

MODULE 5

SOFTWARE QUALITY

13.1 INTRODUCTION

Quality:

- Quality is generally agreed to be ‘**a good thing**’.
- In a practice what people really mean by the ‘quality’ of a system can be vague, undefined attribute.
- We need to define precisely what qualities we require of a system.

Software Quality means different things to different people depends on the role in the project.

Ex: Consider the below example:

Amanda: Represents software developer actively involved in the creation of the system.

Brigitte: Represents customer or decision maker evaluating software package for their organization.

Objective assessment:

- The focus on software quality depends on whether the person is involved in building the system or selecting and using the final system.
- However, we need to go further - we need to judge objectively whether a system meets our quality requirements and this needs measurement.

Now days, delivering a high-quality product is one of the major objectives of all organizations. Traditionally, the quality of a product means that how much it gets fit into its specified purpose. A product is of *good quality if it performs according to the user's requirements*. Good quality software should meet all objectives defined in the SRS document. It is the responsibility of the quality managers to ensure that the software attains a required level of quality.

Waiting until the system is complete to measure quality is too late.

During development, it's important to:

- Assess the likely quality of the final system.
- Ensure development methods will produce the desired quality.

Potential customers, like Amanda at IOE, might focus on:

- Checking if suppliers use the best development methods.
- Ensuring these methods will lead to the required quality in the final system.

13.2 THE PLACE OF SOFTWARE QUALITY IN PROJECT PLANNING

Quality will be of concern at all stages of project planning and execution, but will be particular interest at Stepwise framework .

Step 1 : Identifying project scope and objectives Define specific objectives that relate to the quality of attributes the application to be delivered. These could include performance, reliability, security, usability, maintainability, and scalability goals.

Step 2 : Identifying project infrastructure :Define installation standards and procedures that ensures the quality of installation processes. This includes setting up guidelines for installation, reliability, consistency, and completeness.

Step 3 : Analyze project characteristics. In this activity the application to be implemented will be examined to see if it has any special quality requirements.

Example: Safety-critical applications might require additional activities such as n-version development, where multiple teams develop versions of the same software to cross-check outputs.

Step 4 : Identify the product and activities of the project. It is at that point the entry, exit and process requirement are identified for each activity. Break down the project into manageable activities, ensuring each is planned with quality measures in place.

Step 5:Estimate Effort for Each Activity: Accurate effort estimation is essential to allocate sufficient resources for quality assurance activities, avoiding rushed and low-quality outputs.

Step 6: Identify Activity Risks: Identifying risks early allows for planning mitigation strategies to maintain quality throughout the project

Step 7: Allocate Resources: Allocate resources not just for development but also for quality assurance tasks like testing, code reviews, and quality audits.

Step 8: Review and publicize plan. At this stage the overall quality aspects of the project plan are reviewed. This includes ensuring that quality objectives are clearly defined, measurable, achievable, relevant, and time -bound(SMART criteria).

Step 9: Execute Plan: Execute the project plan with a focus on adhering to quality standards, monitoring progress, and making necessary adjustments to maintain quality.

Step 10: Lower-Level Planning: Detailed planning at lower levels should include specific quality assurance activities tailored to each phase or component of the project.

Review (Feedback Loop): Continuous review and feedback loops help in maintaining and improving quality throughout the project lifecycle.

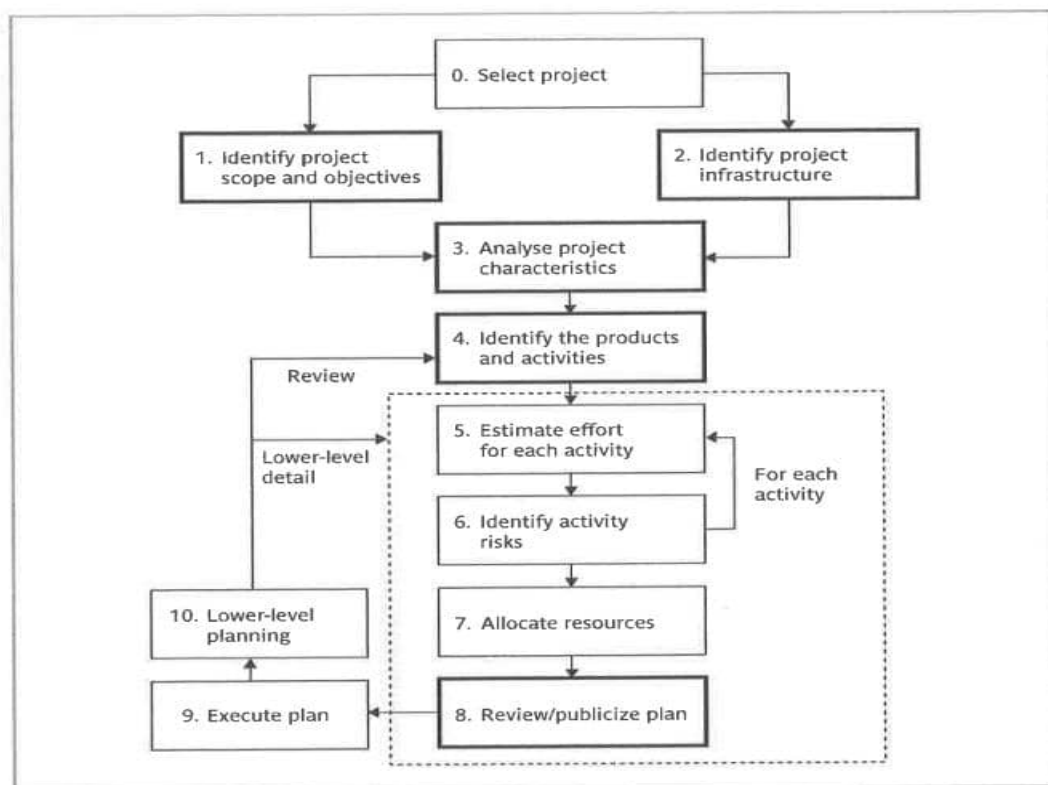


FIGURE 13.1 The place of software quality in Step Wise

13.3 THE IMPORTANCE OF SOFTWARE QUALITY

Now a days, **quality is the important aspect of all organization.** Good quality software is the requirement of all users. There are so many reasons that describe why the quality of software is important; few among of those which are most important are described below:

Increasing criticality of software:

- The final customer or user is naturally anxious about the general quality of software especially about the reliability.
- They are concern about the safety because of their dependency on the software system such as aircraft control system are more safety critical systems.

Earlier detection of errors during development-

- As software is developed through a number of phases; output of one phase is given as input to the other one. So, if error in the initial phase is not found, then at the later stage, it is difficult to fix that error and also the cost indulged is more.

The intangibility of software :

- Difficulty in verifying the satisfactory completion of project tasks.
- Tangibility is achieved by requiring developers to produce "deliverables" that can be examined for quality at each project stage, which can be inspected for quality to ensure tasks are completed satisfactorily.

Accumulating errors during software development :

- Errors in earlier steps can propagate and accumulate in later steps.
- Errors found later in the project are more expensive to fix.
- The unknown number of errors makes the debugging phase difficult to control.

13.4 DEFINING SOFTWARE QUALITY

Defining software quality involves understanding both external user perceptions and internal developer considerations.

External and Internal Quality Mapping: External qualities, such as usability perceived by users, must align with internal factors like well-structured code, which can enhance **reliability** by reducing errors. Mapping these external qualities to measurable internal factors helps ensure alignment with user expectations.

- 1) **Measurable Standards:** Standards like BS ISO/IEC 15939:2007 provide frameworks for measuring software quality. For instance, measuring faults per thousand lines of code offers a more meaningful metric than simply counting total faults. This approach helps in quantifying quality relative to the size and complexity of the software.

- 2) **Direct and Indirect Measurement:** Quality measures can be direct(measuring the quality itself) or indirect (measuring indicators of quality presence). For example user help desk inquiries about software usability reflect its usability quality. Such measures help in clarifying and communicating the presence of desired qualities.
- 3) **Setting Quality Targets:** Project managers set quality targets based on these measurements, ensuring that improvements are meaningful and align with project goals. For instance, increasing the number of errors found in program inspections can reflect the thoroughness of the inspection process, thereby improving overall software quality.
- 4) **Quality Specification:** When specific quality characteristics are essential, a quality specification should include:

- **Definition/Description**

Definition: Clear definition of the quality characteristic.

Description: Detailed description of what the quality characteristic entails.

- **Scale or Unit of Measurement:**

The unit used to measure the quality characteristic (e.g., faults per thousand lines of code).

- **Test**

Practical Test: The method or process used to test the extent to which the quality attribute exists.

- **Minimally Acceptable**

Worst Acceptable Value: The lowest acceptable value, below which the product would be rejected.

- **Target Range**

Planned Range: The range of values within which it is planned that the quality measurement value should lie.

- **Current Value**

Now: The value that applies currently to the quality characteristic.

Measurements Applicable to Quality Characteristics in Software

When assessing quality characteristics in software, multiple measurements may be applicable. For example, in the case of reliability, measurements could include:

1. Availability:

Definition: Percentage of a particular time interval that a system is usable.

Scale: Percentage (%).

Test: Measure the system's uptime versus downtime over a specified period.

Minimally Acceptable: Typically, high availability is desirable; specifics depend on system requirements.

Target Range: E.g., 99.9% uptime.

2. Mean Time Between Failures (MTBF):

Definition: Total service time divided by the number of failures.

Scale: Time (e.g., hours, days).

Test: Calculate the average time elapsed between system failures.

Minimally Acceptable: Longer MTBF indicates higher reliability; minimum varies by system criticality.

Target Range: E.g., MTBF of 10,000 hours.

3. Failure on Demand:

Definition: Probability that a system will not be available when required, or probability that a transaction will fail.

Scale: Probability (0 to 1).

Test: Evaluate the system's response to demand or transaction processing.

Minimally Acceptable: Lower probability of failure is desired; varies by system criticality.

Target Range: E.g., Failure on demand probability of less than 0.01.

4. Support Activity:

Definition: Number of fault reports generated and processed.

Scale: Count (number of reports).

Test: Track and analyze the volume and resolution time of fault reports.

Minimally Acceptable: Lower number of fault reports indicates better reliability.

Target Range: E.g., Less than 10 fault reports per month.

Maintainability and Related Qualities:

- **Maintainability:** How quickly a fault can be corrected once detected.
- **Changeability:** Ease with which software can be modified.
- **Analyzability:** Ease with which causes of failure can be identified, contributing to maintainability. These measurements help quantify and assess the reliability and maintainability of software systems, ensuring they meet desired quality standards.

13.5 SOFTWARE QUALITY MODELS

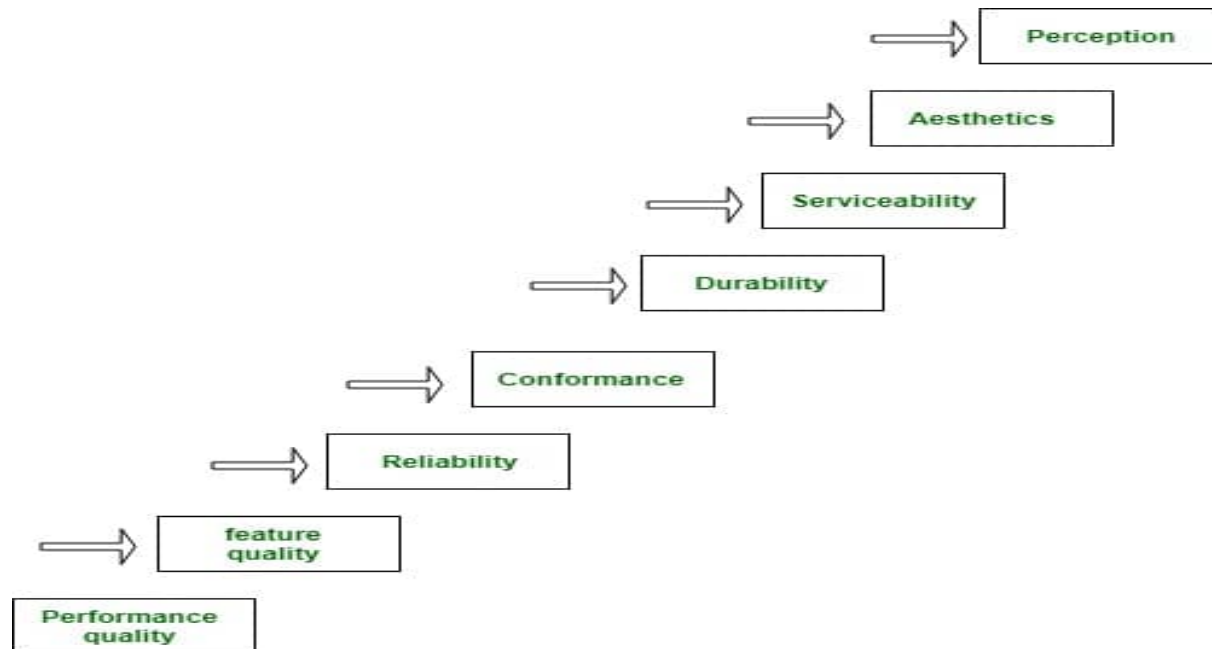
It is hard to directly measure the quality of a software. However, it can be expressed in terms of several attributes of a software that can be directly measured.

The quality models give a characterization (hierarchical) of software quality in terms of a set of characteristics of the software. The bottom level of the hierarchical can be directly measured, thereby enabling a quantitative assessment of the quality of the software.

There are several well-established quality Models including McCall's, Dromey's and Boehm's. Since there was no standardization among the large number of quality models that became available, the ISO 9126 model of quality was developed.

Garvin's Quality Dimensions: David Garvin, a professor of Harvard Business school, defined the quality of any product in terms of eight general attributes of the product.

- **Performance:** How well it performs the jobs.
- **Features:** How well it supports the required features.
- **Reliability:** Probability of a product working satisfactorily within a specified period of time.
- **Conformance:** Degree to which the product meets the requirements.
- **Durability:** Measure of the product life.
- **Serviceability:** Speed and effectiveness maintenance.
- **Aesthetics:** The look and feel of the product.
- **Perceived quality:** User's opinion about the product quality.



Garvin's Dimensions of Quality

1) McCall' Model: McCall's Software Quality Model was introduced in 1977. This model is incorporated with many attributes, termed software factors, which influence software. The model distinguishes between two levels of quality attributes:

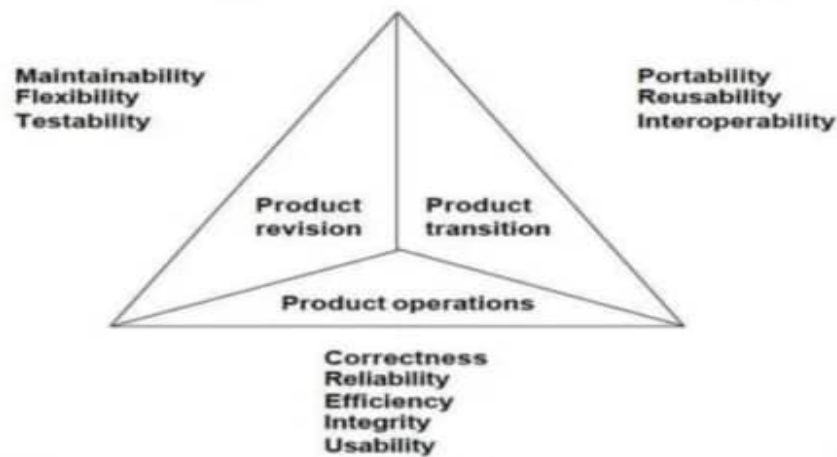
- Quality Factors
- Quality Criteria

Quality Factors: The higher-level quality attributes that can be accessed directly are called quality factors. These attributes are external. The attributes at this level are given more importance by the users and managers.

Quality Criteria: The lower or second-level quality attributes that can be accessed either subjectively or objectively are called Quality Criteria. These attributes are internal. Each quality factor has many second-level quality attributes or quality criteria.

McCall defined the quality of a software in terms of three broad parameters: its operational characteristics, how easy it is to fix defects and how easy it is to part it to different platforms. These three high-level quality attributes are defined based on the following 11 attributes of the software:

McCall's Quality Model Triangle



Product Operations:

Correctness: The extent to which a software product satisfies its specifications.

Reliability: The probability of the software product working satisfactorily over a given duration.

Efficiency: The amount of computing resources required to perform the required functions.

Integrity: The extent to which the data of the software product remain valid.

Usability: The effort required to operate the software product.

Product Revision:

Maintainability: The ease with which it is possible to locate and fix bugs in the software product.

Flexibility: The effort required to adapt the software product to changing requirements.

Testability: The effort required to test a software product to ensure that it performs its intended function.

Product Transition:

Portability: The effort required to transfer the software product from one hardware or software system environment to another.

Reusability: The extent to which a software can be reused in other applications.

Interoperability: The effort required to integrate the software with other software.

- 2) **Dromey's model:** Dromey proposed that software product quality depends on four major high-level properties of the software: Correctness, internal characteristics, contextual characteristics and certain descriptive properties. Each of these high-level properties of a software product, in turn depends on several lower-level quality attributes. Dromey's hierarchical quality model is shown in Fig 13.2

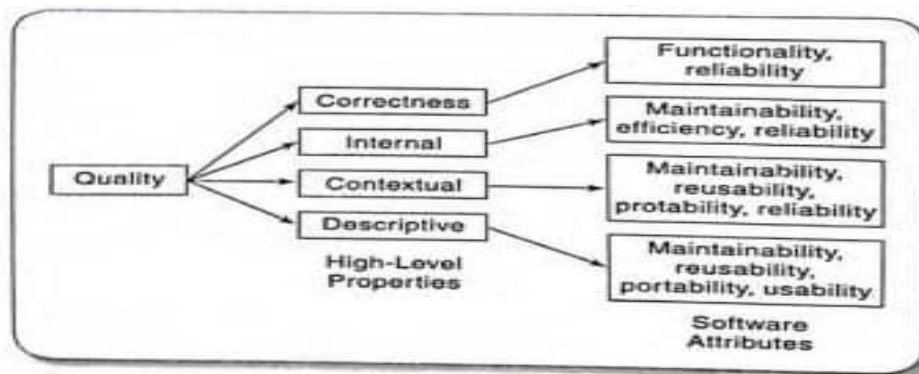


FIGURE 13.2 Dromey's quality model

- 3) **Boehm's Model:** The model represents a hierarchical quality model similar to the McCall Quality Model to define software quality using a predefined set of attributes and metrics, each of which contributes to the overall quality of software. The difference between Boehm's and McCall's Models is that McCall's Quality Model primarily focuses on precise measurement of high-level characteristics, whereas Boehm's Quality Model is based on a wider range of characteristics.

These three high-level characteristics are the following:

As-is -utility: How well (easily, reliably and efficiently) can it be used?

Maintainability: How easy is to understand, modify and then retest the software?

Portability: How difficult would it be to make the software in a changed environment?

Boehm's expressed these high-level product quality attributes in terms of several measurable product attributes. Boehm's hierarchical quality model is shown in **Fig 13.3** .

Quality Factors Associated with Boehm's Model

The next level of Boehm's hierarchical model consists of seven quality factors associated with three primary uses, stated below:

Portability: Effort required to change the software to fit in a new environment.

Reliability: The extent to which software performs according to requirements.

Efficiency: Amount of hardware resources and code required to execute a function.

Usability (Human Engineering): Extent of effort required to learn, operate and understand functions of the software.

Testability: Effort required to verify that software performs its intended functions.

Understandability: Effort required for a user to recognize a logical concept and its applicability.

Modifiability: Effort required to modify software during the maintenance phase.

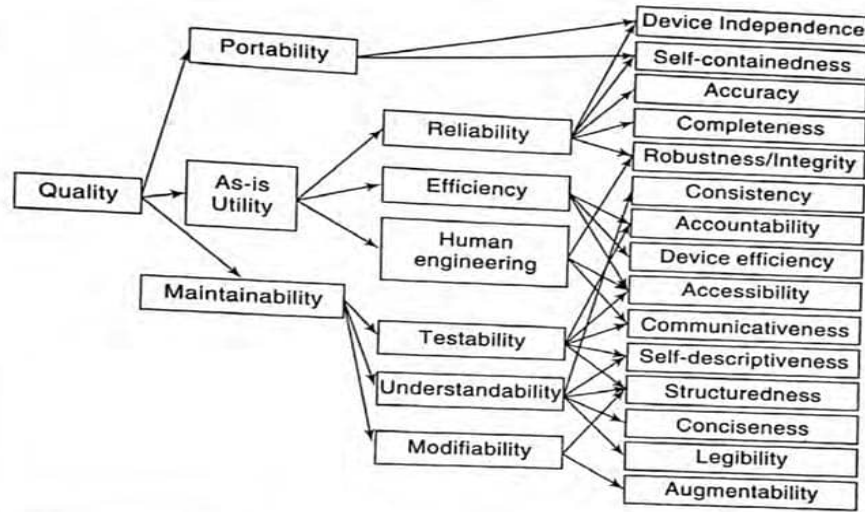


FIGURE 13.3 Boehm's quality model

13.7 PRODUCT AND PROCESS METRICS

Users assess the quality of a software product based on its external attributes, whereas during development, the developers assess the product's quality based on various internal attributes.

The internal attributes may measure either some aspects of product or of the development process (called process metrics).

1. **Product Metrics**
2. **Process Metrics**

1. Product Metrics:

Purpose: Measure the attributes and characteristics of the software product being developed. These metrics focus on evaluating the product's size, complexity, effort, and quality, providing insights to ensure the product meets performance and quality standards.

Examples:

- **Size Metrics:** Such as Lines of Code (LOC) and Function Points, which quantify the size or complexity of the software. Helps in comparing projects with different technologies or programming languages.

- **Effort Metrics:** Like Person-Months (PM), which measure the effort required to develop the software. Helps assess whether resources are allocated efficiently.
- **Time Metrics:** Such as the duration in months or other time units needed to complete the development. Tracks development speed and adherence to timelines.

2. Process Metrics:

Purpose: Measure the effectiveness and efficiency and overall quality of the software of the development process itself. They are used by teams to identify bottlenecks, improve productivity, and ensure continuous process improvement.

Examples:

- **Review Effectiveness:** Measures how thorough and effective code reviews are in finding defects.
- **Defect Metrics:** Average number of defects found per hour of inspection, average time taken to correct defects, and average number of failures detected during testing per line of code.
- **Productivity Metrics:** Measures the efficiency of the development team in terms of output per unit of effort or time.
- **Quality Metrics:** Such as the number of latent defects per line of code, which indicates the robustness of the software after development.

Differences:

- **Focus:** Product metrics focus on the characteristics of the software being built (size, effort, time), while process metrics focus on how well the development process is performing (effectiveness, efficiency, quality).
- **Use:** Product metrics are used to gauge the attributes of the final software product, aiding in planning, estimation, and evaluation. Process metrics help in assessing and improving the development process itself, aiming to enhance quality, efficiency, and productivity.
- **Application:** Product metrics are typically applied during and after development phases to assess the product's progress and quality. Process metrics are applied throughout the development lifecycle to monitor and improve the development process continuously.

By employing both types of metrics effectively, software development teams can better manage projects, optimize processes, and deliver high-quality software products that meet user expectations.

13.8 PRODUCT VERSUS PROCESS QUALITY MANAGEMENT

In software development, managing quality can be approached from two main perspectives: product quality management and process quality management. Here's a breakdown of each approach and their key aspects:

1. **Product Quality Management**
2. **Process Quality Management**

Product Quality Management

Product quality management focuses on evaluating and ensuring the quality of the software product itself. This approach is typically more straightforward to implement and measure after the software has been developed.

Aspects:

1. Measurement Focus: Emphasizes metrics that assess the characteristics and attributes of the final software product, such as size (LOC, function points), reliability (defects found per LOC), performance (response time), and usability (user satisfaction ratings).

2. Evaluation Timing: Product quality metrics are often measured and evaluated after the software product has been completed or at significant milestones during development.

3. Benefits:

- Provides clear benchmarks for evaluating the success of the software development project.
- Facilitates comparisons with user requirements and industry standards.
- Helps in identifying areas for improvement in subsequent software versions or projects.

4. Challenges:

- Predicting final product quality based on intermediate stages (like early code modules or prototypes) can be challenging.
- Metrics may not always capture the full complexity or performance of the final integrated product.

Process Quality Management

Process quality management focuses on assessing and improving the quality of the development processes used to create the software. This approach aims to reduce errors and improve efficiency throughout the development lifecycle.

Aspects:

1. Measurement Focus: Emphasizes metrics related to the development processes themselves, such as defect detection rates during inspections, rework effort, productivity (e.g., lines of code produced per hour), and adherence to defined standards and procedures.

2. Evaluation Timing: Process quality metrics are monitored continuously throughout the development lifecycle, from initial planning through to deployment and maintenance.

3. Benefits:

- Helps in identifying and correcting errors early in the development process, reducing the cost and effort of rework.
- Facilitates continuous improvement of development practices, leading to higher overall quality in software products.
- Provides insights into the effectiveness of development methodologies and practices used by the team.

4. Challenges:

- Requires consistent monitoring and analysis of metrics throughout the development lifecycle.
- Effectiveness of process improvements may not always translate directly into improved product quality without careful management and integration.

Integration and Synergy

- While product and process quality management approaches have distinct focuses, they are complementary.
- Effective software development teams often integrate both approaches to achieve optimal results.
- By improving process quality, teams can enhance product quality metrics, leading to more reliable, efficient, and user-friendly software products.

CHAPTER 2

SOFTWARE PROJECT ESTIMATION

- Software project management begins with a set of activities that are collectively called **project planning**. ----Estimation, Scheduling, Risk analysis, Quality management planning, and Change management planning
- Estimation determines how much money, effort, resources, and time it will take to build a specific system or product
- The software team first estimates
 - The work to be done
 - The resources required
 - The time that will elapse from start to finish
- Then they establish a project schedule that
 - Defines tasks and milestones
 - Identifies who is responsible for conducting each task
 - Specifies the inter-task dependencies

1.1 OBSERVATION ON ESTIMATION

- Planning requires technical managers and the software team to make an initial commitment.
- Process and project metrics can provide a historical perspective and valuable input for generation of quantitative estimates .
- Past- experience can aid greatly as estimates are developed and reviewed.
- Estimation carries inherent risk, and this risk leads to uncertainty.
- Project size is another important factor that can affect the accuracy and efficacy of estimates. As size increases, the interdependency among various elements of the software grows rapidly.
- The degree of structural uncertainty also influences estimation risk.
- The availability of historical information has a strong influence on estimation risk.
- When software metrics are available from past projects
 - Estimates can be made with greater assurance.
 - Schedules can be established to avoid past difficulties.
 - Overall risk is reduced
- Estimation risk is measured by the degree of uncertainty in the quantitative estimates for cost, schedule, and resources .

- Nevertheless, a project manager should not become obsessive about estimation
 - Plans should be iterative and allow adjustments as time passes and more is made certain.

1.2 SOFTWARE PROJECT ESTIMATION

Software cost and effort estimation is dependent on too many variables—human, technical, environmental, political—which affect the ultimate cost of software and effort applied to develop it. However, software project estimation 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, several options arise:

1. Delay estimation until late in the project (obviously, we can achieve 100 per cent accurate estimates after the project is complete!). ---But this is not a practical approach since cost estimates must be provided up-front.
2. Base estimates on similar projects that have already been completed -It works only well when current project is similar to past efforts and other project influences (e.g., the customer, business conditions, the software engineering environment, deadlines, Unfortunately, past experience has not always been a good indicator of future results.
3. Use relatively simple decomposition techniques to generate project cost and effort estimates. --Decomposition techniques take a **divide-and-conquer approach** to software project estimation. This provides cost and effort estimation in a stepwise fashion.
4. Use one or more empirical models for software cost and effort estimation. It offers a potentially valuable estimation approach. A model is based on experience (historical data) and takes the form

$$d = f(v_i)$$

where d is one of a number of estimated values (e.g., effort, cost, project duration) and v_i are selected independent parameters (e.g., estimated LOC or FP).

1.3 DECOMPOSITION TECHNIQUES

Software project estimation is a form of problem solving. If a problem needs to be solved (i.e., developing a cost and effort estimate for a software project) is too complex to be considered in one piece. For this reason, you should decompose the problem, recharacterizing it as a set of smaller (and hopefully, more manageable) problems.

- Before an estimate can be made and decomposition techniques applied, the planner must
 - Understand the scope of the software to be built
 - Generate an estimate of the software's size
- Then one of two approaches are used
 - Problem-based estimation ---Based on either source lines of code or function point estimates
 - Process-based estimation ---Based on the effort required to accomplish each task.

1.3.1 Software Sizing

In the context of project planning, size refers to a quantifiable outcome of the software project.

If a direct approach is taken, size can be measured in lines of code (LOC). If an indirect approach is chosen, size is represented as function points (FP).

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

- Fuzzy Logic Sizing
 - To apply this approach, the planner must identify the type of application, establish its magnitude on a qualitative scale, and then refine the magnitude within the original range.
- Function point sizing
 - Develop 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
 - Types of modifications include reuse, adding code, changing code, and deleting code
 - An effort ratio is then used to estimate each type of change and the size of the change.

1.3.2 Problem -Based Estimation

- 1) Start with a bounded statement of scope.
 - 2) Decompose the software into problem functions that can each be estimated individually
 - 3) Compute an LOC or FP value for each function.
 - 4) Derive cost or effort estimates by applying the LOC or FP values to your baseline productivity metrics (e.g., LOC/person-month or FP/person-month)
 - 5) 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.
 - 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.

pm=person month

- For both approaches, the planner uses lessons learned to estimate an optimistic, most likely, and pessimistic size value for each function or count (for each information domain value)
- Then the expected size value S is computed as follows:

$$S = (S_{\text{opt}} + 4S_m + S_{\text{pess}})/6$$

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

1.3.3 An Example of LOC-Based Estimation

As an example of LOC (Lines of code) and FP (Function Point) problem-based estimation techniques.

LOC-based estimation is a method for predicting the size, effort, and cost of a software project by estimating the **total lines of code (LOC)** required to implement the project. It is particularly useful when the project's scope and functional requirements are well-defined.

Example: Consider the development of a **Computer-Aided Design (CAD)** application for mechanical components. Key details include:

- The application must interface with multiple peripherals, such as a mouse, digitizer, high-resolution display, and laser printer.
- Functionalities like **user interface control, 2D and 3D geometric analysis, database management, and design analysis** are identified.
- The estimation begins with a preliminary description of the software's scope, which must be refined further into measurable functions.

Following the decomposition technique for LOC, an estimation table (Figure 26.2) is developed

The project is divided into smaller, measurable **functional components**, each assigned an estimated LOC range. For example:

- **User interface and control facilities (UICF):** 2,300 LOC
- **3D geometric analysis (3DGA):** 6,800 LOC
- **Design analysis modules (DAM):** 8,400 LOC

FIGURE 26.2

Estimation
table for the
LOC methods

Function	Estimated LOC
User interface and control facilities (UICF)	2,300
Two-dimensional geometric analysis (2DGA)	5,300
Three-dimensional geometric analysis (3DGA)	6,800
Database management (DBM)	3,350
Computer graphics display facilities (CGDF)	4,950
Peripheral control function (PCF)	2,100
Design analysis modules (DAM)	8,400
<i>Estimated lines of code</i>	<i>33,200</i>

Historical Data and Productivity:

The estimation uses **historical productivity data** to convert LOC into effort and cost:

- **Average productivity:** 620 LOC per person-month.
- **Labor cost per person-month:** \$8,000.

- **Cost per line of code:** Approximately \$13.

Based on this data:

- The **effort required** is $\frac{33,200 \text{ LOC}}{620 \text{ LOC/person-month}} = 54 \text{ person-months}$.
- The **project cost** is $33,200 \text{ LOC} \times 13 = \$431,000$.

1.3.4 An Example of FP-Based Estimation

Decomposition for Function Point (FP) estimation measures the **size** and **complexity** of a software project based on its functional requirements. Unlike LOC-based estimation, which depends on lines of code, FP focuses on the software's functionalities, making it independent of the programming language or coding style.

- 1) It focuses on identifying information Domain Values. Referring to the table presented in Figure 26.3, you would estimate inputs, outputs, inquiries, files, and external interfaces for the CAD software.

FIGURE 26.3

Estimating
information
domain values

Information domain value	Opt.	Likely	Pess.	Est. count	Weight	FP count
Number of external inputs	20	24	30	24	4	97
Number of external outputs	12	15	22	16	5	78
Number of external inquiries	16	22	28	22	5	88
Number of internal logical files	4	4	5	4	10	42
Number of external interface files	2	2	3	2	7	15
Count total						320

- 2) Each component is weighted based on its complexity (low, average, or high). In this example:
 - Likely values are used for the **estimated count**.
 - Weights are predefined (e.g., inputs = 4, outputs = 5, etc.).
- 3) **Calculating the FP Count:** The FP count is calculated by multiplying the estimated count of each component by its weight and summing up the results.

Example:

$$\text{FP Count} = (24 \times 4) + (16 \times 5) + (22 \times 5) + (4 \times 10) + (2 \times 7) = 320$$

- 4) **Applying the Value Adjustment Factor (VAF):** The **complexity adjustment factor** accounts for system-specific attributes like performance, reusability, and flexibility. A formula is applied

Factor	Value
Backup and recovery	4
Data communications	2
Distributed processing	0
Performance critical	4
Existing operating environment	3
Online data entry	4
Input transaction over multiple screens	5
Master files updated online	3
Information domain values complex	5
Internal processing complex	5
Code designed for reuse	4
Conversion/installation in design	3
Multiple installations	5
Application designed for change	5
Value adjustment factor	1.17

$$FP_{estimated} = \text{Count Total} \times [0.65 + (0.01 \times \sum F_i)]$$

$\sum F_i$ = Sum of complexity adjustment factors.

VAF = 1.17 in this example.

Final FP Estimate = $320 \times 1.17 = 375$.

After determining the FP estimate:

- Average productivity: 6.5 FP/person-month.
- Cost per FP: Approximately \$1,230.
- Effort required:

$$\text{Effort (person-months)} = \frac{FP}{\text{Productivity}} = \frac{375}{6.5} = 58 \text{ person-months.}$$

- Total cost

$$\text{Cost} = 375 \times 1,230 = 461,000 \text{ USD.}$$

1.3.5 Process Based Estimation

The process is decomposed into a relatively small set of tasks and the effort required to accomplish each task is estimated. Process-based estimation begins with:

- 1) Identify the set of functions that the software needs to perform as obtained from the project scope.
- 2) Identify the series of framework activities that need to be performed for each function.
- 3) Estimate the effort (in person months) that will be required to accomplish each software process activity for each function.
- 4) Apply average labor rates (i.e., cost/unit effort) to the effort estimated for each process activity
- 5) Compute the total cost and effort for each function and each framework activity as in table 26.4.
- 6) 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.

FIGURE 26.4

Process-based
estimation
table

Activity →	CC	Planning	Risk analysis	Engineering		Construction release		CE	Totals
Task →				Analysis	Design	Code	Test		
Function ↓									
UICF				0.50	2.50	0.40	5.00	n/a	8.40
2DGA				0.75	4.00	0.60	2.00	n/a	7.35
3DGA				0.50	4.00	1.00	3.00	n/a	8.50
CGDF				0.50	3.00	1.00	1.50	n/a	6.00
DBM				0.50	3.00	0.75	1.50	n/a	5.75
PCF				0.25	2.00	0.50	1.50	n/a	4.25
DAM				0.50	2.00	0.50	2.00	n/a	5.00
Totals	0.25	0.25	0.25	3.50	20.50	4.50	16.50		46.00
% effort	1%	1%	1%	8%	45%	10%	36%		

CC = customer communication CE = customer evaluation

1.3.6 An Example of Process-Based Estimation

This estimation uses the **CAD software** as an example, referring to the completed process-based table shown in Figure 26.4, estimates of effort (in person-months) for each software engineering activity are provided for each CAD software function.

The framework activities (or phases of the software life cycle) typically include:

- Customer communication
- Planning and risk analysis
- Requirements analysis

- Design
- Coding
- Testing
- Deployment

Each activity is further subdivided into specific tasks (e.g., requirements analysis, user interface design, module coding, unit testing).

For the **CAD software example**, a table (Figure 26.4) details the effort estimates (in person-months) for each **software function** under each activity.

The **front-end engineering tasks** (requirements analysis and design) account for **53% of the total effort**, [45%+8%] emphasizing their importance in the overall project.

Using a **burdened labor rate** of \$8000 per month:

$$\text{Total Cost} = \text{Total Effort (person-months)} \times \text{Labor Rate}$$

- For 46 person-months:

$$\text{Total Cost} = 46 \times 8000 = 368,000 \text{ USD}$$

1.3.6 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 (e.g., interactions) that involve many functions and features.

Use case-based estimation leverages the information in **use cases** to predict the **Lines of Code (LOC)** or **effort** required for development. However, this approach requires several adjustments and considerations due to the variability and abstraction levels of use cases.

The formula to estimate LOC from use cases is:

$$LOC_{estimate} = N \times LOC_{avg} + \left[\left(\frac{S_h}{S_d} - 1 \right) + \left(\frac{P_h}{P_d} - 1 \right) \right] \times LOC_{adjust}$$

N : Actual number of use cases.

LOC_{avg} : Historical average LOC per use case for a given subsystem.

LOC_{adjust} : Adjustment based on n% of LOC_{avg} , representing the difference between this project and historical projects.

S_d : Actual scenarios per use case.

S_h : Average scenarios per use case for this subsystem type.

P_d : Actual pages per use case.

P_h : Average pages per use case for this subsystem type.

Assume:

- $N=50$ (50 use cases in the project)
- $LOC_{avg}=500$ (historical average LOC per use case)
- $LOC_{adjust}=50$ (adjustment factor)
- $S_d=12, S_h=10$ (actual vs. historical average scenarios)
- $P_d=8, P_h=6$ (actual vs. historical average pages)
- **Step 1:** Compute adjustments:

$$\frac{S_h}{S_d} - 1 = \frac{12}{10} - 1 = 0.2$$

$$\frac{P_h}{P_d} - 1 = \frac{8}{6} - 1 = 0.33$$

$$LOC_{estimate} = 50 \times 500 + [0.2 + 0.33] \times 50$$

$$LOC_{estimate} = 25,000 + 27 \times 50 = 25,000 + 1,350 = 26,350 \text{ LOC.}$$

Use case-based estimation is a valuable supplementary method when detailed use cases are available. It accounts for the variability in use case complexity by incorporating adjustments based on scenarios and page lengths.

1.3.7 An Example of Use-Case–Based Estimation

FIGURE 26.5

Use-case estimation	use cases	scenarios	pages	scenarios	pages	LOC	LOC estimate
User interface subsystem	6	10	6	12	5	560	3,366
Engineering subsystem group	10	20	8	16	8	3,100	31,233
Infrastructure subsystem group	5	6	5	10	6	1,650	7,970
Total LOC estimate							42,568

1. Use Case Characteristics

- **User Interface Subsystem:**
 - 6 use cases, each with no more than 10 scenarios and 6 pages.
 - Historical average LOC per use case: 560.
 - LOC estimate: 3,366.
- **Engineering Subsystem Group:**
 - 10 use cases, each with no more than 20 scenarios and 8 pages.
 - Historical average LOC per use case: 3,100.
 - LOC estimate: 31,233.
- **Infrastructure Subsystem Group:**
 - 5 use cases, each with no more than 6 scenarios and 5 pages.
 - Historical average LOC per use case: 1,650.
 - LOC estimate: 7,970.
- The total estimated LOC for the CAD software system is the sum of the estimates for all three subsystems:

$$\text{Total LOC} = 3,366 + 31,233 + 7,970 = 42,568$$

Productivity: 620 LOC/person-month.

Labor Rate: \$8,000 per month

Cost per LOC: \$13.

Effort:

$$\text{Effort} = \frac{\text{Total LOC}}{\text{Productivity}} = \frac{42,568}{620} \approx 68.66 \text{ person-months}$$

Total Cost:

$$\text{Cost} = \text{Total LOC} \times \text{Cost per LOC} = 42,568 \times 13 = \$552,000$$

1.3.8 Reconciling Estimates

- The results gathered from the various estimation techniques must be reconciled to produce a single estimate of effort, project duration, and cost.
- If widely divergent estimates occur, investigate the following causes
 - The scope of the project is not adequately understood or has been misinterpreted by the planner
 - Productivity data** used for problem-based estimation techniques is inappropriate for the application, obsolete (i.e., outdated for the current organization), or has been misapplied.
- The planner must determine the cause of divergence and then reconcile the estimates.

1.4 EMPIRICAL ESTIMATION MODELS

- Estimation models for computer software use empirically derived formulas to predict effort as a function of LOC (line of code) or FP(function point).
- Resultant values computed for LOC or FP are entered into an estimation model.
- The empirical data for these models are derived from a limited sample of projects.
 - Consequently, the models should be calibrated to reflect local software development conditions.

1.4.1 The Structure of Estimation Models:

A typical estimation model is derived using regression analysis on data collected from past software projects. The basic structure of these model:

$$E = A + B \times (e_v)^C$$

Where:

- **E: Effort in person-months.**
- **e_v : Estimation variable (e.g., LOC or FP).**
- **A, B, C: Empirically derived constants.**

In addition to the relationship noted in the above equation, the majority of estimation models have some form of project adjustment component that enables E to be adjusted by other project characteristics (e.g. problem-complexity, staff experience, development environment). Among the many **LOC-oriented estimation models proposed are:**

$$E = 5.2 \times (KLOC)^{0.91}$$

$$E = 5.5 + 0.73 \times (KLOC)^{1.16}$$

$$E = 3.2 \times (KLOC)^{1.05}$$

$$E = 5.288 \times (KLOC)^{1.047}$$

Walston-Felix model
Bailey-Basili model
Boehm simple model
Doty model for KLOC > 9

FP-Based Estimation Models have also been proposed: For Function Point (FP)-based estimation:

$$E = -91.4 + 0.355 \text{ FP}$$

Albrecht and Gaffney model

$$E = -37 + 0.96 \text{ FP}$$

Kemerer model

$$E = -12.88 + 0.405 \text{ FP}$$

Small project regression model

1.4.2 The COCOMO II model:

- Stands for Constructive Cost Model.
- Introduced by Barry Boehm in 1981 in his book “Software Engineering Economics”.
- Became one of the well-known and widely used estimation models in the industry.
- It has evolved into a more comprehensive estimation model called COCOMO II.
- COCOMO II is a hierarchy of three estimation models.

COCOMO II Models

- **Application composition model** - Used during the early stages of software engineering when the following are important
 - Prototyping of user interfaces
 - Consideration of software and system interaction
 - Assessment of performance
 - Evaluation of technology maturity
- **Early design stage model** – Used once requirements have been stabilized and basic software architecture has been established.
- **Post-architecture stage model** – Used during the construction of the software.

Like all the estimation models of the software, , it requires sizing information and accepts it in three forms: object points, function points, and lines of source code.

The COCOMO II application composition model uses object points and is illustrated in the Fig 26.6

FIGURE 26.6

Complexity weighting for object types.

Source: [Boe96].

Object type	Complexity weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL component			10

- Like function points, the object point is an indirect software measure that is computed using counts of the number of (1) screens (at the user interface), (2) reports, and (3) components likely to be required to build the application.
- Each object instance (e.g., a screen or report) is classified into one of three complexity levels (i.e., simple, medium, or difficult).
- Once complexity is determined, the number of screens, reports, and components are weighted according to the table illustrated in Figure 26.6.
- The object point count is then determined by multiplying the original number of object instances by the weighting factor in the figure and summing to obtain a total object point count.
- When component-based development or general software reuse is to be applied, the percent of reuse (%reuse) is estimated, and the object point count is adjusted.

$$NOP = (\text{object point}) \times [(100 - \% \text{reuse}) / 100]$$

where NOP is defined as new object points.

- To derive an **estimate of effort based on the computed NOP** value, a “productivity rate” must be derived. Figure 26.7 presents the productivity rate for different levels of developer experience and development environment maturity.

$$PROD = \frac{NOP}{\text{person} - \text{month}}$$

FIGURE 26.7 Productivity rate for object points.

Source: [Boe96].

Developer's experience/capability	Very low	Low	Nominal	High	Very high
Environment maturity/capability	Very low	Low	Nominal	High	Very high
PROD	4	7	13	25	50

- Once the productivity rate has been determined, an estimate of project effort is computed using

$$\text{Estimated Effort} = \frac{NOP}{PROD}$$

1.4.3 The Software Equation

The **software equation** is a dynamic multivariable model proposed by Putnam and Myers to estimate effort based on productivity data. The equations used are:

$$E = \frac{LOC \times B^{0.333}}{P^3} \times \frac{1}{t^{41}}$$

where

E = effort in person-months or person-years

t = project duration in months or years

B = "special skills factor"¹³

P = "productivity parameter"

Typical values might be $P=2000$ for development of real-time embedded software, $P=10,000$ for telecommunication and systems software, and $P=28,000$ for business systems applications.

The software equation has two different parameters:

- 1) An estimate of size (in LOC) and 2) an indication of project duration in calendar months or years.

To simplify the estimation process and use a more common form for their estimation model, Putnam and Myers [Put92] suggest a set of equations derived from the software equation. Minimum development time is defined as

$$t_{\min} = 8.14 \frac{LOC}{P^{0.43}} \text{ in months for } t_{\min} > 6 \text{ months} \quad (26.5a)$$

$$E = 180 B t^3 \text{ in person-months for } E \geq 20 \text{ person-months} \quad (26.5b)$$

Note that t in Equation (26.5b) is represented in years.

Using Equation (26.5) with $P = 12,000$ (the recommended value for scientific software) for the CAD software discussed earlier in this chapter,

$$t_{\min} = 8.14 \times \frac{33,200}{12,000^{0.43}} = 12.6 \text{ calendar months}$$

$$E = 180 \times 0.28 \times (1.05)^3 = 58 \text{ person-months}$$

COCOMO II is a robust estimation framework that combines object points, function points, and LOC based sizing methods. It uses complexity weighting, reuse factors, and productivity data to provide accurate cost, effort, and schedule estimates. Combined with the **Software Equation**, it ensures a dynamic, data-driven approach to project estimation.