ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore - 560 028

DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

LECTURE NOTES

COURSE: SOFTWARE ENGINEERING & PROJECT MANAGEMENT

COURSE CODE: BCS501

SEMESTER: V

COURSE COORDINATOR: Mr. Chetan G

(ACADEMIC YEAR 2026-27)

MODULE 5

SOFTWARE QUALITY and SOFTWARE PROJECT ESTIMATION

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.

Objective assessment:

- We need to judge objectively whether a system meets our quality requirements and this need measurement.
- Critical for package selection, e.g., Brigette at Brightmouth College.

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**.**

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 Some objective could relate to the quality of the application to be delivered.

Step 2 : Identifying project infrastructure Within this step activity 2.2 involves identifying installation standards and procedures. Some of these will almost certainly be about quality requirements.

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.

Step 8: Review and publicize plan. At this stage the overall quality aspects of the project plan are reviewed.

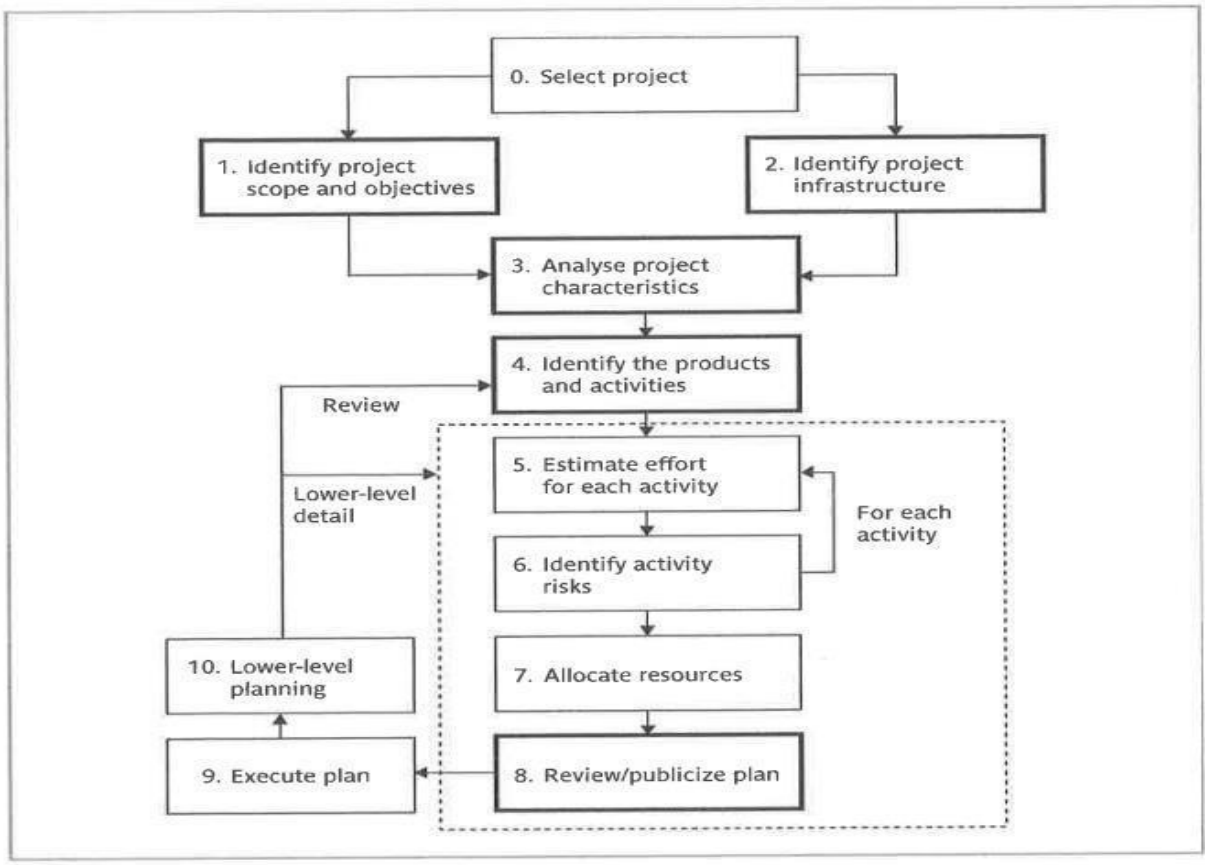


FIGURE 13.1 The place of software quality in Step Wise

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:

Increasingly 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.

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.

DEFINING SOFTWARE QUALITY

Quality is a rather vague term and we need to define carefully what we mean by it. For any software system there should be three specifications.

- **A functional specification** describing what the system is to do
- A **quality specification** concerned with how well the function are to operate
- A **resource specification** concerned with how much is to be spent on the system.

Attempt to identify specific product qualities that are appropriate to software, for instance, grouped software qualities into three sets. Product operation qualities, Product revision qualities and product transition qualities.

Product operation qualities:

Correctness: The extent to which a program satisfies its specification and fulfil user objective.

Reliability: the extent to which a program can be expected to perform its intended function with required precision.

Efficiency: The amounts of computer resource required by software.

Integrity: The extent to which access to software or data by unauthorized persons can be controlled.

Usability: The effort required to learn, operate, prepare input and interprets output.

Product Revision Qualities:

Maintainability: The effort required to locate and fix an error in an operational program

Testability: The effort required to test a program to ensure it performs its intended function.

Flexibility: The effort required to modify an operational program.

Product Transition Qualities:

Portability: The efforts require to transfer a program from one hardware configuration and or software system environment to another.

Reusability: The extent to which a program can be used in other applications.

Interoperability: The efforts required to couple one system to another.

When there is concerned about the need for a specific quality characteristic in a software product then a quality specification with the following minimum details should be drafted.

1. **Definition/Description**
   Definition: Clear definition of the quality characteristic.
   Description: Detailed description of what the quality characteristic entails.

2. **Scale o Unit of Measurement:**
   The unit used to measure the quality characteristic (e.g., faults per thousand lines of code).

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

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

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

6. **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.

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.

1. McCall's model.
2. Dromey's Model.
3. Boehm's Model.
4. ISO 9126 Model.

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

- Performance: How well if 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.

1) **McCall' Model:** 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 eleven attributes of the software:

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.
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.
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

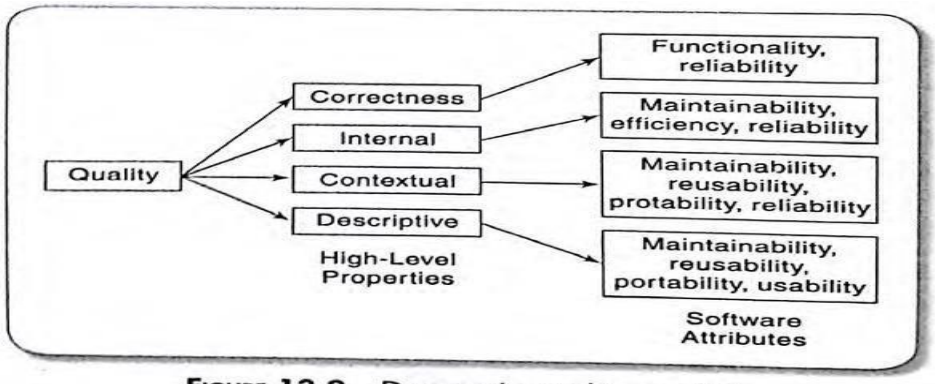


FIGURE 13.2 Dromey's quality model

3) **Boehm's Model:** Boehm's suggested that the quality of a software can be defined based on these high-level characteristics that are important for the users of the software. 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.

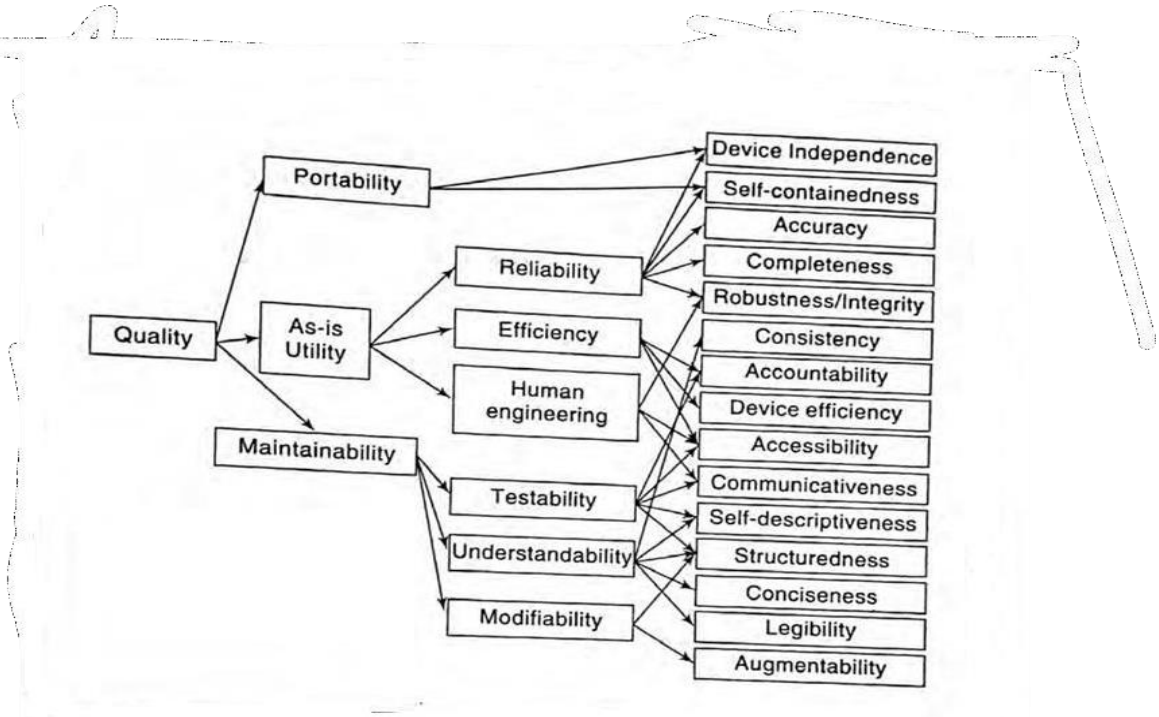


FIGURE 13.3 Boehm's quality model

ISO 9126

- ISO 9126 standards was first introduced in 1991 to tackle the question of the definition of software quality. The original 13-page document was designed as a foundation upon which further more detailed standard could be built. ISO9126 documents are now very lengthy.

Motivation might be-

- **Acquires** who are obtaining software from external suppliers
- **Developers** who are building a software product
- **Independent evaluators** who are accessing the quality of a software product, not for themselves but for a community of user.

- ISO 9126 also introduces another type of elements – quality in use- for which following element has been identified
  - **Effectiveness** : the ability to achieve user goals with accuracy and completeness;
  - **Productivity** : avoiding the excessive use of resources, such as staff effort, in achieving user goals;
  - **Safety**: within reasonable levels of risk of harm to people and other entities such as business, software, property and the environment
  - **Satisfaction**: smiling users

ISO 9126 is a significant standard in defining software quality attributes and providing a framework for assessing them. Here are the key aspects and characteristics defined by ISO 9126:

ISO 9126 identifies six major external software quality **characteristics**:

1. **Functionality:**

   Definition: The functions that a software product provides to satisfy user needs.

   Sub-characteristics: Suitability, accuracy, interoperability, security, compliance.

Characteristic	Sub-characteristics
Functionality	Suitability
	Accuracy
	Interoperability
	Functionality compliance
	Security

'**Functionality Compliance'** refers to the degree to which the software adheres to application-related standard or legal requirements. Typically, these could be auditing requirement. '**Interoperability**' refers to the ability of software to interact with others.

2. Reliability:

Definition: The capability of the software to maintain its level of performance under stated conditions**.**

Sub-characteristics: Maturity, fault tolerance, recoverability**.**

Characteristic	Sub-characteristics
Reliability	Maturity
	Fault tolerance
	Recoverability
	Reliability compliance

Maturity refers to frequency of failures due to fault in software more identification of fault more changes to remove them. **Recoverability** describes the control of access to a system.

3. Usability:

Definition: The effort needed to use the software**.**

Sub-characteristics: Understandability, learnability, operability, attractiveness.

Characteristic	Sub-characteristics
Usability	Understandability
	Learnability
	Operability
	Attractiveness
	Usability compliance

Understand ability is a clear quality to grasp. Although the definition attributes that bear on the user efforts for recognizing the logical concept and its applicability in our actually makes it less clear. **Learnability** has been distinguished from **operability**. A software tool might be easy to learn but time consuming to use say it uses a large number of nested menus. This is for a package that is used only intermittently but not where the system is used or several hours each day by the end user. In this case learnability has been incorporated at the expense of operability

4. **Efficiency**:

   Definition: The ability to use resources in relation to the amount of work done**.**

   Sub-characteristics: Time behavior, resource utilization.

5. **Maintainability:**

   Definition: The effort needed to make changes to the software.

   Sub-characteristics: Analyzability, modifiability, testability.

Characteristic	Sub-characteristics
Efficiency	Time behaviour
	Resource utilization
	Efficiency compliance
Maintainability	Analysability
	Changeability
	Stability
	Testability
	Maintainability compliance

Analyzability is the quality that McCall called diagnose ability, the ease with which the cause of failure can be determined. **Changeability** is the quality that others call flexibility: the latter name implies suppliers of the software are always changing it. **Stability** means that there is a low risk of a modification to software having unexpected effects.

6. **Portability:**

   Definition: The ability of the software to be transferred from one environment to another**.** \

   Sub-characteristics: Adaptability, install ability, co-existence.

Characteristic	Sub-characteristics
Portability	Adaptability
	Installability
	Coexistence
	Replaceability
	Portability compliance

Portability compliance relates to those standards that have a bearing on portability. **Replaceability** refers to the factors that give upward compatibility between old software components and the new ones. **'Coexistence'** refers to the ability of the software to share resources with other software components; unlike' interoperability', no direct data passing is necessarily involved

ISO 9126 provides guidelines for the use of the quality characteristics.

ISO 9126 provides structured guidelines for assessing and managing software quality characteristics based on the specific needs and requirements of the software product. It emphasizes the variation in importance of these characteristics depending on the type and context of the software product being developed.

Steps Suggested After Establishing Requirements

Once the software product requirements are established, ISO 9126 suggests the following steps:

1. **Specify Quality Characteristics**: Define and prioritize the relevant quality characteristics based on the software's intended use and stakeholder requirements.
2. **Define Metrics and Measurements**: Establish measurable criteria and metrics for evaluating each quality characteristic, ensuring they align with the defined objectives and user expectations.
3. **Plan Quality Assurance Activities**: Develop a comprehensive plan for quality assurance activities, including testing, verification, and validation processes to ensure adherence to quality standards.
4. **Monitor and Improve Quality:** Continuously monitor software quality throughout the development lifecycle, identifying areas for improvement and taking corrective actions as necessary.
5. **Document and Report:** Document all quality-related activities, findings, and improvements, and provide clear and transparent reports to stakeholders on software quality status and compliance.

1. **Judge the importance of each quality characteristic for the application.**
   - Identify key quality characteristics (e.g., Usability, Efficiency, and Maintainability).
   - Assign importance ratings to these characteristics based on their relevance to the application.

Importance of Quality Characteristics:

Reliability: Critical for safety-critical systems where failure can have severe consequences. Measures like mean time between failures (MTBF) are essential.

Efficiency: Important for real-time systems where timely responses are crucial. Measures such as response time are key indicators.

2. **Select the external quality measurements within the ISO 9126 framework relevant to the qualities prioritized above.**

Determine appropriate **external quality measurements** that correspond to each quality characteristic.

- **Reliability**: Measure with MTBF or similar metrics.
- **Efficiency**: Measure with response time or time behavior metrics.

3. **Map measurements onto ratings that reflect user satisfaction. For example, the mappings mightbe as in Table 13.1.**

For response time, user satisfaction could be mapped as follows (hypothetical example):
Excellent: Response time < 1 second
Good: Response time 1-3 seconds
Acceptable: Response time 3-5 seconds
Poor: Response time > 5 seconds

Response time (seconds)	Rating
<2	Exceeds expectation
2–5	Within the target range
6–10	Minimally acceptable
>10	Unacceptable

TABLE 13.1 Mapping measurements to user satisfaction

4. **Identify the relevant internal measurements and the intermediate products in which they appear.**
   - Identify and track internal measurements such as cyclomatic complexity, code coverage, defect density, etc.
   - Relate these measurements to intermediate products like source code, test cases, and documentation.

5. **Overall assessment of product quality: To what extent is it possible to combine ratings for different quality characteristics into a single overall rating for the software?**
   - Use weighted quality scores to assess overall product quality.
   - Focus on key quality requirements and address potential weaknesses early to avoid the need for an overall quality rating later.

Based on the ISO 9126 framework and your points:

1. Measurement Indicators Across Development Stages:

- **Early Stages**: Qualitative indicators like checklists and expert judgments are used to assess compliance with predefined criteria. These are subjective and based on qualitative assessments.

- **Later Stages**: Objective and quantitative measurements become more prevalent as the software product nears completion. These measurements provide concrete data about software performance and quality attributes.

2. Overall Assessment of Product Quality:

- Combining ratings for different quality characteristics into a single overall rating for software is challenging due to:
  - Different measurement scales and methodologies for each quality characteristic.
  - Sometimes, enhancing one quality characteristic (e.g., efficiency) may compromise another (e.g., portability).
- Balancing these trade-offs can be complex and context-dependent.

3. Purpose of Quality Assessment:

Software Development: Assessment focuses on identifying weaknesses early, guiding developers to meet quality requirements, and ensuring continuous improvement throughout the development lifecycle.

Software Acquisition: Helps in evaluating software products from external suppliers based on predefined quality criteria.

Independent Assessment: Aims to provide an unbiased evaluation of software quality for stakeholders like regulators or consumers.

It seems like you're describing a method for evaluating and comparing software products based on their quality characteristics. Here's a summary and interpretation of your approach:

Table 13.2 Mapping response times onto user satisfaction

Response time (seconds)	Quality score
<2	5
2–3	4
4–5	3
6–7	2
8–9	1
>9	0

The table 13.2 shows how different response times are mapped to quality scores on a scale of 0-5, with shorter response times receiving higher scores. A rating scale (e.g., 1-5) is used to reflect the importance of various quality characteristics.

1. **Rating for User Satisfaction:**
   - Products are evaluated based on mandatory quality levels that must be met. Beyond these mandatory levels, user satisfaction ratings in the range of 0 to 5 are assigned for other desirable characteristics.
   - Objective measurements of functions are used to determine different levels of user satisfaction, which are then mapped to numerical ratings (see Table 13.2 for an example).
2. **Importance Weighting**:
   - Each quality characteristic (e.g., usability, efficiency, maintainability) is assigned an importance rating on a scale of 1 to 5.
   - These importance ratings reflect how critical each quality characteristic is to the overall evaluation of the software product.
3. **Calculation of Overall Score:**
   - Weighted scores are calculated for each quality characteristic by multiplying the quality score by its importance weight.
   - The weighted scores for all characteristics are summed to obtain an overall score for each software product.
4. **Comparison and Preference Order:**
   - Products are then ranked in order of preference based on their overall scores. Higher scores indicate products that are more likely to satisfy user requirements and preferences across the evaluated quality characteristics.

This method provides a structured approach to evaluating software products based on user satisfaction ratings and importance weights for quality characteristics. It allows stakeholders to compare andprioritize products effectively based on their specific needs and preferences.

TABLE 13.3 Weighted quality scores

Product quality	Importance rating (a)	Product A		Product B	
		Quality score (b)	Weighted score (a × b)	Quality score (c)	Weighted score (a × c)
Usability	3	1	3	3	9
Efficiency	4	2	8	2	8
Maintainability	2	3	6	1	2
Overall			17		19

- This table 13.3 provides a comparison of two products (A and B) based on weighted quality scores.
- Each product quality (Usability, Efficiency, and Maintainability) is given an importance rating.
- Product A and B are scored for each quality, and these scores are multiplied by the importance rating to obtain weighted scores.

- The total weighted scores are summed for each product to determine their overall ranking.

It involves assessing software products by assigning quality scores to various characteristics, weighting these scores by their importance, and summing them to get an overall score. This approach helps to objectively compare products based on user satisfaction and key quality metrics.

By assigning scores to various qualities, weighting them by their importance, and summing these to get an overall score, it provides a comprehensive way to compare and rank software products. This ensures that both essential and desirable characteristics are considered in the assessment, leading to a more balanced and objective evaluation.

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 (calledprocess metrics).

1. **Product Metrics:**

   Purpose: Measure the characteristics of the software product being developed.

   Examples:

   Size Metrics: Such as Lines of Code (LOC) and Function Points, which quantify the size or complexity of the software.

   Effort Metrics: Like Person-Months (PM), which measure the effort required to develop the software.

   Time Metrics: Such as the duration in months or other time units needed to complete the development.

2. **Process Metrics:**

   Purpose: Measure the effectiveness and efficiency of the development process itself**.**

   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.

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:

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

2. Software product, such as size (LOC, function points), reliability (defects found per LOC), performance(response time), and usability (user satisfaction ratings).

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

4. 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.

5. 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 developmentlifecycle, from initial planning through to deployment and maintenance.

3. Benefits:

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

4. Challenges:

- Requires consistent monitoring and analysis of metrics throughout the development lifecycle.
- Effectiveness of process improvements may not always translate directly into improved productquality 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 morereliable, efficient, and user-friendly software products.

26.1 Observations on Estimation

Planning requires you to make an initial commitment, even though it's likely that this "commitment" will be proven wrong. Whenever estimates are made, you look into the future and accept some degree of uncertainty as a matter of course. To quote Frederick Brooks [Bro95]:

> . . . our techniques of estimating are poorly developed. More seriously, they reflect an unvoiced assumption that is quite untrue, i.e., that all will go well. because we are uncertain of our estimates, software managers often lack the courteous stubbornness to make people wait for a good product.

Although estimating is as much art as it is science, this important action need not be conducted in a haphazard manner. Useful techniques for time and effort estimation do exist. Process and project metrics can provide historical perspective and powerful input for the generation of quantitative estimates. Past experience (of all people involved) can aid immeasurably as estimates are developed and reviewed. Because estimation lays a foundation for all other project planning actions, and project planning provides the road map for successful software engineering, we would be ill advised to embark without it.

Estimation of resources, cost, and schedule for a software engineering effort requires experience, access to good historical information (metrics), and the courage to commit to quantitative predictions when qualitative information is all that exists. Estimation carries inherent risk,[1] and this risk leads to uncertainty.

Project complexity has a strong effect on the uncertainty inherent in planning. Complexity, however, is a relative measure that is affected by familiarity with past effort. The first-time developer of a sophisticated e-commerce application might consider it to be exceedingly complex. However, a Web engineering team developing its tenth e-commerce WebApp would consider such work run-of-the-mill. A number of quantitative software complexity measures have been proposed [Zus97]. Such measures are applied at the design or code level and are therefore difficult to use during

software planning (before a design and code exist). However, other, more subjective assessments of complexity (e.g., function point complexity adjustment factors described in Chapter 23) can be established early in the planning process.

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.[2] Problem decomposition, an important approach to estimating, becomes more difficult because the refinement of problem elements may still be formidable. To paraphrase Murphy's law: "What can go wrong will go wrong"—and if there are more things that can fail, more things will fail.

The *degree of structural uncertainty* also has an effect on estimation risk. In this context, structure refers to the degree to which requirements have been solidified, the ease with which functions can be compartmentalized, and the hierarchical nature of the information that must be processed.

The availability of historical information has a strong influence on estimation risk. By looking back, you can emulate things that worked and improve areas where problems arose. When comprehensive software metrics (Chapter 25) are available for past projects, estimates can be made with greater assurance, schedules can be established to avoid past difficulties, and overall risk is reduced.

Estimation risk is measured by the degree of uncertainty in the quantitative estimates established for resources, cost, and schedule. If project scope is poorly understood or project requirements are subject to change, uncertainty and estimation risk become dangerously high. As a planner, you and the customer should recognize that variability in software requirements means instability in cost and schedule.

However, you should not become obsessive about estimation. Modern software engineering approaches (e.g., evolutionary process models) take an iterative view of development. In such approaches, it is possible—although not always politically acceptable—to revisit the estimate (as more information is known) and revise it when the customer makes changes to requirements.

26.2 The Project Planning Process

The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule. In addition, estimates should attempt to define best-case and worst-case scenarios so that project outcomes can be bounded. Although there is an inherent degree of uncertainty, the software team embarks on a plan that has been established as a consequence of these tasks. Therefore, the plan must be adapted and updated as the project proceeds. In the following sections, each of the actions associated with software project planning is discussed.

TASK SET

Task Set for Project Planning

1. Establish project scope.
2. Determine feasibility.
3. Analyze risks (Chapter 28).
4. Define required resources.
   a. Determine required human resources.
   b. Define reusable software resources.
   c. Identify environmental resources.
5. Estimate cost and effort.
   a. Decompose the problem.
   b. Develop two or more estimates using size, function points, process tasks, or use cases.
   c. Reconcile the estimates.
6. Develop a project schedule (Chapter 27).
   a. Establish a meaningful task set.
   b. Define a task network.
   c. Use scheduling tools to develop a time-line chart.
   d. Define schedule tracking mechanisms.

26.3 SOFTWARE SCOPE AND FEASIBILITY

Software scope describes the functions and features that are to be delivered to end users; the data that are input and output; the "content" that is presented to users as a consequence of using the software; and the performance, constraints, interfaces, and reliability that *bound* the system. Scope is defined using one of two techniques:

1. A narrative description of software scope is developed after communication with all stakeholders.
2. A set of use cases[3] is developed by end users.

Functions described in the statement of scope (or within the use cases) are evaluated and in some cases refined to provide more detail prior to the beginning of estimation. Because both cost and schedule estimates are functionally oriented, some degree of decomposition is often useful. Performance considerations encompass processing and response time requirements. Constraints identify limits placed on the software by external hardware, available memory, or other existing systems.

Once scope has been identified (with the concurrence of the customer), it is reasonable to ask: "Can we build software to meet this scope? Is the project feasible?" All too often, software engineers rush past these questions (or are pushed past them by impatient managers or other stakeholders), only to become mired in a project that is doomed from the onset. Putnam and Myers [Put97a] address this issue when they write:

> [N]ot everything imaginable is feasible, not even in software, evanescent as it may appear to outsiders. On the contrary, software feasibility has four solid dimensions: *Technology*—Is a project technically feasible? Is it within the state of the art? Can defects be reduced to a level matching the application's needs? *Finance*—Is it financially feasible? Can development be completed at a cost the software organization, its client, or the market can

afford? *Time*—Will the project's time-to-market beat the competition? *Resources*—Does the organization have the resources needed to succeed?

Putnam and Myers correctly suggest that scoping is not enough. Once scope is understood, you must work to determine if it can be done within the dimensions just noted. This is a crucial, although often overlooked, part of the estimation process.

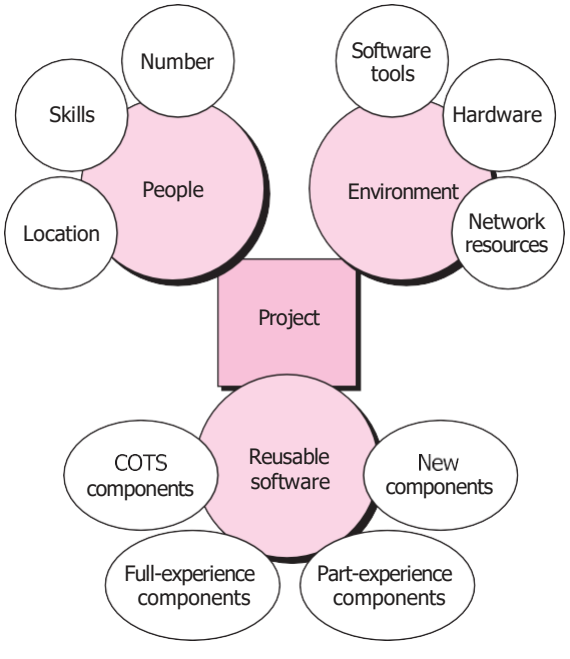
26.4 Resources

The second planning task is estimation of the resources required to accomplish the software development effort. Figure 26.1 depicts the three major categories of software engineering resources—people, reusable software components, and the development environment (hardware and software tools). Each resource is specified with four characteristics: description of the resource, a statement of availability, time when the resource will be required, and duration of time that the resource will be applied. The last two characteristics can be viewed as a *time window*. Availability of the resource for a specified window must be established at the earliest practical time.

26.4.1 Human Resources

The planner begins by evaluating software scope and selecting the skills required to complete development. Both organizational position (e.g., manager, senior software engineer) and specialty (e.g., telecommunications, database, client-server) are

Project resources



specified. For relatively small projects (a few person-months), a single individual may perform all software engineering tasks, consulting with specialists as required. For larger projects, the software team may be geographically dispersed across a number of different locations. Hence, the location of each human resource is specified.

The number of people required for a software project can be determined only after an estimate of development effort (e.g., person-months) is made. Techniques for estimating effort are discussed later in this chapter.

26.4.2 Reusable Software Resources

Component-based software engineering (CBSE)[4] emphasizes reusability—that is, the creation and reuse of software building blocks. Such building blocks, often called *components,* must be cataloged for easy reference, standardized for easy application, and validated for easy integration. Bennatan [Ben00] suggests four software resource categories that should be considered as planning proceeds:

Off-the-shelf components. Existing software that can be acquired from a third party or from a past project. COTS (commercial off-the-shelf) components are purchased from a third party, are ready for use on the current project, and have been fully validated.

Full-experience components. Existing specifications, designs, code, or test data developed for past projects that are similar to the software to be built for the current project. Members of the current software team have had full experience in the application area represented by these components. Therefore, modifications required for full-experience components will be relatively low risk.

Partial-experience components. Existing specifications, designs, code, or test data developed for past projects that are related to the software to be built for the current project but will require substantial modification. Members of the current software team have only limited experience in the application area represented by these components. Therefore, modifications required for partial-experience components have a fair degree of risk.

New components. Software components must be built by the software team specifically for the needs of the current project.

Ironically, reusable software components are often neglected during planning, only to become a paramount concern later in the software process. It is better to specify software resource requirements early. In this way technical evaluation of the alternatives can be conducted and timely acquisition can occur.

26.4.3 Environmental Resources

The environment that supports a software project, often called the *software engineering environment* (SEE), incorporates hardware and software. Hardware provides

a platform that supports the tools (software) required to produce the work products that are an outcome of good software engineering practice.[5] Because most software organizations have multiple constituencies that require access to the SEE, you must prescribe the time window required for hardware and software and verify that these resources will be available.

When a computer-based system (incorporating specialized hardware and software) is to be engineered, the software team may require access to hardware elements being developed by other engineering teams. For example, software for a robotic device used within a manufacturing cell may require a specific robot (e.g., a robotic welder) as part of the validation test step; a software project for advanced page layout may need a high-speed digital printing system at some point during development. Each hardware element must be specified as part of planning.

26.5 Software Project Estimation

Software cost and effort estimation will never be an exact science. Too many variables—human, technical, environmental, political—can 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, a number of options arise:

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

Unfortunately, the first option, however attractive, is not practical. Cost estimates must be provided up-front. However, you should recognize that the longer you wait, the more you know, and the more you know, the less likely you are to make serious errors in your estimates.

The second option can work reasonably well, if the current project is quite similar to past efforts and other project influences (e.g., the customer, business conditions, the software engineering environment, deadlines) are roughly equivalent. Unfortunately, past experience has not always been a good indicator of future results.

The remaining options are viable approaches to software project estimation. Ideally, the techniques noted for each option should be applied in tandem; each used as a cross-check for the other. Decomposition techniques take a divide-and-conquer

approach to software project estimation. By decomposing a project into major functions and related software engineering activities, cost and effort estimation can be performed in a stepwise fashion. Empirical estimation models can be used to complement decomposition techniques and offer a potentially valuable estimation approach in their own right. 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).

Automated estimation tools implement one or more decomposition techniques or empirical models and provide an attractive option for estimating. In such systems, the characteristics of the development organization (e.g., experience, environment) and the software to be developed are described. Cost and effort estimates are derived from these data.

Each of the viable software cost estimation options is only as good as the historical data used to seed the estimate. If no historical data exist, costing rests on a very shaky foundation. In Chapter 25, we examined the characteristics of some of the software metrics that provide the basis for historical estimation data.

26.6 Decomposition Techniques

Software project estimation is a form of problem solving, and in most cases, the problem 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.

In Chapter 24, the decomposition approach was discussed from two different points of view: decomposition of the problem and decomposition of the process. Estimation uses one or both forms of partitioning. But before an estimate can be made, you must understand the scope of the software to be built and generate an estimate of its "size."

26.6.1 Software Sizing

The accuracy of a software project estimate is predicated on a number of things: (1) the degree to which you have properly estimated the size of the product to be built; (2) the ability to translate the size estimate into human effort, calendar time, and dollars (a function of the availability of reliable software metrics from past projects); (3) the degree to which the project plan reflects the abilities of the software team; and (4) the stability of product requirements and the environment that supports the software engineering effort.

In this section, I consider the *software sizing* problem. Because a project estimate is only as good as the estimate of the size of the work to be accomplished, sizing

represents your first major challenge as a planner. 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 [Put92] suggest four different approaches to the sizing problem:

- *"Fuzzy logic" sizing.* This approach uses the approximate reasoning techniques that are the cornerstone of fuzzy logic. 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.* The planner develops estimates of the information domain characteristics discussed in Chapter 23.
- *Standard component sizing.* Software is composed of a number of different "standard components" that are generic to a particular application area. For example, the standard components for an information system are subsystems, modules, screens, reports, interactive programs, batch programs, files, LOC, and object-level instructions. The project planner estimates the number of occurrences of each standard component and then uses historical project data to estimate the delivered size per standard component.
- *Change sizing.* This approach is used when a project encompasses the use of existing software that must be modified in some way as part of a project. The planner estimates the number and type (e.g., reuse, adding code, changing code, deleting code) of modifications that must be accomplished.

Putnam and Myers suggest that the results of each of these sizing approaches be combined statistically to create a *three-point* or *expected-value* estimate. This is accomplished by developing optimistic (low), most likely, and pessimistic (high) values for size and combining them using Equation (26.1), described in Section 26.6.2.

26.6.2 Problem-Based Estimation

In Chapter 25, lines of code and function points were described as measures from which productivity metrics can be computed. LOC and FP data are used in two ways during software project estimation: (1) as estimation variables to "size" each element of the software and (2) as baseline metrics collected from past projects and used in conjunction with estimation variables to develop cost and effort projections.

LOC and FP estimation are distinct estimation techniques. Yet both have a number of characteristics in common. You begin with a bounded statement of software scope and from this statement attempt to decompose the statement of scope into problem functions that can each be estimated individually. LOC or FP (the estimation variable) is then estimated for each function. Alternatively, you may choose another component for sizing, such as classes or objects, changes, or business processes affected.

Baseline productivity metrics (e.g., LOC/pm or FP/pm[6]) are then applied to the appropriate estimation variable, and cost or effort for the function is derived. Func- tion estimates are combined to produce an overall estimate for the entire project.

It is important to note, however, that there is often substantial scatter in productivity metrics for an organization, making the use of a single-baseline productivity metric suspect. In general, LOC/pm or FP/pm averages should be computed by project domain. That is, projects should be grouped by team size, application area, complexity, and other relevant parameters. Local domain averages should then be computed. When a new project is estimated, it should first be allocated to a domain, and then the appropriate domain average for past productivity should be used in generating the estimate.

The LOC and FP estimation techniques differ in the level of detail required for decomposition and the target of the partitioning. When LOC is used as the estimation variable, decomposition is absolutely essential and is often taken to considerable levels of detail. The greater the degree of partitioning, the more likely reasonably accurate estimates of LOC can be developed.

For FP estimates, decomposition works differently. Rather than focusing on function, each of the information domain characteristics—inputs, outputs, data files, inquiries, and external interfaces—as well as the 14 complexity adjustment values discussed in Chapter 23 are estimated. The resultant estimates can then be used to derive an FP value that can be tied to past data and used to generate an estimate.

Regardless of the estimation variable that is used, you should begin by estimating a range of values for each function or information domain value. Using historical data or (when all else fails) intuition, estimate an optimistic, most likely, and pessimistic size value for each function or count for each information domain value. An implicit indication of the degree of uncertainty is provided when a range of values is specified.

A three-point or expected value can then be computed. The *expected value* for the estimation variable (size) S can be computed as a weighted average of the optimistic (s_{opt}), most likely (s_m), and pessimistic (s_{pess}) estimates. For example,

$$S = \frac{s_{\text{opt}} + 4s_m + s_{\text{pess}}}{6} \tag{26.1}$$

gives heaviest credence to the "most likely" estimate and follows a beta probability distribution. We assume that there is a very small probability the actual size result will fall outside the optimistic or pessimistic values.

Once the expected value for the estimation variable has been determined, historical LOC or FP productivity data are applied. Are the estimates correct? The only reasonable answer to this question is: "You can't be sure." Any estimation technique, no matter how sophisticated, must be cross-checked with another approach. Even then, common sense and experience must prevail.

26.6.3 An Example of LOC-Based Estimation

As an example of LOC and FP problem-based estimation techniques, I consider a software package to be developed for a computer-aided design application for mechanical components. The software is to execute on an engineering workstation and must interface with various computer graphics peripherals including a mouse, digitizer, high-resolution color display, and laser printer. A preliminary statement of software scope can be developed:

> The mechanical CAD software will accept two- and three-dimensional geometric data from an engineer. The engineer will interact and control the CAD system through a user interface that will exhibit characteristics of good human/machine interface design. All geometric data and other supporting information will be maintained in a CAD database. Design analysis modules will be developed to produce the required output, which will be displayed on a variety of graphics devices. The software will be designed to control and interact with peripheral devices that include a mouse, digitizer, laser printer, and plotter.

This statement of scope is preliminary—it is not bounded. Every sentence would have to be expanded to provide concrete detail and quantitative bounding. For example, before estimation can begin, the planner must determine what "characteristics of good human/machine interface design" means or what the size and sophistication of the "CAD database" are to be.

For our purposes, assume that further refinement has occurred and that the major software functions listed in Figure 26.2 are identified. Following the decomposition technique for LOC, an estimation table (Figure 26.2) is developed. A range of LOC estimates is developed for each function. For example, the range of LOC estimates for the 3D geometric analysis function is optimistic, 4600 LOC; most likely, 6900 LOC; and pessimistic, 8600 LOC. Applying Equation 26.1, the expected value for the 3D geometric analysis function is 6800 LOC. Other estimates are derived in a similar

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
Estimated lines of code	*33,200*

fashion. By summing vertically in the estimated LOC column, an estimate of 33,200 lines of code is established for the CAD system.

Do not succumb to the temptation to use this result as your project estimate. You should derive another result using a different approach.

A review of historical data indicates that the organizational average productivity for systems of this type is 620 LOC/pm. Based on a burdened labor rate of $8000 per month, the cost per line of code is approximately $13. Based on the LOC estimate and the historical productivity data, the total estimated project cost is $431,000 and the estimated effort is 54 person-months.[7]

SAFEHOME

Estimating

The scene: Doug Miller's office as project planning begins.

The players: Doug Miller (manager of the *SafeHome* software engineering team) and Vinod Raman, Jamie Lazar, and other members of the product software engineering team.

The conversation:

Doug: We need to develop an effort estimate for the project and then we've got to define a micro schedule for the first increment and a macro schedule for the remaining increments.

Vinod (nodding): Okay, but we haven't defined any increments yet.

Doug: True, but that's why we need to estimate.

Jamie (frowning): You want to know how long it's going to take us?

Doug: Here's what I need. First, we need to functionally decompose the *SafeHome* software … at a high level … then we've got to estimate the number of lines of code that each function will take . . . then . . .

Jamie: Whoa! How are we supposed to do that?

Vinod: I've done it on past projects. You begin with use cases, determine the functionality required to implement each, guesstimate the LOC count for each piece of the function. The best approach is to have everyone do it independently and then compare results.

Doug: Or you can do a functional decomposition for the entire project.

Jamie: But that'll take forever and we've got to get started.

Vinod: No . . . it can be done in a few hours . . . this morning, in fact.

Doug: I agree . . . we can't expect exactitude, just a ballpark idea of what the size of *SafeHome* will be.

Jamie: I think we should just estimate effort . . . that's all.

Doug: We'll do that too. Then use both estimates as a cross-check.

Vinod: Let's go do it . . .

26.6.4 An Example of FP-Based Estimation

Decomposition for FP-based estimation focuses on information domain values rather than software functions. Referring to the table presented in Figure 26.3, you would estimate inputs, outputs, inquiries, files, and external interfaces for the CAD software. An FP value is computed using the technique discussed in Chapter 23. For the purposes of this estimate, the complexity weighting factor is assumed to be average. Figure 26.3 presents the results of this estimate.

7 Estimates are rounded to the nearest $1000 and person-month. Further precision is unnecessary and unrealistic, given the limitations of estimation accuracy.

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*

Each of the complexity weighting factors is estimated, and the value adjustment factor is computed as described in Chapter 23:

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**

Finally, the estimated number of FP is derived:

$$FP_{estimated} = \text{count total} \times [0.65 + 0.01 \times \text{o}(F_i)] = 375$$

The organizational average productivity for systems of this type is 6.5 FP/pm. Based on a burdened labor rate of \$8000 per month, the cost per FP is approximately \$1230. Based on the FP estimate and the historical productivity data, the total estimated project cost is \$461,000 and the estimated effort is 58 person-months.

26.6.5 Process-Based Estimation

The most common technique for estimating a project is to base the estimate on the process that will be used. That is, the process is decomposed into a relatively small set of tasks and the effort required to accomplish each task is estimated.

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

Like the problem-based techniques, process-based estimation begins with a delineation of software functions obtained from the project scope. A series of framework activities must be performed for each function. Functions and related framework activities[8] may be represented as part of a table similar to the one presented in Figure 26.4.

Once problem functions and process activities are melded, you estimate the effort (e.g., person-months) that will be required to accomplish each software process activity for each software function. These data constitute the central matrix of the table in Figure 26.4. Average labor rates (i.e., cost/unit effort) are then applied to the effort estimated for each process activity. It is very likely the labor rate will vary for each task. Senior staff are heavily involved in early framework activities and are generally more expensive than junior staff involved in construction and release.

Costs and effort for each function and framework activity are computed as the last step. If process-based estimation is performed independently of LOC or FP estimation, we now have two or three estimates for cost and effort that may be compared and reconciled. If both sets of estimates show reasonable agreement, there is good reason to believe that the estimates are reliable. If, on the other hand, the results of these decomposition techniques show little agreement, further investigation and analysis must be conducted.

26.6.6 An Example of Process-Based Estimation

To illustrate the use of process-based estimation, consider the CAD software introduced in Section 26.6.3. The system configuration and all software functions remain unchanged and are indicated by project scope.

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 (abbreviated for brevity). The engineering and construction release activities are subdivided into the major software engineering tasks shown. Gross estimates of effort are provided for customer communication, planning, and risk analysis. These are noted in the total row at the bottom of the table. Horizontal and vertical totals provide an indication of estimated effort required for analysis, design, code, and test. It should be noted that 53 percent of all effort is expended on front-end engineering tasks (requirements analysis and design), indicating the relative importance of this work.

Based on an average burdened labor rate of $8000 per month, the total estimated project cost is $368,000 and the estimated effort is 46 person-months. If desired, labor rates could be associated with each framework activity or software engineering task and computed separately.

26.6.7 Estimation with Use Cases

As I have noted throughout Part 2 of this book, use cases provide a software team with insight into software scope and requirements. However, developing an estimation approach with use cases is problematic for the following reasons [Smi99]:

- 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.

Unlike an LOC or a function point, one person's "use case" may require months of effort while another person's use case may be implemented in a day or two.

Although a number of investigators have considered use cases as an estimation input, no proven estimation method has emerged to date.[9] Smith [Smi99] suggests that use cases can be used for estimation, but only if they are considered within the context of the "structural hierarchy" that they are used to describe.

Smith argues that any level of this structural hierarchy can be described by no more than 10 use cases. Each of these use cases would encompass no more than 30 distinct scenarios. Obviously, use cases that describe a large system are written at a much higher level of abstraction (and represent considerably more development effort) than use cases that are written to describe a single subsystem. Therefore,

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

To illustrate how this computation might be made, consider the following relationship:[10]

$$\text{LOC estimate} = N \times \text{LOC}_{\text{avg}} + [(S_a/S_h - 1) + (P_a/P_h - 1)] \times \text{LOC}_{\text{adjust}} \tag{26.2}$$

where

N	= actual number of use cases
LOC_{avg}	= historical average LOC per use case for this type of subsystem
$\text{LOC}_{\text{adjust}}$	= represents an adjustment based on n percent of LOC_{avg} where n is defined locally and represents the difference between this project and "average" projects
S_a	= actual scenarios per use case
S_h	= average scenarios per use case for this type of subsystem
P_a	= actual pages per use case
P_h	= average pages per use case for this type of subsystem

Expression (26.2) could be used to develop a rough estimate of the number of LOC based on the actual number of use cases adjusted by the number of scenarios and the page length of the use cases. The adjustment represents up to n percent of the historical average LOC per use case.

26.6.8 An Example of Use-Case–Based Estimation

The CAD software introduced in Section 26.6.3 is composed of three subsystem groups: user interface subsystem (includes UICF), engineering subsystem group (includes the 2DGA, 3DGA, and DAM subsystems), and infrastructure subsystem group (includes CGDF and PCF subsystems). Six use cases describe the user interface subsystem. Each use case is described by no more than 10 scenarios and has an average length of six pages. The engineering subsystem group is described by 10 use cases (these are considered to be at a higher level of the structural hierarchy). Each of these use cases has no more than 20 scenarios associated with it and has an average length of eight pages. Finally, the infrastructure subsystem group is described by five use cases with an average of only six scenarios and an average length of five pages.

scenarios	pages	LOC
12	5	560
16	8	3100
10	6	1650

FIGURE 26.5

Use-case estimation

	u e cases	scenarios	pages	LOC estimate
User interface subsystem	6	10	6	3,366
Engineering subsystem group	10	20	8	31,233
Infrastructure subsystem group	5	6	5	7,970
Total LOC estimate				42,568

Using the relationship noted in Expression (26.2) with $n = 30$ percent, the table shown in Figure 26.5 is developed. Considering the first row of the table, historical data indicate that UI software requires an average of 800 LOC per use case when the use case has no more than 12 scenarios and is described in less than five pages. These data conform reasonably well for the CAD system. Hence the LOC estimate for the user interface subsystem is computed using expression (26.2). Using the same approach, estimates are made for both the engineering and infrastructure subsystem groups. Figure 26.5 summarizes the estimates and indicates that the overall size of the CAD is estimated at 42,500 LOC.

Using 620 LOC/pm as the average productivity for systems of this type and a burdened labor rate of $8000 per month, the cost per line of code is approximately $13. Based on the use-case estimate and the historical productivity data, the total estimated project cost is $552,000 and the estimated effort is 68 person-months.

26.6.9 Reconciling Estimates

The estimation techniques discussed in the preceding sections result in multiple estimates that must be reconciled to produce a single estimate of effort, project duration, or cost. To illustrate this reconciliation procedure, I again consider the CAD software introduced in Section 26.6.3.

The total estimated effort for the CAD software ranges from a low of 46 person-months (derived using a process-based estimation approach) to a high of 68 person-months (derived with use-case estimation). The average estimate (using all four approaches) is 56 person-months. The variation from the average estimate is approximately 18 percent on the low side and 21 percent on the high side.

What happens when agreement between estimates is poor? The answer to this question requires a reevaluation of information used to make the estimates. Widely divergent estimates can often be traced to one of two causes: (1) the scope of the project is not adequately understood or has been misinterpreted by the planner, or (2) productivity data used for problem-based estimation techniques is inappropriate for the application, obsolete (in that it no longer accurately reflects the software engineering organization), or has been misapplied. You should determine the cause of divergence and then reconcile the estimates.

INFO

Automated Estimation Techniques for Software Projects

Automated estimation tools allow the planner to estimate cost and effort and to perform what-if analyses for important project variables such as delivery date or staffing. Although many automated estimation tools exist (see sidebar later in this chapter), all exhibit the same general characteristics and all perform the following six generic functions [Jon96]:

1. *Sizing of project deliverables.* The "size" of one or more software work products is estimated. Work products include the external representation of software (e.g., screen, reports), the software itself (e.g., KLOC), functionality delivered (e.g., function points), and descriptive information (e.g., documents).
2. *Selecting project activities.* The appropriate process framework is selected, and the software engineering task set is specified.
3. *Predicting staffing levels.* The number of people who will be available to do the work is specified. Because the relationship between people available and work (predicted effort) is highly nonlinear, this is an important input.
4. *Predicting software effort.* Estimation tools use one or more models (Section 26.7) that relate the size of the project deliverables to the effort required to produce them.
5. *Predicting software cost.* Given the results of step 4, costs can be computed by allocating labor rates to the project activities noted in step 2.
6. *Predicting software schedules.* When effort, staffing level, and project activities are known, a draft schedule can be produced by allocating labor across software engineering activities based on recommended models for effort distribution discussed later in this chapter.

When different estimation tools are applied to the same project data, a relatively large variation in estimated results can be encountered. More important, predicted values sometimes are significantly different than actual values. This reinforces the notion that the output of estimation tools should be used as one "data point" from which estimates are derived—not as the only source for an estimate.

26.7 EMPIRICAL ESTIMATION MODELS

An estimation model for computer software uses empirically derived formulas to predict effort as a function of LOC or FP.[11] Values for LOC or FP are estimated using the approach described in Sections 26.6.3 and 26.6.4. But instead of using the tables described in those sections, the resultant values for LOC or FP are plugged into the estimation model.

The empirical data that support most estimation models are derived from a limited sample of projects. For this reason, no estimation model is appropriate for all classes of software and in all development environments. Therefore, you should use the results obtained from such models judiciously.

An estimation model should be calibrated to reflect local conditions. The model should be tested by applying data collected from completed projects, plugging the data into the model, and then comparing actual to predicted results. If agreement is poor, the model must be tuned and retested before it can be used.

26.7.1 The Structure of Estimation Models

A typical estimation model is derived using regression analysis on data collected from past software projects. The overall structure of such models takes the form [Mat94]

$$E = A + B \times (e_v)^C \tag{26.3}$$

where A, B, and C are empirically derived constants, E is effort in person-months, and e_v is the estimation variable (either LOC or FP). In addition to the relationship noted in Equation (26.3), 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 in the literature are

$E = 5.2 \times (\text{KLOC})^{0.91}$	Walston-Felix model
$E = 5.5 + 0.73 \times (\text{KLOC})^{1.16}$	Bailey-Basili model
$E = 3.2 \times (\text{KLOC})^{1.05}$	Boehm simple model
$E = 5.288 \times (\text{KLOC})^{1.047}$	Doty model for KLOC > 9

FP-oriented models have also been proposed. These include

$E = -91.4 + 0.355\ \text{FP}$	Albrecht and Gaffney model
$E = -37 + 0.96\ \text{FP}$	Kemerer model
$E = -12.88 + 0.405\ \text{FP}$	Small project regression model

A quick examination of these models indicates that each will yield a different result for the same values of LOC or FP. The implication is clear. Estimation models must be calibrated for local needs!

26.7.2 The COCOMO II Model

In his classic book on "software engineering economics," Barry Boehm [Boe81] introduced a hierarchy of software estimation models bearing the name COCOMO, for *COnstructive COst MOdel.* The original COCOMO model became one of the most widely used and discussed software cost estimation models in the industry. It has evolved into a more comprehensive estimation model, called COCOMO II [Boe00]. Like its predecessor, COCOMO II is actually a hierarchy of estimation models that address the following areas:

- *Application composition model.* Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.
- *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 estimation models for software, the COCOMO II models require sizing information. Three different sizing options are available as part of the model hierarchy: object points, function points, and lines of source code.

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

The COCOMO II application composition model uses object points and is illustrated in the following paragraphs. It should be noted that other, more sophisticated estimation models (using FP and KLOC) are also available as part of COCOMO II.

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) using criteria suggested by Boehm [Boe96]. In essence, complexity is a function of the number and source of the client and server data tables that are required to generate the screen or report and the number of views or sections presented as part of the screen or report.

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:

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

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

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

for different levels of developer experience and development environment maturity.

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

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

In more advanced COCOMO II models,[12] a variety of scale factors, cost drivers, and adjustment procedures are required. A complete discussion of these is beyond

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			13	25	50

the scope of this book. If you have further interest, see [Boe00] or visit the COCOMO II website.

26.7.3 The Software Equation

The *software equation* [Put92] is a dynamic multivariable model that assumes a specific distribution of effort over the life of a software development project. The model has been derived from productivity data collected for over 4000 contemporary software projects. Based on these data, we derive an estimation model of the form

$$E = \frac{\text{LOC} \times B^{0.333}}{P^3} \times \frac{1}{t^4} \tag{26.4}$$

where

E = effort in person-months or person-years
t = project duration in months or years
B = "special skills factor"[13]
P = "productivity parameter" that reflects: overall process maturity and management practices, the extent to which good software engineering practices are used, the level of programming languages used, the state of the software environment, the skills and experience of the software team, and the complexity of the application

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 productivity parameter can be derived for local conditions using historical data collected from past development efforts.

You should note that the software equation has two independent parameters: (1) an estimate of size (in LOC) and (2) an indication of project duration in calendar

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

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

$$E = 180\, Bt^3 \text{ in person-months for } E \geq 20 \text{ person-months} \tag{26.5b}$$

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,

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

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

The results of the software equation correspond favorably with the estimates developed in Section 26.6. Like the COCOMO model noted in Section 26.7.2, the software equation continues to evolve. Further discussion of an extended version of this estimation approach can be found in [Put97b].