Chapter 1

Software Engineering and Project Management

1.1 Nature and Characteristics of Software

Software:

Software is:

- A set of instruction or computer program that when executed provide desired functions and performance.
- Data structure that enables the programs to adequately manipulate information.
- Descriptive information in both hard copy and virtual forms that describes the operation and use of the programs.

In broad sense, Software is not just the programs but also all associated documentation and configuration data which is needed to make these programs operate correctly.

A software usually consists of a number of separate programs, configuration files which are to set up these programs, system documentation which explains how to use the system, and for software products and websites for users to download recent product information.

In conclusion, we can define software as a collection of programs, configuration files, documentation, user manual, update facilities and support.

Characteristics of Software:

a) Software is developed or engineered, but not manufactured.

   Software is a design of strategies, instruction which finally perform the designed, instructed tasks. And a design can only be developed, not manufactured.

b) Software does not “wear out”.

   Software is not susceptible to the environmental melodies and it does not suffer from any effects with time.

c) Most software is custom-built, rather than being assembled from existing components.

Nature of Software:

a) Software takes on a dual role. It is a product, and at the same time, the vehicle for delivering a product.
    - As a product, it delivers the computing potential embodied by computer hardware or more broadly, by a network of computers that are accessible by local hardware.
    - As the vehicle used to deliver the product, software acts as the basis for the control of the computer (operating sys tems), the communication of information (networks), and the creation and control of other programs (software tools and environments).

b) Software delivers the most important product of our time i.e. information.
    - It transforms personal data (e.g., an individual's financial transactions) so that the data can be more useful in a local context;
    - it manages business information to enhance competitiveness;
    - it provides a gateway to worldwide information networks (e.g., the Internet), and provides the means for acquiring information in all of its forms.

Software Application Domains

a) System Software: Manages and controls hardware, providing a platform for running applications. Examples include operating systems, device drivers, and utilities.

b) Application Software: Designed for end-users to perform specific tasks like word processing, spreadsheets, media players, and gaming. Examples include Microsoft Office, Photoshop, and web browsers.

c) Engineering/Scientific Software: Used for technical and scientific tasks, including simulations, calculations, and complex analysis. Examples include MATLAB, AutoCAD, and software for weather forecasting.

d) Embedded Software: Operates within hardware devices, often with real-time constraints. It's found in devices like cars, washing machines, and medical equipment. Examples include firmware for microcontrollers and software in IoT devices.

e) Product-line Software: Tailored for specific markets or industries, with common core features but customizable for different customers. Examples include ERP systems like SAP and CRM platforms.

f) Web Applications: Accessed via web browsers, delivering functionality over the internet. Examples include Google Docs, Gmail, and e-commerce sites like Amazon.

g) AI Software: Mimics human intelligence and performs tasks like decision-making, speech recognition, and machine learning. Examples include voice assistants (e.g., Siri), recommendation systems, and autonomous driving software.

1.2 Software versus System Engineering

Software Engineering focuses solely on software development, while System Engineering considers both hardware and software to create a complete system.

Software Engineering:

- **Focus**: Concerned with the design, development, testing, and maintenance of software applications.
- **Scope**: Deals with programming, algorithms, databases, user interfaces, and software lifecycle processes like requirement gathering, design, coding, and testing.
- **Output**: Produces software solutions that solve specific problems or automate tasks.
- **Example**: Developing a mobile app or web-based system.

System Engineering

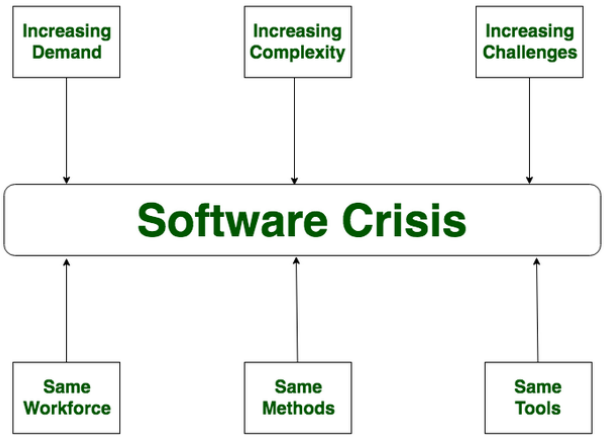
- **Focus**: Broader discipline that integrates both hardware and software components into a complete, functioning system.
- **Scope**: Involves not only software but also hardware, networking, processes, and overall system architecture to ensure the entire system meets specified requirements.
- **Output**: Produces a fully operational system that includes hardware, software, people, and processes.
- **Example**: Designing an automated factory system that integrates robots, sensors, and control software.

1.3 Software Crisis and Myths

Software Crisis:

Software Crisis refers to the challenges faced in the 1960s and 1970s due to the rapid increase in software complexity, leading to project failures, delays, cost overruns, and poor-quality software.

Software Crisis is a term used in computer science for the difficulty of writing useful and efficient computer programs in the required time. The software crisis was due to using the same workforce, same methods, and same tools even though rapidly increasing software demand, the complexity of software, and software challenges. With the increase in software complexity, many software problems arose because existing methods were insufficient.



Factors contributing to Software Crisis:

a) **Increased Complexity:** The complexity of software systems grew rapidly, making it difficult to design, develop, and maintain them effectively. The lack of tools and methodologies to manage this complexity led to software that was error-prone and hard to understand.

b) **Lack of Formal Methods:** In the absence of formalized methods and practices, software development was often ad hoc and lacked systematic approaches. This resulted in unreliable software that was challenging to modify or extend.

c) **Changing Requirements:** Many software projects faced volatile and changing requirements. The inability to handle these changes efficiently led to project delays, cost overruns, and software not aligning with user needs.

d) **Limited Reusability:** Developers struggled to reuse code and components across projects, leading to duplicated efforts and inefficiencies. This hindered productivity and hindered the evolution of software engineering practices.

e) **Poor Quality Assurance:** Inadequate testing and quality assurance processes resulted in software with numerous bugs, security vulnerabilities, and reliability issues. This undermined user confidence and overall system stability.

f) **Lack of Collaboration:** Limited collaboration among development teams, stakeholders, and end-users led to misunderstandings and misalignments. Communication gaps caused delays and hindered the creation of high-quality software.

g) **Rapid Technological Changes:** The fast-paced evolution of technology rendered some software obsolete quickly. This made maintaining and updating software challenging and forced developers to learn new tools and platforms constantly.

h) **Inadequate Project Management:** Weak project management practices, including poor planning, resource allocation, and progress tracking, contributed to missed deadlines, budget overruns, and unsuccessful projects.

i) **Absence of Standards:** The lack of industry-wide standards and best practices resulted in inconsistent approaches to software development. This hindered interoperability, hindered collaboration, and contributed to the crisis.

Solutions to the Software Crisis:

a) Structured Methodologies
b) Formal Software Development
c) Requirements Engineering
d) Quality Assurance and Testing
e) Code Reusability
f) Improved Project Management
g) Standardization and Best Practices

Software Myths:

It is the misconceptions that often lead to misunderstanding or unrealistic expectations about software development.

a) Management Myths:

i) **Myth**: "Adding more people to a late project will speed it up."

   Reality: It often causes further delays due to the learning curve and increased communication overhead (Brooks' Law).

ii) **Myth**: "We can outsource software development and get it done perfectly."

Reality: Outsourcing without proper communication and oversight can lead to issues in quality, timelines, and alignment with goals.

b) Customer Myths:

i) **Myth**: "Software requirements can be changed easily at any point in the project."

Reality: Late changes are costly and time-consuming, impacting timelines and quality.

ii) **Myth**: "A working program is all that's needed; documentation is unnecessary."

Reality: Without documentation, maintaining or enhancing the software becomes difficult and risky over time.

c) Developer Myths:

i) **Myth**: "Once the software is written, the job is done."

Reality: Maintenance, bug fixes, and updates are an ongoing process that often takes more effort than initial development.

ii) **Myth**: "The more code, the better the software."

Reality: More code often leads to more complexity and potential errors; quality code is lean and efficient.

1.4 Four Ps of Software Project Management

In software engineering, the management spectrum describes the management of a software project. The management of a software project starts from requirement analysis and finishes based on the nature of the product, it may or may not end because almost all software products faces changes and requires support. It is about turning the project from plan to reality. It focuses on the four P's; people, product, process and project. Here, the manager of the project has to control all these P's to have a smooth flow in the project progress and to reach the goal

For properly building a product, there's a very important concept that we all should know in software project planning while developing a product The **Four Ps of Software Project Management** are key elements that contribute to the successful management of software projects. These components play a very important role in your project that can help your team meet its goals and objective

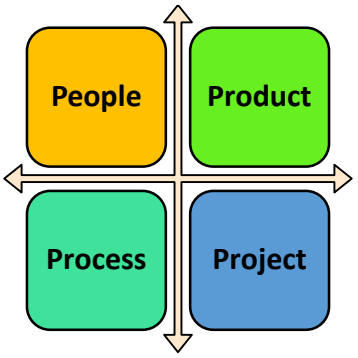


Figure: the 4 Ps

1. **People**

- **Key Aspect**: The individuals and teams responsible for planning, developing, and maintaining the software.
- **Details**:
    - Success of the project depends largely on the skills, motivation, and organization of the team.
    - Roles include project managers, software engineers, testers, and other stakeholders.
    - Emphasizes the importance of communication, collaboration, leadership, and team-building.

2. **Product**

- **Key Aspect**: The actual software or system being developed.
- **Details**:
    - Clearly defining the **scope** and **requirements** of the product is crucial.
    - Understanding the problem the product is solving and the needs of the users ensures alignment with business goals.
    - The product should meet quality standards, be functional, reliable, and maintainable.

3. **Process**

- **Key Aspect**: The methodology and framework used to manage and guide the development.
- **Details**:

- The process provides the structure for planning, monitoring, and controlling the project.
- Software processes can include **Waterfall**, **Agile**, **Scrum**, and others.
- Ensures consistency, repeatability, and that all development activities are carried out systematically to meet the project goals.

4. Project

- **Key Aspect**: The overall management of the work being done to develop the software.
- **Details**:
  - Involves activities such as planning, scheduling, budgeting, risk management, and monitoring progress.
  - Requires setting **realistic goals**, **managing resources** effectively, and delivering the product on time and within budget.
  - Key elements of project management include task allocation, milestone tracking, and ensuring that the software meets the defined requirements.

1.5 Process and Project Metrics

Process Metrics and **Project Metrics** are used to measure, analyze, and improve the performance of software development processes and project management activities. These metrics help in monitoring progress, identifying issues, and ensuring quality.

a) Process Metrics

- **Definition**: Process metrics are the metrics used to evaluate and improve the efficiency and quality of the software development process. They help in understanding how well the process is working and how it can be optimized.
- **Focus**: Improving the overall software development process. They focus on the performance of the overall software development process, aiming for long-term improvements in efficiency, productivity, and quality.
- **Key Metrics**:
  - **Productivity**: Measures output relative to the input (e.g., lines of code per developer per month).
  - **Quality**: Assesses defect rates (e.g., defects per thousand lines of code).

- **Cycle Time**: The total time taken from the start of development to delivery.
- **Process Maturity**: Evaluates how well the development process adheres to best practices (e.g., Capability Maturity Model Integration - CMMI levels).

- **Example**:
  - If the defect rate is too high, process metrics can help identify inefficiencies in the testing phase or coding practices.

b) Project Metrics

- **Definition**: Project metrics are the metrics focused on tracking the performance and progress of a specific software project. They provide insight into whether the project is on track to meet deadlines, budgets, and quality expectations.
- **Focus**: Managing and controlling a specific project. Focus on managing a specific project, ensuring that it is completed on time, within budget, and meets the required quality standards.
- **Key Metrics**:
  - **Effort**: Measures the total time spent on the project (e.g., person-hours, person-days).
  - **Schedule Variance**: Compares actual progress against planned milestones.
  - **Cost Variance**: Compares actual costs to the budgeted costs.
  - **Defects**: Tracks the number of defects found during development and after delivery.
  - **Completion Percentage**: Measures the percentage of work completed versus planned.
- **Example**:
  - If a project is behind schedule, project metrics such as effort and schedule variance can help determine the cause and guide corrective actions.

1.6 Measurement of Software: Metrics, Measures, Indicator

In software engineering, measurement is essential for evaluating various aspects of software development and management. Understanding the terminology associated with measurement is crucial for effective analysis and improvement.

1.6.1 Measures

- **Definition**: A **measure** is a quantitative value derived from a specific attribute of a software product or process. It provides a raw numerical representation of a characteristic.
- **Examples**:
    - **Lines of Code (LOC)**: A measure of the size of a software program.
    - **Defect Count**: The total number of defects identified during testing.
    - **Function Points**: A measure of the functionality delivered by the software based on user requirements.

1.6.2 Metrics

- **Definition**: A **metric** is a calculated value derived from one or more measures. Metrics provide insights into the performance, quality, and efficiency of software development processes or products.
- **Characteristics**:
    - Metrics are often used to compare different projects, assess progress, and guide decision-making.
    - Metrics can be categorized into:
        - **Product Metrics**: Measure characteristics of the software product (e.g., reliability, maintainability).
        - **Process Metrics**: Measure the performance of the software development process (e.g., defect density, cycle time).
        - **Project Metrics**: Measure aspects of a specific project (e.g., effort estimation, schedule variance).
- **Examples**:
    - **Defect Density**: Number of defects per thousand lines of code (Defect Count / LOC).
    - **Development Productivity**: Lines of code produced per person-month.

- **Customer Satisfaction Index**: A measure of user satisfaction based on surveys.

1.6.2 Indicators

1. **Definition**: An **indicator** is a qualitative or quantitative value that provides insights into the health or performance of a software project or process. Indicators are often derived from metrics and help stakeholders make informed decisions.
2. **Characteristics**:
   - Indicators are generally simpler and easier to communicate than metrics.
   - They are used to assess whether certain thresholds or targets are met and can trigger actions based on the results.
3. **Examples**:
   - **Status Reports**: Indicating whether a project is on track, behind schedule, or ahead of schedule.
   - **Customer Feedback Scores**: Used to indicate user satisfaction with the software.
   - **Risk Indicators**: Identifying potential risks in the project based on metrics like defect rates or schedule delays.

1.7 Project Estimation, Empirical Estimation Models

1.7.1 Project estimation

- Project estimation is a crucial part of project management, aiming to predict the resources, time, and costs required to complete a software project successfully.
- Accurate estimations help in planning, resource allocation, budgeting, and scheduling.
- Project estimation involves predicting the necessary resources (time, cost, and effort) for a project based on various inputs, including historical data, project requirements, and complexity.

Types of Estimates:

- **Rough Estimate**: A high-level estimate used for initial project planning, usually expressed in broad ranges (e.g., days, weeks).

- **Detailed Estimate**: A more precise estimate developed after detailed analysis, often broken down into specific tasks and subtasks.
- **Bottom-Up Estimate**: Estimation that starts from the smallest components and aggregates them to determine the overall project estimate.
- **Top-Down Estimate**: Estimation based on overall project requirements and then breaking it down into smaller components.

Factors affecting estimates:

- **Project Scope**: The complexity and size of the project can greatly impact estimates.
- **Team Experience**: The skills and experience of the development team can influence productivity and accuracy.
- **Technology**: Familiarity with the tools and technologies used can affect the speed of development.
- **Requirements Stability**: Frequent changes in requirements can lead to re-estimation and uncertainty.

1.7.2 Empirical Estimation Models:

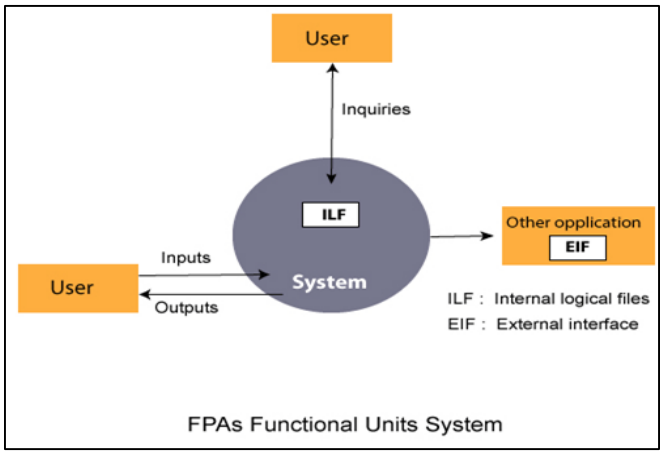
Empirical estimation models use historical data and statistical analysis to predict project outcomes. They leverage past project metrics to inform current estimations. **Empirical Estimation Models** provide structured methods for estimating project effort and cost using historical data, helping improve the accuracy and reliability of predictions.

1. Function Points Estimation

- FPA measures software functionality from the user's perspective, focusing on the functional requirements.
- An FP (Functional Point) is a unit of measurement to express the amount of business functionality that a software provides to a user. It measures functionality from user’s point of view.
- In such method, the software product is directly dependent on the number of functions or features it supports.
- Function points are converted into effort estimates based on historical productivity rates.

Components:

- **External Inputs**: Data inputs into the system (eg. File)
- **External Outputs**: Data outputs generated by the system (Eg. Reports, message)
- **User Inquiries**: Interactive queries from users (Eg. Interactive inputs needing a response)
- **Files**: Internal and external files used in the system (Eg. Files shared with other software systems)
- **Interfaces**: Interaction with other systems.



FPAs Functional Units System

Advantages	Disadvantages
• Not restricted to code • Language independent • More accurate than estimated LOC	• Subjective counting • Hard to automate and difficult to compute. • Ignores quality of output.

For example:

If FP = 375 FP, average productivity = 6.5 FP/PM and labor rate = $8000 per month then

- Cost per FP = ($8000) / (6.5 FP/PM) = $1230
- Total project cost = 375 x $1230 = $461250
- Estimated effort = Total FP / Average productivity = 375 FP / (6.5 FP/PM) = 58 persons

Computation of FP:

Mathematically;

FP= Count Total x CAF

Where, Count Total= sum of all FP entries

CAF =0.65 + 0.01 x $\sum F_i$

CAF→ Complexity Adjustment Factor

The F_i (i=1 to 14) is the value adjusted factor based on response to 14 questions.

Numerical: Given the following values, Compute FP when all complexity adjustment factors and weighting factors are average.

Information Domain Value	Count	Weighting Factor		
		Simple	Average	Complex
External Inputs (EI)	50	3	4	6
External Outputs (EO)	40	4	5	7
External Inquiries (EQ)	35	3	4	6
Internal Logical Files (IFL)	6	7	10	15
External Interface Files (EIF)	4	5	7	10

Solution: FP is given by :

FP= Count Total x CAF

Now , Count Total =50*4 + 40*5 + 35*4 + 6*10 + 4*7 =628

CAF= 0.65 + 0.01 x $\sum F_i$

=0.65 + 0.01 x (14x3) = 1.07 *(Here 14 is taken for 14 questions to be answered)*

Hence FP= 628 x 1.07 =672

Reference for further study: *https://www.javatpoint.com/software-engineering-functional-point-fp-analysis*

https://www.youtube.com/watch?v=nN379biPGxE

2. COCOMO Model

The constructive cost model (COCOMO) is an algorithmic software cost estimation model developed by Barry Boehm and published in 1981 on his book named Software Engineering Economics. The model uses a basic regression formula, with some parameters that are derived from historical project data and current project characteristics.

COCOMO is one of the most widely used empirical models for estimating software project costs and effort.

This model of estimation is of two version:

i) COCOMO 81:

- The original model offers basic estimation formulas. To determine the initial effort E in person-months and Time in months, the equations are:

  Effort $E = a * (KLOC)^b$

  Time $T = c * (KLOC)^d$

  The value of the constant a, b, c and d depends on the project type: organic, semi-organic and embedded.

Project	a_i	b_i	c_i	d_i
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

- The COCOMO model assumes every project is developed in one of the three models:

  a) *Organic mode:* A development project can be treated of the 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 methods of projects. The project requires little innovation. Examples of this type of projects are simple business systems, simple inventory management systems, and data processing systems.

  b) *Semi-organic mode:* A development project can be treated with semidetached type if the development consists of a mixture of experienced and inexperienced staff. Team members may have finite experience in related systems but may be unfamiliar with some aspects of

the order being developed**.** Example of Semidetached system includes developing a new operating system (OS), a Database Management System (DBMS), and complex inventory management system.

c) *Embedded mode:* A development project is treated to be of an embedded type, if the software being developed is strongly coupled to complex hardware, or if the stringent regulations on the operational method exist. Requires a great deal of innovation. For Example**:** ATM, Air Traffic control.

Stages of COCOMO I model:

COCOMO consists of a hierarchy of three increasingly detailed and accurate forms namely: Basic COCOMO, Intermediate COCOMO, and Detailed COCOMO

a) Basic COCOMO:

The basic COCOMO model provide an accurate size of the project parameters. The following expressions give the basic COCOMO estimation model:

Effort $E = a * (KLOC)^b$

Time $T = c * (Effort)^d$

b) Intermediate COCOCMO

This stage uses a set of 15 cost drivers based on various attributes of software engineering known as Effort Adjustment Factor(EAF). The following expressions give the Intermediate COCOMO estimation model:

Effort $E = a * (KLOC)^b * EAF$

Time $T = c * (Effort)^d$

The 15 cost drivers are tabulated as below:

Cost Drivers	Ratings					
	Very Low	Low	Nominal	High	Very High	Extra High
Product attributes						
Required software reliability	0.75	0.88	1.00	1.15	1.40	
Size of application database		0.94	1.00	1.08	1.16	
Complexity of the product	0.70	0.85	1.00	1.15	1.30	1.65
Hardware attributes						
Run-time performance constraints			1.00	1.11	1.30	1.66
Memory constraints			1.00	1.06	1.21	1.56
Volatility of the virtual machine environment		0.87	1.00	1.15	1.30	
Required turnabout time		0.87	1.00	1.07	1.15	
Personnel attributes						
Analyst capability	1.46	1.19	1.00	0.86	0.71	
Applications experience	1.29	1.13	1.00	0.91	0.82	
Software engineer capability	1.42	1.17	1.00	0.86	0.70	
Virtual machine experience	1.21	1.10	1.00	0.90		
Programming language experience	1.14	1.07	1.00	0.95		
Project attributes						
Application of software engineering methods	1.24	1.10	1.00	0.91	0.82	
Use of software tools	1.24	1.10	1.00	0.91	0.83	
Required development schedule	1.23	1.08	1.00	1.04	1.10	

c) Detailed COCOMO

In detailed COCOMO, the whole software is differentiated into multiple modules, and then we apply COCOMO in various modules to estimate effort and then sum the effort.

Numerical:

1. *Suppose a project was estimated to be 400 KLOC. Calculate the effort and development time for each of the three model i.e., organic, semi-detached & embedded.*

Solution: Here, estimated size of project = 400 KLOC

Now:

(i) Organic Mode

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

T= 2.5 * $(1295.31)^{0.38}$ =38.07 months

(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

2. 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 semidetached mode is the most appropriate mode, keeping in view the size, schedule and experience of development time.

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

   T = 2.5 * $(1133.12)^{0.35}$ = 29.3 months

$$\text{Average Staff Size (SS)} = \frac{E}{D} \text{ Persons}$$

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

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

3. The size of organic software is estimated to be 32,000 LOC. The average salary for software engineering is Rs. 15000/- per month. What will be effort and time for the completion of the project?

Solution:

- Effort applied = 2.4 x $(32)^{1.05}$ PM = 91.33 PM (Since: 32000 LOC = 32KLOC)
- Time = 2.5 x $(91.33)^{0.38}$ Month = 13.899 Months
- Cost = Time x Average salary per month = 13.899 x 15000 = Rs. 208480.85
- People required = (Effort applied) / (development time) = 6.57 = 7 persons

ii) COCOMO II:

- The **Constructive Cost Model II (COCOMO II)** is an empirical software cost estimation model developed by Barry Boehm. It is an updated version of the original COCOMO model and is designed to provide more accurate estimates for modern software projects by considering a wider range of factors, including the influence of modern development practices and environments.
- An updated model that incorporates more factors, including new development environments and practices.
- This is the successor of COCOMO I model.

The model allows estimation in three different modes:

a) Application Composition:

- Used during the **early prototyping phase** when software is developed using reusable components (such as RAD tools or component-based development).
- Basic Formula: Effort = Object points x ($\frac{Effort\ Multiplier}{Productivity}$)
- Where:
  - **Object Points**: Represents the total weighted number of screens, reports, and third-generation language (3GL) components
  - **Effort Multiplier**: An adjustment factor based on project characteristics.
  - **Productivity**: Based on the productivity rate (nominally set at 4-7 person-months per thousand object points).

b) Early Design:

- Used when detailed information is not yet available but high-level information about the software architecture and major components is known.
- In this mode, the estimation is based on size in terms of Function Points (FP) or a rough estimate of KLOC

c) Post-Architecture: For projects where detailed design specifications are available.

- Used after the project has been fully specified (detailed design phase), and all major architectural decisions have been made.
- This is the most detailed estimation mode and uses all 17 cost drivers to adjust the effort estimation.

The basic formula for estimating effort in person-months for the second and third mode is:

Effort = a x KLOC ^{b} + EAF

Where:

- a and b are constants derived from the project's characteristics.
- **KLOC**: The estimated number of thousands of lines of code.
- **EAF**: Effort Adjustment Factor, which accounts for the cost drivers

Summary of Constants in COCOMO II

Estimation Mode	Constant a	Exponent b	EAF	Cost Drivers
Application Composition	Varies based on project	N/A	Effort Multiplier	Based on Object Points
Early Design	2.94	1.12	Based on 7 cost drivers	7 key cost drivers
Post-Architecture	2.94	1.10	Based on 17 cost drivers	17 detailed cost drivers

Numerical:

A software development team is tasked with creating a new project management tool. The estimated size of the project is 80 KLOC (80,000 lines of code). The team has decided to use the COCOMO II model with post-Architecture mode for estimation. The cost drivers are:

- Product Reliability: 1.2
- Database Size: 1.0
- Product Complexity: 1.4
- Personnel Capability: 1.1
- Required Development Schedule: 1.3
- Use of Software Tools: 1.2

a) Calculate the Effort Adjustment Factor (EAF) based on the given cost driver ratings.

b) Using the COCOMO II formula, estimate the total effort required in person-months for the project.

Solution:

1. **Effort Adjustment Factor (EAF)**:

   EAF=1.2 × 1.0 × 1.4 × 1.1 × 1.3 × 1.2 =2.88288

2. **Effort Estimation**: Using the COCOMO II formula:

   Effort (Person-Months) $= a \times (KLOC)^{b} \times EAF$

   $= 2.94 \times (80)^{1.10} \times 2.8288$

   ≈ 1031 person-months

1.8 Software Risks, Assumptions, Issues, Dependency

1.8.1 Software Risks:

Software risks refer to potential problems that may cause project delays, cost overruns, or failure to meet requirements. Risk management is critical in software engineering to identify, assess, and mitigate these risks.

Types of Software Risks:

- **Project Risks**: These risks affect project schedules, resources, and overall project management. Examples include:
  - Unrealistic deadlines
  - Budget constraints
  - Resource shortages
- **Technical Risks**: These risks arise from technical challenges in the development process. Examples include:
  - New, unproven technologies
  - Poor system architecture or design
  - Integration issues with legacy systems
- **Business Risks**: These affect the business impact of the software, including market demand, profitability, and client satisfaction. Examples include:

    - Change in market demand
    - Misunderstanding user requirements
    - Stakeholder conflicts
- **External Risks**: These risks are outside the control of the development team. Examples include:
    - Changes in regulations or compliance
    - Natural disasters or geopolitical instability

1.8.2 Assumptions

In software projects, assumptions are beliefs or statements taken to be true without full verification. These assumptions are typically made during the planning phase and form the basis for decision-making.

Common Assumptions in Software Projects:

- **Resource Availability**: Assumptions about team members, tools, and budget being available as needed.
- **Technology Assumptions**: Assuming certain technologies or platforms will be suitable for the project.
- **User Requirements**: Belief that user requirements will not change significantly during development.
- **Timeline Assumptions**: Assuming that project deadlines and schedules are realistic.

Risks of Assumptions:

- If assumptions are incorrect or change during the project, they can lead to **delays**, **cost overruns**, or **project failure**. Therefore, it is essential to document assumptions and revisit them as the project progresses.

1.8.3 Issues

Issues are problems or challenges that arise during the software development process. Unlike risks (which are potential problems), issues are real and present challenges that need immediate attention.

Common Software Development Issues:

- **Requirements Creep**: Gradual addition of new requirements that weren't part of the original scope.

- **Technical Debt**: Accumulation of workarounds and poor code quality, making future development more difficult.
- **Resource Bottlenecks**: Shortages in human or technical resources, leading to delays.
- **Testing and Quality**: Inadequate testing leading to defects and poor-quality software.
- **Team Communication**: Miscommunication among team members or with stakeholders.

Issue Management Process:

1. **Issue Identification**: Recognize the problem.
2. **Issue Logging**: Document the issue with details about its nature and potential impact.
3. **Issue Resolution**: Assign responsibility and take corrective action to resolve the issue.
4. **Monitoring and Tracking**: Continuously monitor the issue until it is resolved.

1.8.4 Dependency

Dependencies in software projects refer to relationships between tasks, teams, or components, where one task or component relies on another for completion. Managing dependencies is crucial for ensuring smooth progress.

Types of Dependencies:

- **Task Dependencies**: One task must be completed before another can start (e.g., coding cannot begin until design is completed).
    - **Finish-to-Start (FS)**: Task A must finish before Task B can start.
    - **Start-to-Start (SS)**: Task A and Task B must start simultaneously.
- **Resource Dependencies**: Shared resources (like team members or hardware) that are needed for multiple tasks.
- **External Dependencies**: Tasks or resources outside the control of the project team (e.g., external vendors or third-party services).
- **Component Dependencies**: One software module or component depends on another for integration or functionality.

Managing Dependencies:

- **Documentation**: Clearly document dependencies during project planning.
- **Schedule Coordination**: Ensure tasks with dependencies are scheduled appropriately to avoid delays.

- **Risk Management**: Identify and mitigate risks related to dependency failures.
- **Communication**: Keep open communication with all parties involved in dependencies to ensure alignment.

1.8.5 Risk Management Process:

Risk Management in software projects involves identifying, assessing, mitigating, and monitoring potential issues that could negatively affect the success of a project. Effective risk management ensures that a project is delivered on time, within budget, and meets quality standards.

a) **Risk Identification**: Identify potential risks that could affect the project.

b) **Risk Analysis**: Assess the probability and impact of each risk.

c) **Risk Prioritization**: Rank risks based on their potential impact.

d) **Risk Mitigation**: Develop strategies to reduce or eliminate risks.

e) **Risk Monitoring**: Continuously track and review risks throughout the project lifecycle.

a) Risk Identification

Risk Identification is the process of finding potential risks that could impact the project negatively. This is the first step in risk management and serves as the foundation for all subsequent risk management activities.

Key Sources of Risks:

1. **Technical Risks**:
    - Adoption of new or unproven technology.
    - Integration issues with existing systems.
    - Unrealistic performance expectations.
2. **Project Risks**:
    - Schedule delays due to resource availability.
    - Budget overruns caused by scope changes.
    - Poor project management and communication.
3. **Business Risks**:

- Market demand changes or loss of business value.
- Changes in client priorities or requirements.

4. **External Risks**:
   - Regulatory changes or compliance issues.
   - Natural disasters or external vendor failures.

Techniques for Risk Identification:

- **Brainstorming**: Group discussion to identify potential risks.
- **Interviews/Workshops**: Talking with stakeholders, team members, or experts to gather insights on risks.
- **Checklists**: Using pre-existing lists of common project risks.
- **SWOT Analysis**: Analyzing the project's strengths, weaknesses, opportunities, and threats.
- **Historical Data**: Reviewing previous project experiences to identify similar risks.

b) Risk Mitigation

Risk Mitigation involves taking proactive steps to reduce the likelihood or impact of a risk. Mitigation strategies are designed and implemented to address risks before they turn into issues.

Risk Mitigation Strategies:

1. **Avoidance**: Changing the project plan to eliminate the risk entirely (e.g., not using a risky technology).
2. **Reduction**: Minimizing the impact or likelihood of the risk (e.g., increasing testing and quality assurance to reduce technical risks).
3. **Transfer**: Shifting the impact of the risk to a third party (e.g., outsourcing risky components to external vendors or purchasing insurance).
4. **Acceptance**: Acknowledging the risk and choosing to do nothing but preparing a contingency plan in case it occurs (e.g., accepting a small budget overrun).

Example Mitigation Actions:

- Conducting **regular code reviews** to mitigate technical defects.

- **Hiring extra resources** to meet tight deadlines.
- Implementing **pilot tests** before full deployment to minimize the impact of technical risks.

c) Risk Monitoring

Risk Monitoring is the continuous process of tracking identified risks, reassessing their status, and identifying any new risks that may arise throughout the project lifecycle.

Steps in Risk Monitoring:

1. **Establish Monitoring Criteria**: Define how risks will be tracked and how frequently.
    - Risk Probability (Likelihood of the risk happening)
    - Risk Impact (Severity of the consequence)
    - Risk Exposure (Combination of probability and impact)
2. **Use Risk Registers**: A risk register or log should be maintained to document each identified risk, its status, and any actions taken.
3. **Regular Risk Reviews**: Hold regular meetings or reviews to reassess the project's risk environment and update the risk register.
4. **Continuous Communication**: Ensure that all team members and stakeholders are aware of ongoing risks and changes in their status.

Risk Monitoring Tools:

- **Risk Matrix**: Helps visualize and prioritize risks based on their likelihood and impact.
- **Dashboard/Tracking Software**: Tools like Jira or MS Project can be used to monitor and report risks.

d) Risk Management

Risk Management is the overall process of handling risks from identification to resolution. It includes the development of strategies to address both existing risks and any new risks that arise during the project.

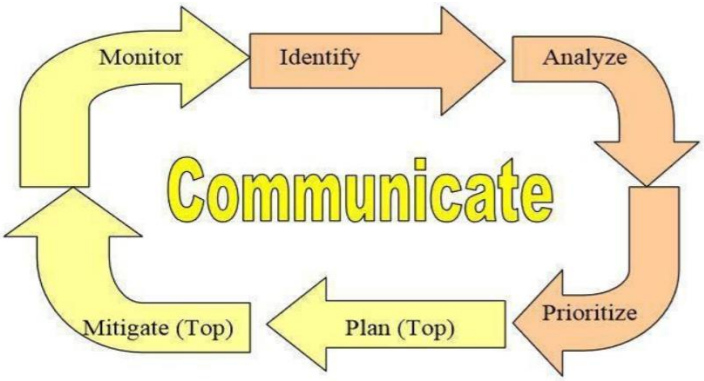


Figure: Risk Management Process

Steps in Risk Management:

1. **Identify Risks**: Discover all potential risks through various techniques (brainstorming, interviews, historical data analysis).
2. **Risk Analysis and Prioritization**:
    - Assess each risk based on its likelihood and impact.
    - Use tools such as the **Risk Matrix** or **Quantitative Risk Analysis** to categorize and prioritize risks.
3. **Plan Risk Responses**:
    - Develop mitigation or contingency plans for high-priority risks.
    - Assign ownership to team members for specific risks.
4. **Implement Risk Responses**:
    - Actively engage in risk mitigation strategies.
    - Monitor the effectiveness of risk response efforts and make adjustments as necessary.
5. **Monitor and Review Risks**:
    - Continuously track risks and update the risk register.
    - Identify any new risks during regular project status updates.
6. **Communicate Risks**:
    - Ensure stakeholders are regularly updated on risk status and any mitigation efforts.
    - Keep open lines of communication within the project team for quick responses.

Case Example of Risk Management:

You are managing a software development project to build a new mobile banking app. The following risks are identified:

1. **Risk 1**: Integration with existing core banking systems might fail due to the complexity of the legacy systems.
    - Mitigation: Conduct a pilot integration with a small subset of services to uncover potential issues early.
    - Monitoring: Track integration progress through weekly status reports.
2. **Risk 2**: The project schedule is at risk due to the limited availability of skilled mobile developers.
    - Mitigation: Hire additional developers or consultants with experience in mobile app development.
    - Monitoring: Monitor team capacity and project velocity in each sprint.

1.10 Software Engineering Ethics and Professional Practice

Software engineering ethics and professional practices are vital to ensuring that software engineers conduct themselves responsibly and in alignment with both legal and ethical standards. Software engineers must not only focus on the technical aspects of their work but also on the broader impact of their work on society.

1.10.1 Code of Ethics and Professional Conduct

Software engineers are expected to adhere to ethical standards established by professional bodies such as the **ACM (Association for Computing Machinery)** and **IEEE (Institute of Electrical and Electronics Engineers)**. These codes of conduct define responsibilities for professional behavior.

Fundamental Ethical Principles:

1. **Public**: Software engineers should act consistently with the public interest.
2. **Client and Employer**: Act in the best interest of the client and employer, provided it is consistent with the public interest.
3. **Product**: Ensure that products meet the highest possible professional standards.
4. **Judgment**: Maintain integrity and independence in professional judgment.

5. **Management**: Promote ethical approaches in the management of software development and maintenance.
6. **Profession**: Advance the integrity and reputation of the profession.
7. **Colleagues**: Be fair and supportive of colleagues.
8. **Self**: Participate in lifelong learning regarding the practice of the profession.

1.10.2 Key Ethical Issues in Software Engineering

1 **Privacy**: Software engineers must ensure that systems respect the privacy of individuals. Collecting and storing personal data should be done responsibly and securely.

2 **Confidentiality**: Engineers often have access to sensitive information. It is essential to maintain confidentiality and not disclose proprietary information unless required by law.

3 **Intellectual Property**: Respect for intellectual property is crucial. Software engineers must avoid unauthorized use or distribution of software, algorithms, or data.

4 **Safety-Critical Systems**: Engineers working on safety-critical systems (e.g., aviation, healthcare, and transportation systems) must ensure that these systems function correctly and reliably to avoid harm to users or the public.

5 **Conflict of Interest**: Engineers should avoid situations where personal interests conflict with their professional duties, ensuring decisions are made based on professional integrity.

1.10.3 Professional Practice in Software Development

Professional practice in software engineering encompasses the ethical standards, responsibilities, and behaviors that engineers must uphold to ensure high-quality, reliable, and safe software development. Key aspects include:

1. **Adherence to Ethical Codes**: Following guidelines from organizations like ACM and IEEE to prioritize public safety, honesty, and confidentiality.
2. **Accountability**: Taking responsibility for the consequences of software products and addressing issues that arise.
3. **Quality and Safety Standards**: Delivering high-quality software, particularly in safety-critical domains, through rigorous testing and validation.
4. **Continuous Learning**: Committing to lifelong education to stay current with technology and industry practices.

5. **Effective Communication**: Clearly conveying technical information to clients and stakeholders, fostering collaboration.
6. **Ethical Decision-Making**: Balancing business needs with ethical responsibilities, especially regarding deadlines and product quality.
7. **Intellectual Property Respect**: Complying with laws related to copyrights and licenses and ensuring proper use of third-party resources.
8. **Sustainability**: Designing software that minimizes environmental impact and promotes energy efficiency.
9. **Risk Management**: Identifying and mitigating technical and project risks throughout the development process.
10. **Mentorship and Leadership**: Guiding junior engineers and fostering an ethical work environment to promote continuous improvement.

***