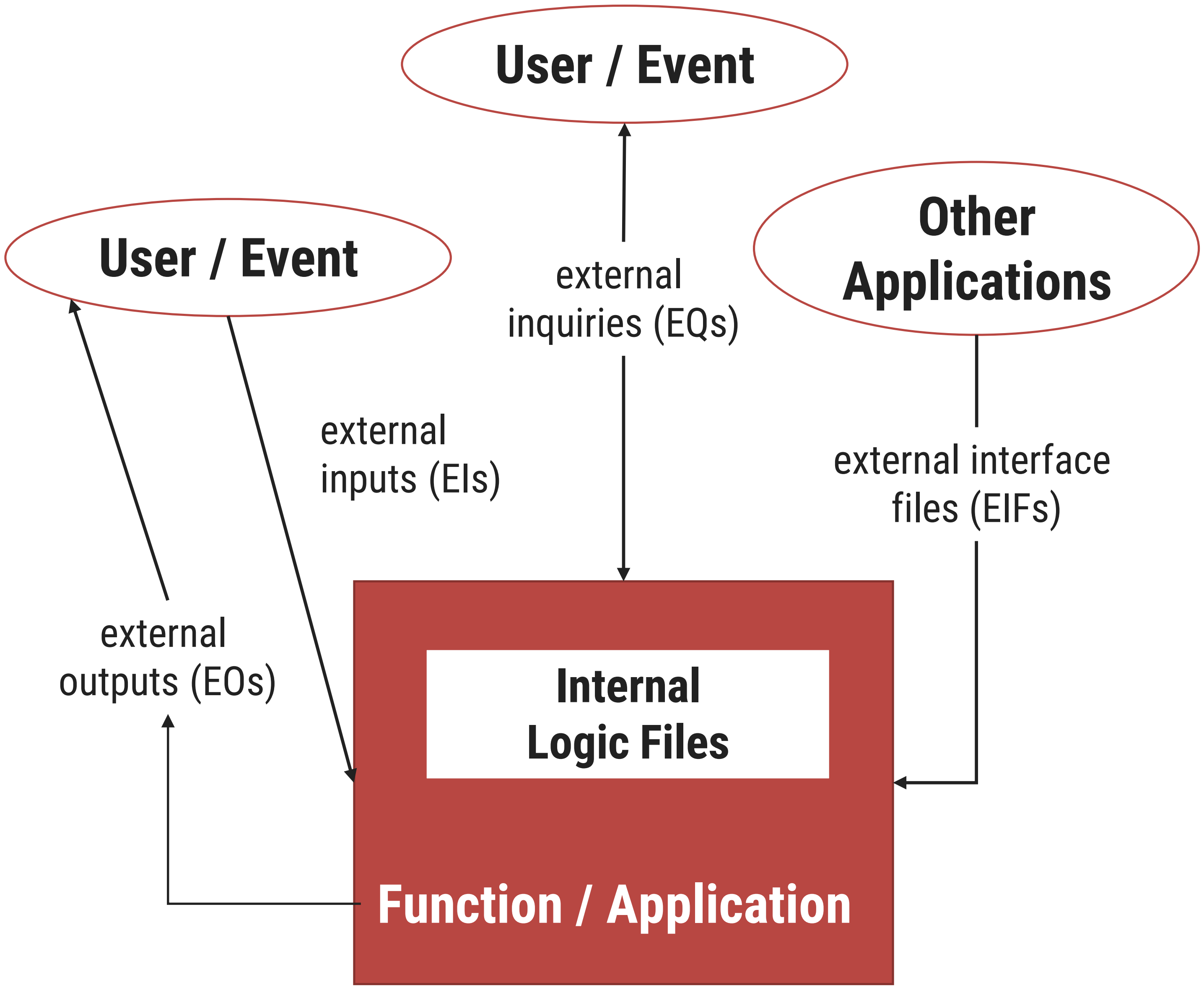
- **Conventional software project metrics** (LOC or FP) **can be used** to estimate object-oriented software projects
- However, these metrics **do not provide enough granularity** (detailing) for the schedule and effort adjustments that are required as you iterate through an evolutionary or incremental process
- Lorenz and Kidd suggest the following **set of metrics for OO projects**
  - Number of **scenario scripts**
  - Number of **key classes** (the highly independent components)
  - Number of **support classes**

Use Case Oriented Metrics

- Like FP, the **use case** is defined early in the software process, allowing it to be **used for estimation before significant** (valuable) **modeling** and **construction** activities are **initiated**
- **Use cases describe** (indirectly, at least) **user-visible functions** and **features** that are basic requirements for a system
- The **use case** is **independent of programming language**, because use cases can be created at vastly different levels of abstraction, there is no standard "size" for a use case
- **Without a standard measure** of what a use case is, its application as a normalization **measure is suspect** (doubtful).
  - Ex., effort expended / use case

Function Point Metrics

- The **function point (FP)** metric can be **used** effectively as a **means for measuring the functionality** delivered by a system
- Using **historical data**, the **FP metric** can be **used** to
  - **Estimate the cost** or **effort** required to design, code, and test the software
  - **Predict the number of errors** that will be encountered during testing
  - **Forecast** the **number of components** and/or the **number of projected source lines** in the implemented system



Function Point Components Cont.

Information domain values (components) are defined in the following manner

- **Number of external inputs (EIs)**
  - input data originates from a user or is transmitted from another application
- **Number of external outputs (EOs)**
  - external output is derived data within the application that provides information to the user
  - output refers to reports, screens, error messages, etc.
- **Number of external inquiries (EQs)**
  - external inquiry is defined as an online input that results in the generation of some immediate software response in the form of an online output
- **Number of internal logical files (ILFs)**
  - internal logical file is a logical grouping of data that resides within the application's boundary and is maintained via external inputs
- **Number of external interface files (EIFs)**
  - external interface file is a logical grouping of data that resides external to the application but provides information that may be of use to the another application

Compute Function Points

$$FP = \text{Count Total} * [\,0.65 + 0.01 * \sum(F_i)\,]$$

Count Total is the sum of all FP entries

Fi (i=1 to 14) are complexity value adjustment factors (**VAF**).

Value adjustment factors are used to provide an indication of problem complexity

Information Domain Value	Count		Weighting factor Simple	Average	Complex		
External Inputs (EIs)		×	3	4	6	=	
External Outputs (EOs)		×	4	5	7	=	
External Inquiries (EQs)		×	3	4	6	=	
Internal Logical Files (ILFs)		×	7	10	15	=	
External Interface Files (EIFs)		×	5	7	10	=	
Count total →							

Value Adjustment Factors

- F1. Data Communication
- F2. Distributed Data Processing
- F3. Performance
- F4. Heavily Used Configuration
- F5. Transaction Role
- F6. Online Data Entry
- F7. End-User Efficiency
- F8. Online Update
- F9. Complex Processing
- F10. Reusability
- F11. Installation Ease
- F12. Operational Ease
- F13. Multiple Sites
- F14. Facilitate Change

Function Point Calculation Example

Information Domain Value	Count		Weighting factor Simple	Average	Complex		
External Inputs (EIs)	3	×	3	4	6	=	9
External Outputs (EOs)	2	×	4	5	7	=	8
External Inquiries (EQs)	2	×	3	4	6	=	6
Internal Logical Files (ILFs)	1	×	7	10	15	=	7
External Interface Files (EIFs)	4	×	5	7	10	=	20
Count total						→	50

FP = Count Total * [0.65 + 0.01 * $\sum(F_i)$]

FP = [50]* [0.65 + 0.01 * 17]

FP = [50]* [0.65 + 0.17]

FP = [50]* [0.82] = 41

Used Adjustment Factors and assumed values are,

F09. Complex internal processing = **3**

F10. Code to be reusable = **2**

F03. High performance = **4**

F13. Multiple sites = **3**

F02. Distributed processing = **5**

Project Adjustment Factor (VAF) = 17

Function Point Calculation Example 2

Study of requirement specification for a project has produced following results

Need for **7 inputs**, **10 outputs**, **6 inquiries**, **17 files** and **4 external interfaces**

Input and **external interface function point** attributes are of **average complexity** and all **other function points** attributes are of **low complexity**

Determine **adjusted function points** assuming complexity **adjustment value is 32**.

Information Domain Value	Count		Weighting factor Simple	Average	Complex		
External Inputs (EIs)	7	×	3	4	6	=	28
External Outputs (EOs)	10	×	4	5	7	=	40
External Inquiries (EQs)	6	×	3	4	6	=	18
Internal Logical Files (ILFs)	17	×	7	10	15	=	119
External Interface Files (EIFs)	4	×	5	7	10	=	28
Count total →							233

Value adjustment factors **(VAF) = 32** given

FP = **Count Total * [0.65 + 0.01 * $\sum(F_i)$]**

= **233** * [0.65 + 0.01 * **32**]

= **233** * 0.97 = **226.01**

Software Project Estimation

It can be **transformed** from a **black art** to a **series of systematic steps** that provide **estimates** with **acceptable risk**

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

- Delay estimation **until late in the project** (obviously, we can achieve 100 percent accurate estimates after the project is complete!)
- Base **estimates** on **similar projects** that have already been completed
- Use relatively simple **decomposition techniques** to generate project cost and effort estimates
- **Use** one or more **empirical models** for software cost and effort estimation.

Software Project Decomposing

- Software **project estimation** is a form of **problem solving** and in most cases, the problem to be solved is **too complex to be considered in one piece**
- For this reason, **decomposing the problem**, re-characterizing it as a set of smaller problems is required
- Before an estimate can be made, the **project planner** must **understand the scope of the software** to be built and must generate an estimate of its "size"

Decomposition Techniques

1. Software Sizing
2. Problem based Estimation
   LOC (Lines of Code) based,
   FP (Function Point) based
3. Process based Estimation
4. Estimation with Use-cases

Software Sizing

Putnam and **Myers** suggest **four** different **approaches** to the **sizing problem**

- **"Fuzzy logic" sizing**
  - This approach uses **the approximate reasoning techniques** that are the cornerstone of fuzzy logic.
- **Function Point sizing**
  - The planner develops **estimates of the information domain characteristics**
- **Standard Component sizing**
  - Estimate the **number of occurrences** of each **standard component**
  - Use **historical** project **data** to determine the **delivered LOC** size per standard component.
- **Change sizing**
  - **Used** when **changes** are being **made** to **existing software**
  - Estimate the **number** and **type of modifications** that must be accomplished
  - An **effort ratio** is then **used** to **estimate** each **type of change** and the **size of** the **change**

Problem Based Estimation

- **Start** with a **bounded** statement of **scope**
- **Decompose** the **software** into **problem functions** that can each be **estimated individually**
- **Compute** an **LOC** or **FP** value for **each function**
- **Derive cost** or **effort estimates** by applying the **LOC** or **FP** values to your **baseline productivity metrics**
  - Ex., LOC/person-month or FP/person-month
- **Combine function estimates** to produce an **overall estimate** for the **entire project**
- In general, the **LOC/pm** and **FP/pm** metrics should be computed by project domain
  - **Important factors** are **team size**, **application area** and **complexity**

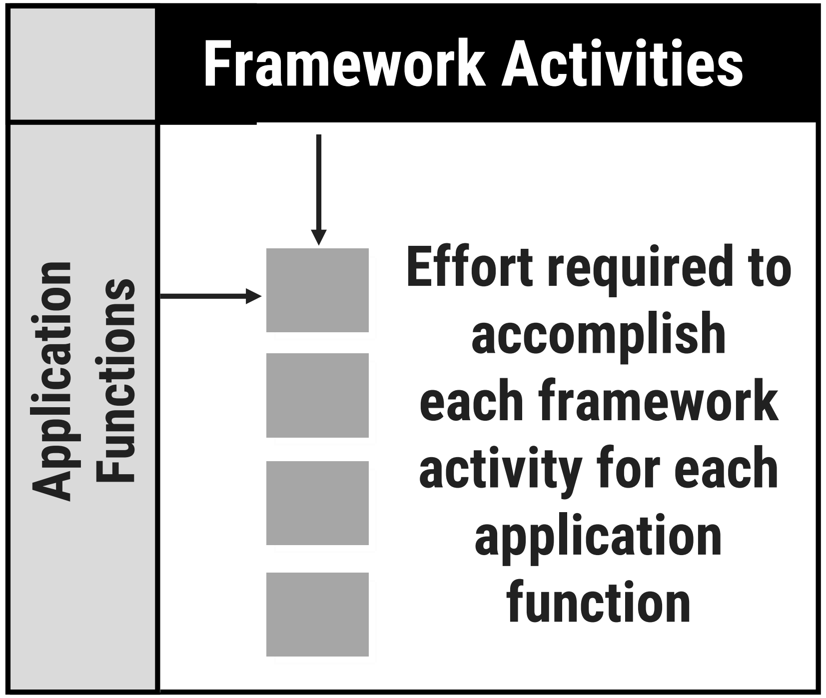
Problem Based Estimation Cont.

- **LOC** and **FP** estimation **differ in the level** of **detail** required for **decomposition** with each value
  - For **LOC**, **decomposition of functions** is **essential** and should go into considerable detail **(the more detail, the more accurate the estimate)**
  - For **FP**, **decomposition** occurs for the **five** information **domain characteristics** and the **14 adjustment factors**
    - **External Inputs, External Outputs, External Inquiries, Internal Logical Files, External Interface Files**
- For **both approaches**, the planner **uses lessons learned** to **estimate**,
  - An **optimistic (S_{opt})**, **most likely (S_m)**, and **pessimistic (S_{pess})** estimates Size (S) value for each function or count
  - Then the expected Size value S is computed as
    - $S = (S_{opt} + 4\,S_m + S_{pess})/6$

Historical LOC or FP data is then compared to S in order to cross-check it.

Process Based Estimation

Process-based estimation is obtained from "process framework"



- This is one of the **most commonly used technique**
- **Identify** the **set of functions** that the software needs to perform as obtained **from the project scope**
- **Identify** the **series of framework activities** that need to be performed **for each function**
- **Estimate the effort** (in **person months**) that will be **required to accomplish** each software process **activity** for **each function**
- **Apply average labor** rates (i.e., **cost/unit** effort) to the **effort estimated** for each process activity
- **Compute** the **total cost** and **effort** for each function and each framework activity.
- **Compare the resulting values** to those obtained by way of the **LOC and FP** estimates
- If **both** sets of **estimates agree**, then your numbers are **highly reliable**
- **Otherwise**, conduct **further investigation** and **analysis** concerning the **function** and **activity breakdown**

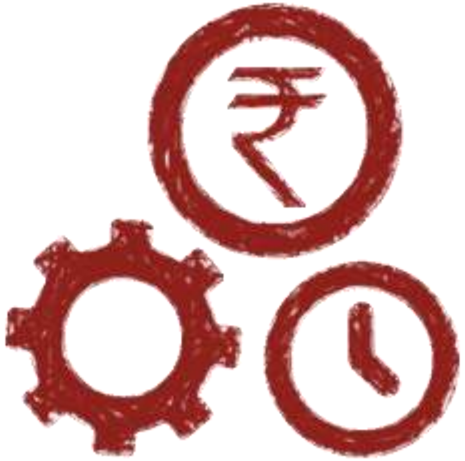
Estimation with Use Cases

Developing an **estimation** approach **with** use cases is **problematic** for the following reasons:

- Use cases are **described using many different formats** and styles—there is no standard form.
- Use cases **represent an external view** (the user's view) of the software and can **therefore** be **written at many different levels** of **abstraction**
- Use cases **do not address** the **complexity of the functions** and features that are described
- Use cases **can describe complex behavior** (Ex., interactions) that **involve many functions** and features
- Although a number of investigators have considered use cases as an estimation input.

- **Before use cases** can be used for **estimation**,
  - the **level within the structural hierarchy** is established,
  - the **average length (in pages) of each use case** is determined,
  - the **type of software** (e.g., real-time, business, engineering/scientific, WebApp, embedded) is defined, and
  - a **rough architecture** for the system is **considered**
- Once these characteristics are established,
  - empirical data may be used to establish the estimated number of LOC or FP per use case (for each level of the hierarchy).
- Historical data are then used to compute the effort required to develop the system.

Empirical Estimation Models



Source Lines of Code (SLOC)

Function Point (FP)

Constructive Cost Model (COCOMO)

Source Lines of Code (SLOC)

- The **project size** helps to determine the resources, effort, and duration of the project.
- **SLOC** is **defined** as the **Source Lines of Code** that are **delivered as part of the product**
- The **effort spent on creating** the **SLOC** is **expressed** in relation to thousand lines of code (**KLOC**)
- This **technique includes** the **calculation of Lines of Code**, **Documentation of Pages**, **Inputs**, **Outputs**, and **Components** of a software program
- The SLOC technique is **language-dependent**
- The **effort required** to calculate **SLOC** may **not be the same** for all **languages**

Software Development Project

Based on the development complexity

Software Development Project Classification

Organic	Semidetached	Embedded
Application programs e.g. data processing programs	**Utility programs** e.g Compilers, linkers	**System programs** e.g OS real-time systems
A **development project** can be considered of **organic** type, if the project deals with **developing** a **well understood application program**, the **size** of the **development team** is reasonably **small**, and the **team members** are **experienced** in **developing similar types of projects**	A **development project** can be considered of **semidetached** type, if the **development consists** of a **mixture** of **experienced** & **inexperienced staff**. Team members may have **limited experience on related systems** but may be unfamiliar with some aspects of the system being developed.	A **development project** is considered to be of **embedded** type, if the **software being developed** is **strongly coupled** to **complex hardware**, or if the **strict regulations** on the **operational procedures** exist

Software Development Project Cont.

Model	Project Size	Nature of Project	Innovation	Dead Line	Development Environment
Organic	Typically **2-50 KLOC**	Small Size Project, Experienced developers in the familiar environment, E.g. Payroll, Inventory projects etc.	Little	Not Tight	Familiar & In-house
Semi Detached	Typically **50-300 KLOC**	Medium Size Project, Medium Size Team, Average Previous Experience, e.g. 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. E.g. ATMs, Air Traffic Controls	Significant Required	Tight	Complex hardware & customer Interfaces

COCOMO Model

COCOMO (Constructive Cost Estimation Model) was proposed by Boehm

According to Boehm, **software cost estimation** should be done through three stages:

- Basic COCOMO
- Intermediate COCOMO
- Complete COCOMO

Basic COCOMO Model

The **basic COCOMO** model gives an **approximate estimate** of the project parameters

The **basic COCOMO estimation** model is given by the **following expressions**

$$\boldsymbol{Effort} = a_1 + (KLOC)^{a_2}\ PM$$

$$\boldsymbol{Tdev} = b_1 \times (Effort)^{b_2}\ Months$$

- **KLOC** is the estimated size of the software product expressed in Kilo Lines of Code
- a_1, a_2, b_1, b_2 are constants for each category of software products,
- **Tdev** is the estimated **time to develop** the software, **expressed in months**,
- **Effort** is the total effort required to develop the software product, expressed in **person months (PMs)**.

Project	a_1	a_2	b_1	b_2
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

Basic COCOMO Model Cont.

- The effort estimation is expressed in **units of person-months (PM)**
- It is the **area under the person-month plot** (as shown in fig.)
- An **effort of 100 PM**
  - **does not** imply that **100 persons** should **work for 1 month**
  - **does not** imply that **1 person** should be **employed for 100 months**
  - **it denotes** the **area under** the person-month curve (fig.)

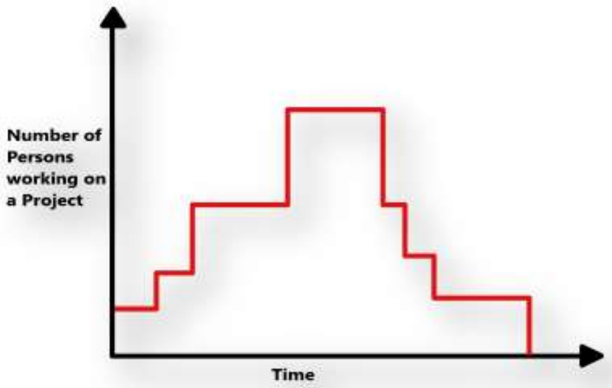


Fig. Person Month Curve

- **Every line of source** text should be **calculated** as **one LOC irrespective of** the **actual number of instructions** on that line
- If a **single instruction spans several lines** (say **n lines**), it is **considered** to be **nLOC**
- The values of a_1, a_2, b_1, b_2 for different categories of products (i.e. organic, semidetached, and embedded) as given by Boehm
- He derived the expressions by examining historical data collected from a large number of actual projects

Basic COCOMO Model Cont.

- **Insight** into the **basic COCOMO** model can be obtained by **plotting the estimated characteristics** for different software sizes
- **Fig.1** shows a **plot of estimated effort** versus **product size**
- From fig. we can observe that the **effort** is somewhat **superlinear** in the **size of the software** product
- The **effort** required to develop a product **increases** very **rapidly with project size**
- The **development time versus the product size** in KLOC is plotted in fig. 2
- From fig., it can be observed that the **development time** is a **sublinear** function of **the size** of the product
- i.e. when the **size of the product** increases by **two times**, the time to develop the **product does not double** but **rises moderately**
- From fig., it can be observed that the development time is roughly the same for all the three categories of products

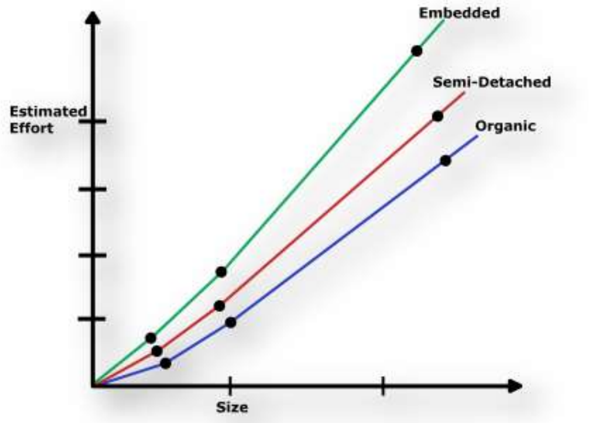


Fig. Effort Verses Product size

Fig. 1

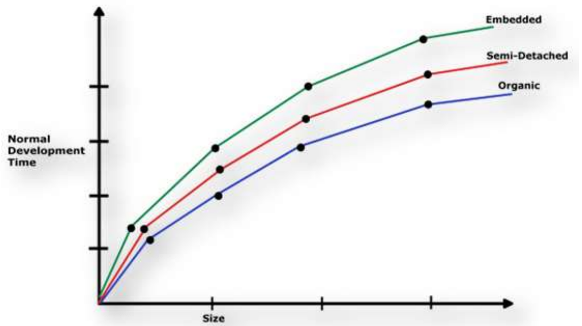


Fig. Development time Verses size

Fig. 2

Basic COCOMO Model Cont.

- **Effort** and the **duration estimations** obtained **using** the **COCOMO** model are called as **nominal effort** estimate and nominal **duration estimate**
- The term **nominal implies** that
  - if anyone **tries to complete the project** in a **time shorter** than the **estimated** duration, then the **cost will increase drastically**
  - But, if anyone **completes the project over a longer period** of time than the **estimated**, then there is almost **no decrease** in the **estimated cost value**

Example: Assume that the **size** of an **organic type** software product **has been estimated** to be **32,000 lines of source code**. Assume that the **average salary** of software **engineers** be **Rs. 15,000/- per month**. **Determine** the **effort required** to develop the software product **and** the **nominal development time**

$$\boxed{Effort = a_1 + (KLOC)^{a_2}\ PM}$$
$$= 2.4 + (32)^{1.05}\ PM$$
$$= 91\ PM$$

$$\boxed{Tdev = b_1 \times (Effort)^{b_2}\ Months}$$
$$= 2.5 \times (91)^{0.38}\ Months$$
$$= 14\ Months$$

Cost required to develop the product = 14 x 15000 = Rs. **2,10,000/-**

Intermediate COCOMO model

- The **basic COCOMO** model **assumes** that **effort** and **development** time **are functions** of the **product size alone**
- However, a **host of other** project **parameters** besides the product size **affect** the **effort required** to develop the product as well as the **development time**
- Therefore, **in order to obtain an accurate estimation** of the effort and project **duration**, the **effect** of all relevant **parameters** must be **taken** into **account**
- The **intermediate COCOMO** model **recognizes this fact** and refines the initial estimate obtained using the basic COCOMO expressions **by using a set of 15 cost drivers (multipliers)** based on **various attributes** of software development
  - For example, if modern programming practices are used, the initial estimates are scaled downward by multiplication with a cost driver having a value less than 1
- It is **requires** the project manager **to rate** these **15** different **parameters** for a particular **project** on a **scale of one to three**.
- Then, **depending** on these **ratings**, **appropriate cost** driver **values** which should be **multiplied with the initial estimate** obtained using the basic COCOMO.

Intermediate COCOMO model Cont.

The cost drivers can be **classified** as being attributes **of the following items**

- **Product:** The characteristics of the product that are considered include the inherent complexity of the product, reliability requirements of the product, etc.
- **Computer:** Characteristics of the computer that are considered include the execution speed required, storage space required etc.
- **Personnel:** The attributes of development personnel that are considered include the experience level of personnel, programming capability, analysis capability, etc.
- **Development Environment:** Development environment attributes capture the development facilities available to the developers. An important parameter that is considered is the sophistication of the automation (CASE) tools used for software development

Complete COCOMO model

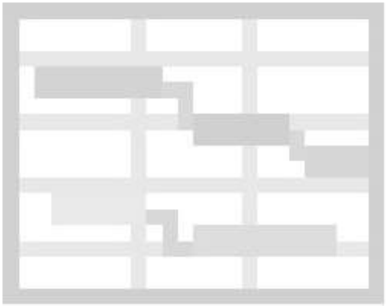
- A major **shortcoming** of both the **basic** and **intermediate COCOMO** models is that they **consider** a software product **as** a **single homogeneous entity**
- Most **large systems** are **made** up several **smaller sub-systems**
- These **sub-systems** may have widely **different characteristics**
  - E.g., some sub-systems may be considered as organic type, some semidetached, and some embedded
  - Also for some subsystems the reliability requirements may be high, for some the development team might have no previous experience of similar development etc.
- The **complete COCOMO** model **considers** these differences in **characteristics** of the **subsystems** and **estimates** the effort and development time **as the sum of the estimates for the individual subsystems**
- The **cost** of each **subsystem** is **estimated separately**
- This approach **reduces** the **margin of error** in the final **estimate**

Project Scheduling & Tracking

It is an **action** that **distributes** estimated **effort across** the **planned** project **duration**, by **allocating** the effort **to specific software engineering tasks**

Scheduling Principles

- Compartmentalization
- Interdependency
- Time Allocation
- Define Responsibilities
- Define Outcomes
- Define Milestones
- Effort Validation



Scheduling Principles

- **Compartmentalization**
  - The **product** and **process** must be **decomposed** into a **manageable number** of **activities** and **tasks**
- **Interdependency**
  - **Tasks** that can be completed in **parallel** must be **separated** from those that must completed **serially**
- **Time Allocation**
  - Every task has **start** and **completion** dates that **take** the **task interdependencies** into account
- **Effort Validation**
  - Project manager must **ensure** that on any **given day** there are **enough staff** members assigned to **complete the tasks** within the **time estimated** in the project plan
- **Define Responsibilities**
  - **Every** scheduled **task** needs to be **assigned to** a **specific** team **member**
- **Define Outcomes**
  - **Every task** in the schedule **needs** to **have** a **defined outcome** (usually a work product or deliverable)
- **Defined Milestones**
  - A **milestone** is **accomplished** when one or more **work products** from an engg task have **passed quality review**

Effort Distribution

- General guideline: **40-20-40 rule**
  - **40%** or more of all effort allocated to **analysis and design tasks**
  - **20%** of effort allocated to **programming**
  - **40%** of effort allocated to **testing**
- **Characteristics** of each **project** dictate the **distribution of effort**
- Although most software organizations encounter the following **projects types**:
  - **Concept Development**
    - initiated to explore new business concept or new application of technology
  - **New Application Development**
    - new product requested by customer
  - **Application Enhancement**
    - major modifications to function, performance or interfaces (observable to user)
  - **Application Maintenance**
    - correcting, adapting or extending existing software (not immediately obvious to user).
  - **Reengineering**
    - rebuilding all (or part) of a existing (legacy) system

Scheduling methods

- Two project scheduling methods that can be applied to software development.
  - **Program Evaluation and Review Technique** (**PERT**)
  - **Critical Path Method** (**CPM**)
- Both techniques are **driven** by **information already** developed in **earlier project planning** activities:
  - estimates of effort
  - a decomposition of the product function
  - the selection of the appropriate process model and task set
  - decomposition of the tasks that are selected
- Both **PERT** and **CPM** provide quantitative tools that allow you to:
  - **Determine the critical path**—the chain of tasks that determines the duration of the project
  - **Establish "most likely" time estimates** for individual tasks by applying statistical models
  - **Calculate "boundary times"** that define a "time window" for a particular task

Project Schedule Tracking

- The project schedule provides a road map for a software project manager.
- It defines the tasks and milestones.
- Several ways to track a project schedule:
  - Conducting **periodic project** status **meeting**
  - **Evaluating** the **review results** in the software process
  - **Determine** if formal **project milestones** have been **accomplished**
  - **Compare actual start date** to **planned start date** for each task
  - Informal **meeting with practitioners**
  - Using **earned value analysis** to **assess progress** quantitatively
- **Project manager** takes the **control** of the **schedule** in the aspects of
  - Project Staffing, Project Problems, Project Resources, Reviews, Project Budget

Gantt chart

A **Gantt chart**, commonly used in **project management**, is one of the most **popular** and **useful ways** of **showing activities (tasks or events)** displayed **against time**.

On **the left of the chart** is a **list of the activities** and along **the top** is a suitable **time scale**.

Each **activity** is **represented** by a **bar**; the **position** and **length** of the bar reflects the **start date**, **duration** and **end date** of the activity. This allows you to see at a glance:

Gantt chart Cont.

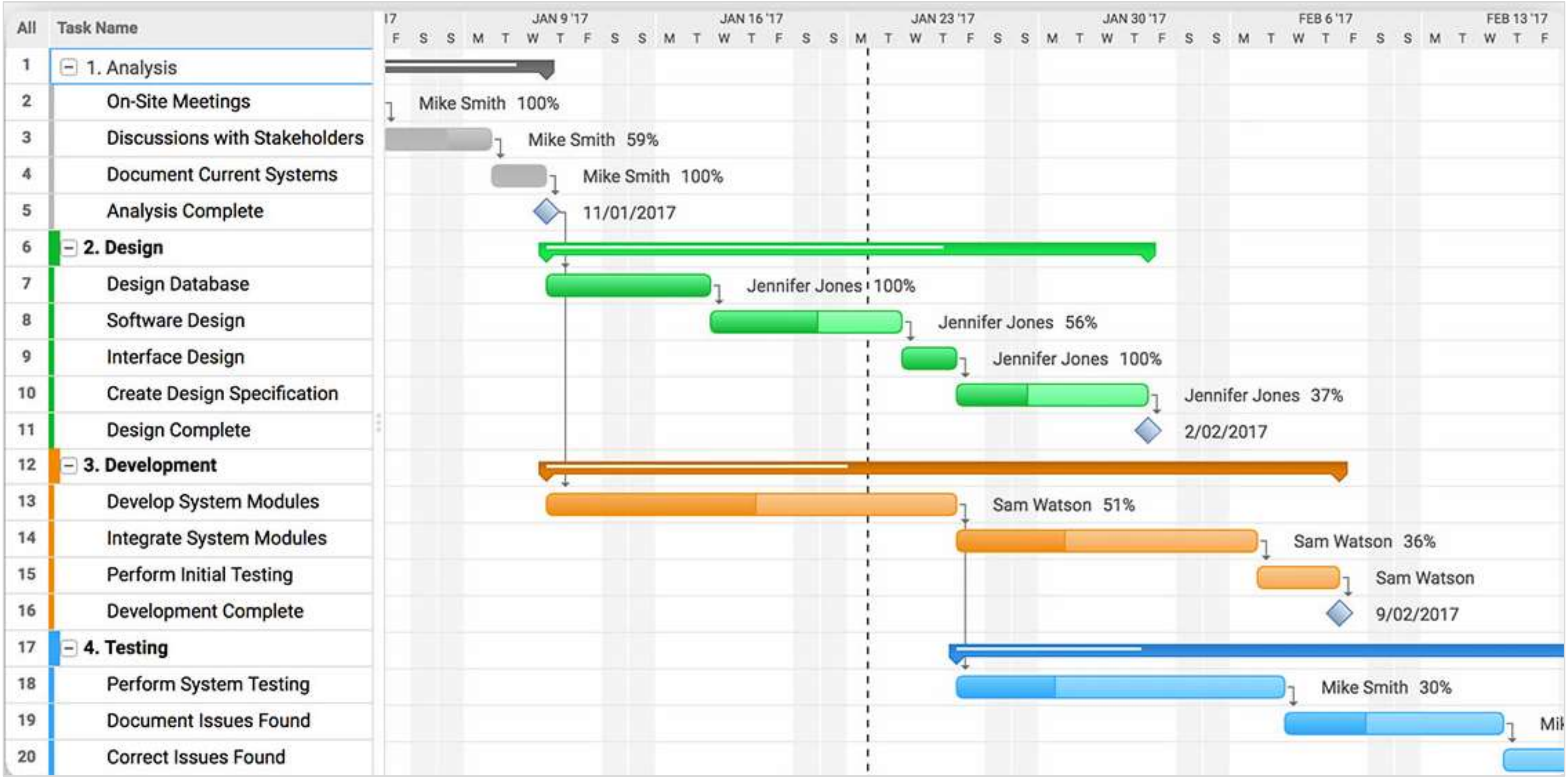
What the various activities are

When each activity begins and ends

How long each activity is scheduled to last

Where activities overlap with other activities, and by how much

The start and end date of the whole project



Thank You