Software Quality

Introduction to System Quality

Concept of Quality:

• Generally agreed to be beneficial.

• Often vaguely defined in practice.

• Requires precise definition of required qualities.

Objective Assessment:

• Judging if a system meets quality requirements needs measurement.

• Critical for package selection, e.g., Brigette at Brightmouth College.

Development Perspective:

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

• During development, it's important to:

Emphasis on Development Methods:

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

o Checking if suppliers use the best development methods.

o Ensuring these methods will lead to the required quality in the final system.

The place of software quality in project planning

Quality Concerns:

• Present at all stages of project planning and execution.

• Key points in the Step Wise framework where quality is particularly emphasized:

Step 1: Identify Project Scope and Objectives

o Objectives may include qualities of the application to be delivered.

Step 2: Identify Project Infrastructure

o Activity 2.2 involves identifying installation standards and procedures, often related to quality.

Step 3: Analyze Project Characteristics

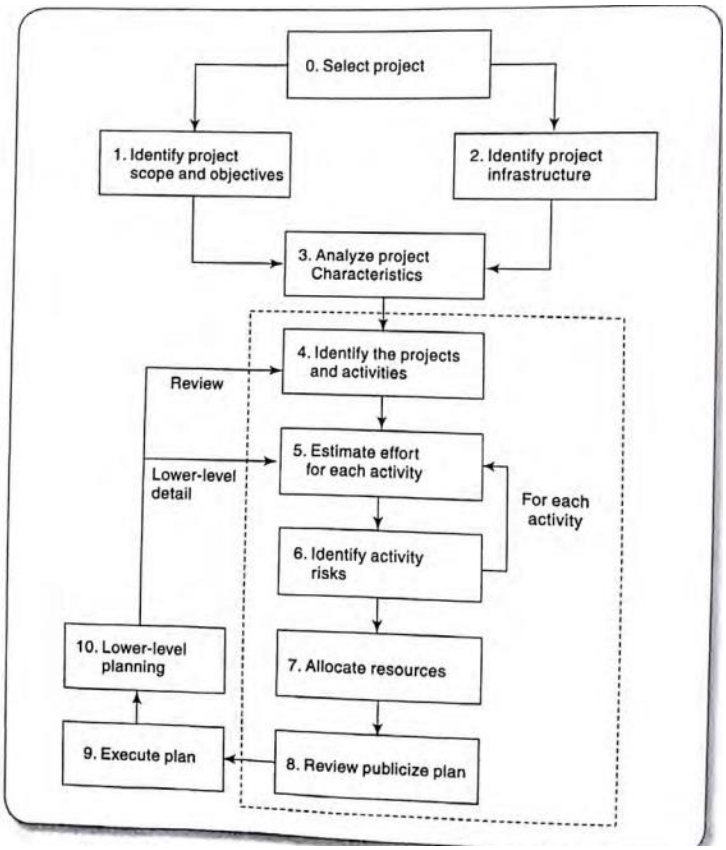
o Activity 3.2 involves analyzing other project characteristics, including quality-based ones.

o 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 Products and Activities of the Project

o Identify entry, exit, and process requirements for each activity.

Step 5: Review and Publicize Plan



Importance of software quality

Quality is a concern for all producers of goods and services

1. Increasing Criticality of Software

Special Demands in Software:

o Final customers and users are increasingly concerned about software quality, particularly reliability.

o Greater reliance on computer systems and use in safety-critical applications (e.g., aircraft control).

2. Intangibility of Software

o Difficulty in verifying the satisfactory completion of project tasks.

O Tangibility is achieved by requiring developers to produce "deliverables" that can be examined for quality.

3. Accumulating Errors During Software Development

o Errors in earlier steps can propagate and accumulate in later steps.

O Errors found later in the project are more expensive to fix.

o The unknown number of errors makes the debugging phase difficult to control.

Defining Software Quality

System Requirements:

• Functional Requirements: Define what the system is to do.

• Resource Requirements: Specify allowable costs.

• Quality Requirements: State how well the system is to operate.

External and Internal Qualities:

• External Qualities: Reflect the user's view, such as usability.

• Internal Factors: Known to developers, such as well-structured code, which may enhance reliability.

Measuring Quality:

• Necessity of Measurement: To judge if a system meets quality requirements, its qualities must be measurable.

• Good Measure: Relates the number of units to the maximum possible (e.g., faults per thousand lines of code).

Direct vs. Indirect Measures:

• Direct Measurement: Measures the quality itself (e.g., faults per thousand lines of code).

• Indirect Measurement: Measures an indicator of the quality (e.g., number of user inquiries at a help desk as an indicator of usability).

Setting Targets:

• Impact on Project Team: Quality measurements set targets for team members.

• Meaningful Improvement: Ensure that improvements in measured quality are meaningful.

o Example: Counting errors found in program inspections may not be meaningful if errors are allowed to pass to the inspection stage rather than being eradicated earlier.

Drafting a Quality Specification for Software Products

When there's concern about a specific quality characteristic in a software product, a

quality specification should include the following details:

1. Definition/Description

o Definition: Clear definition of the quality characteristic.

o 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

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

4. Minimally Acceptable

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

5. Target Range

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

6. Current Value

o 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:

o Definition: Percentage of a particular time interval that a system is usable. o Scale: Percentage (%).

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

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

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

2. Mean Time Between Failures (MTBF):

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

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

o Test: Calculate the average time elapsed between system failures.

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

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

3. Failure on Demand:

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

o Scale: Probability (0 to 1).

o Test: Evaluate the system's response to demand or transaction processing. o Minimally Acceptable: Lower probability of failure is desired; varies by system criticality.

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

4. Support Activity:

o Definition: Number of fault reports generated and processed. o Scale:

Count (number of reports).

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

Minimally Acceptable: Lower number of fault reports indicates better

reliability.

o 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.

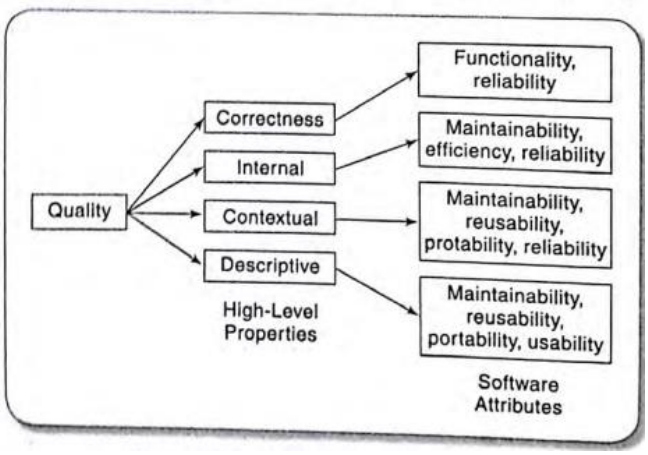
• Durability: Measure of the product life.

• 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.

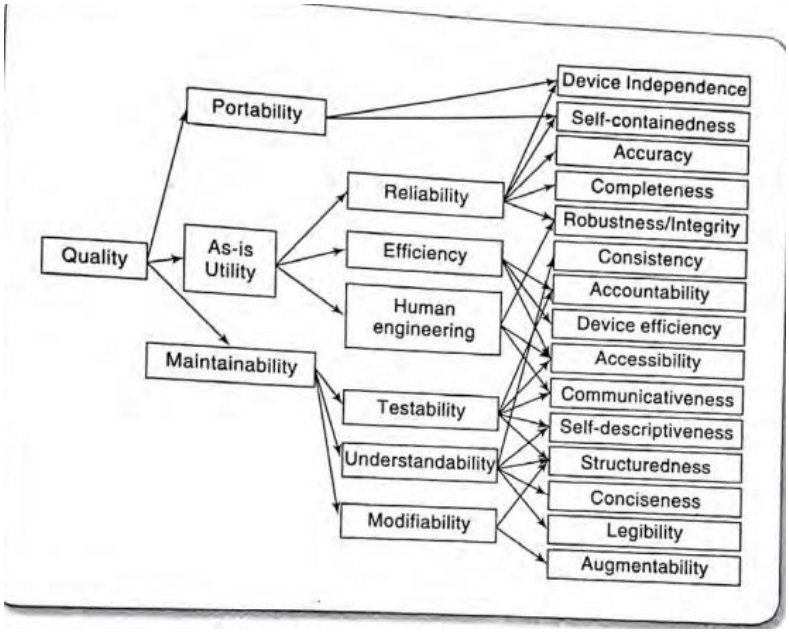
Portability: The effort required to transfer the software product from one hardware or software system environment to another.

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 below Fig



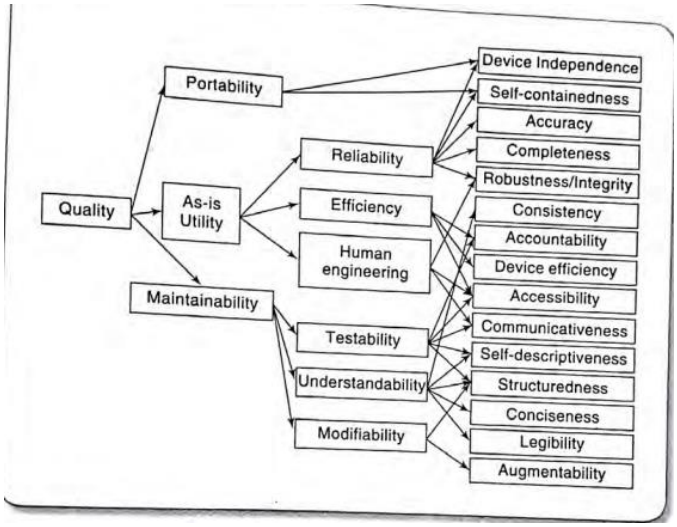
3) Boehm's Model: Boehm's suggested that the quality of a software can be defined based on these high-level characteristics that are important for the users of the software. These three high-level characteristics are the following:

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



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 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

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

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

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

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

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

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 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

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, 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.

3. Map measurements onto ratings that reflect user satisfaction.

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

4. ldentify 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.

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:

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.

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

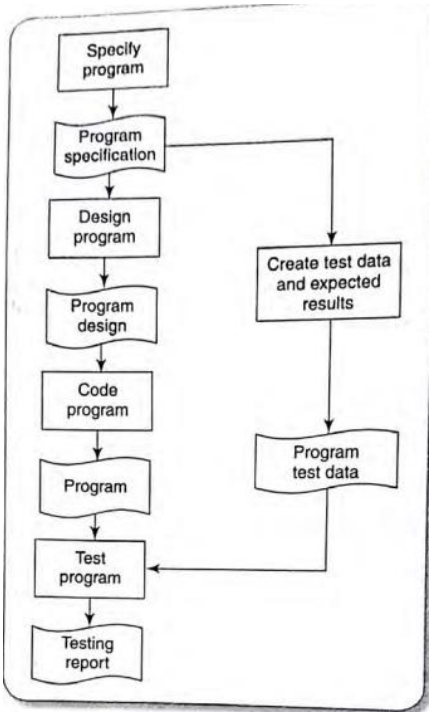
key aspects:

Product Quality Management

Entry Requirements: which have to be in place before an activity can start,

Implementation Requirements: which define how a process to be conducted.

Exit Requirements: which have to be fulfilled before an activity is deemed to have been completed.



Quality Management Systems

BS EN ISO 9001:2000

Principles of BS EN ISO 9001:2000

1. Customer Focus:

Understanding and meeting customer requirements to enhance satisfaction.

2. Leadership:

Providing unity of purpose and direction for achieving quality objectives.

3. Involvement of People:

Engaging employees at all levels to contribute effectively to the QMS.

4. Process Approach:

Focusing on individual processes that create products or deliver services.

Managing these processes as a system to achieve organizational objectives.

5. Continuous Improvement:

Continually enhancing the effectiveness of processes based on objective measurements and analysis.

6. Factual Approach to Decision Making:

Making decisions based on analysis of data and information.

7. Mutually Beneficial Supplier Relationships:

Building and maintaining good relationships with suppliers to enhance capabilities and performance.

Activities in BS EN ISO 9001:2000 Cycle

1. Understanding Customer Needs:

Identifying and defining customer requirements and expectations.

2. Establishing Quality Policy:

Defining a framework for quality objectives aligned with organizational goals.

3. Process Design:

Designing processes that ensure products and services meet quality objectives.

4. Allocation of Responsibilities:

Assigning responsibilities for meeting quality requirements within each process.

5. Resource Management:

Ensuring adequate resources (human, infrastructure, etc.) are available for effective process execution.

6. Measurement and Monitoring:

Designing methods to measure and monitor process effectiveness and efficiency.

Gathering data and identifying discrepancies between actual performance and targets.

Process Capability Models

1. SEI Capability Maturity Model (CMM) and CMMI

Developed by the Software Engineering Institute (SEI), CMM and CMMI provide a framework for assessing and improving the maturity of processes.

SEI CMM Levels:

Level 1: Initial

Characteristics:

- ❖ Chaotic and ad hoc development processes.
- ❖ Lack of defined processes or management practices.
- ❖ Relies heavily on individual heroics to complete projects.

Outcome:

- ❖ Project success depends largely on the capabilities of individual team members.
- ❖ High risk of project failure or delays.

Level 2: Repeatable

Characteristics:

❖ Basic project management practices like planning and tracking costs/schedules are in place.

❖ Processes are somewhat documented and understood by the team.

Outcome:

❖ Organizations can repeat successful practices on similar projects.

❖ Improved project consistency and some level of predictability.

Level 3: Defined

Characteristics:

❖ Processes for both management and development activities are defined and documented.

❖ Roles and responsibilities are clear across the organization.

❖ Training programs are implemented to build employee capabilities.

❖ Systematic reviews are conducted to identify and fix errors early.

Outcome:

❖ Consistent and standardized processes across the organization.

❖ Better management of project risks and quality.

4. Level 4: Managed

Characteristics:

❖ Processes are quantitatively managed using metrics.

❖ Quality goals are set and measured against project outcomes.

❖ Process metrics are used to improve project performance.

Outcome:

- ❖ Focus on managing and optimizing processes to meet quality and performance goals.
- ❖ Continuous monitoring and improvement of project execution.

5. Level 5: Optimizing

Characteristics:

- ❖ Continuous process improvement is ingrained in the organization's culture.
- ❖ Process metrics are analyzed to identify areas for improvement.
- ❖ Lessons learned from projects are used to refine and enhance processes.
- ❖ Innovation and adoption of new technologies are actively pursued.

Outcome:

- ❖ Continuous innovation and improvement in processes.
- ❖ High adaptability to change and efficiency in handling new challenges.
- ❖ Leading edge in technology adoption and process optimization.

CMMI (Capability Maturity Model Integration)

Structure and Levels of CMMI

1. Levels of Process Maturity: Like CMM, CMMI defines five maturity levels, each representing a different stage of process maturity and capability.

These levels are:

- ❖ Level 1: Initial (similar to CMM Level 1)
- ❖ Level 2: Managed (similar to CMM Level 2)
- ❖ Level 3: Defined (similar to CMM Level 3)
- ❖ Level 4: Quantitatively Managed (an extension of CMM Level 4)
- ❖ Level 5: Optimizing (an extension of CMM Level 5)

Implementing process improvement

Implementing process improvement in UVW, especially in the context of software development for machine tool equipment, involves addressing several key challenges identified within the organization.

Here's a structured approach, drawing from CMMI principles, to address these issues and improve process maturity:

Identified Issues at UVW

1. Resource Over commitment:

Issue: Lack of proper liaison between the Head of Software Engineering and Project Engineers leads to resource over commitment across new systems and maintenance tasks simultaneously.

Impact: Delays in software deliveries due to stretched resources.

2. Requirements Volatility:

Issue: Initial testing of prototypes often reveals major new requirements.

Impact: Scope creep and changes lead to rework and delays.

3. Change Control Challenges:

Issue: Lack of proper change control results in increased demands for software development beyond original plans.

Impact: Increased workload and project delays.

4. Delayed System Testing:

Issue: Completion of system testing is delayed due to a high volume of bug fixes.

Impact: Delays in product release and customer shipment.

Steps for Process Improvement

1. Formal Planning and Control

Objective: Introduce structured planning and control mechanisms to assess and distribute workloads effectively.

Actions:

❖ Implement formal project planning processes where software requirements are mapped to planned work packages.

❖ Define clear milestones and deliverables, ensuring alignment with both hardware and software development phases.

❖ Monitor project progress against plans to identify emerging issues early.

Expected Outcomes:

❖ Improved visibility into project status and resource utilization.

❖ Early identification of potential bottlenecks or deviations from planned schedules.

❖ Enable better resource allocation and management across different projects.

2. Change Control Procedures

Objective: Establish robust change control procedures to manage and prioritize system changes effectively.

Actions:

❖ Define a formal change request process with clear documentation and approval workflows.

❖ Ensure communication channels between development teams, testing groups, and project stakeholders are streamlined for change notifications.

❖ Implement impact assessment mechanisms to evaluate the effects of changes on project timelines and resources.

Expected Outcomes:

❖ Reduced scope creep and unplanned changes disrupting project schedules.

❖Enhanced control over system modifications, minimizing delays and rework.

3. Enhanced Testing and Validation

Objective: Improve testing and validation processes to reduce delays in system testing and bug fixes.

Actions:

❖ Strengthen collaboration between development and testing teams to ensure comprehensive test coverage early in the development lifecycle.

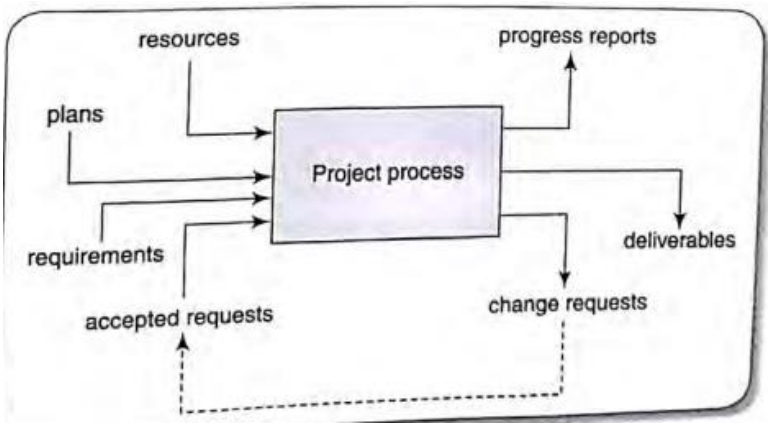
❖ Implement automated testing frameworks where feasible to expedite testing cycles.

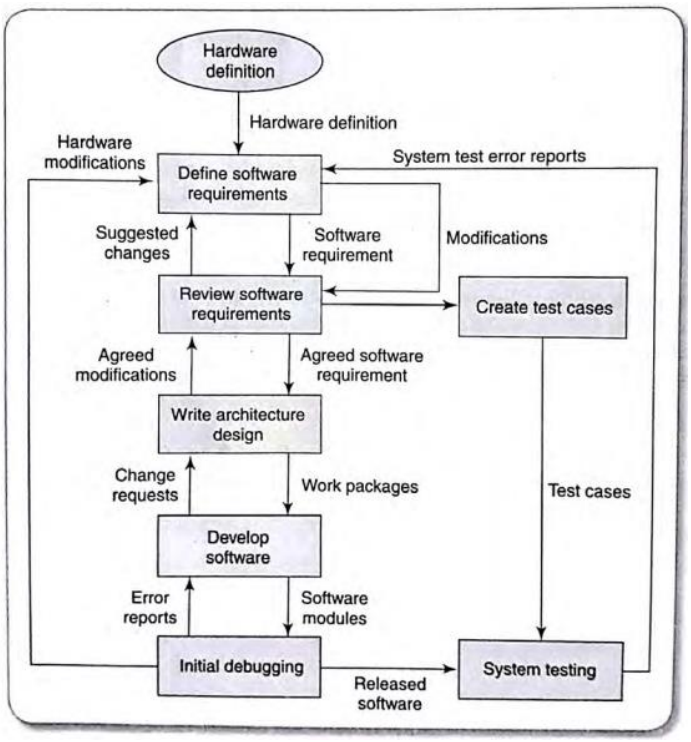
❖ Foster a culture of quality assurance and proactive bug identification throughout the development phases.

Expected Outcomes:

❖ Faster turnaround in identifying and resolving bugs during testing.

❖ Timely completion of system testing phases, enabling on-time product releases.





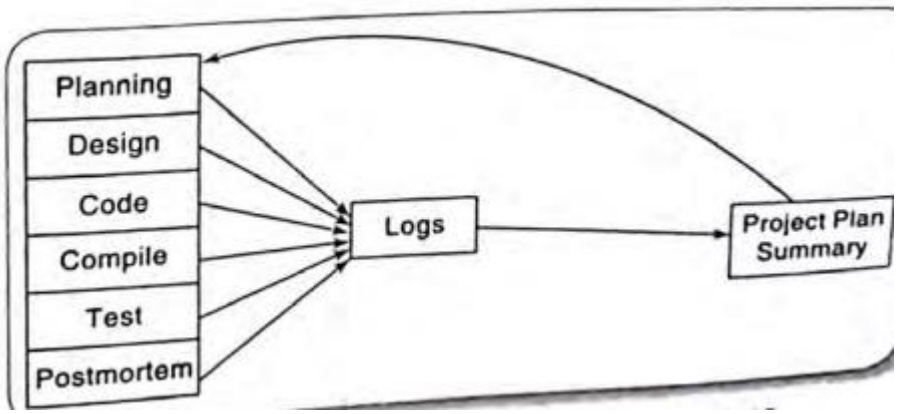
Personal Software Process (PSP)

PSP is based on the work of Watts Humphrey. PSP is suitable for individual use. PSP is a framework that helps engineers to measure and improve the way they work .It helps in developing personal skills and methods by estimating, planning, and tracking performance against plans, and provides a defined process which can be tuned by individuals.

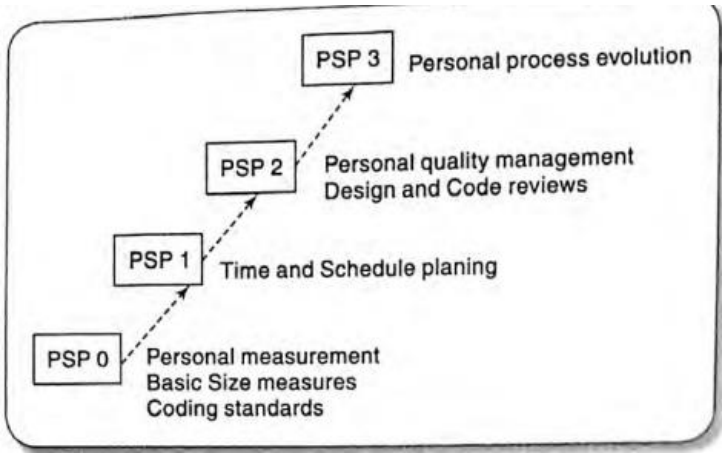
Time Management: PSP advocates that developers should rack the way they spend time. The actual time spent on a task should be measured with the help of a stop-clock to get an objective picture of the time spent. An engineer should measure the time he spends for various development activities such as designing, writing code, testing etc.

PSP Planning: Individuals must plan their project. The developers must estimate the maximum. minimum, and the average LOC required for the product. They record the plan data in a project plan summary.

The PSP is schematically shown in figure below. As an individual developer must plan the personal activities and make the basic plans before starting the development work. While carrying out the activities of different phases of software development, the individual developer must record the log data using time measurement.



During post implementation project review, the developer can compare the log data with the initial plan to achieve better planning in the future projects, to improve his process etc. The four maturity levels of PSP have schematically been shown in figure next. The activities that the developer must perform for achieving a higher level of maturity have also been annotated on the diagram.



Six Sigma

• Motorola, USA , initially developed the six-sigma method in the early 1980s. The purpose of six sigma is to develop processes to do things better, faster, and at a lower cost.

• Six sigma becomes applicable to any activity that is concerned with cost, timeliness, and quality of results. Therefore, it is applicable to virtually every industry.

Steps to Implement Six Sigma at UVW

1. Define:

Objective: Clearly define the problem areas and goals for improvement.

Action: Identify critical processes such as software development, testing, and deployment where defects and variability are impacting quality and delivery timelines.

2. Measure:

Objective: Quantify current process performance and establish baseline metrics.

Action: Use statistical methods to measure defects, cycle times, and other relevant metrics in software development and testing phases.

3. Analyze:

Objective: Identify root causes of defects and variability in processes.

Action: Conduct thorough analysis using tools like root cause analysis, process mapping, and statistical analysis to understand why defects occur and where process variations occur.

4. Improve:

Objective: Implement solutions to address root causes and improve process performance.

Action: Develop and implement process improvements based on the analysis findings. This could include standardizing processes, enhancing communication between teams (e.g., software development and testing), and implementing better change control procedures.

5. Control:

Objective: Maintain the improvements and prevent regression.

Action: Establish control mechanisms to monitor ongoing process performance. Implement measures such as control charts, regular audits, and performance reviews to sustain improvements.

Techniques to Help Enhance Software Quality

The discussion highlights several key themes in software quality improvement over time, emphasizing shifts in practices and methodologies:

Three main themes emerge:

Increasing visibility: A landmark in this movement towards making the software development process more visible was the advocacy by the American software guru, Gerald Weinberg of

egoless programming. Weinberg encouraged the simple practice of programmer looking at each other code.

Procedure structure:

• Initially, software development lacked structured methodologies, but over time, methodologies with defined processes for every stage (like Agile, Waterfall, etc.) have become prevalent.

• Structured programming techniques and 'clean-room' development further enforce procedural rigor to enhance software quality

Checking intermediate stages:

• Traditional approaches involved waiting until a complete, albeit imperfect, version of software was ready for debugging.

• Contemporary methods emphasize checking and validating software components early in development, reducing reliance on predicting external quality from early design documents.

Inspections:

▪ Inspections are critical in ensuring quality at various development stages, not just in coding but also in documentation and test case creation.

It is very effective way of removing superficial errors from a piece of work.

• It motivates developers to produce better structured and self-explanatory software.

• It helps spread good programming practice as the participants discuss the advantages and disadvantages of specific piece of code.

• It enhances team spirit.

▪ Techniques like Fagan inspections, pioneered by IBM, formalize the review process with trained moderators leading discussions to identify defects and improve quality.

The general principles behind Fagan method

• Inspections are carried out on all major deliverables.

• All types of defects are noted.

• Inspection can be carried out by colleagues at all levels except the very top.

• Inspection can be carried using a predefined set of steps.

• Inspection meeting do not last for more than two hours.

• The inspection is led by a moderator who has had specific training in the techniques.

• The participants have define rules.

• Checklist are used to assist the fault-finding process.

• Material is inspected at an optimal rate of about 100 lines an hour.

• Statistics are maintained so that the effectiveness of the inspection process can be monitored.

Formal methods

• Clean-room development, uses mathematical verification techniques. These techniques use unambiguous, mathematically based , specification language of which Z and VDM are examples.

They are used to define preconditions and post-conditions for each procedure.

• Precondition define the allowable states, before processing, of the data items upon which a procedure is to work.

• Post condition define the state of those data items after processing. The mathematical notation should ensure that such a specification is precise and unambiguous.

Software quality circles (SWQC)

▪ SWQCs are adapted from Japanese quality practices to improve software development processes by reducing errors.

• Staff are involved in the identification of sources of errors through the formation of quality circle. These can be set up in all departments of an organizations including those producing software where they are known as software quality circle(SWQC).

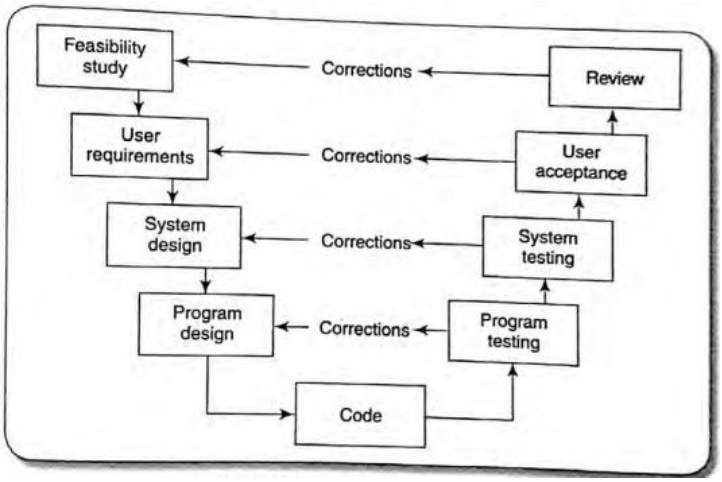
• A quality circle is a group of four to ten volunteers working in the same area who meet for ,say, an hour a week to identify, analyze and solve their work -related problems. One

of their number is a group leader and there could be an outsider a facilitator, who can advise on procedural matters.

• Associated with quality circles is the compilation of most probable error lists. For example, at IOE, Amanda might find that the annual maintenance contracts project is being delayed because of errors in the requirements specifications.

Testing

The final judgement of the quality of the application is whether it actually works correctly when executed.



• Considering the diagrammatic representation of V-Model , which stress the necessity of validation activities that match the activities creating the products of the project.

• The V-process model is expanding the activity box ‘testing’ in the waterfall model.

• Each step has a matching validation process which can ,where defects are found, cause a loop back to the corresponding development stage and a reworking of a following step.

Verification verses Validation:

Both are bug detection techniques which helps to remove errors in software.

The main difference between these two techniques are the following:

❖ Verification is the process of determining whether the output of one phase of software development conforms to that of its previous phase. For example, a verification step can be to check if the design documents produced after the design step conform to the requirement specification.

❖ Validation is the process of determining whether fully developed software conforms to its requirements specification. Validation is applied to the fully developed and integrated software to check if it satisfies customer's requirements.

❖ Verification is carried put during development process to check of the development activities are being carried out correctly , where as

❖ Validation is carried out towards the end of the development process to check if the right product as required by the customer had been developed.

• All the boxes in the right hand of the V-process model of below Figure correspond to verification activities except the system testing block which corresponds to validation activity.

Test case Design

There are two approaches to design test cases:

1) Black-box approach

2) White-box(or glass-box) approach.

In black-box approach, test cases are designed using only the functional specification of the software.

That is, test cases are designed solely based on an analysis of the input/output behavior. Hence black-box testing is also known as functional testing and also as requirement-driven testing.

Design of white-box test cases requires analysis of the source code, It is also called structural testing or structure-driven testing.

Levels of Testing

A software product is normally tested at three different stages or levels. These three testing stages are:

1) Unit Testing

2) Integration Testing

3) System Testing

During unit testing, the individual components (or units) of a program are tested. For every module, unit testing is carried out as soon as the coding for it is complete. Since every module is tested separately, there is a good scope for parallel activities during unit testing.

After testing all the units individually, the units are integrated over a number of steps and tested after each step of integration (Integration testing).

Finally, the fully integration system is tested (System testing).

Testing Activities:

Testing involves performing the following main activities:

1) Test Planning: Test Planning consists of determining the relevant test strategies and planning for any test bed that may be required. A test bed usually includes setting up the hardware or simulator.

2) Test Case Execution and Result Checking: Each test case is run and the results are compared with the expected results. A mismatch between the actual result and expected results indicates a failure. The test cases for which the system fails are noted down for test reporting.

3) Test Reporting: When the test cases are run, the tester may raise issues, that is, report discrepancies between the expected and the actual findings. A means of formally recording these issues and their history is needed. A review body adjudicates these issues. The outcome of this scrutiny would be one of the following:

• The issue is dismissed on the grounds that there has been a misunderstanding of a requirement by the tester.

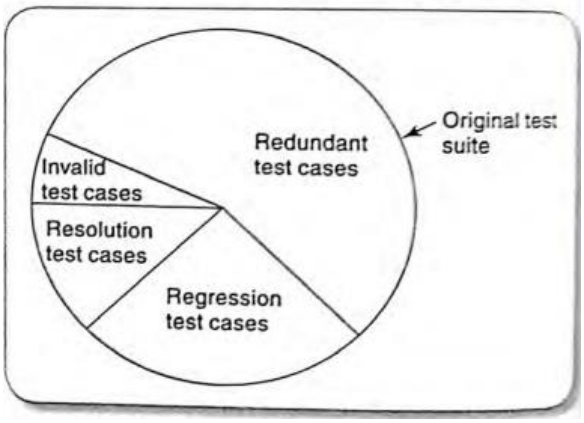
• The issue is identified as a fault which the developers need to correct -Where development is being done by contractors, they would be expected to cover the cost of the correction.

• It is recognized that the software is behaving as specified, but the requirement originally agreed is in fact incorrect.

• The issue is identified as a fault but is treated as an off-specification -It is decided that the application can be made operational with the error still in place.

4) Debugging: For each failure observed during testing, debugging is carried out to identify the statements that are in error.

5) Defect Retesting: Once a defect has been dealt with by the development team, the corrected code is retested by the testing team to check whether the defect has successfully been addressed. Defect retest is also called resolution testing. The resolution tests are a subset of the complete test suite



6) Regression Testing: Regression testing checks whether the unmodified functionalities still continue to work correctly. Thus, whenever a defect is corrected and the change is incorporated in the program code, the change introduced to correct an error could actually introduce errors in functionalities that were previously working correctly.

7) Test Closure: Once the system successfully passes all the tests, documents related to lessons learned, results, logs etc., are achieved for use as a reference in future projects.

Who Performs Testing

Many organizations have separate system testing groups to provide an independent assessment of the correctness of software before release. In other organizations, staff is allocated to a purely testing role but work alongside the develops instead of a separate group.

Test Automation

1) Testing is most time consuming and laborious of all software development. With the growing size of programs and the increased importance being given to product quality, test automation is drawing attention.

2) Test automation is automating one or some activities of the test process. This reduces human effort and time which significantly increases the thoroughness of testing.

3) With automation, more sophisticated test case design techniques can be deployed. By using the proper testing tools automated test results are more reliable and eliminates human errors during testing.

4) Every software product undergoes significant change overtime. Each time the code changes, it needs to be tested whether the changes induce any failures in the unchanged features. Thus the originally designed test suite need to be run repeatedly each time the code changes. Automated testing tools can be used in repeatedly running the same set of test cases.

Estimation of latent errors

The methods of estimating the number of latent errors in software during testing:

1) Using Historical Data :If historical error data from past projects are available, you can use this data to estimate the number of errors per 1000 lines of code. This method allows you to predict the number of errors likely to be found in a new system development of a known size.

2) Seeding Known Errors

- During testing, seed the software with known errors.
- For example, introduce 10 known errors into the code.
- After testing, suppose 30 errors are found, of which 6 are the seeded errors.
- This implies that 60% of the seeded errors were detected.
- Thus, 40% of the errors are still undetected.
- Using this method, you can estimate the total number of errors in the software using the formula:

Estimated Total Errors= (TotalErrors Found/Seeded Errors Found)×Total Number of Seeded errors.

3) Alternative Approach by Tom Gilb---Independent Reviews

- Instead of seeding known errors, use independent reviewers to inspect or test the same code.
- Collect three counts:

    n1: Number of valid errors found by reviewer A

n2: Number of valid errors found by reviewer B

n12: Number of cases where the same error is found by both A and B

• The smaller the proportion of errors found by both A and B compared to those found by only one reviewer, the larger the total number of errors likely to be in the software.

• Estimate the total number of errors using the formula:

$$n=(n1\times n2)/n12$$

This method helps in estimating the number of latent errors without deliberately introducing known errors into the software.

For example, A finds 30 errors and B finds 20 errors of which 15 are common to both A and B. These estimated total number of errors would be:

$$(30*20)/15=40$$

Software Reliability

The reliability of a software product denotes trustworthiness or dependability.

It can be defined as the probability of its working correctly over a given period of time.

Software product having a large number of defects is unreliable. Reliability of the system will improve if the number of defects in it is reduced.

Reliability is an observer dependent, it depends on the relative frequency with which different users invoke the functionalities of a system. It is possible that because of different usage patterns of the available functionalities of software, a bug which frequently shows up for one user, may not show up at all for another user, or may show up very infrequently.

Reliability of the software keeps on improving with time during the testing and operational phases as defects are identified and repaired. The growth of reliability over the testing and operational phases can be modelled using a mathematical expression called Reliability Growth Model (RGM).

Popular RGMs (Explanation in Page 53)

• Jelinski-Moranda model

• Littlewood-Verrall model

• Goel-Okumoto model

RGMs help predict reliability levels during the testing phase and determine when testing can be stopped.

Challenges in Software Reliability which makes which makes difficult to measure than hardware reliability.

• Dependence on the specific location of bugs

• Observer-dependent nature of reliability

• Continuous improvement as errors are detected and corrected.

Hardware vs. Software Reliability

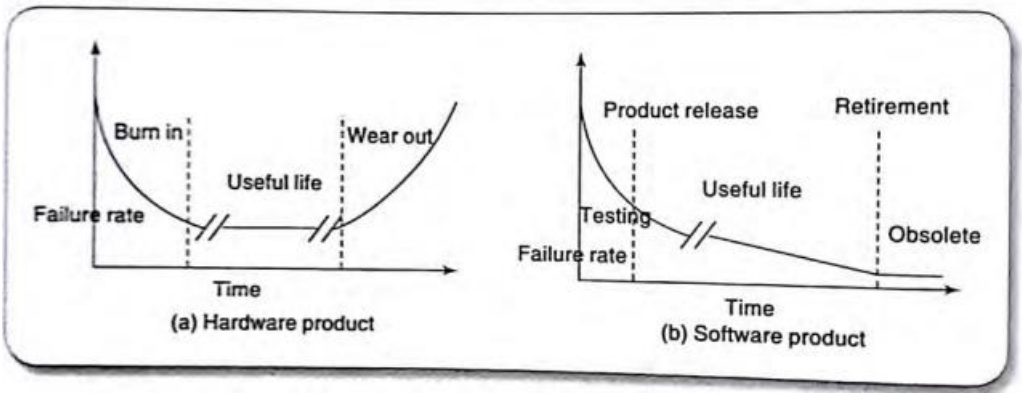
Hardware failures typically result from wear and tear, whereas software failures are due to bugs.

Hardware failure often requires replacement or repair of physical components. Software failures need bug fixes in the code, which can affect reliability positively or negatively.

Hardware Reliability: Concerned with stability and consistent inter-failure times.

Software Reliability: Aims for growth, meaning an increase in inter-failure times as bugs are fixed.

Hardware: Shows a "bathtub" curve where failure rate is initially high, decreases during the useful life, and increases again as components wear out. Software: Reliability generally improves over time as bugs are identified and fixed, leading to decreased failure rates



Software Reliability Metrics

The six metric that correlate with reliability as follows:

1) Rate of Occurrence of Failure (ROCOF):

• Measures the frequency of occurrences of failures.

• Calculated as the ratio of total failures observed to the duration of observation.

• Limited applicability for non-continuously running software. Ex: Library software is idle until a book issue request is made.

2) Mean Time to Failure (MTTF):

• Time between two successive failures, averaged over a large number of failures.

• Calculation involves summing up time intervals between failures and dividing by the number of failures. MTTF can be calculated as

• Only runtime is considered, excluding downtime for fixes.

3) Mean Time to Repair (MTTR):

• Average time required to fix an error.

• Measures the time taken to track and fix failures

4) Mean Time Between Failures (MTBF):

• Sum of MTTF and MTTR. –> MTBF=MTTF+MTTR.

• Indicates the expected time until the next failure after a failure is fixed.

5) Probability of Failure on Demand (POFOD):

• POFOD measures likelihood of system failure when a service request is made. For example, POFOD of 0.001 means that 1 out of every 1000 service requests would result in a failure.

• Suitable for systems not required to run continuously.

6) Availability:

• Measures how likely the system is available for use during a given period.

• Takes into account both failure occurrences and repair times.

Types of Software Failures

• Transient: Failures occurring only for certain inputs while invoking a function.

• Permanent: Failures occurring for all inputs while invoking a function.

• Recoverable: System can recover without shutting down.

• Unrecoverable: System needs to be restarted to recover.

• Cosmetic: Minor

Reliability Growth Modelling

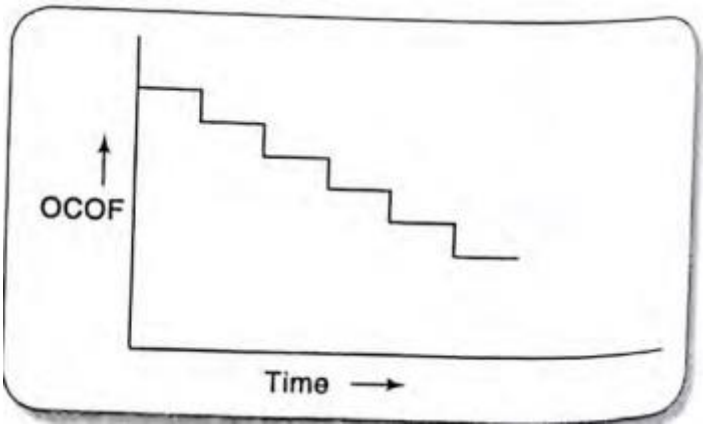
1) Jelinski and Moranda Model

Model Characteristics:

• A step function model assuming reliability increases by a constant increment with each error fix.

• Assumes perfect error fixing, meaning all errors contribute equally to reliability growth.

• Typically, unrealistic as different errors contribute differently to growth. Reliabity growth predicted using this model has been shown in Figure below.



2) Littlewood and Verall's Model

• This model allows for negative reliability growth, acknowledging that repairs can introduce new errors.

• It models the impact of error repairs on product reliability, which diminishes over time as larger contributions to reliability are addressed first.

• Uses Gamma distribution to treat an error's contribution to reliability improvement as an independent random variable.

3) Goel-Okutomo Model

• The rate of failures decreases exponentially over time.

• The model assumes immediate and perfect correction once a failure is detected.

Graph: Illustrates the expected total number of errors over execution time, showing an initial rapid increase in detected errors, which slows down as more errors are corrected.