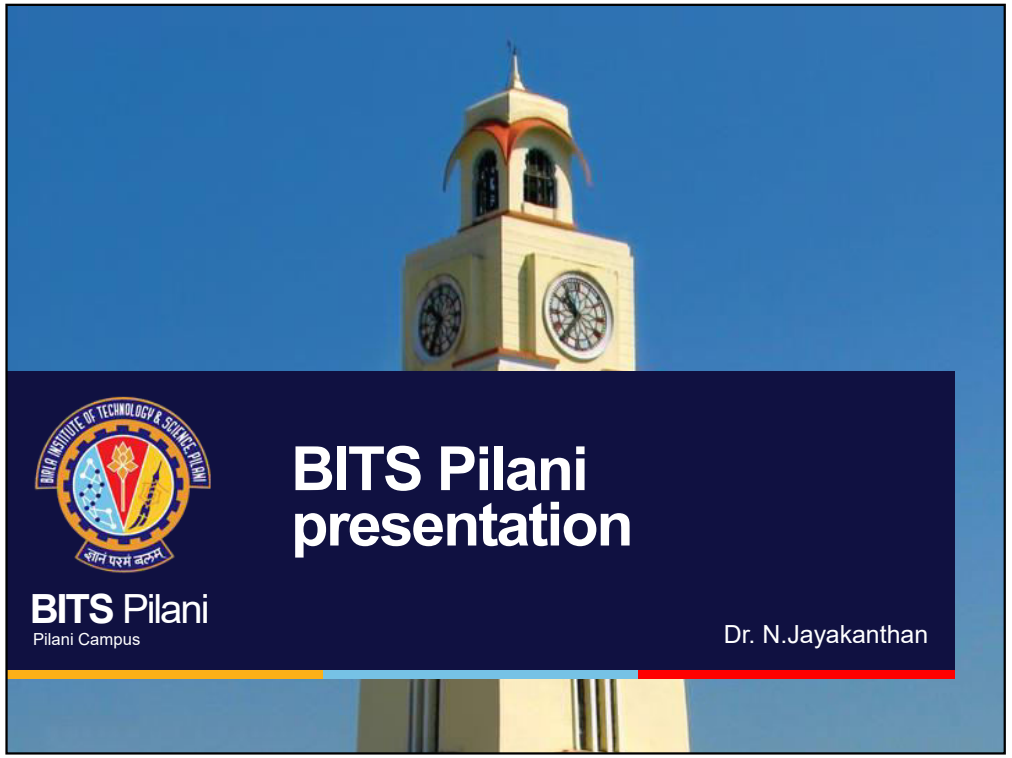
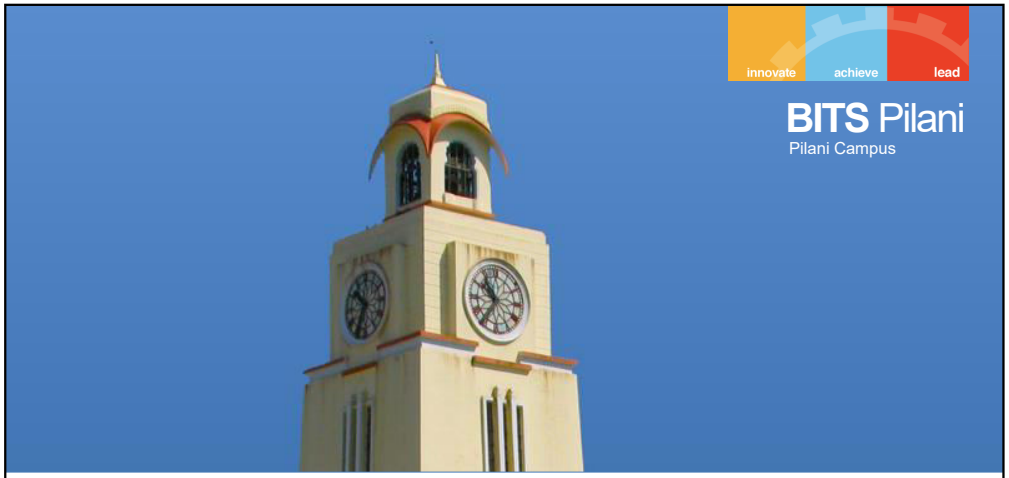


BITS Pilani presentation

BITS Pilani
Pilani Campus

Dr. N.Jayakanthan



innovate achieve lead

BITS Pilani
Pilani Campus

SE ZG501
Software Quality Assurance and Testing
Lecture No. 1

innovate achieve lead

Software Quality Assurance

SQA addresses the global challenge of the **improvement of software quality.**

It seeks to provide an overview of software quality assurance (SQA) practices for customers, managers, auditors, suppliers, and personnel responsible for software projects, development, maintenance, and software services.

innovate achieve lead

In a globally competitive environment, clients and competitors exert a great deal of pressure on organizations.

Clients are increasingly demanding and require, among other things, software that is of high quality, low cost, delivered quickly, and with impeccable after-sales support.

To meet the **demand, quality, and deadlines**, the organization must use **efficient quality assurance practices** for their software activities.

Standards

Standards define ways to maximize performance but managers and employees are largely left to themselves to decide how to practically improve the situation.

They face several problems:

- **increasing pressure to deliver quality products quickly**
- **increasing size and complexity of software and of systems**
- **increasing requirements to meet national, international, and professional standards**
- **subcontracting and outsourcing**
- **distributed work teams**
- **ever changing platforms and technologies.**

INTRODUCTION

Software is developed, maintained, and used by people in a wide variety of situations.

Students ,enthusiasts , professionals address quality problems that arise in the software they are working with

The Guide to the Software Engineering Body of Knowledge (SWEBOK) [SWE 14] constitutes the first international consensus developed on **the fundamental knowledge required by all software engineers**.

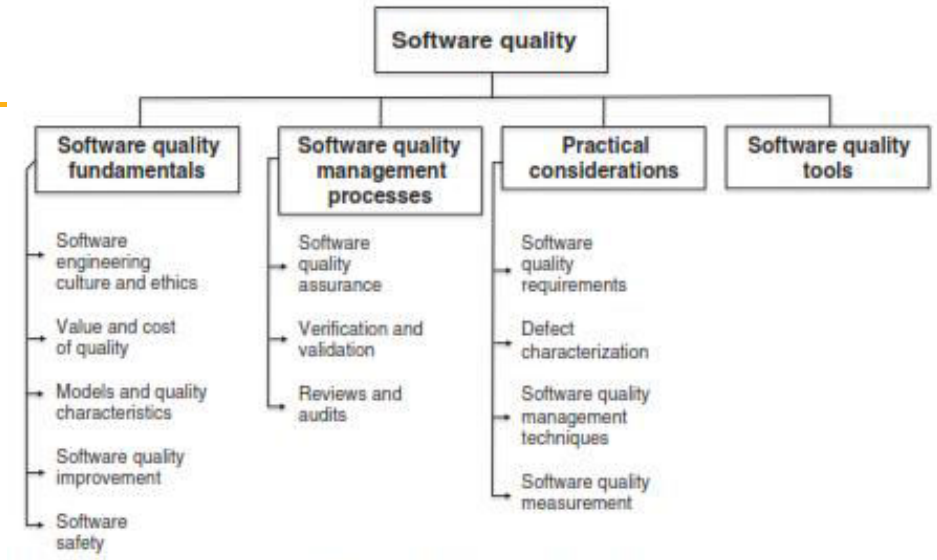


Figure 1.1 Software Quality in the SWEBOK® Guide [SWE 14].

DEFINING SOFTWARE QUALITY

"**software**" "**software quality**" and "**software quality assurance**"

Software [ISO 24765 [ISO 17a]]

1) All or part of **the programs, procedures, rules, and associated documentation** of an information processing system.

2) Computer **programs, procedures, and possibly associated documentation and data** pertaining to the operation of a computer system.

Software found in embedded systems is sometimes called microcode or firmware.

Firmware is present in **commercial mass-market products** and controls machines and devices **used in our daily lives.**

Firmware

Combination of a hardware device and computer instructions or computer data that reside as read-only software on the hardware device.

SOFTWARE ERRORS, DEFECTS, AND FAILURES

- The system crashed during production.
- The designer made an error.
- After a review, we found a defect in the test plan.
- I found a bug in a program today.
- The system broke down.
- The client complained about a problem with a calculation in the payment report.
- A failure was reported in the monitoring subsystem.

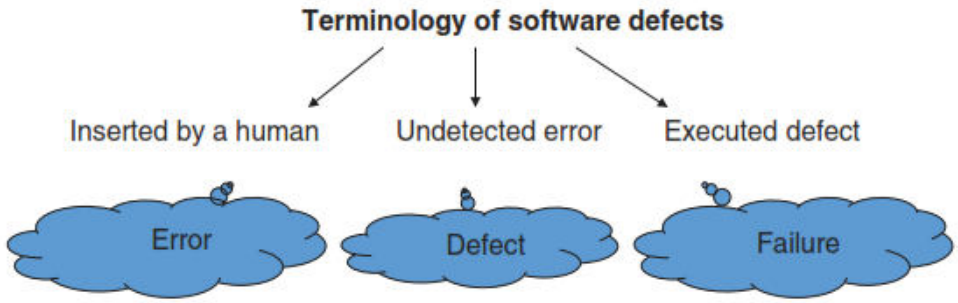
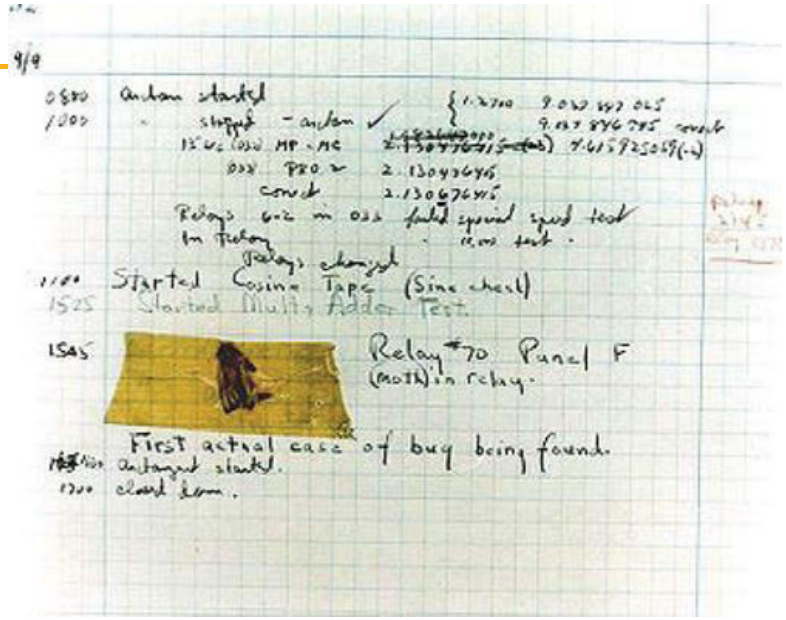


Figure 1.2 Terminology recommended for describing software problems.



A failure (synonymous with a crash or breakdown) is the execution (or manifestation) of a fault in the operating environment.

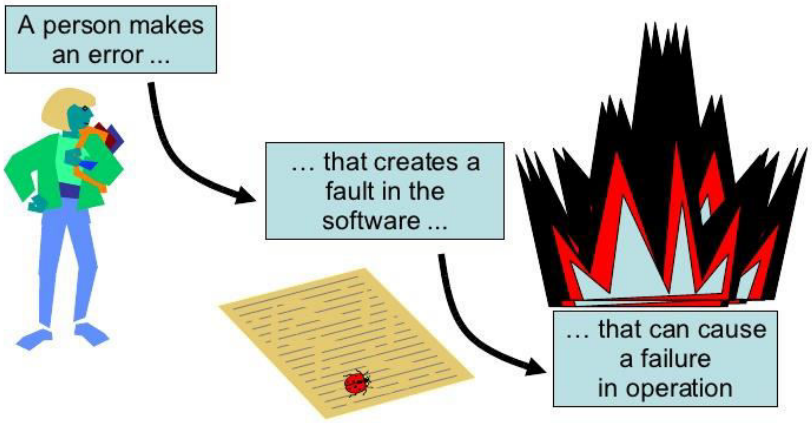
A failure is defined as the **termination of the ability of a component to fully or partially perform a function** that it was designed to carry out.

The origin of a failure lies with a **defect hidden**, that is, not detected by tests or reviews, in the system currently in operation.

Defects (synonym of faults) are human errors that were not detected during software development, quality assurance (QA), or testing.

An error can be found in the documentation, the software source code instructions, the logical execution of the code, or anywhere else in the life cycle of the system.

Error - Fault - Failure



Error, Defect, and Failure

Error

A human action that produces an incorrect result (ISO 24765) [ISO 17a].

Defect

1) A problem (synonym of fault) which, if not corrected, could cause an application to either fail or to produce incorrect results. (ISO 24765) [ISO 17a].
2) An imperfection or deficiency in a software or system component that can result in the component not performing its function, e.g. an incorrect data definition or source code instruction. A defect, if executed, can cause the failure of a software or system component (ISTQB 2011 [IST 11]).

Failure

The termination of the ability of a product to perform a required function or its inability to perform within previously specified limits (ISO 25010 [ISO 11i]).

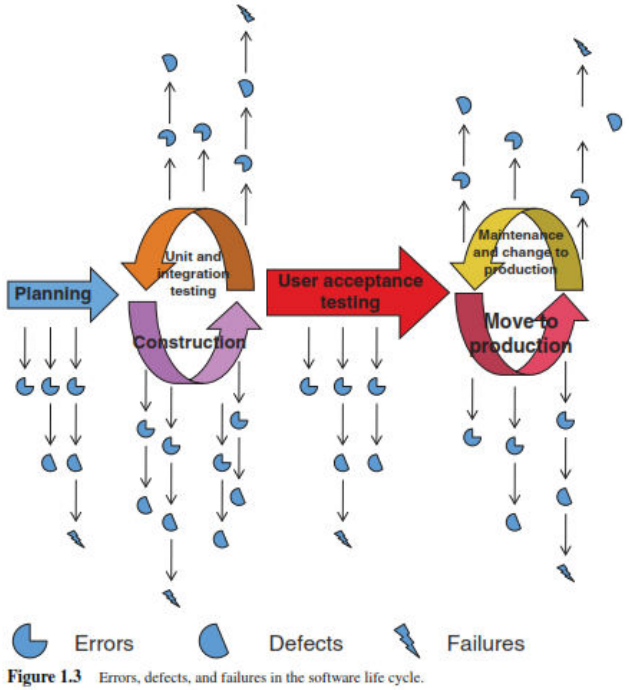


Figure 1.3 Errors, defects, and failures in the software life cycle.

Case 1:

A local pharmacy added a software requirement to its cash register to prevent sales of more than $75 to customers owing more than $200 on their pharmacy credit card.

The programmer did not fully understand the specification and created a sales limit of $500 within the program. This **defect** never caused a failure since no client could purchase more than $500 worth of items given that the pharmacy credit card had a limit of $400.

Case 2: In 2009, a loyalty program was introduced to the clients of American Signature, a large furniture supplier. The specifications described the following business rules: **a customer who makes a monthly purchase that is higher than the average amount of monthly purchases for all customers** will be considered a Preferred Customer.

The Preferred Customer will be identified when making a purchase, and will be immediately given a gift or major discount once a month.

The defect introduced into the system (due to a poor understanding of the algorithm to set up for this requirement) involved only taking into account the average amount of current purchases and not the customer's monthly history. At the time of the software failure, the cash register was identifying far too many Preferred Clients, resulting in a loss for the company.

Development Life Cycle

Software life cycle process that contains the activities of **requirements analysis, design, coding, integration, testing, installation, and support for acceptance** of software products.

Depending on the **business model** of your organization, you will have to allow for **varying degrees of effort in identifying and correcting defects.**

Airbus, Boeing, Bombardier, and Embraer to have **identified and corrected all the defects** in the software for their airplanes **before we board them**!

Research Studies have come to the following conclusions:

- The scope of most defects is very **limited** and **easy to correct.**
- Many **defects occur outside of the coding activity** (e.g., requirement, architecture activities).
- **Poor understanding of the design** is a recurrent problem in programming error studies.
- It is a good idea to **measure** the number and origin of **defects** in your organization **to set targets for improvement**.

SQA need to develop a classification of the causes of software error by category.

1) **problems with defining requirements.**
2) **maintaining effective communication between client and developer.**
3) **deviations from specifications.**
4) **architecture and design errors.**
5) **coding errors (including test code).**
6) **non-compliance with current processes/procedures.**
7) **inadequate reviews and tests.**
8) **documentation errors.**

SOFTWARE QUALITY

- *Conformance to established software requirements; the capability of a software product to satisfy stated and implied needs when used under specified conditions.*
- *The degree to which a software product meets established requirements; however,* ***quality depends upon the degree to which those established requirements accurately represent stakeholder needs, wants, and expectations***

SOFTWARE QUALITY ASSURANCE

Software Engineering

The systematic application of **scientific and technological knowledge, methods, and experience** for the design, implementation, testing, and documentation of software.

Quality Assurance

- **1) a planned and systematic pattern of all actions necessary to provide adequate confidence** that an **item or product conforms to established technical requirements;**
- **2) a set of activities designed to evaluate the process by which products are developed** or manufactured;
- **3) the planned and systematic activities implemented within the quality system, and** demonstrated as needed, **to provide adequate confidence that an entity will fulfil requirements for quality**.

Software Quality Assurance

Definition

A set of activities that define and assess the adequacy of software processes to provide evidence that establishes confidence that the **software processes are appropriate for and produce software products of suitable quality for their intended purposes.**

A key attribute of SQA is the **objectivity of the SQA function with respect to the project.**

The SQA function may also be organizationally **independent of the project**; that is, **free from technical, managerial, and financial pressures from the project**.

SQA Elements

- The need to plan the **quality aspects of a product or service**.
- **Systematic activities** that tell us, **throughout the software life cycle**, that certain corrections are required.
- The quality system is a complete system that must, in the context of quality management, allow for the setting up of a **quality policy and continuous improvement**;
- QA techniques that **demonstrate the level of quality reached** so as to instil confidence in users and lastly.
- **Demonstrate that the quality requirements defined for the project**, for the change or by the software department **have been met**.

BUSINESS MODELS AND THE CHOICE OF SOFTWARE ENGINEERING PRACTICES

Business Model

A business model describes the rationale of how an organization creates, delivers, and captures value (economic, social, or other forms of value).

The essence of a business model is that it defines the manner by which the business enterprise delivers value to customers, entices customers to pay for value, and converts those payments to profit.

Choice of Software Practices

As expected, people from different business sectors chose software engineering practices that would lower the probability of their worst fears(quality and deadlines).

Since their apprehensions are different, their practices are also different.

SUCCESS FACTORS Foster Software Quality

1) SQA techniques adapted to the environment.
2) Clear terminology with regards to software problems.
3) An understanding and specific attention to each major category of software error sources.
4) An awareness of the SQA body of knowledge of the SWEBOK as a guide for SQA.

Factors that may Adversely Affect Software Quality

1) A lack of cohesion between SQA techniques and environmental factors in your organization.
2) Confusing terminology used to describe software problems.
3) A lack of understanding or interest for collecting information on software error sources.
4) Poor understanding of software quality fundamentals.
5) Ignorance or non-adherence with published SQA techniques.

Software quality assurance vs. software quality control

Quality control is defined as "a set of activities designed to evaluate the quality of a developed or manufactured product"
The main objective of **quality assurance is to minimize the cost of guaranteeing** quality by a variety of activities performed throughout the development and manufacturing processes/stages.

These activities prevent the causes of errors, and detect and correct them early in the development process.

Quality assurance activities substantially reduce the rate of products that do not qualify for shipment and, at the same time, reduce the costs of guaranteeing quality in most cases.

The objectives of SQA activities

innovate achieve lead

Software development (process-oriented):

Software maintenance (product-oriented):

innovate achieve lead

Software development (process-oriented):

- 1. Assuring an acceptable level of confidence that the software will conform to functional technical requirements.
- 2. Assuring an acceptable level of confidence that the software will conform to managerial scheduling and budgetary requirements.
- 3. Initiating and managing of activities for the improvement and greater efficiency of software development and SQA activities. This means improving the prospects that the functional and managerial requirements will be achieved while reducing the costs of carrying out the software development and SQA activities.

innovate achieve lead

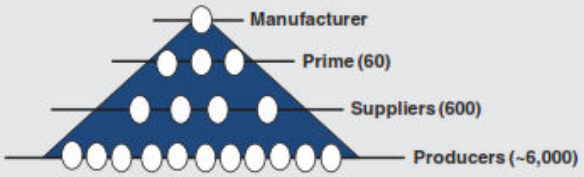
Software maintenance (product-oriented):

- 1. Assuring with an acceptable level of confidence that the software maintenance activities will conform to the functional technical requirements.
- 2. Assuring with an acceptable level of confidence that the software maintenance activities will conform to managerial scheduling and budgetary requirements.
- 3. Initiating and managing activities to improve and increase the efficiency of software maintenance and SQA activities. This involves improving the prospects of achieving functional and managerial requirements while reducing costs.

Quality Culture

innovate achieve lead

A large Japanese electronic product manufacturing company uses a considerable number of suppliers. The supplier pyramid is made up of a first level with some sixty suppliers, a second level with a few hundred suppliers, and a third level with a few thousand very small suppliers.



A software defect for a component produced by a third-level supplier caused a loss of over $200 million for this manufacturer.

Adapted from Shintani (2006) [SHI 06]

Setting up a quality culture and software quality assurance (SQA) principles, as stipulated in the standards, could help solve these problems.

Quality, influenced by organization's senior management and the organization's culture, has a cost, has a positive effect on profits, and must be governed by a code of ethics.

COST OF QUALITY

One of the major factors that explains the resistance to implementing quality assurance is the perception of its **high cost.**

As software engineers, responsible for informing administrators of the **risks that a company takes when not fully committed to the quality of its software**.

A practical manner of beginning the exercise is to **identify the costs of non-quality.**

It is easier to **identify potential savings by studying the problems caused by software**.

Costs of a project

(1) Implementation costs
(2) Prevention costs
(3) Appraisal costs
(4) The costs associated with failures or anomalies.

- If all development activities are error free, then 100% of costs could be implementation costs.
- Given that we make mistakes, we need to be able to identify them. The costs of detecting errors are appraisal costs (e.g., testing).
- Costs due to errors are anomaly costs.
- When we want to reduce the cost of anomalies, we invest in training, tools, and methodology. These are prevention costs.

The “cost of quality” is not calculated in the same way in all organizations.

- There is a certain amount of ambiguity between the notion of the cost of quality, the cost of non-quality, and the cost of obtaining quality.
- Most commonly used model at this time, costs of quality take into account the following five perspectives:

(1) prevention costs, (2) appraisal costs, (3–4) failure costs (internal during development, external on the client’s premises), and (5) costs associated with warranty claims and loss of reputation caused by non-quality.

calculation of the cost of quality in this model is as follows:

Quality costs = Prevention costs
+ Appraisal or evaluation costs
+ Internal and external failure costs
+ Warranty claims and loss of reputation costs

Prevention costs: This is defined as the cost incurred by an organization to prevent the occurrence of errors in the various steps of the development or maintenance process.

- For example, the cost of training employees, the cost of maintenance to ensure a stable manufacturing process, and the cost of making improvements.

Appraisal costs: The cost of verifying or evaluating a product or service during the different steps in the development process. Monitoring system costs (their maintenance and management costs).

Internal failure costs: The cost resulting from anomalies before the product or service has been delivered to the client. Loss of earnings due to non-compliance (cost of making changes, waste, additional human activities, and the use of extra products).

External failure costs: The cost incurred by the company when the client discovers defects. Cost of late deliveries, cost of managing disputes with the client, logistical costs for storing the replacement product or for delivery of the product to the client.

Warranty claims and loss of reputation costs: Cost of warranty execution and damage caused to a corporate image as well as the cost of losing clients.

The first objective of the SQA is to convince management that there are proven benefits to SQA activities.

"identifying an error early in the process can save a lot of time, money and effort."

Figure 2.2 Costs of propagating an error[1] [JON 00].

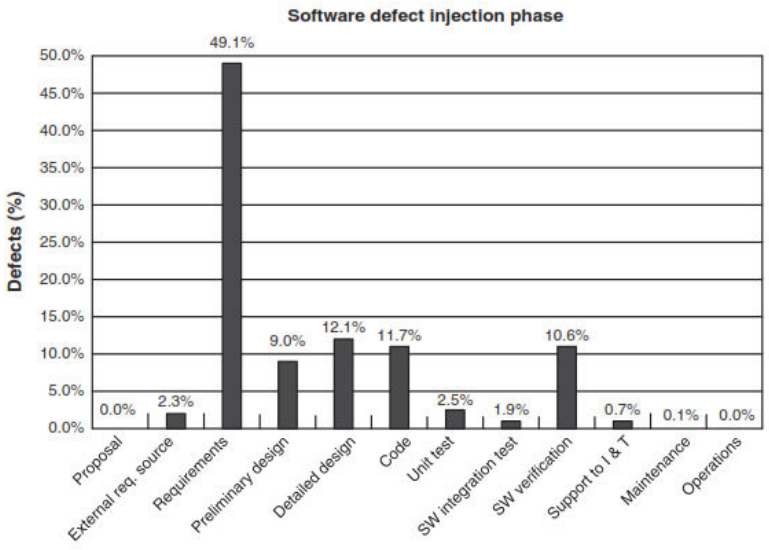
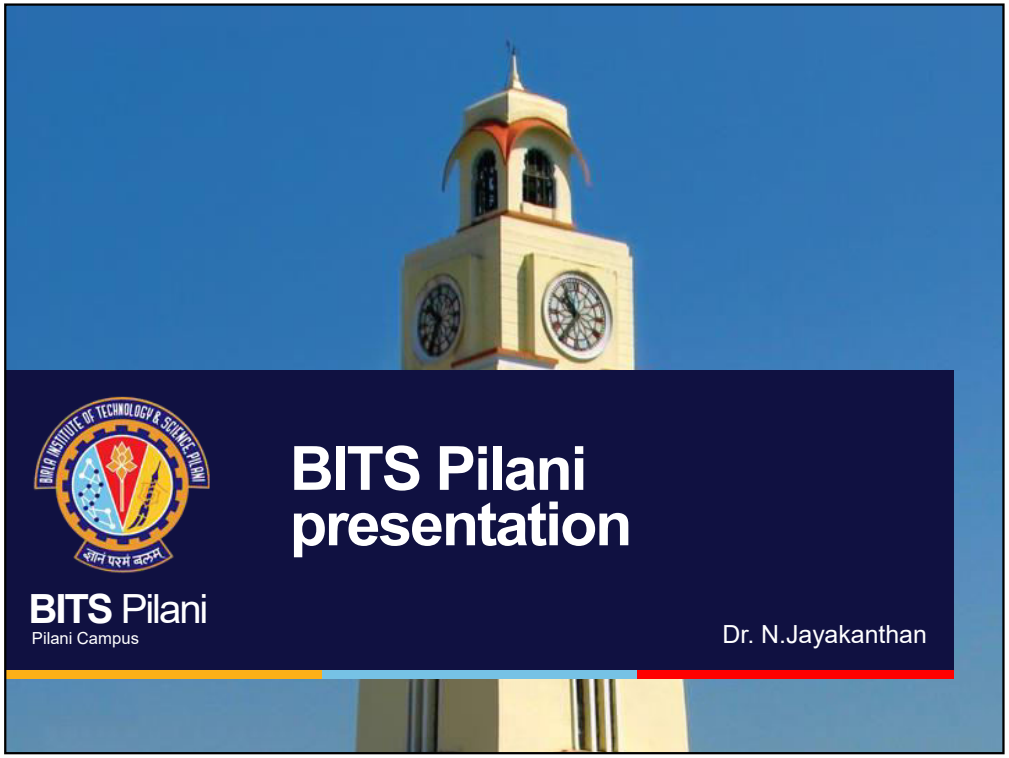
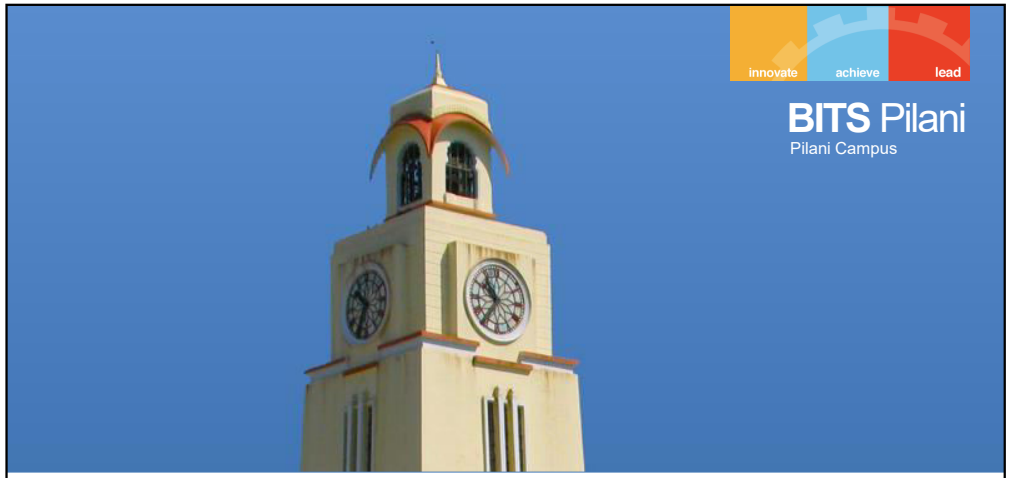


Figure 2.3 Defect injection distribution during the life cycle [SEL 07].





SE ZG501
Software Quality Assurance and Testing
Lecture No. 2

QUALITY CULTURE

Tylor [TYL 10] defined **Human Culture as**

"that complex whole which includes knowledge, belief, art, morals, law, custom, and any other capabilities and habits acquired by man as a member of society."

It is culture that guides the *behaviors, activities, priorities, and decisions* of an individual as well as of an organization.

Wiegers (1996) [WIE 96], in his book "*Creating a Software Engineering Culture,"* illustrates the interaction between the **software engineering culture of an organization, its software engineers, and its projects**

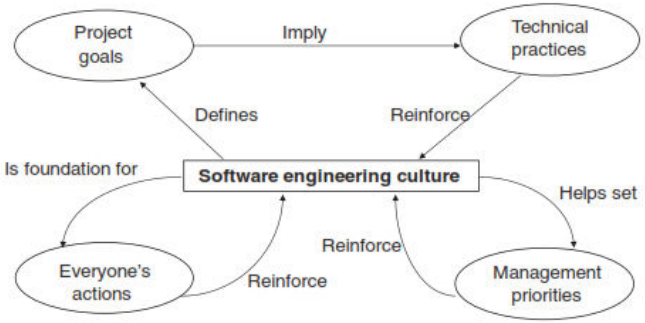


Figure 2.5 Software engineering culture.
Source: Adapted from Wiegers 1996 [WIE 96].

A healthy culture is made up of the following elements:

- The **personal commitment of each developer** to create quality products by systematically applying effective software engineering practices.
- The **commitment to the organization by managers** at all levels to provide an environment in which software quality is a fundamental factor of success and allows each developer to carry out this goal.
- The **commitment of all team members** to constantly improve the processes they use and to always work on improving the products they create.

Figure 2.6 Start coding … I'll go and see what the client wants!
Source: Reproduced with permission of CartoonStock ltd.

Figure 2.7 Dilbert is threatened and must provide an estimate on the fly. DILBERT © 2007 Scott Adams. Used By permission of UNIVERSAL UCLICK. All rights reserved.

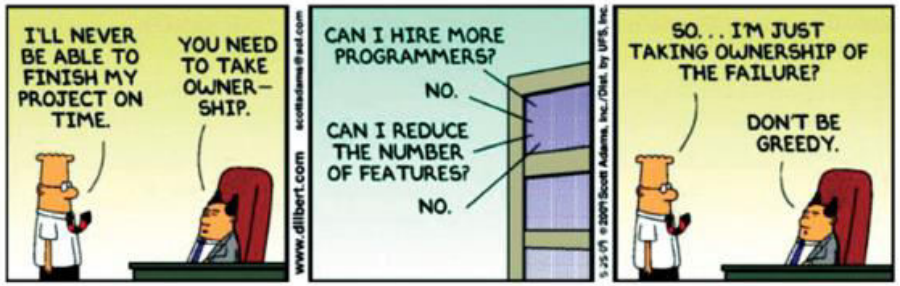


Figure 2.8 Dilbert tries to negotiate a change in his project. DILBERT © 2009 Scott Adams. Used By permission of UNIVERSAL UCLICK. All rights reserved.

Fourteen principles to follow to develop a culture that fosters quality

Table 2.3 Cultural Principles in Software Engineering [WIE 96, p. 17]

1. Never let your boss or client cause you to do poor work.
2. People must feel that their work is appreciated.
3. Continuing education is the responsibility of each team member.
4. Participation of the client is the most critical factor of software quality.
5. Your greatest challenge is to share the vision of the final product with the client.
6. Continuous improvement in your software development process is possible and essential.
7. Software development procedures can help establish a common culture of best practices.
8. Quality is the number one priority; long-term productivity is a natural consequence of quality.
9. Ensure that it is a peer, not a client, who finds the defect.
10. A key to software quality is to repeatedly go through all development steps except coding; coding should only be done once.
11. Controlling error reports and change requests is essential to quality and maintenance.
12. If you measure what you do, you can learn to do it better.
13. Do what seems reasonable; do not base yourself on dogma.
14. You cannot change everything at the same time. Identify changes that will reap the most benefits, and start to apply them as of next Monday.

THE SOFTWARE ENGINEERING CODE OF ETHICS

The first draft of the software engineering code of ethics was developed in cooperation with the Institute of Electrical and Electronics Engineers (IEEE) Computer Society and the Association for Computing Machinery (ACM)

Table 2.5 The Eight Principles of the IEEE's Software Engineering Code of Ethics [IEE 99]

Principle	Description
1. The public	Software engineers shall act consistently with the public interest.
2. Client and Employer	Software engineers shall act in a manner that is in the best interests of their client and employer, consistent with the public interest.
3. Product	Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.
4. Judgment	Software engineers shall maintain integrity and independence in their professional judgment.
5. Management	Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.
6. Profession	Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.
7. Colleagues	Software engineers shall be fair to and supportive of their colleagues.
8. Self	Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Role of SQA in software development life cycle

The Software Development Life Cycle is split into six main phases.

1. Planning
2. Design
3. Implementation
4. Testing
5. Deployment
6. Maintenance

In the planning phase of a software project, Quality assurance is responsible for making sure that the **project meets all quality requirements**, such as scope, budget, timeline, and compliance with standards.

QA can also review user requirements and analyze them to determine if they fit within the scope of the project. This helps ensure that expectations are properly set at the beginning of a project, and that resources are allocated appropriately.

QA is involved in the design phase, they can identify aspects of the design that might cause problems ,while they're still in-progress. This enables the designer or wireframes creator to make changes on the fly.

The role QA plays in the implementation stage:

- Code Reviews
- System Integration Testing
- User Acceptance Testing

The Role of QA in Testing

QA focuses on various aspects, such as *functionality, usability, reliability, performance, and compliance with industry standards*. In order to ensure that the requirements of the application are met before it is released to its users.

The Role of QA in Deployment

In the deployment stage, QA ensures that *all elements of the custom software development process are properly implemented, tested, verified, and deployed*. This ensures the product is released with confidence, free from issues or bugs.

The Role of QA in Maintenance

- Verifying software updates to confirm they are working properly
- Testing changes to make sure they are as expected
- Identifying potential problems with updates
- Following up with customers to ensure they are satisfied with changes and updates
- Documenting all issues related to releases, so that future versions can be improved upon accordingly.
- By investing in QA during maintenance, companies can be sure their products remain reliable and bug-free for customers for the long haul!

Standardizing SQA: Quality Models and Management

- Concepts conveyed by software quality models.
- Characteristics and sub-characteristics of software quality
- Software quality requirements of a software product
- Software traceability
- Standards for Quality Management
- Frameworks (ITIL, ISO, CMMI)

Requirements

The needs or requirements of these systems are typically documented either in a request for quote (RFQ) or request for proposal (RFP) document, a statement of work (SOW), a software requirements specification (SRS) document or in a system requirements document (SRD).

- Using these documents, the software developer must extract the information needed to define specifications for both the functional requirements and performance or non-functional requirements required by the client.

Functional Requirement

A requirement that specifies a function that a system or system component must be able to perform.

ISO 24765 [ISO 17a]

Non-Functional Requirement

A software requirement that describes not what the software will do but how the software will do it. Synonym: design constraint.

ISO 24765 [ISO 17a]

Performance Requirement

The measurable criterion that identifies a quality attribute of a function or how well a functional requirement must be accomplished (IEEE Std 1220™-2005). A performance requirement is always an attribute of a functional requirement.

IEEE 730 [IEE 14]

- Software quality assurance (SQA) must be able to support the practical application of these definitions.
- To achieve this, many concepts proposed by software quality models must be mastered.

Quality Model

A defined set of characteristics, and of relationships between them, which provides a framework for specifying quality requirements and evaluating quality.

ISO 25000 [ISO 14a]

Using software quality model client can:

- Define software quality characteristics that can be evaluated.
- Contrast the different perspectives of the quality model that come into play (i.e., internal - process to develop and external-fulfilling requirements perspectives).
- Carefully choose a limited number of quality characteristics that will serve as the non-functional requirements for the software (i.e., quality requirements);
- Set a measure and its objectives for each of the quality requirements.

Evaluation A systematic examination of the extent to which an entity is capable of fulfilling specified requirements.

SOFTWARE QUALITY MODELS

Unfortunately, in software organizations, software quality models are still rarely used.

A number of stakeholders have put forth the hypothesis that these models do not clearly identify all concerns for all of the stakeholders involved and are difficult to use.

Still Formally defining and evaluating the quality of software before it is delivered to the client in a need.

Five quality perspectives described by Garvin

Transcendental approach to quality:

- "Although I can't define quality, I know it when I see it."
- The main problem with this view is that quality is a personal and individual experience.
- it only takes time for all users to see it.

User-based approach: A second approach is that quality software performs as expected from the user's perspective (i.e., fitness for purpose).

Manufacturing-based approach: quality is defined as complying with specifications, is illustrated by many documents on the quality of the development process.

Product-based approach: The product-based quality perspective involves an internal view of the product. The software engineer focuses on the internal properties of the software components, for example, the quality of its architecture.

These internal properties correspond to source code characteristics and require advanced testing techniques.

Value-based approach: focuses on the *elimination of all activities that do not add value*, for example the drafting of certain documents.

In the software domain, the concept of "value" is synonymous with productivity, increased profitability, and competitiveness.

Initial Model Proposed by McCall

It proposes three perspectives for the user and primarily promotes a product-based view of the software product.

- **Operation**: during its use;
- **Revision**: during changes made to it over the years;
- **Transition**: for its conversion to other environments when the time comes to migrate to a new technology.

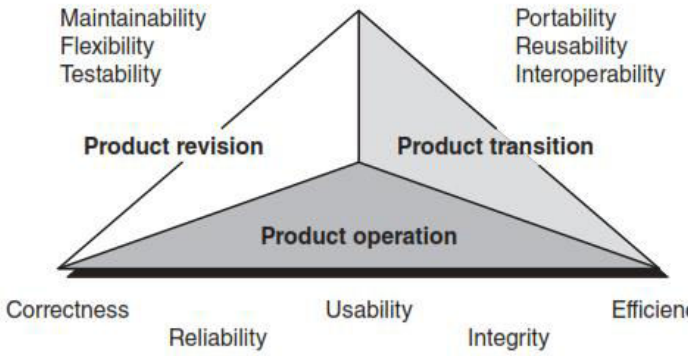


Figure 3.1 The three perspectives and 11 quality factors of McCall et al. (1977) [MCC 77].

- Each perspective is broken down into a number of quality factors.
- The model proposed by McCall and his colleagues lists 11 quality factors.
- Each quality factor can be broken down into several quality criteria (see Figure 3.2).

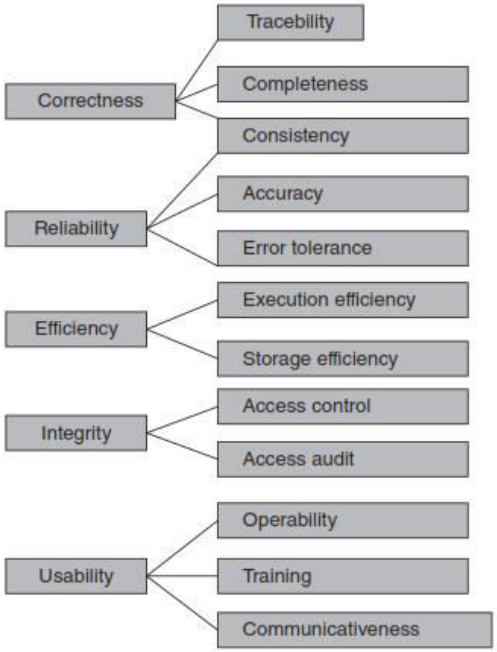


Figure 3.2 Quality factors and criteria from McCall et al. (1977) [MCC 77].

The right side of Figure 3.2 presents the measurable properties (called "quality criteria"), which can be evaluated (through observation of the software) to assess quality.

McCall proposes a subjective evaluation scale of 0 (minimum quality) to 10 (maximum quality).

The McCall quality model was primarily aimed at software product quality (i.e., the internal perspective) and did not easily tie in with the perspective of the user who is not concerned with technical details.

Example: A car owner who is not concerned with the metals or alloys used to make the engine. He expects the car to be well designed so as to minimize frequent and expensive maintenance costs.

The First Standardized Model: IEEE 1061

The IEEE 1061 standard, that is, the *Standard for a Software Quality Metrics Methodology* [IEE 98b], provides a framework for measuring software quality that allows for the establishment and identification of software quality measures based on quality requirements in order to implement, analyze, and validate software processes and products.

This standard claims to adapt to all business models, types of software, and all of the stages of the software life cycle.

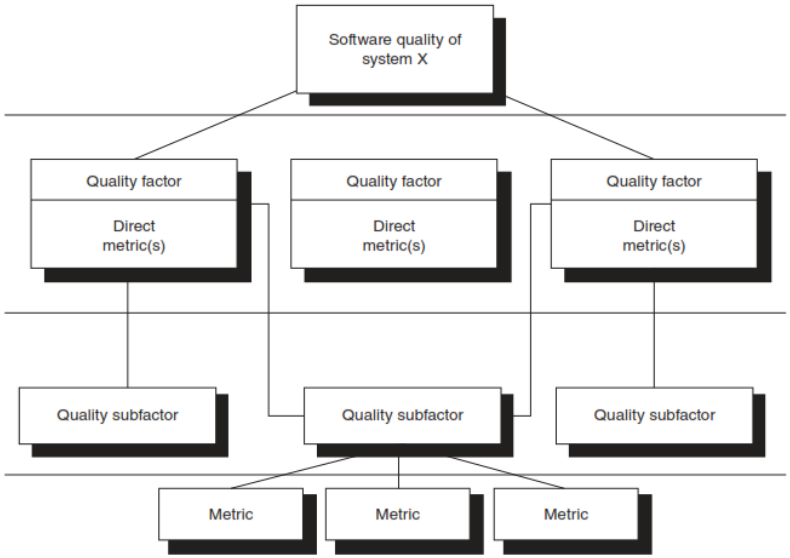


Figure 3.3 Framework for measuring software quality as per the IEEE 1061 [IEE 98b].

- At the top tier, we can see that software quality requires prior specification of a **certain number of quality attributes**, which serve to describe **the final quality desired in the software.**
- The **attributes desired by clients and users** allow for the definition of the software quality requirements.
- Quality factors suggested by this standard are assigned attributes at the next tier.

At the tier below that, and only if necessary, subfactors can then be assigned to each quality factor.

Lastly, measures are associated with each quality factor, allowing for a quantitative evaluation of the quality factor (or subfactor).

As an example, users choose availability as a quality attribute. It is defined in the requirements specifications as being the ability of a software product to maintain a specified level of service when it is used under specific conditions. The team establishes a quality factor, such as mean time between failures (or MTBF).

Users wish to specify that the software should not crash too often, since it needs to perform important activities for the organization. The measurement formula established for Factor A = hours available/(hours available + hours unavailable). It is necessary to identify target values for each directly measured factor. It is also recommended to provide an example of the calculation to clearly define the measure. For example, the work team, when preparing the system specifications, indicates that the MTBF should be 95% to be acceptable (during service hours). If the software must be made available during work hours, that is, 37.5 hours per week, it should therefore not be down for more than 2 hours a week: 37.5/(37.5 + 2) = 0.949%.

Note that if you do not set a target measure (i.e., an objective), there will be no way of determining whether the quality level for the factor was reached when implementing or accepting the software.

This model provides defined steps to use quality measures in the following situations:

Software program acquisition: In order to establish a contractual commitment regarding quality objectives for client-users and verify whether they were met by allowing their measurement when adapting and releasing the software.

Software development: In order to clarify and document quality characteristics on which designers and developers must work in order to respect the customer's quality requirements.

Quality assurance/quality control/audit: In order to enable those outside the development team to evaluate the software quality.

Maintenance: Allow the maintainer to understand the level of quality and service to maintain when making changes or upgrades to the software.

Client/user: Allow users to state quality characteristics and evaluate their presence during acceptance tests (i.e., If the software does not meet the agreed-upon specifications, corrective actions should be taken by the developer)

Steps proposed under the IEEE 1061 [IEE 98b] standard:

- Start By Identifying The List Of Non-functional (Quality) Requirements
- Everybody Involved
- List And Make Sure To Resolve Any Conflicting
- Quantify Each Quality Factor.
- Have Measures And Thresholds Approved.
- Perform A Cost–benefit Study To Identify The Costs Of Implementing The Measures For The Project.
- Implement The Measurement Method
- Analyze The Results
- Validate The Measures

Current Standardized Model: ISO 25000 Set of Standards

The Treasury Board concluded that there were basically two ways to determine the quality of a software product:

(1) assess the quality of the development process,

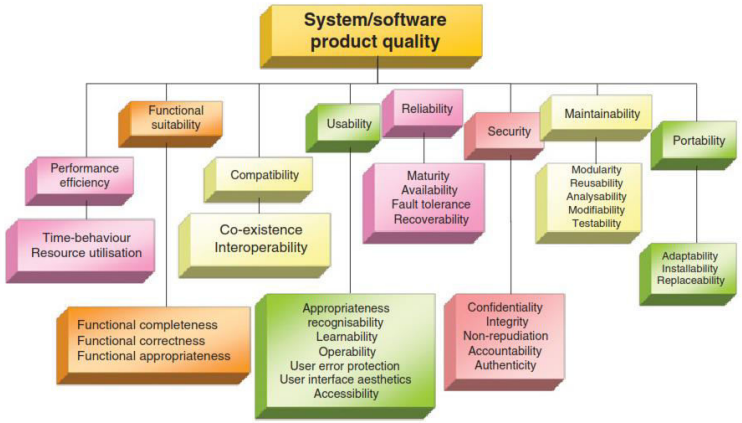
(2) assess the quality of the final product.

The ISO 25000 [ISO 14a] standard allows for the evaluation of the quality of the final software product.

The ISO 25000's series of standards recommends the following four steps [ISO 14a]:

- **Set quality requirements;**
- **Establish a quality model;**
- **Define quality measures;**
- **Conduct evaluations.**

The ISO 25010 standard identifies eight quality attributes for software



To illustrate how this standard is used, we will describe the characteristic of maintainability, which has **five sub-characteristics**: modularity, reusability, analyzability, modifiability, and testability.

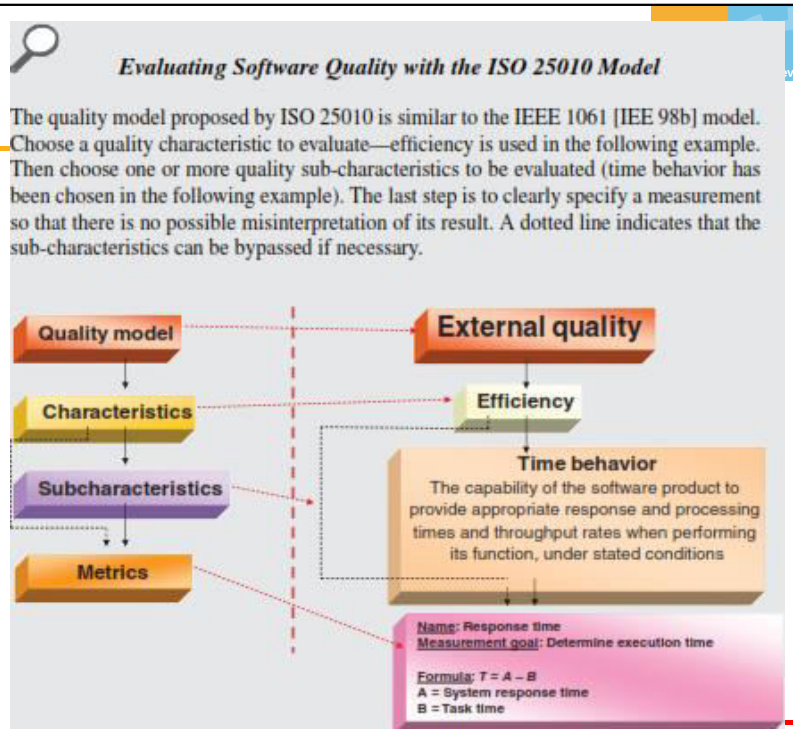
Maintainability is defined as being the level of efficiency and efficacy with which software can be modified. Changes may include software corrections, improvements, or adaptation to changes in the environment, requirements, or functional specifications.

The internal and external points of view, of the maintainability of the software.

External point of view, maintainability attempts to measure the effort required to troubleshoot, analyze, and make changes to specific software.

Internal point of view, maintainability usually involves measuring the attributes of the software that influence this change effort.

Maintainability	Degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers
• Modularity	Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components
• Reusability	Degree to which an asset can be used in more than one system, or in building other assets
• Analyzability	Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified
• Modifiability	Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality
• Testability	Degree of effectiveness and efficiency with which test criteria can be established for a system, product, or component and tests can be performed to determine whether those criteria have been met



DEFINITION OF SOFTWARE QUALITY REQUIREMENTS

Process of defining quality requirements for software (i.e., a process that supports the use of a software quality model).

In the classical engineering approach, requirements are considered to be prerequisites to the design and development stages of a product.

The requirements development phase may have been preceded by a feasibility study, or a design analysis phase for the project.

Once the stakeholders have been identified, activities for software specifications can be broken down into:

– gather: collect all wishes, expectations, and needs of the stakeholders;

– prioritize: debate the relative importance of requirements based on, for example, two priorities (essential, desirable);

– analyze: check for consistency and completeness of requirements;

– describe: write the requirements in a way that can be easily understood by users and developers;

– specify: transform the business requirements into software specifications (data sources, values and timing, business rules).

Requirements are generally grouped into three categories:

1) Functional Requirements: These describe the characteristics of a system or processes that the system must execute. This category includes business requirements and functional requirements for the user.
2) Non-Functional (Quality) Requirements: These describe the properties that the system must have, for example, requirements translated into quality characteristics and sub-characteristics such as security, confidentiality, integrity, availability, performance, and accessibility.
3) Constraints: Limitations in development such as infrastructure on which the system must run or the programming language that must be used to implement the system.

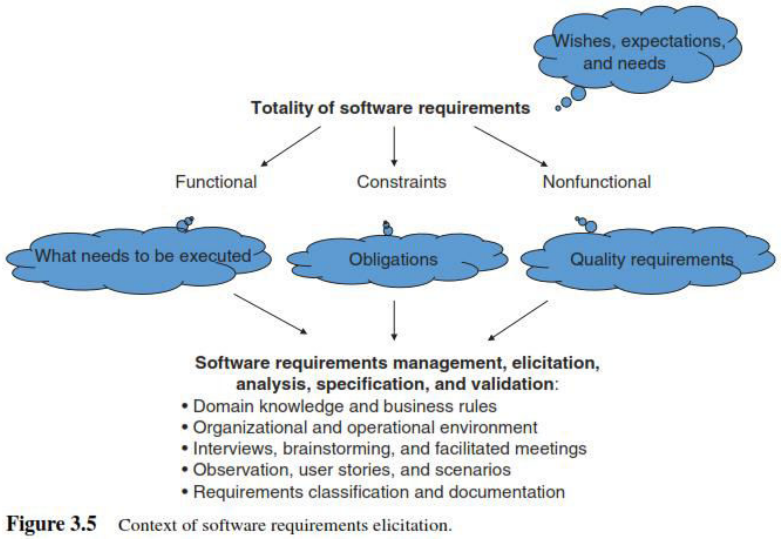
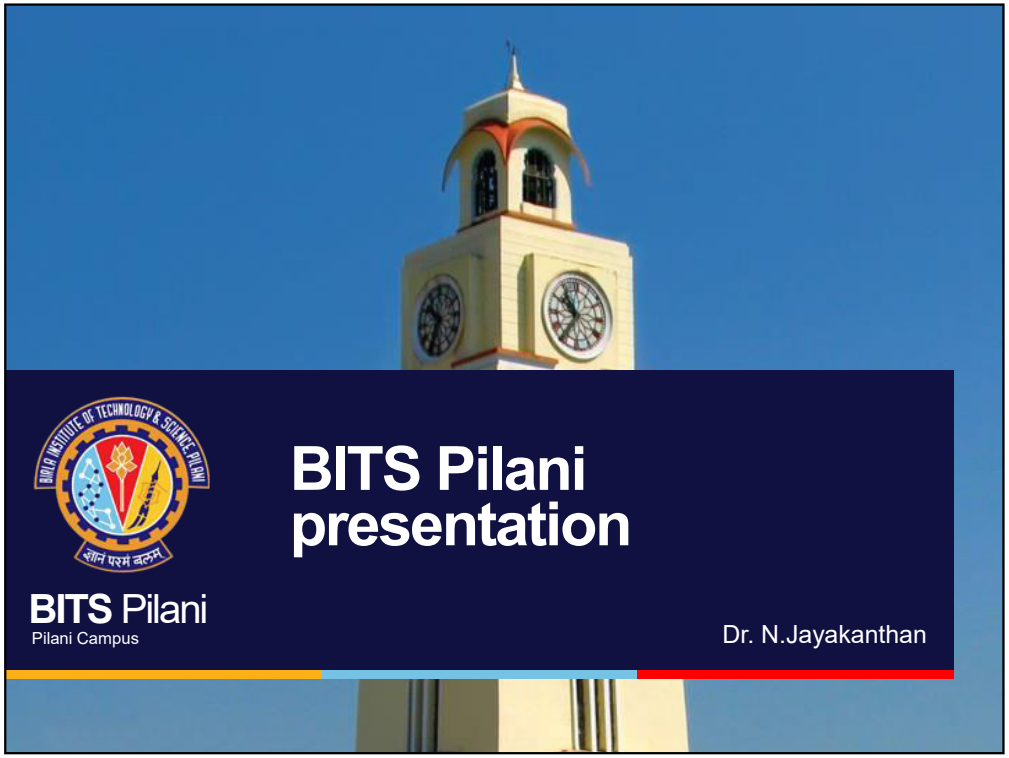


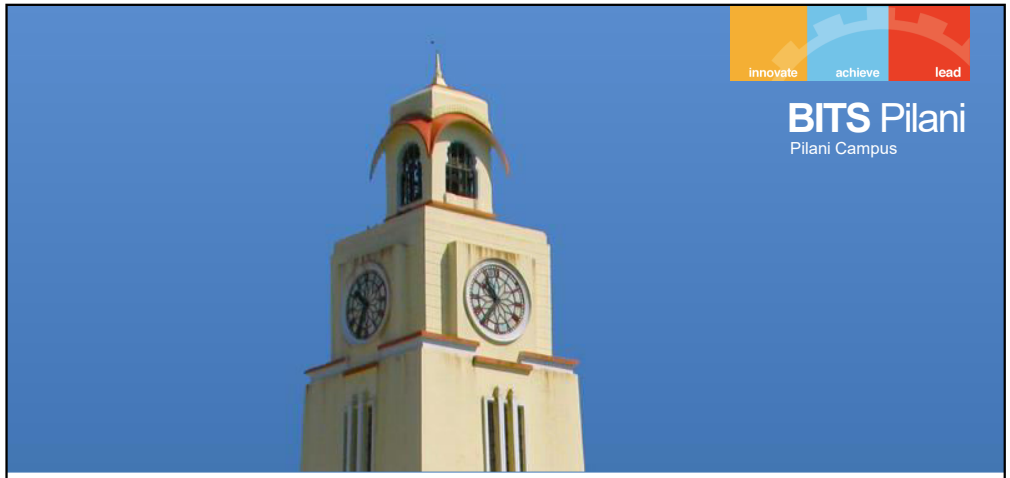
Figure 3.5 Context of software requirements elicitation.

Characteristics to measure quality of a requirement

- Necessary: They must be based on necessary elements
- Unambiguous: clear enough to be interpreted in only one way.
- Concise: They must be stated in a language that is precise, brief, and easy to read.
- Coherent: They must not contradict the requirements.
- Complete: They must all be stated fully
- Accessible: They must be realistic regarding their implementation (time ,budget ,resource)
- Verifiable: inspection, analysis, demonstration, or tests.

THANK YOU





innovate achieve lead

BITS Pilani
Pilani Campus

SE ZG501
Software Quality Assurance and Testing
Lecture No. 3

Characteristics to measure quality of a requirement

innovate achieve lead

- Necessary: They must be based on necessary elements
- Unambiguous: clear enough to be interpreted in only one way.
- Concise: They must be stated in a language that is precise, brief, and easy to read.
- Coherent: They must not contradict the requirements.
- Complete: They must all be stated fully
- Accessible: They must be realistic regarding their implementation (time ,budget ,resource)
- Verifiable: inspection, analysis, demonstration, or tests.

Specifying Quality Requirements: The Process

innovate achieve lead

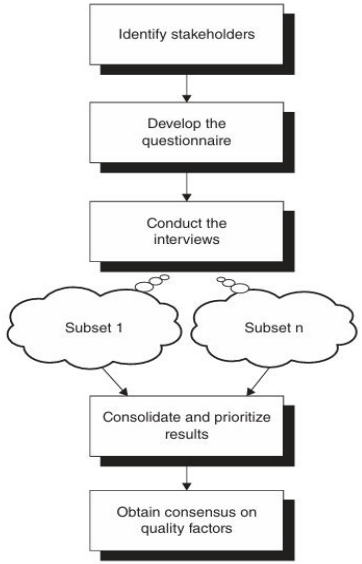


Figure 3.6 Steps suggested for defining non-functional requirements.

innovate achieve lead

- Stakeholders are **any person or organization that has a legitimate interest in the quality of the software**.

  These needs and hopes may **change** during the system life cycle and must be checked when there is any change.

- Developing the **questionnaire** that presents the external quality characteristics in terms that are easy to understand for managers.

Table 3.3 Example of Quality Criteria Documentation

Quality characteristics	Importance
Reliability	Indispensable
User-friendliness	Desirable
Operational safety	Non-applicable

Next, for each characteristic, the quality measure must be described in detail and include the following information

- quality characteristic;
- quality sub-characteristic;
- measure (i.e., formula);
- objectives (i.e., target);
- example.

The last step in defining quality requirements involves having these requirements authorized through consensus.

Evaluation of the Functional Capacity of Software

Users often choose this characteristic. It is defined in the specifications as the ability of a software product to carry out all specified requirements. The *ability* sub-characteristic is chosen by the team and described as the percentage of requirements described in the specification document that must be delivered (%*E*). The measure established is

$$\%E = (\text{Number of functionalities requested}/(\text{Number of functionalities delivered})) \times 100.$$

It is necessary to identify target values as an objective for each measure. It is also recommended to provide an example of the calculations (i.e., measurement) to clearly illustrate the measure. For example, during the writing of the specification, the project team indicates that the %*E* should be 100% of the requirements described in the specifications document and that they are functional and delivered, without defects, before the final acceptance of the software for production.

Alternatively, with a lack of measurable objectives, the general policy is to accept the most stable version of the code having the necessary functionality.

ISO 25010 [ISO 11i]

REQUIREMENT TRACEABILITY DURING THE SOFTWARE LIFE CYCLE

Throughout the life cycle, client needs are documented and developed in different documents, such as **specifications, architecture, code, and user manuals**.

Throughout the life cycle of a system, many changes regarding client needs should be expected.

Every time a need changes, it must be ensured that all documents are updated.

Traceability is a technique that helps us follow the development of needs as well as their changes.

Software Engineering Standards

Other engineering domains such as mechanical, chemical, electrical, or physics engineering are based on the laws of nature as discovered by scientists.

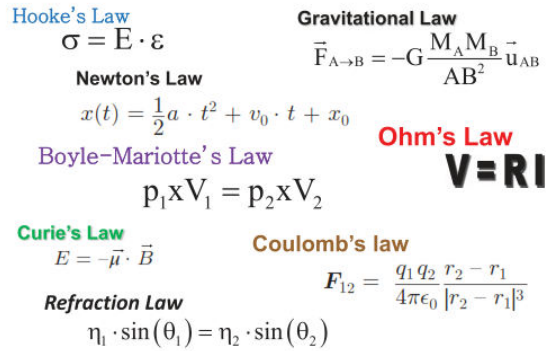


Figure 4.1 A few laws of nature used by some engineering disciplines.

Unfortunately, software engineering, unlike other engineering disciplines, is not based on the laws of nature.

Software engineering, like other disciplines, is based on the use of well-defined practices for ensuring the quality of its products.

In software engineering, there are several standards, which are actually guides for management practices.

A rigorous process serves as the framework for developing and approving standards, including international ISO standards and those from professional organizations like IEEE.

Standard

A set of mandatory requirements established by consensus and maintained by a recognized body to prescribe a disciplined and uniform approach, or to specify a product, with respect to mandatory conventions and practices.

The four principles for the development of ISO standards are:

- **ISO standards meet a market need.**
- **ISO standards are based on worldwide expertise.**
- **ISO standards are the result of a multi-stakeholder process.**
- **ISO standards are based on consensus.**

The ISO standards are developed by consensus

- That all parties were able to express their views.
- The best effort has been made to take into account all opinions and solve all problems (i.e., all the submissions in a vote of the draft of a standard).

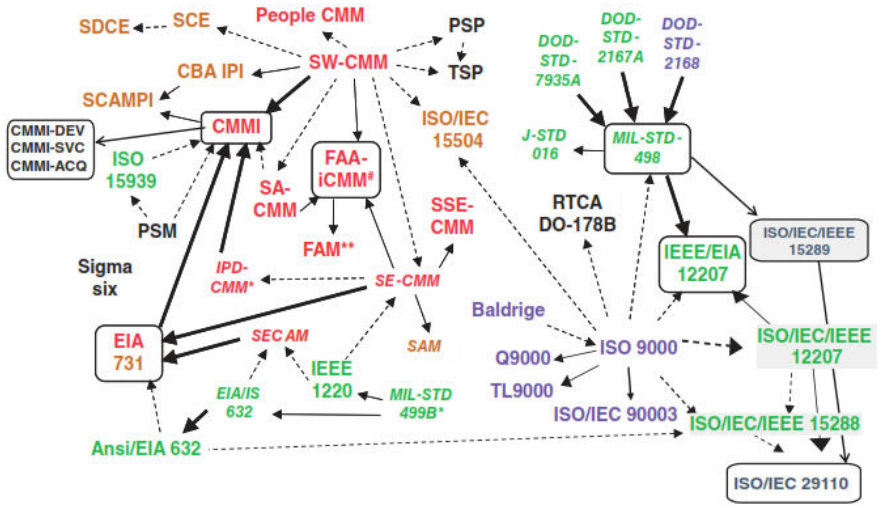


Figure 4.2 The development of standards and models.

- American Department of Defence (DoD) created the "DoD-STD1679A" military standard
- IEEE, the International Organization for Standardization (ISO) European Space Agency (ESA) developed standards
- "*Capability Maturity Model"* (CMM®): developed at the request of the American DoD, by the Software Engineering Institute (SEI) in order to provide a road map of engineering practices to improve the performance of the development, maintenance and service provisioning processes.

Figure 4.3 illustrates the evolution of standards that are maintained and published under the responsibility of the appointed subcommittee for standardized processes, tools, and supporting technologies for software engineering and systems:

The Continuous Evolution of Standards

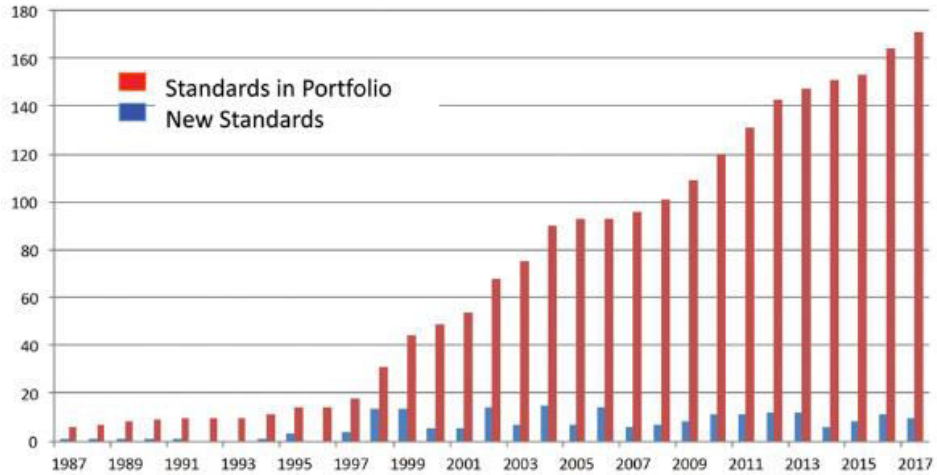


Figure 4.3 The evolution of standards SC7 [SUR 17].

MAIN STANDARDS FOR QUALITY MANAGEMENT

Standards related to the management of software quality:

ISO 9000 [ISO 15b] and ISO 9001 [ISO 15].

The application guide for software, The ISO/IEC 90003 standard.

Standards in the ISO 9000 family include:

- ISO 9001:2015 - sets out the requirements of a quality management system
- ISO 9000:2015 - covers the basic concepts and language
- ISO 9004:2009 - focuses on how to make a quality management system more efficient and effective
- ISO 19011:2011 - sets out guidance on internal and external audits of quality management systems.

- The ISO 9001 standard provides the **basic concepts, principles and vocabulary** of quality management systems (QMS) and is the basis for other standards for QMSs [ISO 15].
- The **"Quality Management Principles"** (QMP) are a set of values, rules, standards, and fundamental convictions regarded as fair and that could be the basis for quality management.

The seven QMP of the ISO 9001

- **Principle 1: Customer focus**
- **Principle 2: Leadership**
- **Principle 3: Involvement of people**
- **Principle 4: Process approach**
- **Principle 5: System approach to management**
- **Principle 6: Factual approach to decision making**
- **Principle 7: Mutually beneficial supplier relationships**

ISO 9001 uses the process approach, the Plan-Do-Check-Act (PDCA) approach, and a risk-based thinking approach [ISO 15].

- The **process approach** allows an organization to plan its processes and their interactions.
- The PDCA cycle allows an organization to ensure that its processes are adequately resourced and appropriately managed and that opportunities for improvement are identified and implemented.
- The risk-based thinking approach allows an organization to determine the factors that may cause deviation from its processes and its QMS in relation to expected results, to implement preventive measures in order to limit negative effects and exploit opportunities when they arise.

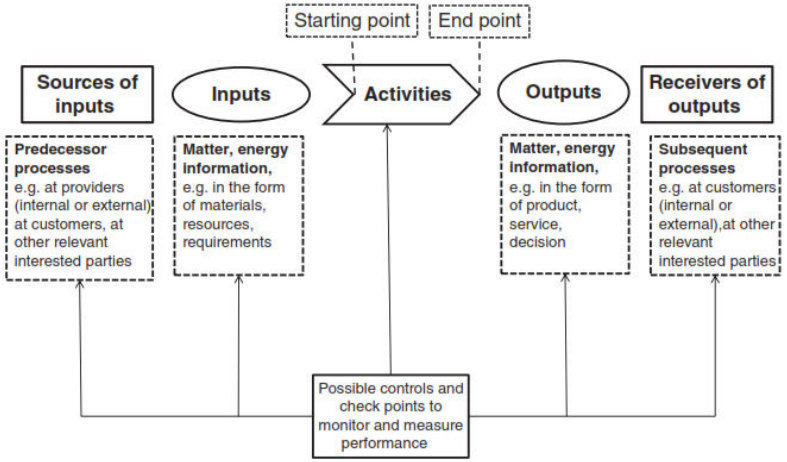


Figure 4.4 Elements of a process [ISO 15].

ISO 9001 describes the elements of the PDCA cycle as follows :

– Plan: establish the objectives of the system, processes and resources to deliver results in accordance with customer requirements and policies of the organization, identify and address risks and opportunities;

– Do: implement what has been planned;

– Check: monitor and measure (if applicable) processes and the products and services obtained against policies, objectives, requirements and planned activities, and report the results;

– Act: take actions to improve performance, as needed.

ISO/IEC 90003 Standard

International Electrotechnical Commission.

- Provides guidelines for the application of the ISO 9001 standard to computer software.
- It provides organizations with instructions for acquiring, supplying, developing, using and maintaining software.
- Explains what a software audit is for the organization wishing to set up a QMS as well as for the QMS auditor.

ISO/IEC/IEEE 12207 STANDARD

Establishes a common framework for software life cycle processes.

It applies to the acquisition of systems and software products and services, development, supply, operation, maintenance, and disposal of software products and the development of the software part of a system whether performed internally or externally to an organization

ISO 12207 [ISO 17] defines four sets of processes as shown in Figure 4.5:

– **Two agreement processes between a customer and a supplier;**

– **Six organizational project-enabling processes;**

– **Eight processes for technology management;**

– **Fourteen technical processes.**

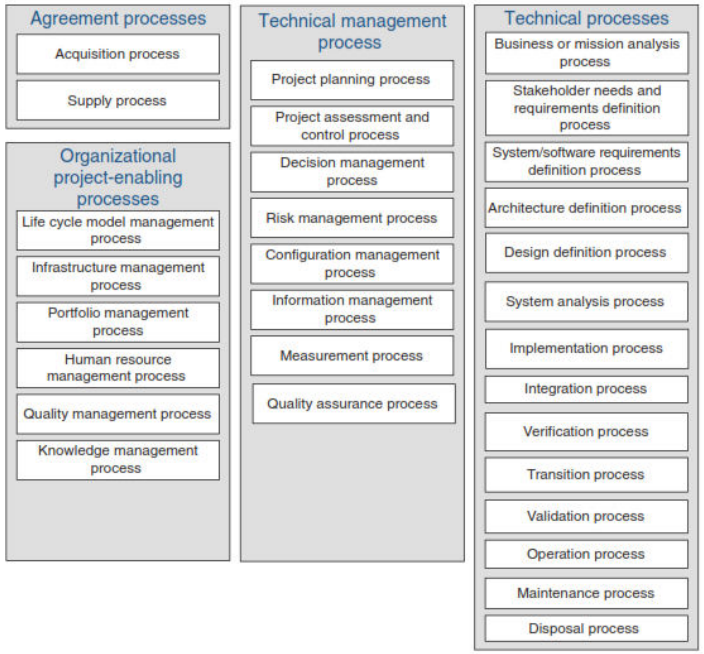


Figure 4.5 The four life cycle process groups of ISO 12207 [ISO 17].

The ISO 12207 standard can be used in one or more of the following modes

For an organization: It aids in establishing a desired process environment, supported by appropriate methods, procedures, techniques, tools, and trained personnel.

For a project: It assists in selecting, structuring, and employing elements from established life cycle processes to deliver products and services effectively.

For an acquirer and a supplier: It facilitates the development of agreements concerning processes and activities, ensuring clarity and mutual understanding.

For organizations and assessors It serves as a process reference model for conducting process assessments, which can support organizational process improvement initiatives.

IEEE 730 STANDARD FOR SQA PROCESSES

QA according to IEEE is a set of proactive measures to ensure the quality of the software product.

The IEEE 730 provides guidance for the SQA activities of products or of services.

The SQA process of the IEEE 730 is grouped into three activities: the implementation of the SQA process, product assurance, and process assurance.

Activities consist of a set of tasks.

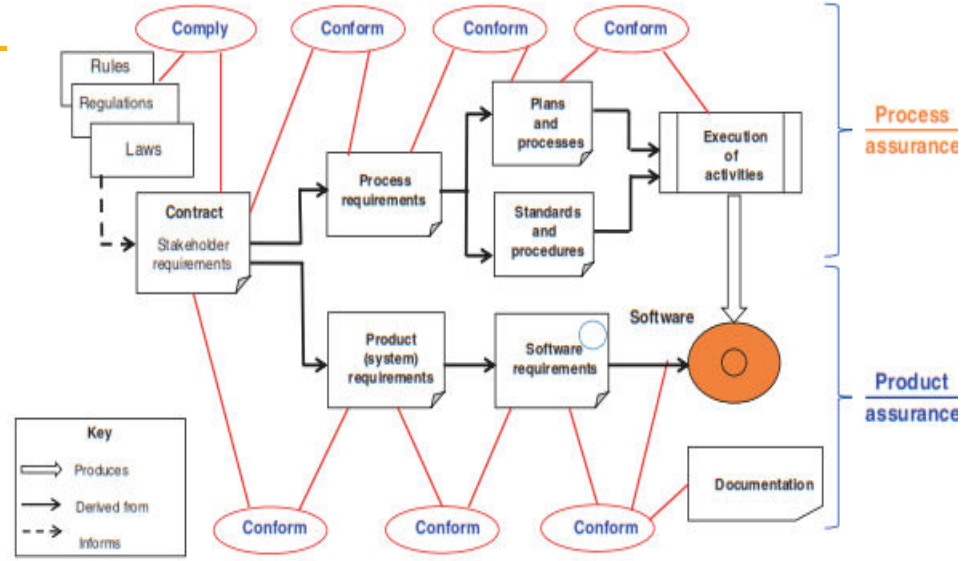


Figure 4.6 The links between requirements and the artifacts of a project [IEE 14].

IEEE 730 [IEE 14] describes what must be done by a project;

- it assumes that the organization has already implemented SQA processes before the start of a project.
- The standard includes a clause that describes what is meant by compliance.

Product Assurance Activities

- Evaluate plans for compliance to contracts, standards, and regulations;
- Evaluate product for compliance to established requirements;
- Evaluate product for acceptability;
- Evaluate the compliance of product support;
- Measure products.

Process Assurance Activities

- **Evaluate compliance of the processes and plans;**
- **Evaluate environments for compliance;**
- **Evaluate subcontractor processes for compliance;**
- **Measure processes;**
- **Assess the skill and knowledge of personnel.**

Capability Maturity Models (CMM®).

- Tool used to improve and refine software development processes.
- Framework that is used to analyze the approach and techniques followed by any organization to develop software products.
- It also provides guidelines to enhance further the maturity of the process used to develop those software products.

The Capability Maturity Model Integration (CMMI)

- An advanced framework designed to improve and integrate processes across various disciplines such as **software engineering, systems engineering, and people management**.
- Helps organizations fulfill customer needs, create value for investors, and improve product quality and market growth.

The CMMI model was developed as two versions:

Initial staged version and continuous version, which is the first CMM model for systems engineering

CMMI-DEV The objective of this model is to encourage organizations to check and continuously improve their development project process and evaluate their level of maturity on a five-level scale as proposed by the staged CMMI model.

The **CMMI for Development (CMMI-DEV)** covers a broader area than its predecessor by adding other practices, such as systems engineering, and the development of integrated processes and products.

The objective of this model is to encourage organizations to check and continuously improve their development project process and evaluate their level of maturity on a five-level scale

Two other CMMI models were developed based on architecture, CMMI for Services (CMMI-SVC) and the CMMI for Acquisition (CMMIACQ)

The CMMI-SVC model provides guidelines for organizations that provide services either internally or externally.

The CMMI-ACQ model provides guidelines for organizations that purchase products or services.

All three CMMI models use 16 common process areas.

- For each level of maturity, a set of process areas are defined.
- Each area encompasses a set of requirements that must be met.
- These requirements define which elements must be produced rather than *how they are produced*

Thereby allowing the organization implementing the process to choose its own life cycle model, its design methodologies, its development tools, its programming languages, and its documentation standard.

This approach enables a wide range of companies to implement this model while having processes that are compatible with other standards.

Maturity levels and process areas for each maturity level in the CMMI-DEV model.

Maturity Level 1: Initial

Processes are usually ad hoc and chaotic.

Maturity level 1 organizations are characterized by a tendency to overcommit, abandon their processes in a time of crisis, and be unable to repeat their successes.

Maturity Level 2: Managed

When these practices are in place, projects are performed and managed according to their documented plans.

Process areas:

- Requirements management
- Project planning
- Project monitoring and control
- Supplier agreement management
- Measurement and analysis
- Process and product quality assurance
- Configuration management

Maturity Level 3: Defined

Processes are well characterized and understood, and are described in standards, procedures. tools. and methods.

Process areas:

- Requirements development
- Technical solution
- Product integration
- Verification
- Validation
- Organizational process focus
- Organizational process definition
- Organizational training integrated project management
- Risk management
- Decision analysis and resolution

Maturity Level 4: Quantitatively managed

The organization and projects establish quantitative objectives for quality and process performance and use them as criteria in managing projects.

Process areas:

- Organizational process performance
- Quantitative project management

Maturity Level 5: Optimizing

An organization continually improves its processes based on a **quantitative understanding of its business objectives and performance needs**.

Process areas

- Organizational performance management
- Causal analysis and resolution

CMMI model structure.

Each process area has generic and specific goals, practices, and sub-practices.

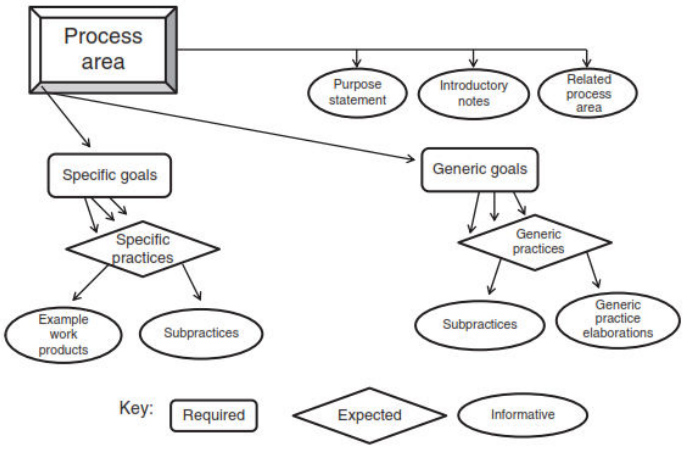


Figure 4.8 Structure of the staged representation of the CMMI [SEI 10a].

Level	Focus	Key process area	Quality productivity
5 Optimizing	*Continuous process improvement*	Organizational performance management causal analysis and resolution	
4 **Quantitatively managed**	*Quantitative management*	Organizational process performance quantitative project management	
3 **Defined**	*Process standardization*	Requirements development Technical solution Product integration Verification Validation Organizational process focus Organizational process definition Organizational training Integrated project management Risk management Decision analysis and resolution	
2 Managed	*Basic project management*	Requirement management Project planning Project monitoring and control Supplier agreement management Measurement and analysis Process and product quality assurance Configuration management	Risk rework
1 Initial			

Figure 4.9 The staged representation of the CMMI® for Development model.

ITIL Framework

The ITIL framework was created in Great Britain based on good management practices for computer services.

It consists of a set of five books providing advice and recommendations in order to offer quality service to IT service users.

IT services are typically responsible for ensuring that the infrastructures are effective and running (backup copies, recovery, computer administration, telecommunications, and production data)

– strategy;
– design;
– transition;
– operation;
– continuous improvement.

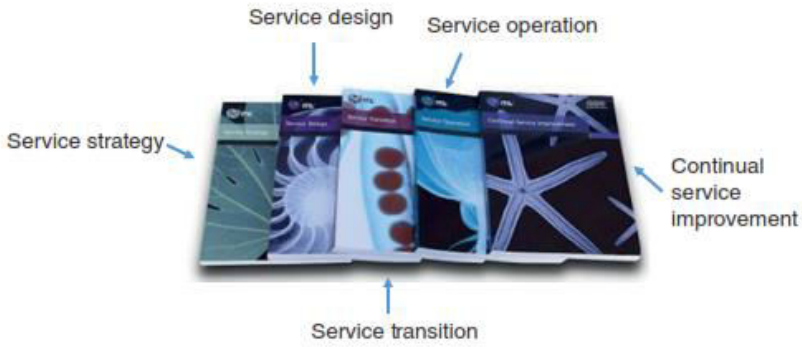


Figure 4.11 The main ITIL guides.

The ITIL framework offers guidance and best practices for managing the five stages of the IT service lifecycle: **service strategy, service design, service transition, service operation and continual service improvement.**

Support processes described in the ITIL are focused on daily operations. Their main goals are to resolve the problems when they arise or to prevent them from happening when there is a change in the computer environment or in the way the organization does things.

Support center function

– Incident management
– Problem management
– Configuration management
– Change management
– Commissioning management

Five processes for service operation:

– Service level management
– Financial management of IT services
– Capacity management
– IT service continuity management
– Availability management

Given the major recognition of ITIL worldwide, an international standard based on ITIL came into being: ISO/IEC 20000-1.

The principles of ITIL were successfully conveyed to many companies of all sizes and from all sectors of activity

The main subjects handled under ITIL

– User support, which includes the management of incidents and is an extension of the concept of a Helpdesk;

– Provision of services which involves managing processes that are dedicated to the daily operations of IT (cost control, management of service levels);

– Management of the production environment infrastructure which involves implementing the means for network management and production tools (scheduling, backup, and monitoring);

– Application management which consists of managing the support of an operational program;

– Security management (confidentiality, data integrity, data availability, etc.) of the SI (security process)

THANK YOU

Staged Representation of CMMI Process Model

Process Area: Requirements Management

- **Purpose Statement**: Ensure that requirements are understood and managed throughout the project lifecycle.
- **Introductory Notes**: Requirements must align with project goals and stakeholder needs.
- **Related Process Areas**: Configuration Management, Project Planning.

Specific Goals:

1. **Understand Requirements**: Clearly define and document stakeholder expectations.
   - **Specific Practices**:
     - Identify all functional and non-functional requirements.
     - Validate requirements with stakeholders.
   - **Example Work Products**: Requirements document, stakeholder approval forms.
   - **Sub practices**: Conduct requirement reviews, update documents after stakeholder feedback.

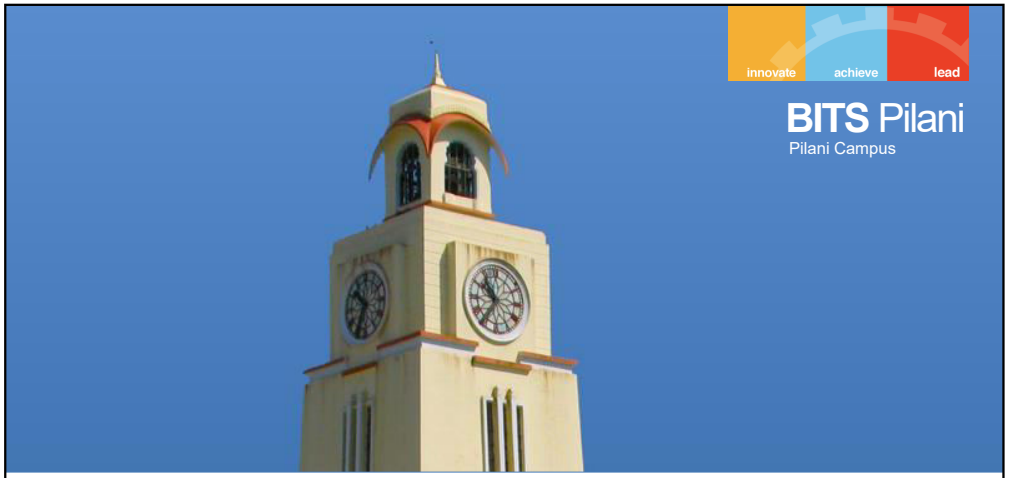
Generic Goals:

1. **Establish and Institutionalize Requirements Management**:
   - **Generic Practices**:
     - Assign roles for managing requirements.
     - Train team members on requirements tools and processes.
   - **Generic Practice Elaborations**:
     - Use tools like JIRA or IBM DOORS for requirements tracking.
     - Perform regular audits to ensure adherence.
   - Sub practises
     - Conduct brainstorming sessions with stakeholders to gather comprehensive inputs.
     - Categorize requirements as functional (e.g., features) or non-functional (e.g., performance, security).

BITS Pilani
Pilani Campus

BITS Pilani presentation

Dr. Nagesh BS



innovate achieve lead

BITS Pilani
Pilani Campus

SE ZG501
Software Quality Assurance and Testing
Lecture No. 4

The need for a comprehensive definition of requirements

innovate achieve lead

To Cover all attributes of software and aspects of the use of software, including usability aspects, reusability aspects, maintainability aspects, and so forth in order to assure the full satisfaction of the users.

The great variety of issues related to the various attributes of software and its use and maintenance, as defined in software requirements documents, can be classified into content groups called *quality factors.*

Classifications of software requirements into software quality factors

innovate achieve lead

The classic model of software quality factors, suggested by McCall, consists of 11 factors.

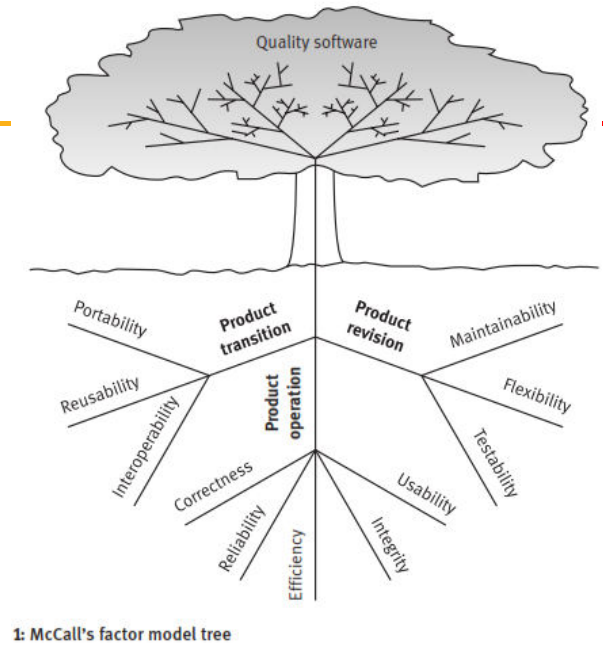
Subsequent models, consisting of 12 to 15 factors

The McCall factor model, despite the quarter of a century of its “maturation”, continues to provide a practical, up-to-date method for classifying software requirements

McCall's factor model

Classifies all software requirements into 11 software quality factors.

- Product operation factors: Correctness, Reliability, Efficiency, Integrity, Usability.
- Product revision factors: Maintainability, Flexibility, Testability.
- Product transition factors: Portability, Reusability, Interoperability.



1: McCall's factor model tree

Product operation software quality factors

Correctness : Correctness requirements are defined in a list of the software system's **required outputs**.

For example:

- Displaying a customer's balance in the sales accounting information system.
- Regulating air supply as a function of temperature, as specified by the firmware of an industrial control unit

Output specifications are usually multidimensional

- The output mission (e.g., sales invoice printout, and red alarms when temperature rises above 250°F).
- The required accuracy of those outputs that can be adversely affected by inaccurate data or inaccurate calculations.
- The completeness of the output information, which can be adversely affected by incomplete data.
- The up-to-dateness of the information (defined as the time between the event and its consideration by the software system).
- The availability of the information (the reaction time, defined as the time needed to obtain the requested information or as the requested reaction time of the firmware installed in a computerized apparatus).
- The standards for coding and documenting the software system.

Example

The correctness requirements of a club membership information system consisted of the following:

- The output mission: A defined list of 11 types of reports, four types of standard letters to members and eight types of queries, which were to be displayed on the monitor on request.
- The required accuracy of the outputs: The probability for a non-accurate output, containing one or more mistakes, will not exceed 1%.
- The completeness of the output information: The probability of missing data about a member, his attendance at club events, and his payments will not exceed 1%.
- The up-to-dateness of the information: Not more than two working days for information about participation in events and not more than one working day for information about entry of member payments and personal data.
- The availability of information: Reaction time for queries will be less than two seconds on average; the reaction time for reports will be less than four hours.
- The required standards and guidelines: The software and its documentation are required to comply with the client's guidelines.

Reliability

Reliability requirements deal with failures to provide service.

They determine the **maximum allowed software system failure rate**, and can refer to the entire system or to one or more of its separate functions.

Example

(1) The failure frequency of a heart-monitoring unit that will operate in a hospital's intensive care ward is required to be less than one in 20 years. Its heart attack detection function is required to have a failure rate of less than one per million cases.

(2) One requirement of the new software system to be installed in the main branch of Independence Bank, which operates 120 branches, is that it will not fail, on average, more than 10 minutes per month during the bank's office hours. In addition, the probability that the off-time (the time needed for repair and recovery of all the bank's services) be more than 30 minutes is required to be less than 0.5%.

Efficiency

Efficiency requirements deal with the hardware resources needed to perform all the functions of the software system in conformance to all other requirements.

computer's processing capabilities, data storage capability, data communication capability of the communication lines

Another type of efficiency requirement deals with the time between recharging of the system's portable units, such as, information systems units located in portable computers, or meteorological units placed outdoors.

Examples

(1) A chain of stores is considering two alternative bids for a software system. Both bids consist of placing the same computers in the chain's headquarters and its branches. The bids differ solely in the storage volume: 20 GB per branch computer and 100 GB in the head office computer (Bid A); 10 GB per branch computer and 30 GB in the head office computer (Bid B). There is also a difference in the number of communication lines required: Bid A consists of three communication lines of 28.8 KBPS between each branch and the head office, whereas Bid B is based on two communication lines of the same capacity between each branch and the head office. In this case, it is clear that Bid B is more efficient than Bid A because fewer hardware resources are required.

Integrity

Integrity requirements deal with the software system security, that is, requirements to **prevent** access to unauthorized persons, to distinguish between the majority of personnel allowed to see the information (“**read permit**”) and a limited group who will be allowed to add and change data **(“write permit”),** and so forth.

Example

The Engineering Department of a local municipality operates a GIS (Geographic Information System). The Department is planning to allow citizens access to its GIS files through the Internet. The software requirements include the possibility of viewing and copying but not inserting changes in the maps of their assets as well as any other asset in the municipality's area (“read only” permit). Access will be denied to plans in progress and to those maps defined by the Department's head as limited access documents.

Usability

Usability requirements deal with the scope of staff resources needed to train a new employee and to operate the software system.

Example

The software usability requirements document for the new help desk system initiated by a home appliance service company lists the following specifications:

(a) A staff member should be able to handle at least 60 service calls a day.

(b) Training a new employee will take no more than two days (16 training hours), immediately at the end of which the trainee will be able to handle 45 service calls a day.

Product revision software quality factors

Maintainability

Maintainability requirements determine the *efforts that will be needed by users and maintenance personnel to identify the reasons for software failures, to correct the failures, and to verify the success of the corrections*.

This factor's requirements refer to the modular structure of software, the internal program documentation, and the programmer's manual, among other items.

Example

Typical maintainability requirements:

(a) The size of a software module will not exceed 30 statements.

(b) The programming will adhere to the company coding standards and guidelines.

Flexibility

Flexibility in software means how easily it can be updated or adjusted with minimal effort. It includes adapting the software for different customers, scales of operation, or product ranges in the same industry with minimal resources like time or effort.

Example

TSS (teacher support software) deals with the documentation of pupil achievements, the calculation of final grades, the printing of term grade documents, and the automatic printing of warning letters to parents of failing pupils. The software specifications included the following flexibility requirements:

(a) The software should be suitable for teachers of all subjects and all school levels (elementary, junior and high schools).

(b) Non-professionals should be able to create new types of reports according to the schoolteacher's requirements and/or the city's education department demands.

Testability

Testability refers to how easily an information system can be tested during development and operation. Testability requirements focus on features that make testing easier, such as:

- Predefined intermediate results to verify specific parts of the system.
- Log files to track system behavior and identify issues.

These features help testers quickly find and fix problems.

Example

An industrial computerized control unit is programmed to calculate various measures of production status, report the performance level of the machinery, and operate a warning signal in predefined situations.

One testability requirement demanded was to develop a set of standard test data with known system expected correct reactions in each stage. This standard test data is to be run every morning, before production begins, to check whether the computerized unit reacts properly.

Product transition software quality factors

Portability

Portability requirements tend to the adaptation of a software system to other environments consisting of different hardware, different operating systems, and so forth.

Example

A software package designed and programmed to operate in a Windows 2000 environment is required to allow low-cost transfer to Linux and Windows NT environments.

Reusability

Reusability requirements deal with the use of software modules originally designed for one project in a new software project currently being developed.

They may also enable future projects to make use of a given module or a group of modules of the currently developed software.

The reuse of software is expected to save development resources, shorten the development period, and provide higher quality modules.

These benefits of higher quality are based on the assumption that most of the software faults have already been detected by the quality assurance activities performed on the original software, by users of the original software, and during its earlier reuses.

Example

A software development unit has been required to develop a software system for the operation and control of a hotel swimming pool that serves hotel guests and members of a pool club. Although the management did not define any reusability requirements, the unit's team leader, after analyzing the information processing requirements of the hotel's spa, decided to add the reusability requirement that some of the software modules for the pool should be designed and programmed in a way that will allow its reuse in the spa's future software system, which is planned to be developed next year.

These modules will allow:

- Entrance validity checks of membership cards and visit recording.
- Restaurant billing.
- Processing of membership renewal letters.

Interoperability

Interoperability requirements focus on creating interfaces with other software systems or with other equipment firmware.

Interoperability requirements can specify the name(s) of the software or firmware for which interface is required.

Example

The firmware of a medical laboratory's equipment is required to process its results (output) according to a standard data structure that can then serve as input for a number of standard laboratory information systems.

Alternative models of software quality factors

- The Evans and Marciniak factor model (Evans and Marciniak, 1987).
- The Deutsch and Willis factor model (Deutsch and Willis, 1988).

Comparison of the alternative models

Both alternative models exclude only one of McCall's 11 factors, namely the testability factor.

- The Evans and Marciniak factor model consists of 12 factors that are classified into three categories.
- The Deutsch and Willis factor model consists of 15 factors that are classified into four categories.

Taken together, five new factors were suggested by the two alternative factor models.

- Verifiability (by both models)
- Expandability (by both models)
- Safety (by Deutsch and Willis)
- Manageability (by Deutsch and Willis)
- Survivability (by Deutsch and Willis).

Table 3.1: Comparison of McCall's factor model and alternative models

No.	Software quality factor	McCall's classic model	Alternative factor models: Evans and Marciniak	Alternative factor models: Deutsch and Willis
1	Correctness	+	+	+
2	Reliability	+	+	+
3	Efficiency	+	+	+
4	Integrity	+	+	+
5	Usability	+	+	+
6	Maintainability	+	+	+
7	Flexibility	+	+	+
8	Testability	+		
9	Portability	+	+	+
10	Reusability	+	+	+
11	Interoperability	+	+	+
12	Verifiability		+	+
13	Expandability		+	+
14	Safety			+
15	Manageability			+
16	Survivability			+

Software testing (or "testing") was the first software quality assurance tool applied to control the software product's quality before its shipment or installation at the customer's premises.

SQA professionals were encouraged to extend testing to the partial in-process products of coding, which led to software module (unit) testing and integration testing.

Definition

"Testing is the process of executing a program with intention of finding errors."

Software testing is a formal process carried out by a specialized testing team in which a software unit, several integrated software units or an entire software package are examined by running the programs on a computer.

All the associated tests are performed according to approved test procedures on approved test cases.

Formal – Software test plans are part of the project's development and quality plans, scheduled in advance and often a central item in the development agreement signed between the customer and the developer.

Specialized testing team – An independent team or external consultants who specialize in testing are assigned to perform these tasks mainly in order to eliminate bias and to guarantee effective testing by trained professionals.

Running the programs – Any form of quality assurance activity that does not involve running the software, for example code inspection, cannot be considered as a test.

Approved test procedures – The testing process performed according to a test plan and testing procedures that have been approved as conforming to the SQA procedures adopted by the developing organization.

Approved test cases – The test cases to be examined are defined in full by the test plan. No omissions or additions are expected to occur during testing.

Software testing objectives

Direct objectives

- To identify and reveal as many errors as possible in the tested software.
- To bring the tested software, after correction of the identified errors and retesting, to an acceptable level of quality.
- To perform the required tests efficiently and effectively, within budgetary and scheduling limitations.

Indirect objective

- To compile a record of software errors for use in error prevention (by corrective and preventive actions).

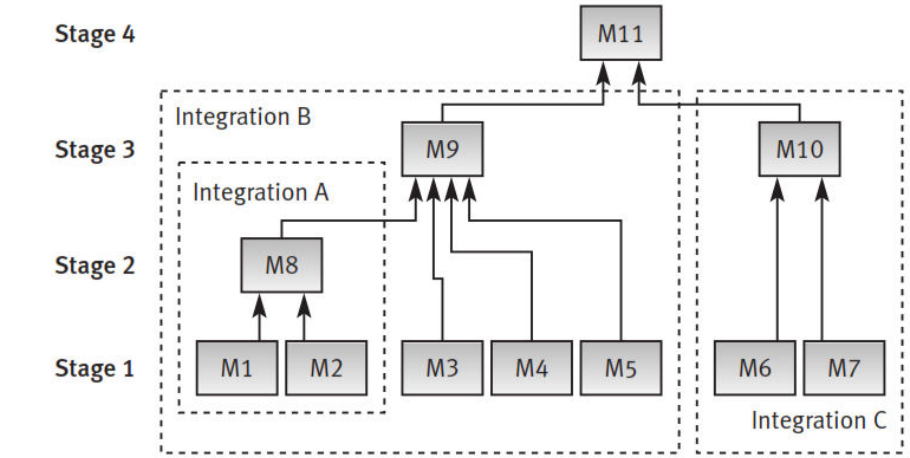
Software testing strategies

To test the software in its entirety, once the completed package is available; known as "big bang testing".

To test the software piecemeal, in modules, as they are completed (unit tests); then to test groups of tested modules integrated with newly completed modules (integration tests). This process continues until all the Package modules have been tested. Once this phase is completed, the entire package is tested as a whole (system test). This testing strategy is usually termed "incremental testing".

Incremental testing is also performed according to two basic strategies: bottom-up and top-down.

Both incremental testing strategies assume that the software package is constructed of a hierarchy of software modules.



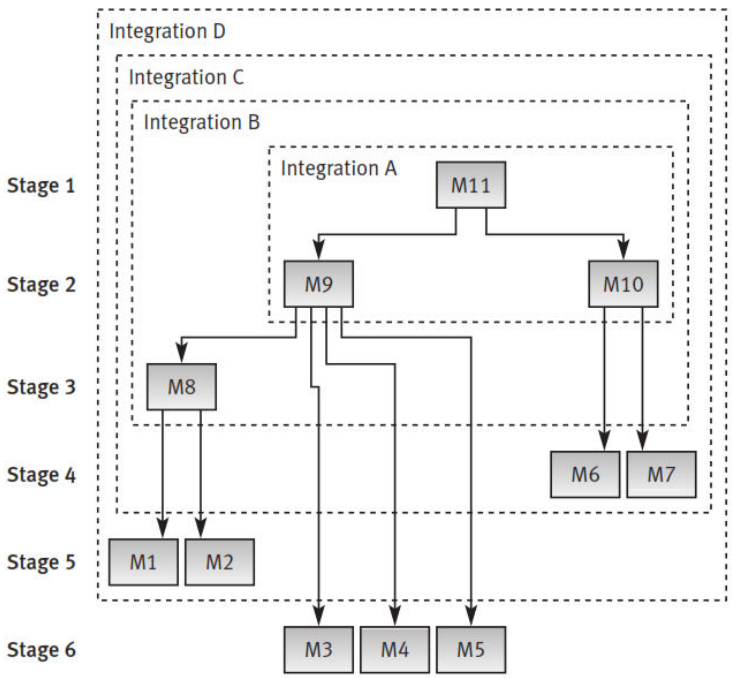
(a) Bottom-up testing

Stage 1: Unit tests of modules 1 to 7.

Stage 2: Integration test A of modules 1 and 2, developed and tested in stage 1, and integrated with module 8, developed in the current stage.

Stage 3: Two separate integration tests, B, on modules 3, 4, 5 and 8, integrated with module 9, and C, for modules 6 and 7, integrated with module 10.

Stage 4: System test is performed after B and C have been integrated with module 11, developed in the current stage.



(b) Top-down testing

Stage 1: Unit tests of module 11.

Stage 2: Integration test A of module 11 integrated with modules 9 and 10, developed in the current stage.

Stage 3: Integration test B of A integrated with module 8, developed in the current stage.

Stage 4: Integration test C of B integrated with modules 6 and 7, developed in the current stage.

Stage 5: Integration test D of C integrated with modules 1 and 2, developed in the current stage.

Stage 6: System test of D integrated with modules 3, 4 and 5, developed in the current stage.

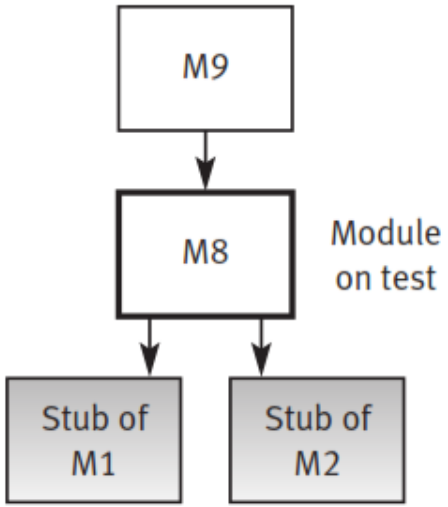
Stubs and drivers for incremental testing

Stubs and drivers are software replacement simulators required for modules not available when performing a unit or an integration test.

A stub (often termed a "dummy module") replaces an unavailable lower level module, subordinate to the module tested.

Stubs are required for topdown testing of incomplete systems. In this case, the stub provides the results of calculations the subordinate module, yet to be developed (coded), is designed to perform.

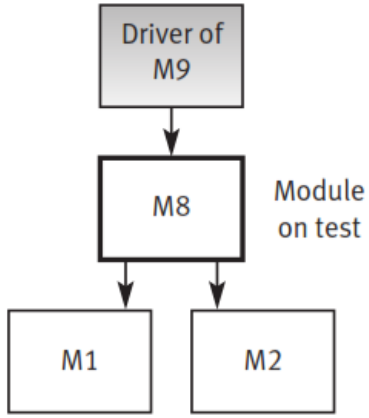
(a) Implementing top-down tests (Stage 3 testing of the example shown in Figure 9.1)



A driver is a substitute module but of the upper level module that activates the module tested. The driver is passing the test data on to the tested module and accepting the results calculated by it.

Drivers are Required in bottom-up testing until the upper level modules are developed (coded).

(b) Implementing bottom-up tests (Stage 2 testing of the example shown in Figure 9.1)



THE TESTING PROCESS

Testing is different from debugging.

- Removing errors from your programs is known as *debugging but testing aims to locate as yet undiscovered errors.*
- Starts from the requirements analysis phase and goes until the last maintenance phase.

Static testing: Requirement analysis and designing stage.

SRS is tested to check whether it is as per user requirements or not. We use techniques of code reviews, code inspections, walkthroughs, and software technical reviews (STRs) to do static testing

Dynamic Testing: This begins when the code or a specific unit/module is ready. It involves executing the code to validate its functionality. Common techniques for dynamic testing include black-box testing (focusing on inputs and outputs without knowing internal details), gray-box testing (a mix of internal knowledge and external testing), and white-box testing (testing internal structures and logic)..

Five distinct levels of testing

a. Debug: It is defined as the successful correction of a failure.

b. Demonstrate: The process of showing that major features work with typical input.

c. Verify: The process of finding as many faults in the application under test (AUT) as possible.

d. Validate: The process of finding as many faults in requirements, design, and AUT.

e. Prevent: To avoid errors in development of requirements, design, and implementation by self-checking techniques, including “test before design.”

Popular equation of software testing

Software Testing = Software Verification + Software Validation

- Software Verification "It is the process of evaluating, reviewing, inspecting and doing desk checks of work products such as requirement specifications, design specifications and code."
- Verification means **Are we building the product right?**

software validation

- "It is defined as the process of evaluating a system or component during or at the end of development process to determine whether it satisfies the specified requirements. It involves executing the actual software. It is a computer based testing process."
- Validation means **Are we building the right product?**
- Both verification and validation (V&V) are complementary to each other.

Software test classifications

Classification according to testing concept

Black box testing:

(1) Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions.

(2) Testing conducted to evaluate the compliance of a system or component with specified functional requirements.

White box testing:

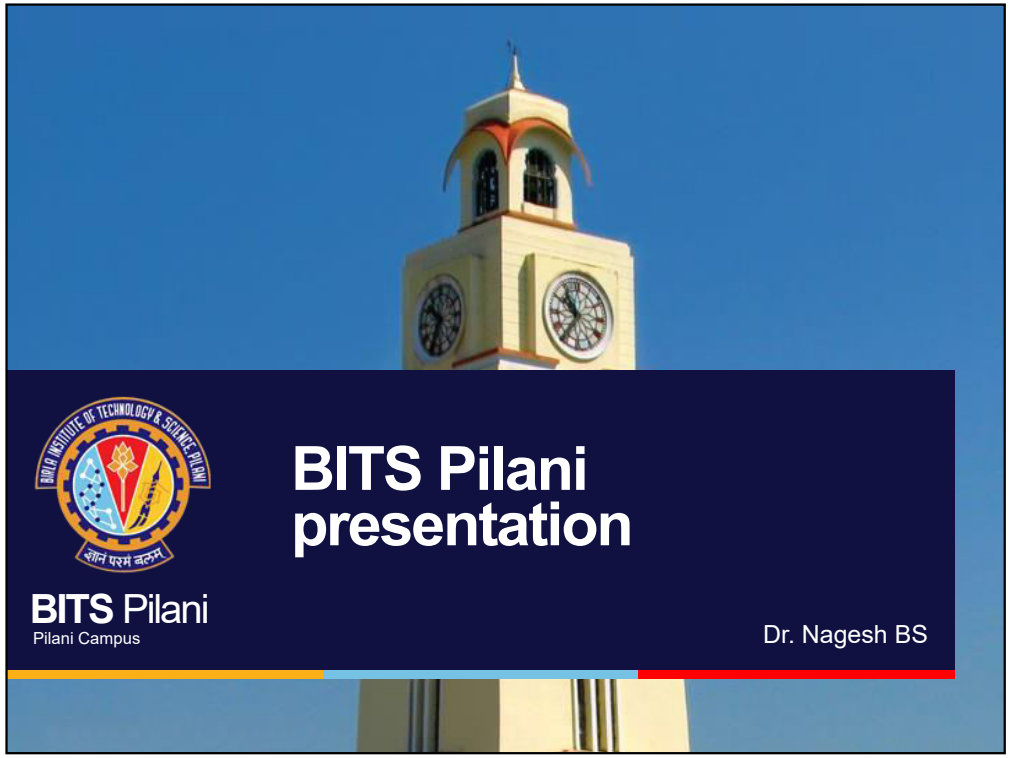
Testing that takes into account the internal mechanism of a system or component.

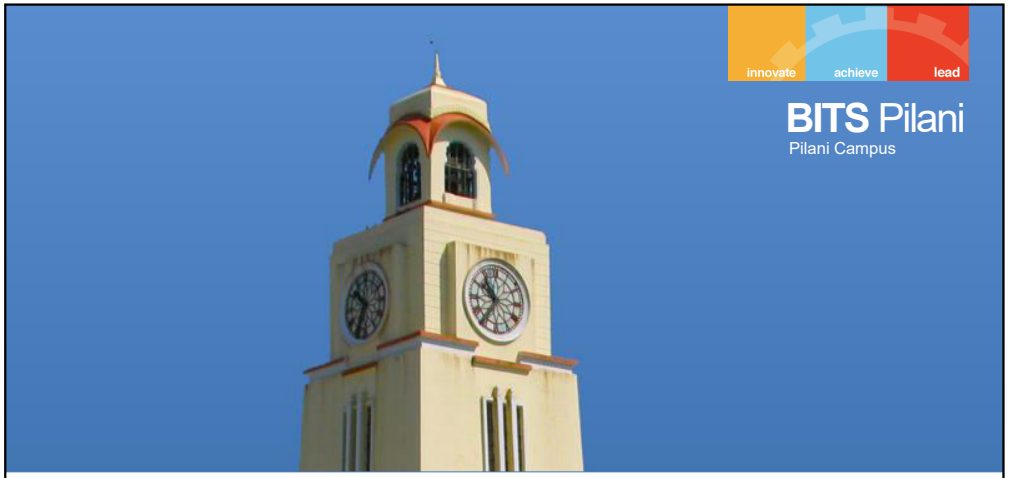
Classification according to requirements

Factor category	Quality requirement factor	Quality requirement sub-factor	Test classification according to requirements
Operation	1. Correctness	1.1 Accuracy and completeness of outputs, accuracy and completeness of data	1.1 Output correctness tests
		1.2 Accuracy and completeness of documentation	1.2 Documentation tests
		1.3 Availability (reaction time)	1.3 Availability (reaction time) tests
		1.4 Data processing and calculations correctness	1.4 Data processing and calculations correctness tests
		1.5 Coding and documentation standards	1.5 Software qualification tests
	2. Reliability		2. Reliability tests
	3. Efficiency		3. Stress tests (load tests, durability tests)
	4. Integrity		4. Software system security tests
	5. Usability	5.1 Training usability 5.2 Operational usability	5.1 Training usability tests 5.2 Operational usability tests

Revision	6. Maintainability		6. Maintainability tests
	7. Flexibility		7. Flexibility tests
	8. Testability		8. Testability tests
Transition	9. Portability		9. Portability tests
	10. Reusability		10. Reusability tests
	11. Interoperability	11.1 Interoperability with other software	11.1 Software interoperability tests
		11.2 Interoperability with other equipment	11.2 Equipment interoperability tests

THANK YOU

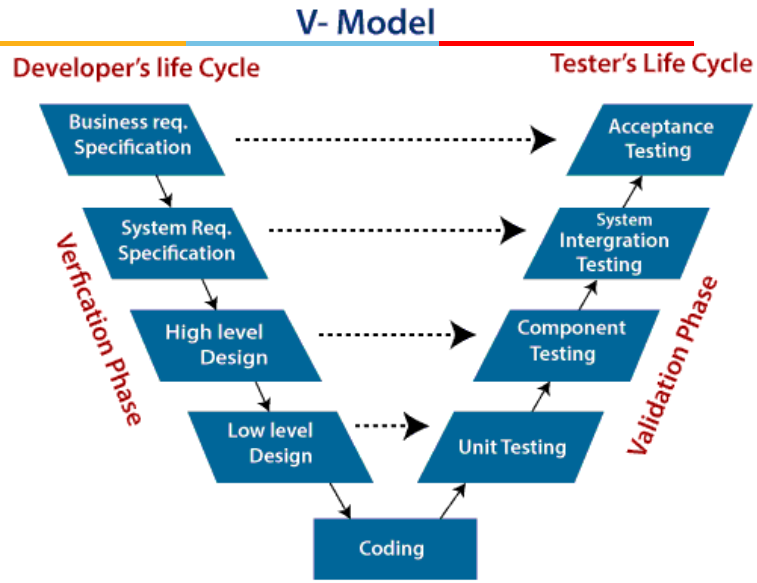




SE ZG501
Software Quality Assurance and Testing
Lecture No. 5

Deep driving SQA: Software Testing Techniques

V-Model in Software Testing



V-Model in Software testing is an SDLC model where the **test execution** takes place in a hierarchical manner. The execution process makes a V-shape.

It is also called a Verification and Validation model that undertakes the testing process for every development phase.

According to the waterfall model, testing is a post-development activity.

The spiral model took one step further by breaking the product into increments each of which can be tested separately.

V-model brings in a new perspective that different types of testing apply at different levels.

The V-model splits testing into two parts

- DESIGN
- EXECUTION.

Verification phases on one side and the Validation phases on the other side.

Verification and Validation process is joined by coding phase in V-shape.

Test: Testing is concerned with errors, faults, failures, and incidents. A test is the act of exercising software with test cases. A test has two distinct goals—**to find failures or to demonstrate correct execution.**

Test case: A test case has an **identity** and is associated with program behavior. A test case also has a set of **inputs and a list of expected outputs**. The essence of software testing is to **determine a set of test cases for the item to be tested.**

Test case template

```
Test Case ID
Purpose
Preconditions
Inputs
Expected Outputs
Postconditions
Execution History
Date        Result        Version        Run By
```


Test suite: A collection of **test scripts or test cases** that is used for validating bug fixes (or finding new bugs) within a logical or physical area of a product.

For example, an acceptance test suite contains all of the test cases that were used to verify that the software has met certain predefined acceptance criteria.

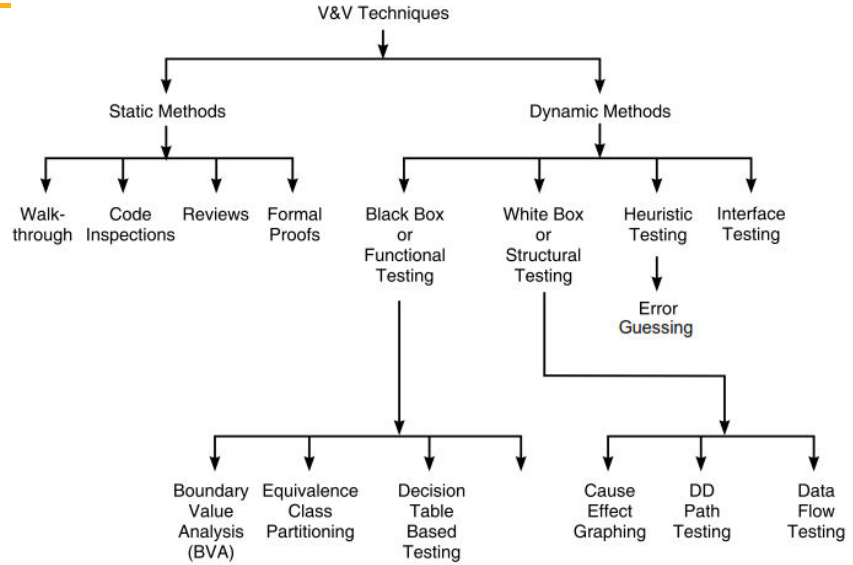
Test script: The step-by-step instructions that describe how a test case is to be executed. It may contain one or more test cases.

Test cases for ATM:

Preconditions: System is started.

Test case ID	Test case name	Test case description	Test steps			Test status (P/F)	Test priority
			Step	Expected result	Actual result		
Session 01	Verify Card	To verify whether the system reads a customer's ATM card.	Insert a readable card	Card is accepted; System asks for entry of PIN			High
			Insert an unreadable card	Card is ejected; System displays an error screen; System is ready to start a new session			High
	Validate PIN	To verify whether the system accepts customer's PIN	Enter valid PIN	System displays a menu of trans-action types			High
			Enter invalid PIN	Customer is asked to re-enter			High
			Enter incorrect PIN the first time, then correct PIN the second time	System displays a menu of transaction types.			High
			Enter incorrect PIN the first time and second time, then correct PIN the third time	System displays a menu of transaction types.			High
			Enter incorrect PIN three times	An appropriate message is displayed; Card is retained by machine; Session is terminated			High

CATEGORIZING V&V TECHNIQUES



BLACK-BOX (OR FUNCTIONAL TESTING)

- The term Black-Box refers to the software which is treated as a black-box.
- The system or source code is not checked at all.
- It is done from the customer's viewpoint.
- The test engineer engaged in black-box testing only knows the set of inputs and expected outputs and is unaware of how those inputs are transformed into outputs by the software.

BOUNDARY VALUE ANALYSIS (BVA)

It is a **black-box testing technique** based on the principle that defects are more likely to occur at the boundaries of input ranges rather than in the middle. This is because boundary conditions are more prone to errors due to incorrect implementation of logical conditions. This is done for the following reasons:

i. Programmers usually are not able to decide whether they have to use <= operator or < operator when trying to make comparisons.

ii. Different terminating conditions of for-loops, while loops, and repeat loops may cause defects to move around the boundary conditions.

iii. The requirements themselves may not be clearly understood, especially around the boundaries, thus causing even the correctly coded program to not perform the correct way.

The basic idea of BVA is to use input variable values at their minimum, just above the minimum, a nominal value, just below their maximum, and at their maximum.

{min, min+, nom, max–, max}

- When more than one variable for the same application is checked then one can use a single fault assumption.
- Holding all but one variable to the extreme value and allowing the remaining variable to take the extreme value.

BVA is based upon a critical assumption that is known as single fault assumption theory.

The **single fault assumption** states that if an error occurs, it is due to a fault in only **one variable at a time**, while all other variables are kept constant at their extreme values.

For example, if a system has **n input variables**, when testing one variable, all other variables are held at their minimum or maximum values while the selected variable is tested at its boundary value

•For n variable to be checked:

•Maximum of 4n+1 test cases

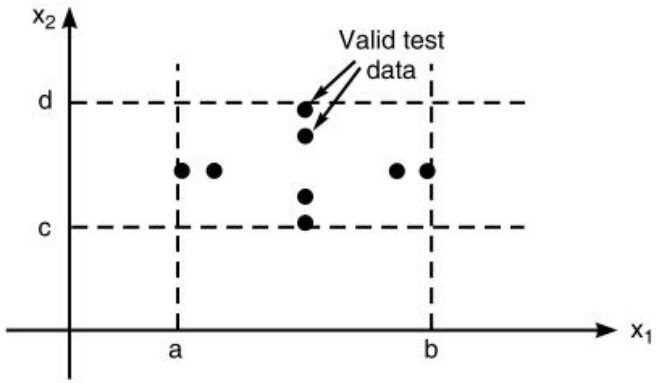


FIGURE 3.1 BVA Test Cases.

Problem: Consider a Program for determining the Previous Date.

Input: *Day, Month, Year with valid ranges as-*

1 ≤ Month≤12
1 ≤ Day ≤31
1900 ≤ Year ≤ 2000

Design Boundary Value Test Cases.

Solution:

1) Year as a Single Fault Assumption

Test Cases	Month	Day	Year	Output
1	6	15	1900	14 June 1900
2	6	15	1901	14 June 1901
3	6	15	1960	14 June 1960
4	6	15	1999	14 June 1999
5	6	15	2000	14 June 2000

Day as Single Fault Assumption

Test Case	Month	Day	Year	Output
6	6	1	1960	31 May 1960
7	6	2	1960	1 June 1960
8	6	30	1960	29 June 1960
9	6	31	1960	Invalid day

Month as Single Fault Assumption

Test Case	Month	Day	Year	Output
10	1	15	1960	14 Jan 1960
11	2	15	1960	14 Feb 1960
12	11	15	1960	14 Nov 1960
13	12	15	1960	14 Dec 1960

For the n variable to be checked Maximum of 4n + 1 test case will be required.

Therefore, for n = 3, the maximum test cases are

4 × 3 + 1 =13

Consider a system that accepts ages from 18 to 56.

Boundary Value Analysis(Age accepts 18 to 56)

Invalid (min-1)	Valid (min, min + 1, nominal, max – 1, max)	Invalid (max + 1)
17	18, 19, 37, 55, 56	57

Valid Test cases: Valid test cases for the above can be any value entered greater than 17 and less than 57.

Enter the value- 18.

Enter the value- 19.

Enter the value- 37.

Enter the value- 55.

Enter the value- 56.

Invalid Testcases: When any value less than 18 and greater than 56 is entered.

Enter the value- 17.

Enter the value- 57.

EQUIVALENCE CLASS TESTING

The use of equivalence classes as the basis for functional testing has two motivations:

a. We want exhaustive testing

b. We want to avoid redundancy

This is not handled by the BVA technique as we can see massive redundancy in the tables of test cases.

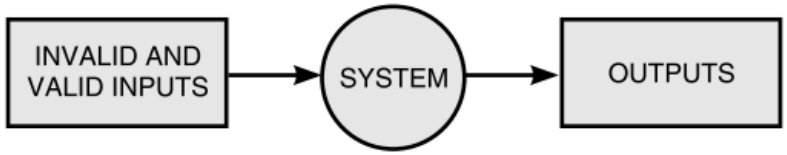


FIGURE 3.4 Equivalence Class Partitioning.

Process

- The input and the output domain **is divided into a** finite number of equivalence classes.
- select one representative of each class and test our program against it.
- It is assumed by the tester that if one representative from a class is able to detect error then why should we consider other cases.
- if this single representative test case did not detect any error then we assume that no other test case of this class can detect error.
- we consider both valid and invalid input domains.

- The key and the craftsmanship lies in the choice of the equivalence relation that determines the classes.
- The equivalence class testing can be categorized into four different types.
- **Weak Normal Equivalence Class Testing:** In this first type of equivalence class testing, one variable from each equivalence class is tested by the team. Moreover, the values are identified in a systematic manner. Weak normal equivalence class testing is also known as **single fault assumption**.

Strong Normal Equivalence Class Testing: Termed as **multiple fault assumption**, in strong normal equivalence class testing the team selects test cases from each element of the Cartesian product of the equivalence. This ensures the notion of completeness in testing, as it covers all equivalence classes and offers the team of each possible combinations of inputs.

Weak Robust Equivalence Class Testing: Like weak normal equivalence, weak robust testing too tests one variable from each equivalence class. However, unlike the former method, it is also focused on testing test cases for invalid values.

Strong Robust Equivalence Class Testing: Another type of equivalence class testing, strong robust testing produces test cases for all valid and invalid elements of the product of the equivalence class. However, it is incapable of reducing the redundancy in testing.

White Box Testing

- White-box testing is a way of testing the external functionality of the code by examining and testing the program code that realizes the external functionality.
- White-box testing is used to test the program code, code structure, and the internal design flow.

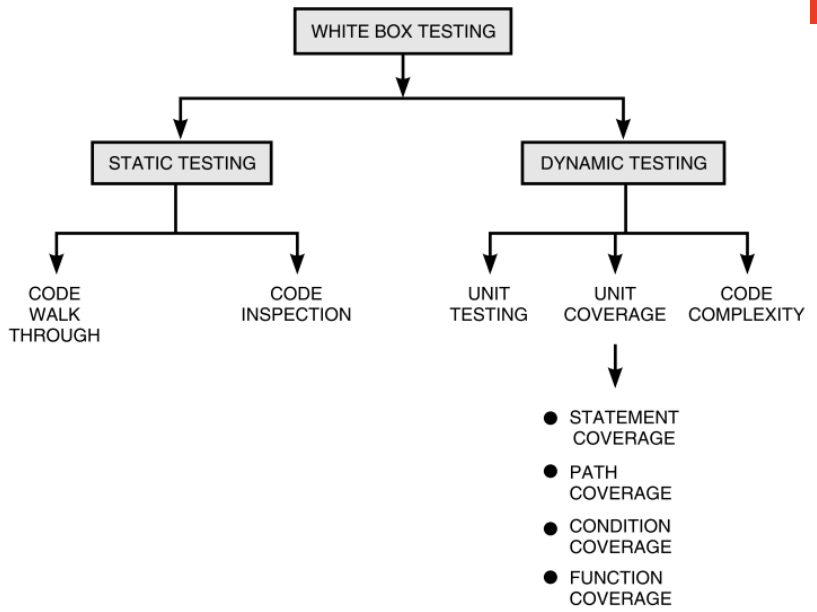


FIGURE 4.1 Classification of White-Box Testing.

CODE COVERAGE TESTING

•Designing and executing test cases and finding out the percentage of code that is covered by testing.

•The percentage of code covered by a test is found by adopting a technique called the instrumentation of code.

•*STATEMENT COVERAGE*

•*PATH COVERAGE*

•*CONDITION COVERAGE*

•*FUNCTION COVERAGE*

Test levels and types

Three levels of testing:

1. **Unit testing**
2. **Integration testing**
3. **System testing**

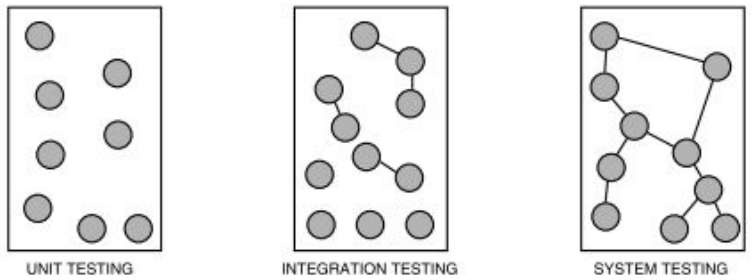
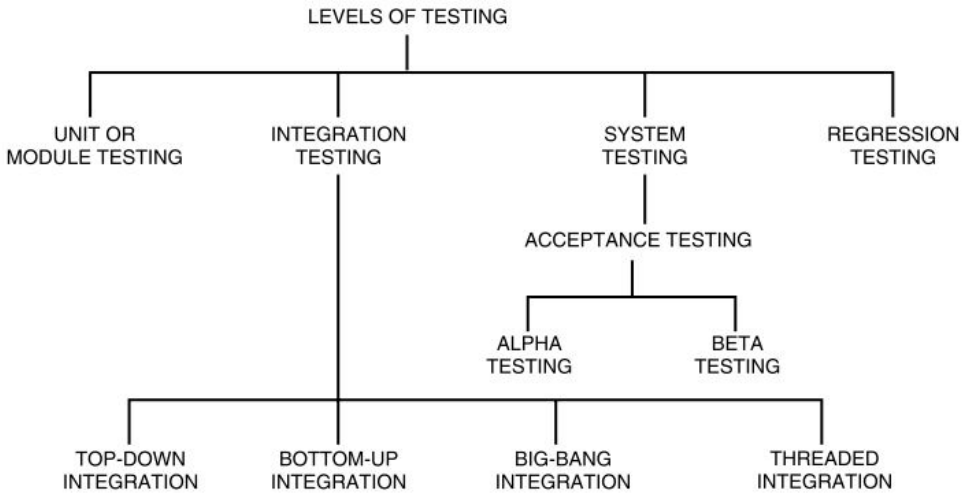


FIGURE 7.1 Levels of Testing.



Unit (or module) testing

Unit (or module) testing "is the process of taking a module (an atomic unit) and running it in isolation from the rest of the software product by using prepared test cases and comparing the actual results with the results predicted by the specification and design module."

It is a white-box testing technique.

Importance of unit testing:

1. Because modules are being tested individually, testing becomes easier.
2. It is more exhaustive.
3. Interface errors are eliminated.

INTEGRATION TESTING

- *A system is composed of multiple components or modules that comprise hardware and software.*
- *Integration is defined as the set of interactions among components.*
- *Testing the interaction between the modules and interaction with other systems externally is called integration testing.*
- *The architecture and design can give the details of interactions within systems.*

Classification of integration testing

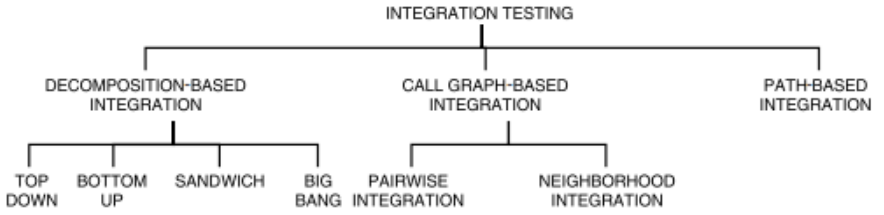


FIGURE 7.4

Decomposition-based Integration:

Decomposition-based integration involves functionally decomposing the system under test into a hierarchical structure, represented as a tree or in textual form. The primary objective is to evaluate the interfaces between individually tested units

Types Of Decomposition-based Techniques :Top-down Integration Approach

It begins with the main program, i.e., the **root of the tree**.

Any lower-level unit that is called by the main program appears as a "stub."

A stub is a piece of throw-away code that emulates a called unit.

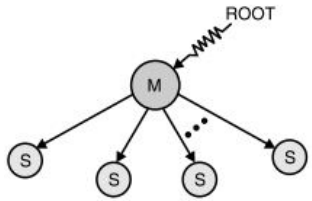


FIGURE 7.5 Stubs.

Bottom-up Integration Approach

It is a mirror image to the top-down order with the difference that stubs are replaced by driver modules that emulate units at the next level up in the tree.

start with the leaves of the decomposition tree and test them with specially coded drivers. Less throw-away code exists in drivers than there is in stubs.

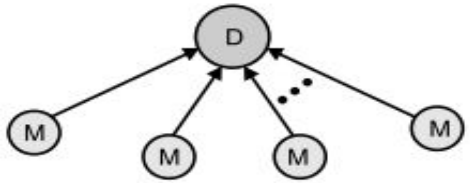


FIGURE 7.6 Drivers.

Sandwich Integration Approach

The **Sandwich Integration Approach**, also known as **Bidirectional Integration**, combines **Top-Down** and **Bottom-Up** integration testing methods.

- **Hybrid Approach**: It merges top-down and bottom-up techniques to accelerate integration.
- **Reduced Stub and Driver Effort**: Since both directions are integrated simultaneously, fewer stubs and drivers are needed.
- **Initial Testing with Stubs & Drivers**: Early integration relies on stubs (to replace lower modules) and drivers (to simulate higher modules).
- **Focus on Key Components**: Emphasis is placed on testing newly developed or critical components efficiently.

Big-bang Integration

- Instead of integrating component by component and testing, **this approach waits until all the components arrive and one round of integration testing is done**. This is known as *big-bang integration.*
- *It reduces testing effort and* **removes duplication in testing** for the multi-step component integrations.
- Big-bang integration is **ideal for a product where the interfaces are stable with fewer number of defects**.

7.2.2.6. GUIDELINES TO CHOOSE INTEGRATION METHOD AND CONCLUSIONS

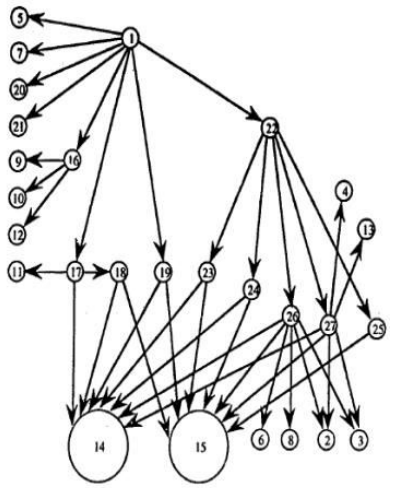
S. No.	Factors	Suggested method
1.	Clear requirements and design.	Top-down approach.
2.	Dynamically changing requirements, design, and architecture.	Bottom-up approach.
3.	Changing architecture and stable design.	Sandwich (or bi-directional) approach.
4.	Limited changes to existing architecture with less impact.	Big-bang method.
5.	Combination of above.	Select any one after proper analysis.

Call Graph-based Integration

- Structural testing(shows **how functions/modules interact**)
- A **Call Graph** is a **directed graph** that represents **function/method calls** within a program.
- Call graph is a directed graph thus, we can use it as a program graph.
  - *Pairwise Integration*
  - *Neighborhood Integration*

Table 13.1 SATM Units and Abbreviated Names

Unit Number	Level Number	Unit Name
1	1	SATM System
A	1.1	Device Sense & Control
D	1.1.1	Door Sense & Control
2	1.1.1.1	Get Door Status
3	1.1.1.2	Control Door
4	1.1.1.3	Dispense Cash
E	1.1.2	Slot Sense & Control
5	1.1.2.1	WatchCardSlot
6	1.1.2.2	Get Deposit Slot Status
7	1.1.2.3	Control Card Roller
8	1.1.2.3	Control Envelope Roller
9	1.1.2.5	Read Card Strip
10	1.2	Central Bank Comm.
11	1.2.1	Get PIN for PAN
12	1.2.2	Get Account Status
13	1.2.3	Post Daily Transactions
B	1.3	Terminal Sense & Control
14	1.3.1	Screen Driver
15	1.3.2	Key Sensor
C	1.4	Manage Session
16	1.4.1	Validate Card
17	1.4.2	Validate PIN
18	1.4.2.1	GetPIN
F	1.4.3	Close Session
19	1.4.3.1	New Transaction Request
20	1.4.3.2	Print Receipt
21	1.4.3.3	Post Transaction Local
22	1.4.4	Manage Transaction
23	1.4.4.1	Get Transaction Type
24	1.4.4.2	Get Account Type
25	1.4.4.3	Report Balance
26	1.4.4.4	Process Deposit
27	1.4.4.5	Process Withdrawal



Pairwise Integration

- Pairwise Integration reduces the effort needed to create **stubs and drivers** by directly integrating **two related units at a time** from the **call graph**.
- Instead of developing temporary stubs and drivers, we test real code components that interact.
- Each integration focuses on a single pair of units, ensuring their functionality before moving to the next pair.
- The total number of test sessions remains similar to top-down or bottom-up approaches, but the effort in creating extra test code is significantly reduced.

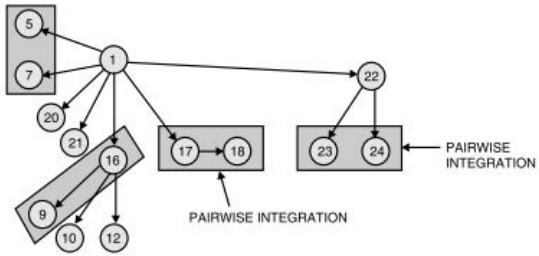


FIGURE 7.7 Pairwise Integration.

Neighborhood Integration

The neighborhood of a node in a graph is the set of nodes that are one edge away from the given node.

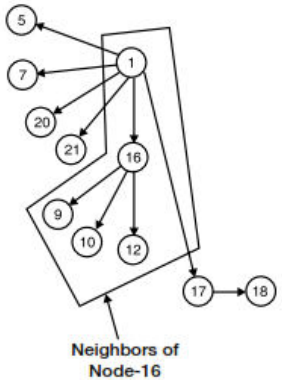


FIGURE 7.8 Neighborhood Integration.

Path-based Integration

- Path-Based Integration Testing is a **structural testing approach** that focuses on testing **all possible execution paths** between interacting modules or components. Instead of only checking individual interfaces, it ensures that different **execution flows and conditions are**
- The combination of structural and functional testing.
- structural and functional testing are highly desirable at the unit level and it would be nice to have a similar capability for integration and system testing.
- "*Instead of testing interfaces among separately developed and tested units, we focus on interactions among these units.*"
- Here, *cofunctioning might be a* good term.
- Interfaces are structural whereas interaction is behavioral.

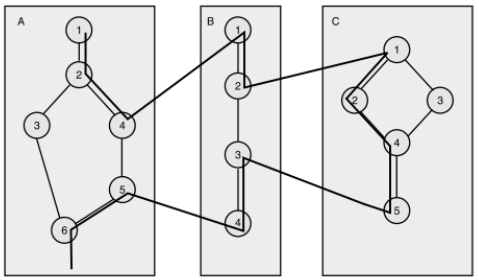


FIGURE 7.9 MM-Path Across Three Units (A, B, and C).

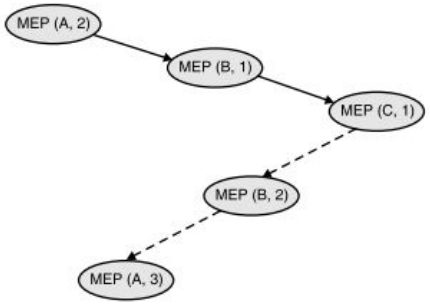


FIGURE 7.10 MM-Path Graph Derived from Figure 7.9.

SYSTEM TESTING

The testing that is conducted on the complete integrated products and solutions to evaluate system compliance with specified requirements on functional and non functional aspects is called *system testing. It is done after* unit, and integration testing phases.

System testing is done to:

1. Provide independent perspective in testing as the team becomes more quality centric.
2. Bring in customer perspective in testing.
3. Provide a "fresh pair of eyes" to discover defects not found earlier by testing.
4. Test product behavior in a holistic, complete, and realistic environment.
5. Test both functional and non functional aspects of the product.
6. Build confidence in the product.
7. Analyze and reduce the risk of releasing the product.
8. Ensure all requirements are met and ready the product for acceptance testing.

Thank You

BVA

single fault assumption.

Understanding the Formula

The formula **4n+1**represents the **maximum number of test cases required** when applying Boundary Value Analysis (BVA) under the **single fault assumption**.

Breaking It Down

1. **For a Single Variable (n = 1)**:
    - When testing a single variable, we check at **four boundary points**:
        - **Lower boundary** (Min)
        - **Lower boundary +1** (Min + 1)
        - **Upper boundary** (Max)
        - **Upper boundary -1** (Max - 1)
    - We also test **one nominal value** (some midpoint value).
    - **Total test cases for one variable = 4 + 1 = 5.**
2. **For Multiple Variables (n > 1)**:
    - **Single Fault Assumption:** We assume only **one variable** is faulty at a time, while all others remain at their extreme values.
    - So, for **each variable**, we need **4 tests (boundary values)** while keeping other variables constant at their extreme values.
    - Since there are **n variables**, we multiply by 4: **4n**.
    - We also include **one additional nominal test case**, where all variables are set to their typical mid-range values.

Thus, for **n variables**:

Total Test Cases) = 4n + 1

Example: Applying 4n + 1 Formula

Case 1: Single Variable (n = 1)

Consider an input **X** with a valid range **[10, 100]**:

- **Boundary values to test:**
    - **Lower bound:** 10
    - **Lower bound +1:** 11
    - **Upper bound:** 100
    - **Upper bound -1:** 99
    - **Nominal value (midpoint):** 55
- **Total Test Cases = 4(1) + 1 = 5**

Case 2: Two Variables (n = 2)

Consider **X** in [10, 100] and **Y** in [20, 200]:

- **Variable X** boundary tests: Keep **Y** at **extreme values** (either 20 or 200).
    - X = 10, Y = 20
    - X = 11, Y = 20
    - X = 99, Y = 20
    - X = 100, Y = 20
- **Variable Y** boundary tests: Keep **X** at **extreme values** (either 10 or 100).
    - Y = 20, X = 10
    - Y = 21, X = 10
    - Y = 199, X = 10
    - Y = 200, X = 10
- **Nominal test case**: X = 55, Y = 110

Total Test Cases = 4(2) + 1 = 9

Equivalence Class Partitioning (ECP) - All 4 Types Explained Simply

Equivalence Class Partitioning (ECP) is a **black-box testing technique** that divides input values into **valid and invalid classes** to reduce the number of test cases while ensuring full coverage.

There are **four types** of Equivalence Class Testing:

1. **Weak Normal Equivalence Class Testing**
2. **Strong Normal Equivalence Class Testing**
3. **Weak Robust Equivalence Class Testing**
4. **Strong Robust Equivalence Class Testing**

---

1. Weak Normal Equivalence Class Testing

✔ **Single fault assumption** (only one variable changes at a time).
✔ **One test case is selected per equivalence class per variable.**
✔ **Does not include invalid values.**

Example: Age Input (Valid: 18 - 60)

Equivalence Class	Representative Test Case
Valid Age (18 - 60)	**30**

- Here, we **only** test **one valid value** (e.g., 30).
- **Invalid values (e.g., 17, 61) are not tested.**

👉 **Key Point:** Only valid classes are tested, and only one test case per class is chosen.

---

2. Strong Normal Equivalence Class Testing

✔ **Multiple fault assumption** (tests all possible combinations of valid classes).
✔ **Uses the Cartesian product of all equivalence classes.**
✔ **Does not include invalid values.**

Example: Online Order System

- **Two Inputs:** Payment Type & Shipping Method
- **Valid Equivalence Classes:**
    - **Payment:** (Credit Card, PayPal)
    - **Shipping**: (Standard, Express)

Test Cases (All Valid Combinations)

Test Case	Payment Type	Shipping Method
TC1	Credit Card	Standard
TC2	Credit Card	Express
TC3	PayPal	Standard
TC4	PayPal	Express

👉 **Key Point:** Tests **all possible valid combinations**.

---

3. Weak Robust Equivalence Class Testing

✔ **Single fault assumption** (only one variable changes at a time).
✔ **Includes both valid and invalid values.**
✔ **One test case is selected per class per variable.**

Example: ATM Withdrawal (Valid: $100 - $5000)

Equivalence Class	Representative Test Case
Valid Amount (100 - 5000)	**$1000**
Invalid Amount (<100)	**$50**
Invalid Amount (>5000)	**$6000**

👉 **Key Point:** Both **valid and invalid cases** are tested, but **only one variable at a time**.

---

4. Strong Robust Equivalence Class Testing

✔ **Multiple fault assumption** (tests all possible combinations).
✔ **Includes both valid and invalid values.**
✔ **Tests all possible valid + invalid combinations (Cartesian product).**

Example: Online Order System

- **Payment Types:** (Credit Card ✔, PayPal ✔, Invalid ✘)
- **Shipping Methods:** (Standard ✔, Express ✔, Invalid ✘)

Test Cases (Valid + Invalid Combinations)

Test Case	Payment Type	Shipping Method
TC1	Credit Card ✔	Standard ✔
TC2	Credit Card ✔	Express ✔

Test Case	Payment Type	Shipping Method
TC3	PayPal ✅	Standard ✅
TC4	PayPal ✅	Express ✅
TC5	Invalid ❌	Standard ✅
TC6	Invalid ❌	Express ✅
TC7	Credit Card ✅	Invalid ❌
TC8	PayPal ✅	Invalid ❌
TC9	Invalid ❌	Invalid ❌

👉 **Key Point: Every combination** (valid + invalid) is tested.

Final Summary

Type	Fault Assumption	Includes Invalid Cases?	Test Case Selection
Weak Normal	Single Fault	❌ No	One valid case per class
Strong Normal	Multiple Fault	❌ No	All valid combinations
Weak Robust	Single Fault	✅ Yes	One valid + one invalid per class
Strong Robust	Multiple Fault	✅ Yes	All valid + invalid combinations

---

Conclusion

- **Weak Normal → Tests only valid cases, one per class.**
- **Strong Normal → Tests all valid combinations.**
- **Weak Robust → Adds invalid cases, but tests one at a time.**
- **Strong Robust → Tests every possible combination of valid and invalid cases.**

Would you like a **Python script to generate these test cases automatically**? 🚀

Test Case ID

Test Case ID: TC001
Purpose: Verify that the login functionality works with **valid credentials**.

Preconditions:

1. User is **registered** in the system.
2. The application is accessible and running.

Inputs:

- Username: testuser
- Password: password123

Expected Outputs:

- The user is successfully logged in and redirected to the dashboard.

Postconditions:

- The user session is active, and the dashboard is displayed.

Execution History:

Date	Result	Version	Run By
2025-02-10	Pass	v1.0	Tester A
2025-02-11	Fail	v1.1	Tester B

V-Model

The V-Model, as depicted in the diagram, represents a software development lifecycle model emphasizing a **structured and sequential development and testing process**. It is called the "**V-Model" because of its V-shaped structure**, where the left side represents the **verification phase** (developer's lifecycle), and the right side represents the **validation phase** (tester's lifecycle). The center of the "V" is the coding phase, which bridges the two halves.

Here's a detailed explanation of each stage in the diagram:

Verification Phase (Left Side of the V)

This phase focuses on **planning, designing, and verifying that the product is being built correctly according to requirements.**

1. **Business Requirement Specification (BRS)**
   - Description: Defines the **high-level business goals and user needs.**
   - Output: **A business requirement document (BRD).**
   - Corresponding Test Activity: **Acceptance Testing**.
2. **System Requirement Specification (SRS)**
   - Description: Elaborates on **detailed functional and non-functional requirements**.
   - Output: **A system requirement specification document**.
   - Corresponding Test Activity: System Integration Testing.
3. **High-Level Design (HLD)**
   - Description: Describes the **architecture and design of the system**, **breaking down modules and their interactions.**
   - Output: High-level design documents.
   - Corresponding Test Activity: Component Testing.
4. **Low-Level Design (LLD)**
   - Description: Focuses **on detailed designs for each module or component**.
   - Output: Low-level design documents.
   - Corresponding Test Activity: **Unit Testing**.
5. **Coding**
   - **Description**: The actual development of the software system, converting designs into executable code.

Validation Phase (Right Side of the V)

This phase involves testing to ensure the product meets the business and technical requirements.

1. **Unit Testing**
   - **Tests individual components or modules as defined in the LLD**.
   - Ensures each unit functions as intended.
2. **Component Testing**
   - Verifies the **integration and interaction between multiple components.**
   - Ensures data flow between modules works as expected.
3. **System Integration Testing**
   - Validates **the overall system's functionality** against the **system requirements**.
   - Ensures the system works in an integrated environment.
4. **Acceptance Testing**
   - Confirms that the **final product meets the business requirements (BRS).**
   - Often performed by the client or end-users.

Key Features of the V-Model

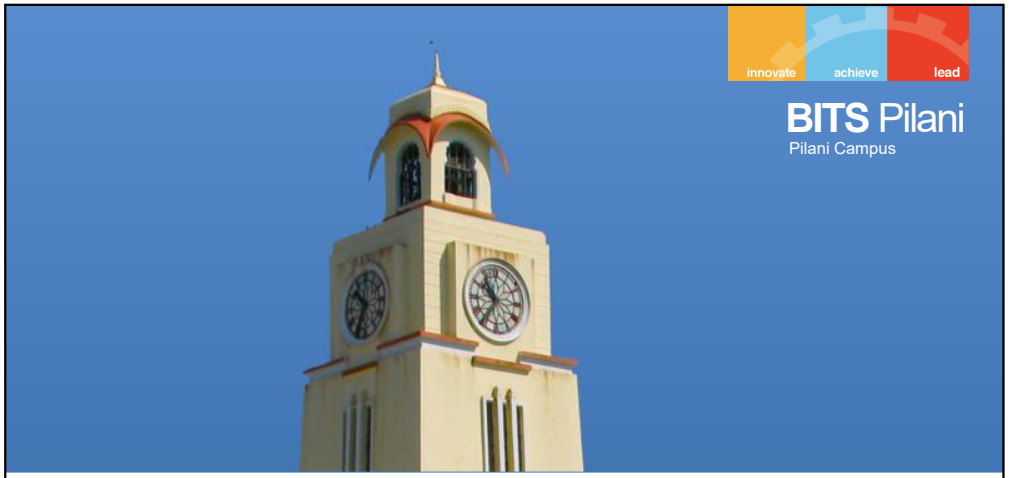
- **Validation at Every Stage**: Each stage in the verification phase has a corresponding testing stage in the validation phase to ensure the deliverables meet requirements.
- **Sequential and Rigid**: The model does not allow going back once a phase is complete, making it suitable for projects with well-defined requirements.
- **Early Defect Detection**: Verification activities help catch defects early in the lifecycle.

This model is best suited for projects where requirements are clear and unlikely to change, as it emphasizes thorough planning and systematic testing.

BITS Pilani presentation

BITS Pilani
Pilani Campus

Dr. Nagesh BS



innovate achieve lead

BITS Pilani
Pilani Campus

SE ZG501
Software Quality Assurance and Testing
Session 6

SYSTEM TESTING

innovate achieve lead

The testing that is conducted on the complete integrated products and solutions to evaluate system compliance with specified requirements on functional and non functional aspects is called *system testing*. *It is done after* unit, component, and integration testing phases.

innovate achieve lead

System testing is done to:

1. Provide independent perspective in testing as the team becomes more quality centric.
2. Bring in customer perspective in testing.
3. Provide a "fresh pair of eyes" to discover defects not found earlier by testing.
4. Test product behavior in a holistic, complete, and realistic environment.
5. Test both functional and non functional aspects of the product.
6. Build confidence in the product.
7. Analyze and reduce the risk of releasing the product.
8. Ensure all requirements are met and ready the product for acceptance testing.

- An independent test team normally does system testing.
- This independent test team is different from the team that does the component and integration testing.
- The system test team generally reports to a manager other than the product-manager to avoid conflicts and to provide freedom to testing team during system testing.
- Testing the product with an independent perspective and combining that with the perspective of the customer makes system testing unique, different, and effective.

Functional testing	Non functional testing
1. It involves the product's functionality.	1. It involves the product's quality factors.
2. Failures, here, occur due to code.	2. Failures occur due to either architecture, design, or due to code.
3. It is done during unit, component, integration, and system testing phase.	3. It is done in our system testing phase.
4. To do this type of testing only domain of the product is required.	4. To do this type of testing, we need domain, design, architecture, and product's knowledge.
5. Configuration remains same for a test suite.	5. Test configuration is different for each test suite.

Test Execution Process

Test plan

The primary purpose of a test plan is to define **the scope, approach, resources, and schedule** for testing activities. It provides a **systematic approach** to ensure that the software meets specified requirements and quality standards.

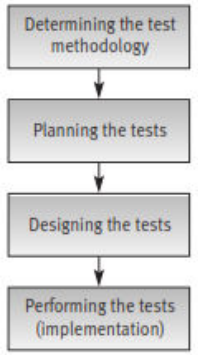
Key components of a test plan

- Objectives,
- Scope,
- Test Items,
- Test Environment,
- Testing Strategy,
- Test Schedule,
- Resource Requirements,
- Risk Management
- Test Deliverables.

The testing process

Planning, design and performance of testing are carried out throughout the software development process.

These activities are divided in phases, beginning in the design stage and ending when the software is installed at the customer's site.



The testing process

Determining the test methodology phase

The key challenges in testing methodology include :

- The appropriate required software quality standard
- The software testing strategy.

Determining the appropriate software quality standard

The level of quality standard selected for a project depends mainly on the characteristics of the software's application.

Example 1: A software package for a hospital patient bed monitor requires the highest software quality standard considering the possibly severe consequences of software failure.

Determining the software testing strategy

Key decisions to be made include:

- Selecting a testing approach: **Big Bang vs. Incremental**. If incremental, should it follow a **bottom-up or top-down** approach?
- Identifying which parts of the test plan should follow the **white-box testing** model.
- Determining which test cases should be executed using **automated testing?**

Planning the tests

The tests to be planned include:

- Unit tests
- Integration tests
- System tests.

Consider the following issues before initiating a specific test plan:

- What to test?
- Which sources to use for test cases?
- Who is to perform the tests?
- Where to perform the tests?
- When to terminate the tests?

Test planning documentation

The planning stage of the software system tests is commonly documented in a "software test plan" (STP).

Frame 10.2 The software test plan (STP) – template

1 Scope of the tests

1.1 The software package to be tested (name, version and revision)
1.2 The documents that provide the basis for the planned tests (name and version for each document)

2 Testing environment

2.1 Testing sites
2.2 Required hardware and firmware configuration
2.3 Participating organizations
2.4 Manpower requirements
2.5 Preparation and training required of the test team

3 Test details (for each test)

3.1 Test identification
3.2 Test objective
3.3 Cross-reference to the relevant design document and the requirement document
3.4 Test class
3.5 Test level (unit, integration or system tests)
3.6 Test case requirements
3.7 Special requirements (e.g., measurements of response times, security requirements)
3.8 Data to be recorded

4 Test schedule (for each test or test group) including time estimates for the following:

4.1 Preparation
4.2 Testing
4.3 Error correction
4.4 Regression tests

Test design

The products of the test design stage are:

- Detailed design and procedures for each test.
- Test case database/file.

The test design is carried out on the basis of the software test plan as documented by STP.

The test procedures and the test case database/file may be documented in a "software test procedure" document and "test case file" document or in a single document called the "software test description" (STD).

Frame 10.3 **Software test descriptions (STD) – template**

1 Scope of the tests

1.1 The software package to be tested (name, version and revision)
1.2 The documents providing the basis for the designed tests (name and version for each document)

2 Test environment (for each test)

2.1 Test identification (the test details are documented in the STP)
2.2 Detailed description of the operating system and hardware configuration and the required switch settings for the tests
2.3 Instructions for software loading

3 Testing process

3.1 Instructions for input, detailing every step of the input process
3.2 Data to be recorded during the tests

4 Test cases (for each case)

4.1 Test case identification details
4.2 Input data and system settings
4.3 Expected intermediate results (if applicable)
4.4 Expected results (numerical, message, activation of equipment, etc.)

5 Actions to be taken in case of program failure/cessation

6 Procedures to be applied according to the test results summary

Test implementation

- The testing implementation phase activities consist of a series of tests, corrections of detected errors and re-tests (regression tests).
- Testing is culminated when the re-test results satisfy the developers.
- The tests are carried out by running the test cases according to the test procedures.
- Documentation of the test procedures and the test case database/file comprises the “software test description” (STD)
- Re-testing (also termed “regression testing”) is conducted to verify that the errors detected in the previous test runs have been properly corrected, and that no new errors have entered as a result of faulty corrections.

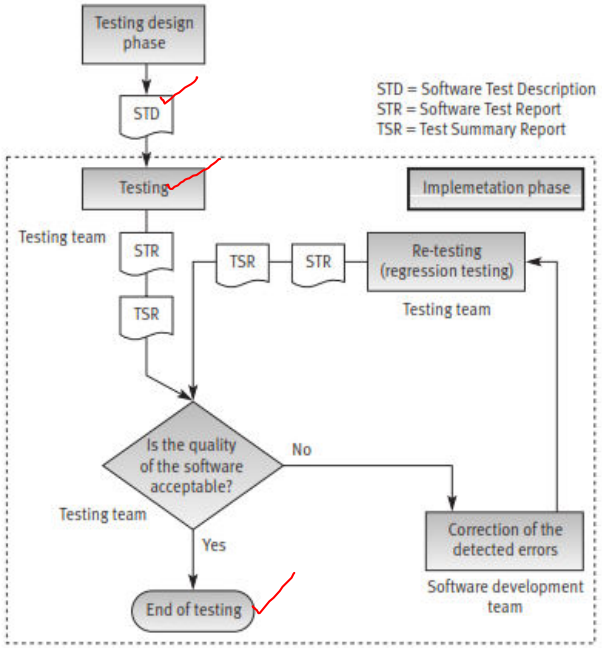


Figure 10.2: Implementation phase activities

The summary of the set of tests planned for a software package (or software development project) is documented in the “test summary report” (TSR).

Frame 10.4 **Software test report (STR) – template**

1 Test identification, site, schedule and participation
1.1 The tested software identification (name, version and revision)
1.2 The documents providing the basis for the tests (name and version for each document)
1.3 Test site
1.4 Initiation and concluding times for each testing session
1.5 Test team members
1.6 Other participants
1.7 Hours invested in performing the tests

2 Test environment
2.1 Hardware and firmware configurations
2.2 Preparations and training prior to testing

3 Test results
3.1 Test identification
3.2 Test case results (for each test case individually)
3.2.1 Test case identification
3.2.2 Tester identification
3.2.3 Results: OK / failed
3.2.4 If failed: detailed description of the results/problems

4 Summary tables for total number of errors, their distribution and types
4.1 Summary of current tests
4.2 Comparison with previous results (for regression test summaries)

5 Special events and testers' proposals
5.1 Special events and unpredicted responses of the software during testing
5.2 Problems encountered during testing
5.3 Proposals for changes in the test environment, including test preparations
5.4 Proposals for changes or corrections in test procedures and test case files

Test case design

Test case data components

A test case is a documented set of the **data inputs** and **operating conditions** required to **run a test item** together **with the expected results** of the run.

The tester is expected to run the program for the test item according to the test case documentation, and then compare the actual results with the expected results noted in the documents.

If the obtained results completely agree with the expected results, no error is present or at least has been identified. When some or all of the results do not agree with the expected results, a potential error is identified.

Example

Consider the following test cases for the basic annual municipal property tax on apartments. The basic municipal property tax (before discounts to special groups of city dwellers) is based on the following parameters:

S, the size of the apartment (in square yards)
N, the number of persons living in the apartment
A, B or C, the suburb's socio-economic classification.

The municipal property tax (MPT) is calculated as follows:

For class A suburbs: $MPT = (100 \times S) / (N + 8)$
For class B suburbs $MPT = (80 \times S) / (N + 8)$
For class C suburbs $MPT = (50 \times S) / (N + 8)$

The following are three test cases for the software module used to calculate the basic municipal property tax on apartments:

	Test case 1	Test case 2	Test case 3
Size of apartment – (square yards), *S*	250	180	98
Suburb class	A	B	C
No. of persons in the household, *N*	2	4	6
Expected result: municipal property tax (MPT)	**$2500**	**$1200**	**$350**

Automated testing

- Automated testing represents an additional step in the integration of computerized tools into the process of software development.
- These tools have joined computer aided software engineering (CASE) tools in performing a growing share of software analysis and design tasks.

Factors have motivated the development of automated testing tools:

Anticipated cost savings, shortened test duration, heightened thoroughness of the tests performed, improvement of test accuracy, improvement of result reporting as well as statistical processing and subsequent reporting.

The process of automated testing

Automated software testing requires **test planning, test design, test case preparation, test performance, test log and report preparation, re-testing after correction of detected errors (regression tests), and final test log and Report preparation including comparison reports**.

The last two activities may be repeated several times.

Types of automated tests

Code auditing : The computerized code auditor checks the **compliance of code to specified standards and procedures of coding**. The auditor's report includes a list of the deviations from the standards and a statistical summary of the findings.

Coverage monitoring: Coverage monitors produce reports about the line coverage achieved when implementing a given test case file. The monitor's output includes the percentage of lines covered by the test cases as well as listings of uncovered lines.

These features make coverage monitoring a vital tool for white-box tests.

Functional tests : Automated functional tests often replace manual black-box correctness tests.

Prior to performance of these tests, the test cases are recorded into the test case database.

The tests are then carried out by executing the test cases through the test program.

The test results documentation includes listings of the errors identified in addition to a variety of summaries and statistics as demanded by the testers' specifications.

*Load tests :*The history of software system development contains many sad chapters of systems that succeeded in correctness tests but severely failed – and caused enormous damage – once they were required to operate under standard full load.

The damage in many cases was extremely high because the failure occurred "unexpectedly", when the systems were supposed to start providing their regular software services.

*Test management :*Testing involves **many participants occupied in actually carrying out the tests and correcting the detected errors**.

Testing typically **monitors performance of every item** on long lists of test case files.

This workload makes timetable follow-up important to management. Computerized test management supports these and other testing management goals.

Advantages of automated tests

(1) Accuracy and completeness of performance.
(2) Accuracy of results log and summary reports.
(3) Comprehensiveness of information.
(4) Few manpower resources required to perform tests.
(5) Shorter duration of testing.
(6) Performance of complete regression tests.
(7) Performance of test classes beyond the scope of manual testing.

Disadvantages of automated testing

1) **High investments required in package purchasing and training.**
2) **High package development investment costs.**
3) **High manpower requirements for test preparation.**
4) **Considerable testing areas left uncovered.**

Frame 10.7 Automated software testing: advantages and disadvantages

Advantages	Disadvantages
1. Accuracy and completeness of performance	1. High investments required in package purchasing and training
2. Accuracy of results log and summary reports	2. High package development investment costs
3. Comprehensive information	3. High manpower resources for test preparation
4. Few manpower resources for test execution	4. Considerable testing areas left uncovered
5. Shorter testing periods	
6. Performance of complete regression tests	
7. Performance of test classes beyond the scope of manual testing	

Alpha and beta site testing programs

- Alpha site and beta site tests are employed to obtain comments about quality from the package's potential users.
- They are additional commonly used tools to identify software design and code errors in software packages in commercial over-the-counter sale (COTS).

Alpha site tests :"Alpha site tests" are tests of a new software package that are performed at the developer's site.

Beta site tests : Once an advanced version of the software package is available, the developer offers it free of charge to one or more potential users.

The users install the package in their sites (usually called the "beta sites"), with the understanding that they will inform the developer of all the errors revealed during trials or regular usage.

REGRESSION TESTING TECHNIQUE

- Regression testing is used to confirm that **previously fixed bugs remain resolved** and that **no new bugs have been introduced**.
- How many cycles of regression testing are required will depend upon the **project size**. Cycles of regression testing may be **performed once per milestone or once per build**. Regression tests can be **automated**.

The Regression–Test Process

The regression-test process is shown in Figure 6.6.

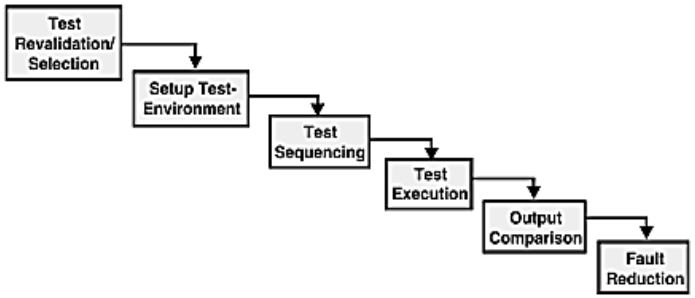


FIGURE 6.6

This process assumes that P′ (modified is program) available for regression testing. There is a long series of tasks that lead to P′ from P.

Quality Audits and Project Assessments

- Humphrey (2005) [HUM 05] : years of data from thousands of software engineers showing that they unintentionally inject 100 defects per thousand lines of code.
- He also indicates that commercial software typically includes from one to ten errors per thousand lines of code [HUM 02].
- These errors are like hidden time bombs that will explode when certain conditions are met.
- Put practices in place to identify and correct these errors at each stage of the development and maintenance cycle.

Cost of quality

Quality costs = Prevention costs
+ Appraisal or evaluation costs
+ Internal and external failure costs
+ Warranty claims and loss of reputation costs

The detection cost is the cost of verification or evaluation of a product or service during the various stages of the development process.

- One of the detection techniques is conducting reviews.
- Another technique is conducting tests.

- Quality of a software product begins in the first stage of the development process (defining requirements and specifications).
- Reviews will detect and correct errors in the early phase of development
- Tests will only be used when the code is available.
- we should not wait for the testing phase to begin to look for errors.
- Reviews range from informal to formal

Informal reviews

An **informal review** lacks structure and standardization, leading to inconsistencies. Key issues include:

- **No documented process** – Different people conduct reviews in different ways.
- **Undefined roles** – No clear responsibilities for participants.
- **No specific objective** – Reviews don't focus on detecting faults.
- **Unplanned** – Conducted without proper scheduling.
- **No defect tracking** – Defects are not measured or recorded.
- **No management oversight** – Effectiveness is not evaluated.
- **No standards or checklists** – No guidelines to follow for identifying defects.

Review

A process or meeting during which a work product, or set of work products, is presented to project personnel, managers, users, customers, or other interested parties for comment or approval.

ISO 24765 [ISO 17a]

A process or meeting during which a software product, set of software products, or a software process is presented to project personnel, managers, users, customers, user representatives, auditors, or other interested parties for examination, comment, or approval.

IEEE 1028 [IEE 08b]

IEEE 1028 Standard

Types of Reviews in Software Engineering:

1. **Walk-through & Inspection:**
   1. Formal review methods to identify defects and improve quality.
2. **Personal Review & Desk Check:**
   1. Individual verification of code, documents, or design before peer review.
3. **Peer Reviews:**
   1. Conducted by colleagues during development, maintenance, or operations.
   2. Aims to identify errors, present alternatives, and discuss solutions.

✅ **Purpose:** Improve software quality by detecting defects early and enhancing collaboration.

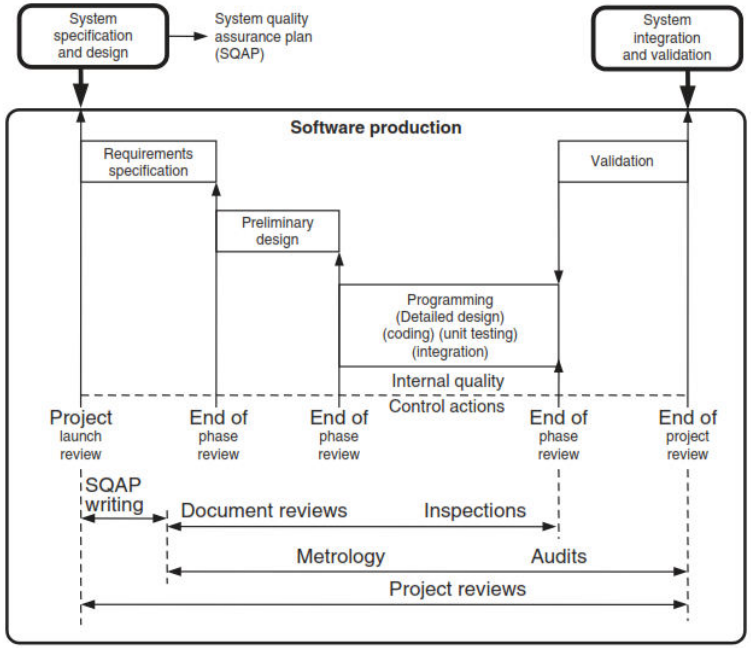


Figure 5.1 Types of reviews used during the software development cycle [CEG 90] (© 1990 – ALSTOM Transport SA).

- Identify defects
- Assess / measure the quality of a document (e.g., the number of defects per page)
- Reduce the number of defects by correcting the defects identified
- Reduce the cost of preparing future documents (i.e., by learning the type of defects each developer makes, it is possible to reduce the number of defects injected in a new document)
- Estimate the effectiveness of a process (e.g., the percentage of fault detection)
- Estimate the efficiency of a process (e.g., the cost of detection or correction of a defect)
- Estimate the number of residual defects (i.e., defects not detected when software is delivered to customer)
- Reduce the cost of tests
- Reduce delays in delivery
- Determine the criteria for triggering a process
- Determine the completion criteria of a process
- Estimate the impacts (e.g., cost) of continuing with current plans, e.g. cost of delay, recovery, maintenance, or fault remediation
- Estimate the productivity and quality of organizations, teams and individuals
- Teach personnel to follow the standards and use templates
- Teach personnel how to follow technical standards
- Motivate personnel to use the organization's documentation standards
- Prompt a group to take responsibility for decisions
- Stimulate creativity and the contribution of the best ideas with reviews
- Provide rapid feedback before investing too much time and effort in certain activities
- Discuss alternatives
- Propose solutions, improvements
- Train staff
- Transfer knowledge (e.g., from a senior developer to a junior)
- Present and discuss the progress of a project
- Identify differences in specifications and standards
- Provide management with confirmation of the technical state of the project
- Determine the status of plans and schedules
- Confirm requirements and their assignment in the system to be developed

Figure 5.2 Objectives of a review.

Figure 5.3 Process of developing a document.

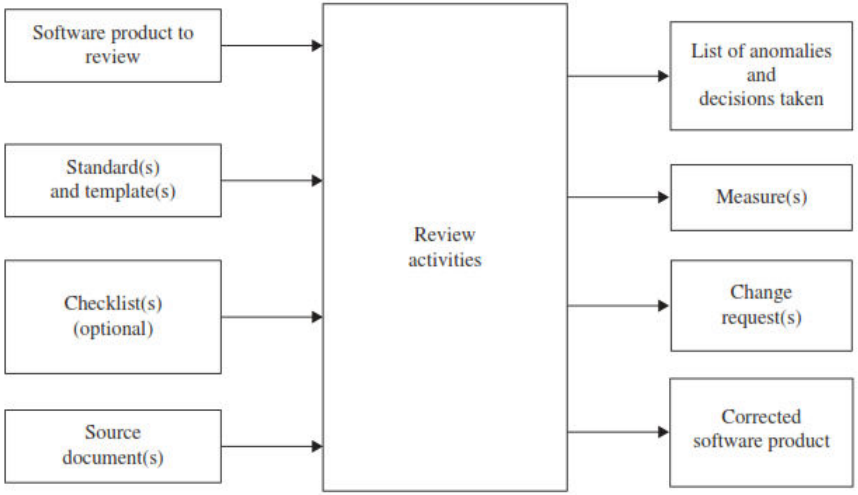


Figure 5.4 Review process.

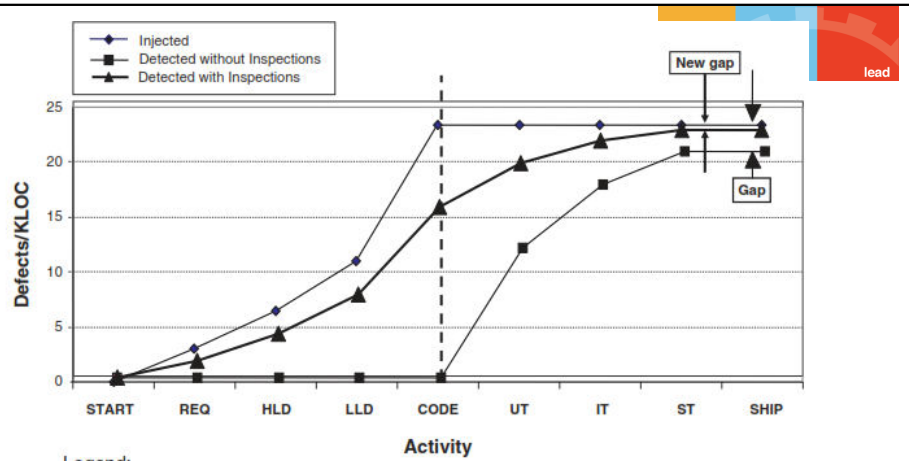
PERSONAL REVIEW AND DESK-CHECK REVIEW

- Inexpensive and very easy to perform.
- Personal reviews do not require the participation of additional reviewers,
- Desk-check reviews require at least one other person to review the work of the developer of a software product.

Personal Review

Done by the person reviewing his own software product in order to find and fix the most defects possible.

- Principles of a personal review
  - Find and correct all defects in the software product.
  - Use a checklist produced from your personal data, if possible, using the type of defects that you are already aware of.
  - Follow a structured review process;
  - Use measures in your review;
  - Use data to improve your review;
  - Use data to determine where and why defects were introduced and then change your process to prevent similar defects in the future.



Legend:
- REQ = requirement
- HLD = high-level architecture design
- LLD = detailed design
- CODE = coding and debugging
- UT = unit testing
- IT = integration testing
- ST = system testing
- SHIP = delivery to customer
- KLOC = thousand lines of code

Figure 5.5 Error detection during the software development life cycle [RAD 02].

Checklist

A checklist is used as a memory aid. A checklist includes a list of criteria to verify the quality of a product. It also ensures consistency and completeness in the development of a task. An example of a checklist is a list that facilitates the classification of a defect in a software product (e.g., an oversight, a contradiction, an omission).

Practices - to develop an effective and efficient personal review

- Pause between the development of a software product and its review.
- Examine products in hard copy rather than electronically.
- Check each item on the checklist.
- Update the checklists periodically to adjust to your personal data.
- Build and use a different checklist for each software product.
- Verify complex or critical elements with an in depth analysis.

ENTRY CRITERIA
- None

INPUT
- Software product to review

ACTIVITIES
1. Print:
- Checklist for the software product to be reviewed
- Standard (if applicable)
- Software product to review

2. Review the software product, using the first item on the checklist and cross this item off when the review of the software product is completed
3. Continue review of the software product using the next item on the checklist and repeat until all the items in the list have been checked
4. Correct any defects identified
5. Check that each correction did not create other defects.

EXIT CRITERIA
- Corrected software product

OUTPUT
- Corrected software product

MEASURE
- Effort used to review and correct the software product measured in person-hours with an accuracy of +/–15 minutes.

Figure 5.6 Personal review process.

Source: Adapted from Pomeroy-Huff et al. (2009) [POM 09].

THANK YOU

Test Plan

1. **Objectives** – **Defines the purpose of testing and what it aims to achieve.**
   - *Example:* Ensuring that the **login functionality works correctly** for all user roles.
2. **Scope** – **Specifies what will be tested and what will not be covered.**
   - *Example:* Testing only the web version of an application, excluding mobile apps.
3. **Test Items** – **Identifies the components, modules, or features to be tested.**
   - *Example:* Testing the **checkout process in an e-commerce application.**
4. **Test Environment** – **Defines the hardware, software, and network configurations required for testing.**
   - *Example:* A test server running **Windows 11 with a MySQL database**.
5. **Testing Strategy** – **Describes the approach, techniques, and levels of testing to be used.**
   - *Example:* Using automated regression testing and manual exploratory testing.
6. **Test Schedule** – **Specifies timelines, milestones, and deadlines for test execution.**
   - *Example:* Unit testing from March 1–5, system testing from March 6–10.
7. **Resource Requirements** – Lists the **personnel, tools, and infrastructure needed for testing.**
   - *Example:* Requiring 3 testers, Selenium automation tool, and a dedicated test server.
8. **Risk Management** – **Identifies potential risks that could impact testing and mitigation strategies.**
   - *Example:* Risk: Delayed development could shorten testing time. Mitigation: Prioritize critical test cases.
9. **Test Deliverables** – **Defines the documents, reports, and output produced during testing.**
   - *Example:* Test plan document, defect report, test summary report.

Software Test Plan (STP) Template

1. Scope of the Tests

Defines what will be tested, including:

- The software package (name, version, and revision).
- Relevant documents supporting the test plan.

2. Testing Environment

Specifies the setup needed for testing:

- Testing locations/sites.
- Hardware and firmware requirements.
- Involved organizations and manpower needs.
- Training and preparation for the test team.

3. Test Details (for each test)

Includes specific details of each test case:

- **Test identification & objective** – Defines the purpose of the test.
- **Reference documents** – Links to design and requirement documents.
- **Test class & level** – Identifies whether it's unit, integration, or system testing.
- **Test case requirements** – Lists conditions for executing test cases.
- **Special requirements** – Factors like response time, security, or performance metrics.
- **Data recording** – Information to be logged during tests.

4. Test Schedule

Provides estimated time for various testing phases:

- **Preparation** – Setting up resources and test environments.
- **Testing** – Executing test cases.
- **Error correction** – Debugging and fixing detected issues.
- **Regression testing** – Retesting to ensure fixes do not introduce new defects.

This structured approach ensures a **systematic and well-documented** testing process for software projects.

Software Test Descriptions (STD) – Template,

1. Scope of the Tests

- Defines the **software package** to be tested (name, version, revision).
- Lists the **documents** that form the basis of the test design.

2. Test Environment (for each test)

- Identifies the **test case details** (linked to the Software Test Plan - STP).
- Describes the **operating system and hardware configuration** required.
- Provides **instructions for software loading**.

3. Testing Process

- Step-by-step **input instructions** for the test.
- Specifies **data to be recorded** during the test execution.

4. Test Cases (for each case)

- Includes **test case identification details**.
- Lists **input data and system settings**.
- Defines **expected intermediate results** (if applicable).
- Specifies **expected final results** (numerical, messages, system behavior, etc.).

5. Actions to Be Taken in Case of Program Failure/Cessation

- Defines contingency actions when failures occur.

6. Procedures to Be Applied According to the Test Results Summary

- Describes the **post-test procedures**, including result evaluation and reporting.

This template helps ensure **consistency and completeness** in software testing documentation.

Software Test Report (STR)

A **Software Test Report (STR)** is a **comprehensive document** that records the **test execution details**, **results, and observations**.

---

1. Test Identification, Site, Schedule, and Participation

This section documents the **fundamental details** of the test, including the **software, location, team, and effort involved**.

Example:

- **1.1 Tested Software Identification** – Software: **XYZ Application v2.0 (Build 1021)**
- **1.2 Documents Providing Test Basis** – Test Plan, Requirement Specification Document
- **1.3 Test Site** – Testing conducted in **Lab Environment – Server 192.168.1.10**
- **1.4 Test Duration** – Start: **March 10, 2025, 9:00 AM** | End: **March 12, 2025, 5:00 PM**
- **1.5 Test Team Members** – John (Lead Tester), Alice (Automation Engineer), Bob (Manual Tester)
- **1.6 Other Participants** – Developers and Product Managers
- **1.7 Hours Invested in Testing** – **15 hours total**

---

2. Test Environment

This section describes the **hardware, software, and preparation** needed before testing.

Example:

- **2.1 Hardware & Firmware Configuration**
  - Server: **Intel Xeon 8-core, 32GB RAM, 1TB SSD**
  - Database: **PostgreSQL v13**
  - Operating System: **Windows Server 2019**
- **2.2 Preparations & Training Prior to Testing**
  - Installed **latest software build** on test server.
  - Set up **test accounts and access credentials**.
  - Conducted **training session** for new testers.

---

3. Test Results

This section provides **test execution details** including **test case outcomes** and failures.

Example:

3.1 Test Identification

- Test Name: **Login Functionality Test**
- Test ID: **TC-001**

3.2 Test Case Results

Test Case ID	Tester	Expected Result	Actual Result	Status
TC-001	Alice	User logs in successfully	Login successful	✅ Pass
TC-002	Bob	Incorrect password shows error	Error displayed	✅ Pass
TC-003	John	Forgot Password link redirects	**Page not found error**	❌ Fail

- **Failed Test Case Details (TC-003):**
  - **Issue**: Clicking the "Forgot Password" link shows a **404 Page Not Found** error.
  - **Logs/Error Code**: Error: HTTP 404 - Page Not Found
  - **Recommendation**: Fix the **broken link to Forgot Password page** in the next release.

---

4. Summary Tables for Errors & Regression Testing

This section **summarizes errors found** and compares results with previous test cycles.

Example:

- **4.1 Summary of Current Test Execution**
  - **Total Test Cases Executed**: **50**
  - **Passed**: **45**
  - **Failed**: **5**
  - **Skipped**: **0**
- **4.2 Comparison with Previous Results (Regression Testing)**
  - Previous Cycle Failures: **7**
  - Current Cycle Failures: **5** (✅ **Improvement**)
  - **Issues Resolved from Last Test**: Login page responsiveness fixed.

---

5. Special Events & Testers' Proposals

Records **unexpected behaviours, problems, and improvement suggestions**.

Example:

- **5.1 Special Events & Unexpected Software Responses**
  - **During peak load testing**, the application **crashed** when handling 10,000 concurrent users.
  - The **API failed** when too many requests were sent in a short time, causing **timeouts.**
- **5.2 Problems Encountered During Testing**
  - **Database response time was slow** for queries returning more than **100,000 records**.
  - **Some UI elements did not load correctly** in Safari browser.
- **5.3 Test Environment & Process Improvement Suggestions**
  - **Upgrade the test servers** to handle **high-load scenarios**.
  - Implement **automated test scripts** for **repetitive functional testing**.
- **5.4 Suggested Changes in Test Cases & Procedures**
  - Add **stress testing scenarios** to check how the system performs under high load.
  - Improve test documentation for **new team members**.

Test case design Example

Explanation of the Municipal Property Tax (MPT) Calculation with Test Cases

The municipal property tax (MPT) is calculated based on three factors:

1. **S** = Size of the apartment (in square yards).
2. **N** = Number of people living in the apartment.
3. **A, B, or C** = Suburb classification, which determines the tax rate.

Formula for MPT Calculation

- **Class A**: MPT = **(100 × S) / (N + 8)**
- **Class B**: MPT = **(80 × S) / (N + 8)**
- **Class C**: MPT = **(50 × S) / (N + 8)**

Applying the Formula to the Given Test Cases

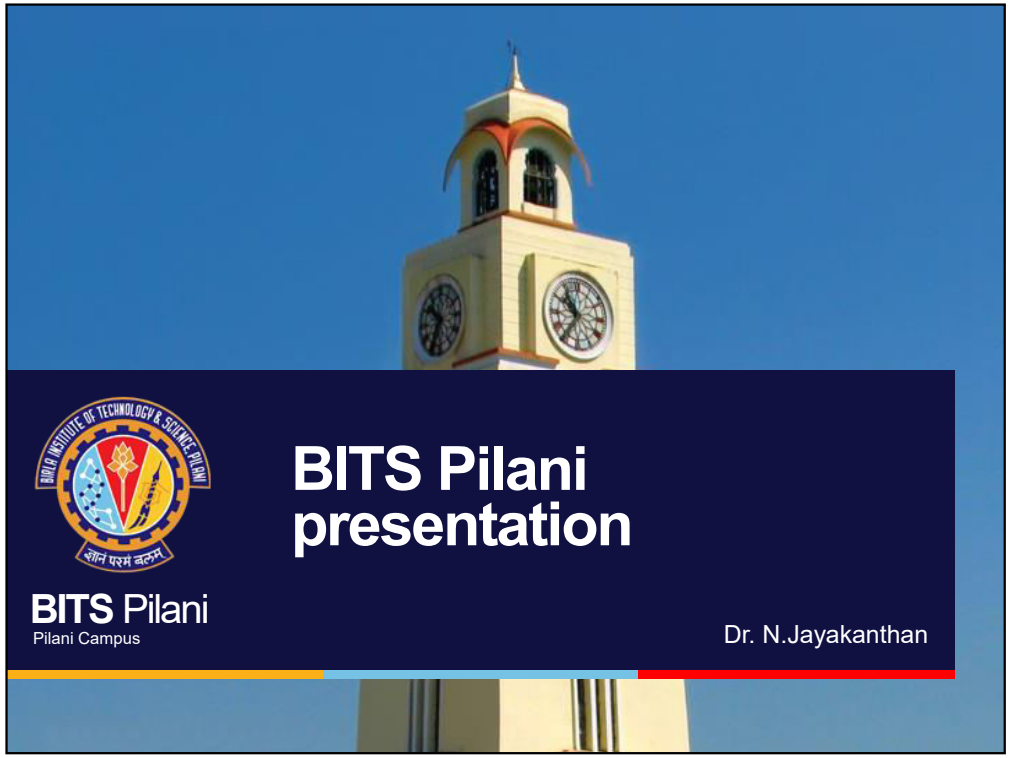
Test Case	Size of Apartment (S)	Suburb Class	No. of People (N)	Calculation	MPT (Expected Result)
Test Case 1	250	A	2	(100 × 250) / (2 + 8) = **25000 / 10**	**$2500**
Test Case 2	180	B	4	(80 × 180) / (4 + 8) = **14400 / 12**	**$1200**
Test Case 3	98	C	6	(50 × 98) / (6 + 8) = **4900 / 14**	**$350**

Key Observations

- ✅ **Higher S value = Higher MPT** (Larger apartments pay more tax).
- ✅ **Higher N value = Lower MPT** (More people in a house reduce tax per person).
- ✅ **Suburb Class Impact**:
  - **Class A** (highest rate) → $2500 tax for 250 sq. yards.
  - **Class B** (moderate rate) → $1200 tax for 180 sq. yards.
  - **Class C** (lowest rate) → $350 tax for 98 sq. yards.

Note:

The term **"/ (N + 8)"** in the municipal property tax (MPT) formula represents **division** by the sum of the number of persons living in the apartment (**N**) plus 8. This denominator is used to normalize the tax amount based on household size.



Practices - to develop an effective and efficient personal review

- Pause between the development of a software product and its review.
- Examine products in hard copy rather than electronically.
- Check each item on the checklist once completed.
- Update the checklists periodically to adjust to your personal data.
- Build and use a different checklist for each software product;
- Verify complex or critical elements with an in depth analysis.

ENTRY CRITERIA
- None

INPUT
- Software product to review

ACTIVITIES
1. Print:
- Checklist for the software product to be reviewed
- Standard (if applicable)
- Software product to review

2. Review the software product, using the first item on the checklist and cross this item off when the review of the software product is completed
3. Continue review of the software product using the next item on the checklist and repeat until all the items in the list have been checked
4. Correct any defects identified
5. Check that each correction did not create other defects.

EXIT CRITERIA
- Corrected software product

OUTPUT
- Corrected software product

MEASURE
- Effort used to review and correct the software product measured in person-hours with an accuracy of +/–15 minutes.

Figure 5.6 Personal review process.
Source: Adapted from Pomeroy-Huff et al. (2009) [POM 09].

Desk-Check Reviews

- A type of peer review that is not described in standards is the desk-check review (Pass around) .

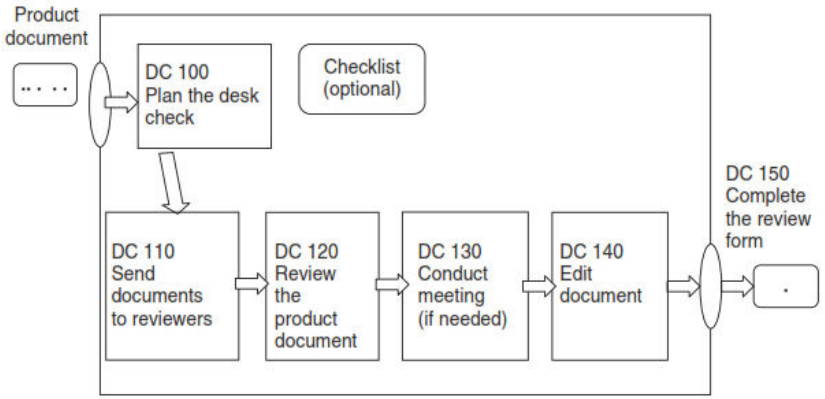


Figure 5.7 Desk-check review.

There are six steps.

- Initially, the author plans the review by identifying the reviewer(s) and a checklist.
- A checklist is an important element of a review as it enables the reviewer to focus on only one criterion at a time.
- A checklist is a reflection of the experience of the organization.
- Then, individuals review the software product document and note comments on the review form provided by the
- author.
- When completed, the review form can be used as "evidence" during an audit.

Important features of checklists:

- Each checklist is designed for a specific type of document (e.g., project plan, specification document).
- Each item of a checklist targets a single verification criteria.
- Each item of a checklist is designed to detect major errors. Minor errors, such as misspellings, should not be part of a checklist.
- Each checklist should not exceed one page, otherwise it will be more difficult to use by the reviewers.
- Each checklist should be updated to increase efficiency.
- Each checklist includes a version number and a revision date.

- The following text box presents a generic checklist, that is, a checklist that can be used for almost any type of document to be reviewed (e.g., project plan, architecture).
- For each type of software product (e.g., requirements or design), a specific checklist will be used.

Generic Checklist

LG 1 (COMPLETE). All pertinent information should be included or referenced.
LG 2 (RELEVANT). All information must be relevant to the software product.
LG 3 (BRIEF). Information must be stated succinctly.
LG 4 (CLEAR). Information must be clear to all reviewers and users of the document.
LG 5 (CORRECT). Information does not contain errors.
LG 6 (COHERENT). Information must be consistent with all other information in the document and its source document(s).
LG 7 (UNIQUE). Ideas must be described once and referenced afterward.

Adapted from Gilb and Graham (1993) [GIL 93]

- In the **third step of the desk-check process**, the reviewers verify the document and record their comments on the review form.
- The **author reviews the comments as part of step 4**.
- If the author agrees with all the comments, he incorporates them into his document. **After this meeting, one of three options should be considered:** the comment is incorporated as is, the comment is ignored, or it is incorporated with modifications.

Next step, the author can make the corrections and note the effort spent reviewing and correcting the document, that is, the time spent by the reviewers as well as the time spent by the author to correct the document and conduct the meeting if this is the case.

In the final step, the author completes the review form illustrated in Figure 5.9.

ENTRY CRITERIA
- The document is ready for a review

INPUT
- Software product to review

DC 100. Plan the Desk-Check
Author:
- Identifies reviewers
- Chooses the checklist(s) to use
- Completes the first part of the review form

DC 110. Send documents to reviewers
Author:
- Provides the following documents to the reviewers:
  - Software product to review
  - Review form
  - Checklist(s)

DC 120. Review the software product
Reviewers:
- Check the software product against the checklist
- Complete the review form with
  - comments
  - effort to conduct the review
- Sign and return the form to the author

DC 130. Call a meeting (if needed)
Author:
- Reviews the comments
  - If the author agrees with all the comments, they are incorporated in the software product
  - If the author does not agree with all the comments, or believes some comments have a significant impact, then the author:
    - Convenes a meeting with the reviewers
    - Leads the meeting to discuss the comments and determine course of action:
      - Incorporate the comment as is
      - Ignore the comment
      - Incorporate the comment with modifications

DC 140. Correct the software product
The author incorporates the comments received.
DC 150. Complete the review form
Author:
- Completes the review form with:
  - Total effort (i.e., by all the reviewers) required to review the software product
  - Total effort required to correct the software product
- Signs the review form

EXIT CRITERIA
- Corrected software product

OUTPUT
- Corrected software product
- Completed and signed review form

MEASURE
- Effort required to review and correct the software product (person hours).

Figure 5.8 Desk-check review activities.

Desk check review form

Name of author: _____ Review date (yyyy-mm-dd): _____
Document title: _____ Reviewer name: _____
Document version: _____

Comment No.	Document page	Line # / location	Comments	Disposition of comments*	Remarks
1					
2					
3					
4					
5					
6					
7					
8					
9					
10					
11					
12					
13					
14					
15					
16					
17					
18					
19					
20					
21					
22					
23					
24					

Disposition of comment: Inc: Incorporate as is; NOT: Not incorporate, MOD: Incorporate with modification

Effort to review document (hour): _____
Effort to correct document (hour): _____

Signature of reviewer: _____

Signature of author: _____

Figure 5.9 Desk-Check review form.

The IEEE 1028 Standard

The **IEEE 1028-2008 Standard** defines **five types of software reviews and audits**, detailing their purpose and process.

It provides guidelines on **how to conduct systematic reviews** during **software acquisition, development, operation, and maintenance** to ensure quality and compliance.

- ✓ Defines **what to review** and **how to review it**
- ✓ Ensures **software quality, correctness, and reliability**
- ✓ Covers **processes for structured reviews and audits**

This standard provides descriptions of the particular types of reviews and audits included in the standard as well as tips.

a) Introduction to review: describes the objectives of the systematic review and provides an overview of the systematic review procedures;

b) Responsibilities: defines the roles and responsibilities needed for the systematic review.

c) Input: describes the requirements for input needed by the systematic review;

d) Entry criteria: describes the criteria to be met before the systematic review can begin, including the following:
  1) Authorization,
  2) Initiating event;

e) Procedures: details the procedures for the systematic review, including the following:
  1) Planning the review;
  2) Overview of procedures;
  3) Preparation;
  4) Examination/evaluation/recording of results;
  5) Rework/follow-up;

f) Exit criteria: describe the criteria to be met before the systematic review can be considered complete;

g) Output: describes the minimum set of deliverables to be produced by the systematic review.

IEEE 1028 -The types of reviews and audits

1. **Management Review**
   - Evaluates project **progress, schedules, and requirements.**
   - Conducted by management to ensure alignment with goals.
2. **Technical Review**
   - Assesses software for **functional suitability and compliance.**
   - Performed by technical experts to detect specification gaps.
3. **Inspection**
   - A **formal, structured** review to **identify errors and deviations.**
   - Involves step-by-step checks against standards.
4. **Walkthrough**
   - The author **presents the software** to the team.
   - Participants **ask questions, give feedback, and suggest improvements.**
5. **Audit**
   - An **independent assessment** for **standards and contract compliance.**
   - Ensures the software meets regulatory and quality requirements.

WALK-THROUGH

- The purpose of a walk-through is to evaluate a software product.
- A walk-through can also be performed to create discussion for a software product"
- Objectives of the walk-through:
  - Find anomalies.
  - Improve the software product.
  - Consider alternative implementations.
  - Evaluate conformance to standards and specifications.
  - Evaluate the usability and accessibility of the software product.

Usefulness of a Walk-Through

- Identify errors to reduce their impact and the cost of correction.
- Improve the development process.
- Improve the quality of the software product.
- Reduce development costs.
- Reduce maintenance costs.

Table: Characteristics of Reviews and Audits (IEEE 1028 Standard)

Characteristic	Management Review	Technical Review	Inspection	Walk-through	Audit
Objective	Monitor progress	Evaluate conformance to specifications and plans	Find anomalies, verify resolution, ensure product quality	Identify anomalies, examine alternatives, improve product	Independently verify compliance with standards and regulations
Recommended Group Size	2 or more people	2 or more people	3-6	2-7	1-5
Volume of Material	Moderate to High	Moderate to High	Relatively low	Relatively low	Moderate to High
Leadership	Responsible manager	Lead engineer	Trained facilitator	Facilitator or author	Lead auditor
Management Participation	Yes	Sometimes, if evidence is required	No	No	No, but may be called for evidence
Output	Management review documentation	Technical review documentation	Anomaly list, summary, inspection report	Anomaly list, action items, follow-up proposals	Formal audit report, findings, deficiencies

INSPECTION REVIEW

This section briefly describes the inspection process that Michael Fagan developed at IBM in the 1970s to increase the quality and productivity of software development.

The purpose of the inspection, according to the IEEE 1028 standard, is to detect and identify anomalies of a software product including errors and deviations from standards and specifications.

Throughout the development or maintenance process, developers prepare written materials that unfortunately have errors. It is more economical and efficient to detect and correct errors as soon as possible. Inspection is a very effective method to detect these errors or anomalies.

According to the IEEE 1028 standard, inspection allows us to

a) verify that the software product satisfies its specifications;

b) check that the software product exhibits the specified quality attributes;

c) verify that the software product conforms to applicable regulations, standards, guidelines, plans, specifications, and procedures;

d) identify deviations from provisions of items (a), (b), and (c);

e) collect data, for example, the details of each anomaly and effort associated with their identification and correction;

f) request or grant waivers for violation of standards where the adjudication of the type and extent of violations are assigned to the inspection jurisdiction;

g) use the data as input to project management decisions as appropriate (e.g., to make trade-offs between additional inspections versus additional testing).

Major steps of the inspection process, Each step is composed of a series of inputs, tasks and outputs.

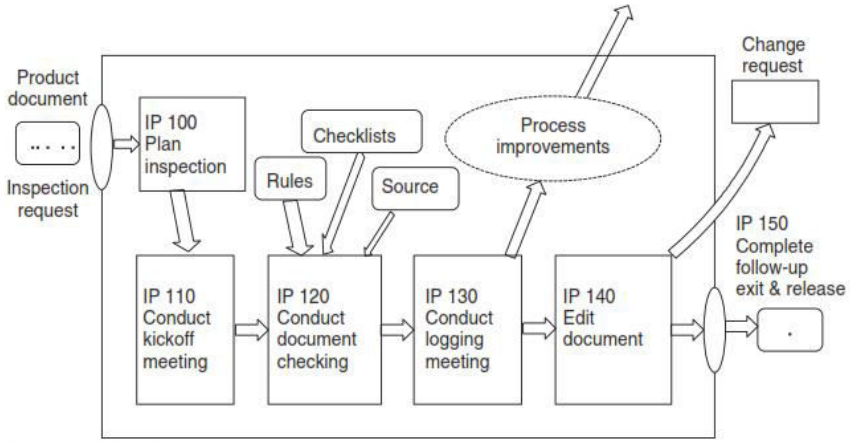


Figure 5.12 The inspection process.

PROJECT LAUNCH REVIEWS AND PROJECT ASSESSMENTS

In the SQAP of their projects, many organizations plan a project launch or kick-off meeting as well as a project assessment review, also called a lessons learned review.

QUIZ

1. What is the main purpose of a **desk-check review**?

   A) To test software functionality.

   B) To pass around a document for individual review and collect feedback.

   C) To perform automated code testing.

   D) To execute the software and detect runtime errors.

What is the main purpose of a **desk-check review**?

A) To test software functionality.

B) To pass around a document for individual review and collect feedback.

C) To perform automated code testing.

D) To execute the software and detect runtime errors.

Answer: B) To pass around a document for individual review and collect feedback

QUIZ

2. What are the **major benefits** of conducting an **inspection review**? *(Select all that apply.)*

   A) Detect and identify anomalies early in the process
   B) Improve software quality
   C) Reduce development and maintenance costs
   D) Automate software debugging

What are the **major benefits** of conducting an **inspection review**? *(Select all that apply.)*

A) Detect and identify anomalies early in the process
B) Improve software quality.
C) Reduce development and maintenance costs.
D) Automate software debugging.

Correct Answers: A, B, C

Project Launch Review

The project launch review is a management review of: the milestone dates, requirements, schedule, budget constraints, deliverables, members of the development team, suppliers, etc.

Some organizations also conduct kick-off reviews at the beginning of each of the major phases of the project when projects are spread over a long period of time

Before the start of a project, team members ask themselves the following questions:

- Who will the members of my team ?
- Who will be the team leader?
- What will my role and responsibilities ?
- What are the roles of the other team members and their responsibilities?
- Do the members of my team have all the skills and knowledge to work on this project?

MEASURES

Measures are mainly used to answer the following questions:

- How many reviews were conducted?
- What software products have been reviewed?
- How effective were the reviews (e.g., number of errors detected by number of hours for the review)?
- How efficient were the reviews (e.g., number of hours per review)?
- What is the density of errors in software products?
- How much effort is devoted to reviews?
- What are the benefits of reviews?

The measures that allow us to answer these questions are:

- Number of reviews held.
- Identification of the revised software product.
- Size of the software product (e.g., number of lines of code, number of pages);
- Number of errors recorded at each stage of the development process;
- Effort assigned to review and correct the defects detected.

SELECTING THE TYPE OF REVIEW

To determine the type of review and its frequency, the criteria to be considered are:

- The risk associated with the software to be developed.
- The criticality of the software.
- Software complexity.
- The size and experience of the team,
- The deadline for completion, and software size.

Table 5.5 Example of a Matrix for the Selection of a Type of Review

	Technical drivers—complexity		
Product	Low	Medium	High
Software requirements	Walk-through	Inspection	Inspection
Design	Desk-check	Walk-through	Inspection
Software code and unit test	Desk-check	Walk-through	Inspection
Qualification test	Desk-check	Walk-through	Inspection
User/operator manuals	Desk-check	Desk-check	Walk-through
Support manuals	Desk-check	Desk-check	Walk-through
Software documents, for example, Version Description Document (VDD), Software Product Specification (SPS), Software Version Description (SVD)	Desk-check	Desk-check	Desk-check
Planning documents	Walk-through	Walk-through	Inspection
Process documents	Desk-check	Walk-through	Inspection

Tools for conducting Reviews

- **dutils** – Finds and organizes program identifiers.
- **Egrep** – Searches text using patterns.
- **Find** – Locates files on a system.
- **Diff** – Compares files to find differences.
- **Cscope** – Helps browse C code.
- **LXR** – Shows source code online with cross-referencing.

AUDITS

Audits are one of the most formal types of reviews in software quality assurance.

Different types of audits serve **different purposes**, such as:

- Ensuring a software company follows industry standards.
- Verifying that a supplier meets a client's requirements.

The **cost and independence of the auditor** depend on the type of audit being conducted.

This helps organizations **maintain compliance, improve quality, and build trust** with clients.

Key Components of a Software Audit

Management System – Defines policies, objectives, and processes for quality compliance.

Audit Criteria – Standards or policies used to evaluate compliance (e.g., ISO 9001, CMMI).

Audit Evidence – Documents, reports, and test results proving adherence to audit criteria.

Audit Process – Steps include planning, evidence collection, evaluation, reporting, and corrective actions.

Internal audits

- Called first party audits
- Conducted by the organization itself, or on its behalf, for management review and other internal purposes (e.g. to confirm the effectiveness of the management system or to obtain information for the improvement of the management system).
- Internal audits can form the basis for an organization's self-declaration of conformity.

External audits

- Includes second and third party audits.
- Second party audits are conducted by parties having an interest in the organization, such as customers, or by other persons on their behalf.
- Third party audits are conducted by independent auditing organizations, such as regulators or those providing certification.

There are different types of audit:

– audits to verify the compliance to a standard, such as the audits described in International Organization for Standardization (ISO) standards such as ISO 9001 and IEEE 1028;

– Compliance audits for a model such as the Capability Maturity Model Integration (CMMI) that are used to select a supplier before awarding a contract or assess a supplier during a contract;

– Audits ordered by the management team of the organization to verify the progress of a project against its approved plan.

Why audit?

Software project audits are usually requested by management to ensure that the software team and contracted suppliers:

- know their duties and obligations toward the public, their employers, their customers, and their colleagues;
- use the processes, practices, techniques, and methods suggested by the company;
- reveal any deficiencies and shortcomings in daily operations and try to identify required corrective actions (CAs);
- are encouraged to develop a personal training plan for their professional skills;
- are monitored in the course of their work on high profile projects of the company.

TYPES OF AUDITS

Internal Audit

- A **first-party audit** is an internal audit conducted by an organization to assess its own processes,
- First-party audit, can be useful for a software supplier wanting to obtain an ISO 9001 certification.
- It is the least expensive approach to prepare for conformity to an international standard.

Second-Party Audit

- conducted by parties having an interest in the organization, such as customers, or by other persons on their behalf.

Third party audits are conducted by independent auditing organizations, such as regulators or those providing certification.

- ISO does not provide certification services or issue certificates.
- Instead, certification bodies use standards like ISO 19011 for auditing guidelines and ISO/IEC 17021-1 for certification requirements.
- These standards ensure compliance with certifications like ISO 9001.

Project Assessment and Control Process

- **Align and Validate Plans:** Ensure project **plans are feasible and aligned with business objectives**.
- **Monitor Progress:** Assess the **current status of the project, technical performance, and process efficiency**.
- **Guide Execution:** Ensure the **project stays on schedule, within budget, and meets technical goals**.
- **Mandatory Reviews:** Conduct **management and technical reviews, audits, and inspections to verify compliance and performance**

CORRECTIVE ACTIONS

- After an internal or external audit, an organization must perform CAs **to correct the observed deficiencies**.
- The CA (Corrective Action) process can also be used to **address preventive actions, incident reports, and customer complaints**.
- The CA aims at **eliminating potential causes of non-conformity, a defect, or any other adverse event** to prevent their recurrence.
- This process should cover **software products, agreements and software development plans**.

Corrective Action

Action to eliminate the cause of a nonconformity and to prevent recurrence.

Note 1: There can be more than one cause for a nonconformity.

Note 2: Corrective action is taken to prevent recurrence whereas preventive action is taken to prevent occurrence.

ISO 9000 [ISO 15b]

An intentional activity that realigns the performance of the project work with the project management plan.

PMBOK® Guide [PMI 13]

Preventive Action

An intentional activity that ensures the future performance of the project work is aligned with the project management plan.

PMBOK® Guide [PMI 13]

Corrective Actions Process

- The problems encountered when developing systems that include software, or that occur during their operation, can come from defects in the software, in the development process itself, or in the hardware of the system.
- To facilitate the identification of problem sources and apply appropriate CAs, it is desirable that a **centralized system is developed to track issues through to resolution and to determine their root cause.**

A CA process, in a closed loop, may include the following elements:

Inputs:

- Audit report
- Non-compliance issue
- Problem report

Activities:

1. **Register non-conformities** in the organization's issue tracking tool.
2. **Analyze and validate** the problem to ensure resources are not wasted.
3. **Classify and prioritize** the issue/problem based on impact.
4. **Conduct trend analysis** to identify recurring problems and address them proactively.

Problem Resolution Process

Steps to Address the Problem:

1. **Propose a Solution** – Identify and suggest a fix for the problem.
2. **Solve the Problem** – Implement the solution and ensure it does not create new issues.
   - If the problem cannot be resolved within the project, escalate it to the appropriate management level.
3. **Verify the Resolution** – Check that the issue has been fully resolved.
4. **Inform Stakeholders** – Communicate the resolution to all relevant parties.
5. **Archive Documentation** – Store records of the problem and its resolution for future reference.
6. **Update the Issue Tracking Tool** – Ensure the tracking system reflects the latest resolution status.

Outputs:

- **Resolution File** – A documented report of the issue and how it was resolved.
- **Corrected Software Version** – An updated version of the software with the issue fixed.

Problem report

Priority:_____ Project name: _____ Date: _____

Process name: _____ Phase number: ___Raised by: _____

Number of days to answer: _____ Close date: _____

Number of days to fix this problem: _____

Finding: _____

Requirement/Standard impacted:

_____

Immediate solution proposed:

_____

Root cause: _____

Permanent solution proposed:

_____

Acceptance date of permanent solution: _____

Follow-up action (if necessary):

_____

Figure 6.5 Problem report and resolution proposal form.

THANK YOU

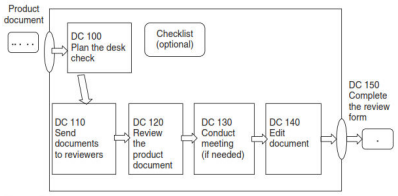


Figure 5.7 Desk-check review.

A **Desk-Check Review** is an informal software review process used to identify errors in a product document. The process includes:

1. **DC 100 - Plan the Desk Check**: Define review objectives and participants.
2. **DC 110 - Send Documents**: Distribute the product document to reviewers.
3. **DC 120 - Review Product Document**: Reviewers examine the document for errors.
4. **DC 130 - Conduct Meeting (if needed)**: Discuss findings and clarify issues.
5. **DC 140 - Edit Document**: The author applies necessary corrections based on feedback.
6. **DC 150 - Complete Review Form**: Finalize and document review results.

Desk Check Review Form.

ENTRY CRITERIA

- The document is ready for a review

INPUT

- Software product to review

DC 100. Plan the Desk-Check

Author:

- Identifies reviewers
- Chooses the checklist(s) to use
- Completes the first part of the review form

DC 110. Send documents to reviewers

Author:

- Provides the following documents to the reviewers:
  - Software product to review
  - Review form
  - Checklist(s)

DC 120. Review the software product

Reviewers:

- Check the software product against the checklist
- Complete the review form with:
  - Comments
  - Effort to conduct the review
- Sign and return the form to the author

DC 130. Call a meeting (if needed)

Author:

- Reviews the comments
  - If the author agrees with all the comments, they are incorporated into the software product
  - If the author does not agree with all the comments, or believes some comments have a significant impact, then the author:
    - Convenes a meeting with the reviewers
    - Leads the meeting to discuss the comments and determine the course of action:
      - Incorporate the comment as is
      - Ignore the comment
      - Incorporate the comment with modifications

DC 140. Correct the software product

- The author incorporates the comments received.

DC 150. Complete the review form

Author:

- Completes the review form with:
  - Total effort (i.e., by all the reviewers) required to review the software product
  - Total effort required to correct the software product
- Signs the review form

EXIT CRITERIA

- Corrected software product

OUTPUT

- Corrected software product
- Completed and signed review form

MEASURE

- Effort required to review and correct the software product (person hours).

Systematic Review Process

This document outlines the **steps for conducting a systematic review**.

1. **Introduction** – Explains the purpose and objectives of the review.
2. **Responsibilities** – Defines the roles of people involved in the review.
3. **Input** – Lists the required documents or data needed for the review.
4. **Entry Criteria** – Conditions that must be met before the review starts:
   - **Authorization** – Approval to conduct the review.
   - **Initiating Event** – A trigger that starts the review (e.g., project milestone).
5. **Procedures** – Steps for conducting the review:
   - **Planning** – Organizing the review process.
   - **Overview of Procedures** – Defining how the review will be carried out.
   - **Preparation** – Gathering necessary documents and resources.
   - **Examination/Evaluation** – Reviewing and recording findings.
   - **Rework/Follow-up** – Making necessary corrections based on findings.
6. **Exit Criteria** – Defines when the review is considered complete.
7. **Output** – The final results or deliverables from the review process.

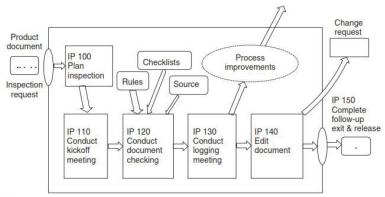


Figure 5.12 The inspection process.

1. **Plan Inspection (IP 100)** – Define scope, objectives, and assign roles.
2. **Conduct Kickoff Meeting (IP 110)** – Align team on process and expectations.
3. **Document Checking (IP 120)** – Review documents using checklists and rules.
4. **Logging Meeting (IP 130)** – Identify and record defects for further action.
5. **Edit Document (IP 140)** – Correct errors and improve document quality.
6. **Complete Follow-up, Exit & Release (IP 150)** – Verify corrections and finalize inspection.

✔ **Outcome:** Improved process efficiency and defect-free documentation.

(IP – Inspection Process)

Problem Report Form

Priority: High
Project Name: Inventory Management System Upgrade
Date: 25-Feb-2025

Process Name: Order Processing
Phase Number: 2
Raised By: John Doe

Number of Days to Answer: 2
Close Date: 02-Mar-2025

Number of Days to Fix this Problem: 5

Finding:
The system fails to update inventory levels after an order is placed, causing stock mismatches.

Requirement/Standard Impacted:
Real-time inventory update as per company policy.

Immediate Solution Proposed:
Manually update inventory records after each order as a temporary solution.

Root Cause:
Database synchronization failure between the order management module and the inventory system.

Permanent Solution Proposed:
Modify the system to implement an automated real-time database sync process.

Acceptance Date of Permanent Solution: 05-Mar-2025

Follow-up Action (if necessary):
Monitor inventory updates for one month to ensure the issue is fully resolved.

SE ZG501
Software Quality Assurance and Testing
Module – Session 8

Effective Test Management and Planning

innovate achieve lead

- During the testing phase of software development, testing activities are managed well to complete the testing process smoothly and on time as well.
- **Test Management** is a process where testing activities are managed to ensure high-quality and high-end testing of software applications.
- This method consists of tracking, organization, controlling processes and checking the visibility of the testing process to deliver a high-quality software application.

- Test management is concerned with both test resource and test environment management.
- It is the role of test management to ensure that new or modified service products meet the business requirements for which they have been developed or enhanced.
- It makes sure the software testing process runs as expected.

key elements of Test Management

- **Test organization**
- **Test planning**
- **Detailed test design and test specifications**
- **Test monitoring and assessment**
- **Product quality assurance**

TEST ORGANIZATION

- Test organization involves structuring and managing the testing process by defining clear roles and responsibilities within the team. It includes:
- Establishing a structured framework for testing activities.
- Reviewing the project scope and creating high-level test plans.
- Scheduling resources and defining timelines.
- Setting up configuration standards and preparing the test environment.

A well-organized test setup ensures efficient execution, better collaboration, and higher software quality.

- Since testing is viewed as a process, it must have an organization such that a testing group works for better testing and high quality software.
- The testing group is responsible for the following activities:
  - Maintenance and application of test policies
  - Development and application of testing standards
  - Participation in requirement, design, and code reviews
  - Test planning
  - Test execution
  - Test measurement
  - Test monitoring
  - Defect tracking
  - Acquisition of testing tools
  - Test reporting

The staff members of such a testing group are called test specialists or test engineers or simply testers.

A tester is not a developer or an analyst. He does not debug the code or repair it.

He is responsible for ensuring that testing is effective and quality issues are being addressed.

Skills a Tester

1. Personal and Managerial Skills: -

- Testers must be able to contribute in policy-making and planning the testing activities.
- Testers must be able to work in a team.
- Testers must be able to organize and monitor information, tasks, and people.
- Testers must be able to interact with other engineering professionals, software quality assurance staff, and clients.
- Testers should be capable of training and mentoring new testers.
- Testers should be creative, imaginative, and experiment-oriented.
- Testers must have written and oral communication skills.

2. Technical Skills: -

- Testers must be technically sound, capable of understanding software engineering principles and practices.
- Testers must be good in programming skills.
- Testers must have an understanding of testing basics, principles, and practices.
- Testers must have a good understanding and practice of testing strategies and methods to develop test cases.
- Testers must have the ability to plan, design, and execute test cases with the goal of logic coverage.
- Testers must have technical knowledge of networks, databases, operating systems, etc. needed to work in a the project environment.
- Testers must have the knowledge of configuration management.
- Testers must have the knowledge of test ware and the role of each document in the testing process.
- Testers must have known about quality issues and standards.

STRUCTURE OF TESTING GROUP

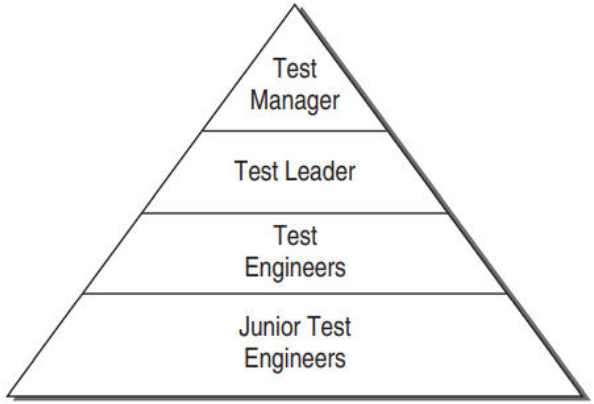


Figure 9.1 Testing Group Hierarchy

Test Manager

A **Test Manager** holds the highest position in the testing hierarchy and is responsible for managing the testing process efficiently.

Key Responsibilities:

1. Acts as the main point of contact between the testing team, project management, quality assurance, and marketing.
2. Defines **test strategies**, prepares the **master test plan**, and schedules testing activities.
3. Communicates with customers regarding **quality-related issues**.
4. Acquires **testing resources**, including tools and infrastructure.
5. **Monitors testing progress** and ensures smooth execution.
6. Participates in **static verification meetings** for reviewing documents and test cases.
7. **Manages the test team**, including hiring, evaluating, and, if necessary, dismissing members.

Test Leader: Assists the test manager in meeting testing and quality goals. To lead a team of test engineers who are working at the leaf-level of the hierarchy.

- i. Planning the testing tasks given by the test manager.
- ii. Assigning testing tasks to test engineers who are working under him.
- iii. Supervising test engineers.
- iv. Helping the test engineers in test case design, execution, and reporting.
- v. Providing tool training, if required.
- vi. Interacting with customers.

Test Engineers: Test engineers are highly experienced testers.

- i. Designing test cases.
- ii. Developing Developing automated testing frameworks to execute test cases efficiently..
- iii. Set-up test laboratories and environment.
- iv. Maintain the test and defect repositories.

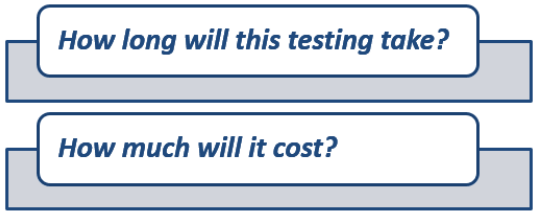
Junior Test Engineers: -

- Junior test engineers are newly hired testers. They usually are trained about the test strategy, test process, and testing tools.
- They participate in test design and execution with experienced test engineers.

Test Estimation and Scheduling

- Test estimation techniques refer to the methods and approaches used to determine or estimate the effort, time, and resources required for testing activities in software development projects.
- Software test estimation is a managerial task that involves assessing and approximating the required time, resources, and costs for executing tests in a specific environment.
- It serves as a projection that aids in preventing time constraints and exceeding budgets.

Why Test Estimation?



Why Test Estimation?

Project Planning: With correct estimations done, the overall project timeline can be kept under check. Project managers can create realistic schedules and allocate resources as per need if they know the time required for testing activities well in advance. This allows for effective coordination with development and other project activities.

Resource Allocation: With test estimations in place, the resource allocations: namely number of testers, testing tools, and testing environments required, can be allocated carefully and efficiently. It helps to ensure that overallocation or underutilization of resources is avoided.

Budgeting and Cost Control: Budgets are key to project success. Efficient testing estimation helps predict costs like **personnel, infrastructure, and tools**, enabling accurate budgeting, expense monitoring, and cost control

Risk Management: Estimation identifies potential testing risks, highlighting areas needing extra attention or resources. This enables proactive risk mitigation strategies.

Stakeholder Expectations: Estimation helps set realistic expectations regarding deadlines, costs, and potential risks for project stakeholders, including clients, managers, and development teams.

Project Optimization: Accurate estimation allows for better planning and optimization of testing activities. It helps identify opportunities for process improvements, resource optimization, and automation, leading to increased efficiency and productivity in the testing process.

What to Estimate?

Resources: Resources are required to **carry out** any project tasks. They can be people, equipment, facilities, funding, or anything else required for the completion of a project activity.

Times : Time is the most valuable resource in a project. Every project has a deadline to delivery.

Human Skills : Human skills mean the **knowledge** and the **experience** of the Team members. They affect to your estimation. For example, a team, whose members have low testing skills, will take more time to finish the project than the one which has high testing skills.

Cost: Cost is the project **budget**. Generally speaking, it means **how much money** it takes to finish the project.

How to estimate?
Software Test Estimation Techniques

Estimation Method	Simple Explanation
Work Breakdown Structure	Breaks large tasks into smaller, manageable parts for easier execution.
3-Point Estimation	Estimates tasks based on three scenarios: Best case, Most likely, and Worst case.
Wideband Delphi Method	Experts discuss and agree on the most accurate estimate.
Functional Point Analysis	Estimates tasks based on size, cost, and time required.
Agile Estimation	Uses past data and continuously updates estimates with new information.
Distribution in Percentage	Assigns effort to different project stages using percentages to balance workload.

Example : Bank Case Study

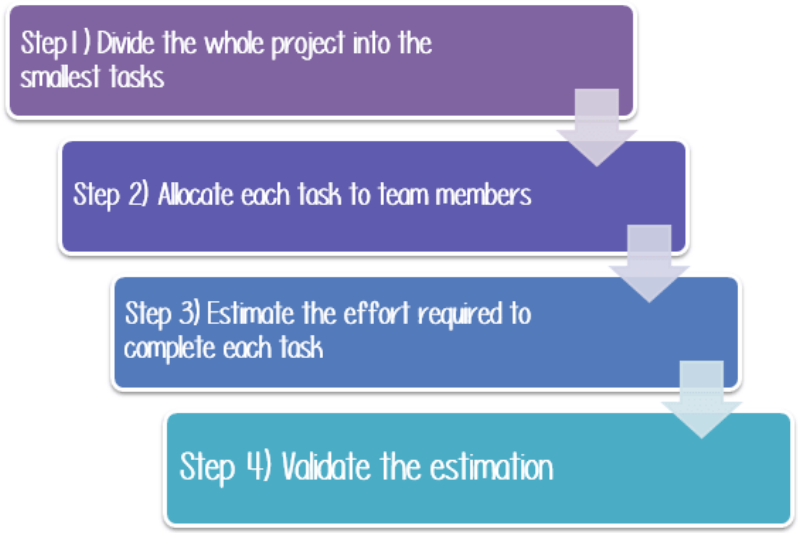
Work Breakdown Structure (WBS)
- Breaking down the test project into small pieces

Three Point Estimation
- Estimation method is based on statistical data

Functional Point Method
- Measure the size and give weightage to each function point

- **Combine these techniques to find the estimate.**

Following 4 Step process to arrive at an estimate



Quick Brain Teaser: Mastering Test Management!

Which of the following are key elements of Test Management?

A) Test organization

B) Test planning

C) Software debugging

D) Detailed test design and specifications

E) Test monitoring and assessment

Answer Reveal: Test Management Essentials!

Which of the following are key elements of Test Management?

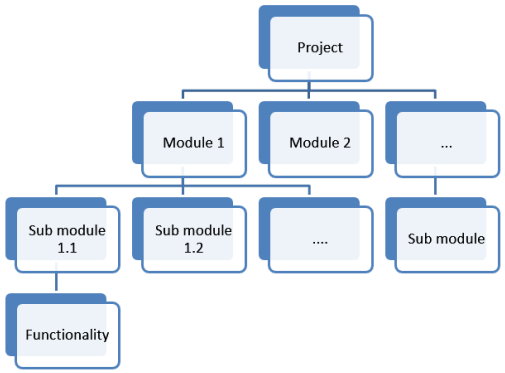
A) Test organization
B) Test planning
C) Software debugging
D) Detailed test design and specifications
E) Test monitoring and assessment

Correct Answers: A, B, D, E, and F.

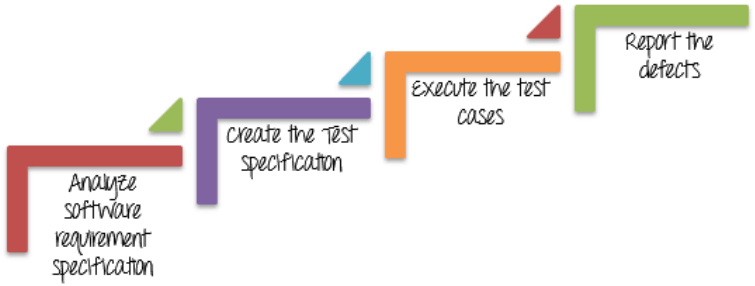
Step 1) Divide the whole project task into subtasks

Work Breakdown Structure technique. : a complex project is divided into modules. The modules are divided into sub-modules. Each sub-module is further divided into functionality. It means divide the whole project task into the **smallest** tasks.

- Using Work Break Down structure to break out the Bank project into 5 smaller tasks



- After that, you can break down each task into subtasks. The purpose of this activity is to create tasks as detailed as possible.



Task	Sub task
Analyze software requirement specification	Investigate the soft requirement specs
	Interview with the developer & other stakeholders to know more about the website
Create the Test Specification	Design test scenarios
	Create test cases
	Review and revise test cases
Execute the test cases	Build up the test environment
	Execute the test cases
	Review test execution results
Report the defects	Create the Defect reports
	Report the defects

Step 2) Allocate each task to team member

Each task is assigned to the **appropriate** member in the project team.

Task	Members
Analyze software requirement specification	All the members
Create the test specification	Tester/Test Analyst
Build up the test environment	Test Administrator
Execute the test cases	Tester, Test Administrator
Report defects	Tester

Step 3) Effort Estimation For Tasks

There are 2 techniques which you can apply to estimate the effort for tasks

- **Functional Point Method**
- **Three Point Estimation**

Method 1) Function Point Method
to estimate the effort for tasks

- The Test Manager estimates Size, Duration, and Cost for the tasks

Step A) Size

Step B) Duration

Step C) Cost

Step A: Estimating Task Size

After breaking the project into smaller tasks using the **Work Breakdown Structure (WBS)** method, the next step is to estimate the size of each task.

Example:

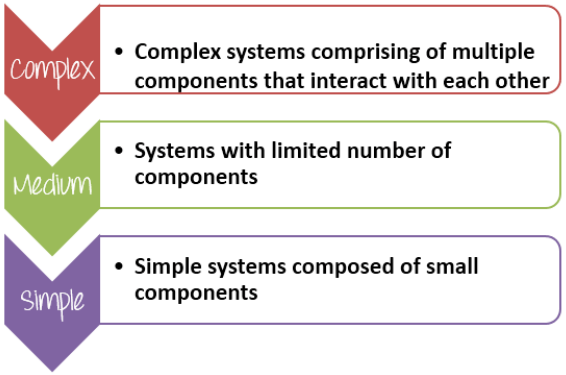
Task – *"Create the test specification"*

The size of this task depends on the number of system functions.

- **More functions → Larger and more complex task**
- **Fewer functions → Smaller and easier task**

A system with many features increases complexity and requires more effort to complete.

Prior to start actual estimating tasks effort, functional points are divided into three groups like **Complex**, **Medium Simple** as following:



Based on the complex of software functions, the Test Manger has to give enough **weightage** to each functional point. For example

Group	Weightage
Complex	5
Medium	3
Simple	1

Let's take a simple example exercise

- By looking at SRS the software specification of Bank website, where the software engineer have already described the software modules in detail, we can determine the **complexity** of website's features by giving the weightage for each modules.
- More complex the function point, more is the effort to test it is. The website is divided into **12 function** points, you can determine the **complexity** of each function points as follows-

No.	Module Name	Applicable Roles	Description	Weightage
1.	Balance Enquiry	Manager Customer	**Customer:** A customer can have multiple bank accounts. He can view balance of his accounts only **Manager:** A manager can view balance of all the customers who come under his supervision	3
2.	Fund Transfer	Manager Customer	**Customer:** A customer can have transfer funds from his "own" account to any destination account. **Manager:** A manager can transfer funds from any source bank account to destination account	5

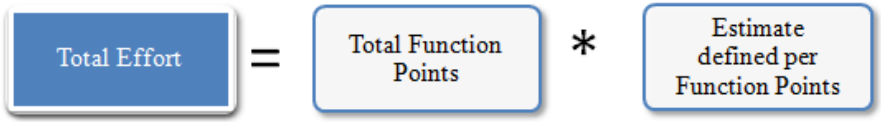
3.	Mini Statement	Manager Customer	A Mini statement will show last 5 transactions of an account **Customer:** A customer can see mini-statement of only his "own" accounts **Manager:** A manager can see mini-statement of any account	3
4.	Customized Statement	Manager Customer	A customized statement allows you to filter and display transactions in an account based on date, transaction value **Customer:** A customer can see Customized-statement of only his "own" accounts **Manager:** A manager can see Customized - statement of any account	5

5.	Change Password	Manager Customer	**Customer:** A customer can change password of only his account. **Manager:** A manager can change password of only his account. He cannot change passwords of his customers	1
6.	New Customer	Manager	**Manager:** A manager can add a new customer. **Manager:** A manager can edit details like address, email, telephone of a customer.	3
7.	New Account	Manager	Currently system provides 2 types of accounts • Saving • Current A customer can have multiple saving accounts (one in his name, other in a joint name etc). He can have multiple current accounts for different companies he owns.	5

8.	Edit Account	Manager	**Manager:** A manager can add an edit account details for an existing account	1
9.	Delete Account	Manager	**Manager:** A manager can add a delete an account for a customer.	1
10.	Delete Customer	Manager	A customer can be deleted only if he/she has no active current or saving accounts **Manager:** A manager can delete a customer.	1
11.	Deposit	Manager	**Manager:** A manager can deposit money into any account. Usually done when cash is deposited at a bank branch.	3
12.	Withdrawal	Manager	**Manager:** A manager can withdraw money from any account. Usually done when cash is withdrawn at a bank branch.	3

STEP B) Estimate duration for the task

After classifying the **complexity** of the function points, you have to estimate the **duration** to test them. Duration means **how much** time needs to finish the task.



Key Components:

1. **Total Effort**: The total time and resources required to test all functions of the website.
2. **Total Function Points**: The total number of functional modules in the website.
3. **Effort per Function Point**: The estimated time required to test one function point. This value depends on the **productivity** of the team member assigned to the task.

Suppose your project team has estimated defined per Function Points of 5 hours/points. You can estimate the total effort to test all the features of Bank website as follows:

	Weightage	# of Function Points	Total
Complex	5	3	15
Medium	3	5	15
Simple	1	4	4
Function Total Points			**34**
Estimate define per point			**5**
Total Estimated Effort (Person Hours)			**170**

So the total effort to complete the task **"Create the test specification"** of Bank website is is around 170 man-hours.

1 man hour = work completed in an hour of uninterrupted effort by an average worker.

Resource Allocation and Cost Estimation

- Once the required effort is determined, you can allocate resources by assigning team members and tools to the task. This helps in estimating the **time required** (duration) to complete the task effectively. Based on this estimation, you can also assess both **labor and non-labor costs**.

- This process highlights the importance of skilled team members. **Experienced and talented team members** can complete tasks more efficiently, reducing the overall time required. As a result, your project is more likely to be completed **on time or even ahead of schedule**.

STEP C) Estimate the cost for the tasks

"**How much does it cost?"**

- Suppose, on average your team salary is $5 per hour.
- The time required for "Create Test Specs" task is 170 hours. Accordingly, the cost for the task is 5*170= $850.
- Now you can calculate budget for other activities in WBS and arrive at overall budget for the project.

Method 2) Three Point Estimation

- Three-Point estimation is one of the techniques that could be used to estimate a task.
- The simplicity of the Three-point estimation makes it a very useful tool for a Project Manager that who wants to estimate.
- In three-point estimation, **three** values are produced initially for every task based on **prior experience** or **best-guesses** as follows

When estimating a task, the Test Manager needs to provide three values, as specified above.

The three values identified, estimate what happens in an **optimal state**, what is the **most likely**, or what we think it would be the **worst case scenario.**

Example: "**Create the test specification**", for bank website

You can estimate as following

The **best case** to complete this task is **120** man-hours (around 15 days). In this case, you have a talented team, they can finish the task in smallest time.

The **most likely** case to complete this task is **170** man-hours (around 21 days). This is a normal case, you have enough resource and ability to complete the task

The **worst case** to complete this task is **200** man-hours (around 25 days). You need to perform much more work because your team members are not experienced.

Now, assign the value to each parameter as below

$$a = 120 \quad m = 170 \quad b = 200$$

The effort to complete the task can be calculated using **double-triangular distribution** formula as follows-

$$E = (a + 4m + b)/6$$

$$E = (120 + 4 * 170 + 200)/6$$

$$E = 166.6\ (man - hours)$$

- Parameter E is known as **Weighted Average.**
- It is the estimation of the task “Create the test specification”.

In the above estimation, you just determine a **possible** value, and not a **certain** one., we must know about the **probability** that the estimation is correct. You can use the other formula:

Since estimation involves uncertainty, it is essential to understand the **probability** that the estimated effort is correct

$$SD = (b - a)/6$$

$$SD = (200 - 120)/6$$

$$SD = 13.33\ (man - hours)$$

To measure this uncertainty, we use **Standard Deviation (SD)**, which quantifies how much the actual effort may **deviate from the estimated value**..

- Now you can conclude the estimation for the task “Create the test specification”
- To complete the task “Create the test specification” of Bank website, you need **166.6 ± 13.33** Man-hour (153.33 to 179.99 man-hour)

THANK YOU

How to estimate? Software Test Estimation Techniques

Estimation Method	Simple Explanation	Example
Work Breakdown Structure	Breaks a big task into smaller, manageable parts for easier execution.	If you're developing a website, you break it down into tasks like design, coding, testing, and deployment.
3-Point Estimation	Estimates tasks based on three scenarios: Best case (fastest), Most likely, and Worst case (slowest).	If testing a feature takes 5 days (best case), 7 days (most likely), or 10 days (worst case), the final estimate considers all three.
Wideband Delphi Method	Experts discuss and agree on the best estimate based on collective knowledge.	A team of experienced testers discusses how long a testing phase will take and agrees on a reasonable estimate.
Functional Point Analysis	Measures the effort required for a task based on its **size, complexity, and cost.**	A simple login page might take 2 days, while a complex payment system could take 10 days due to more functionality.
Agile Estimation	Uses past project data and continuously updates estimates with new information.	A software team estimates testing time based on similar past projects and adjusts if new challenges arise.
Distribution in Percentage	Assigns effort to different project stages using percentages to balance workload.	If testing takes 40% of the total project time, the team allocates resources accordingly.

Double-Triangular Distribution Explained

The **Double-Triangular Distribution** is a statistical approach used for **effort estimation** in project management and software engineering. It is a modified version of the **PERT (Program Evaluation and Review Technique) formula**, which accounts for different possible estimates:

Formula:

$$E = \frac{a + 4m + b}{6}$$

Where:

- **a** = Optimistic estimate (best-case scenario)
- **m** = Most likely estimate (realistic case)
- **b** = Pessimistic estimate (worst-case scenario)
- **E** = Expected effort (in man-hours or time units)

How It Works:

- It assumes that most tasks follow a **triangular probability distribution** where the **most likely estimate (m)** has the highest weight (4 times more) in the formula.
- The formula helps in calculating a **balanced expected effort** based on **best-case, worst-case, and most likely case.**

Use Case Example:

If a task has:

- **Optimistic estimate (a) = 120 hours**
- **Most likely estimate (m) = 170 hours**
- **Pessimistic estimate (b) = 200 hours**

Then, using the **double-triangular distribution formula**:

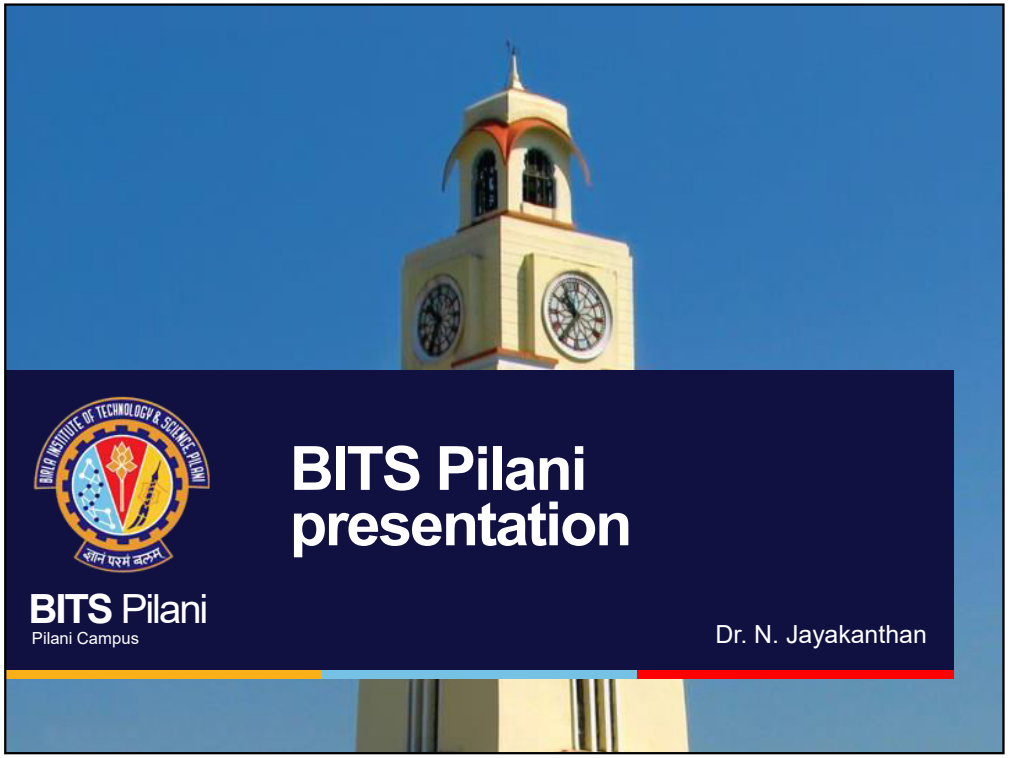
$$E = \frac{120 + 4(170) + 200}{6} = \frac{120 + 680 + 200}{6} = \frac{1000}{6} = 166.6 \text{ man-hours}$$

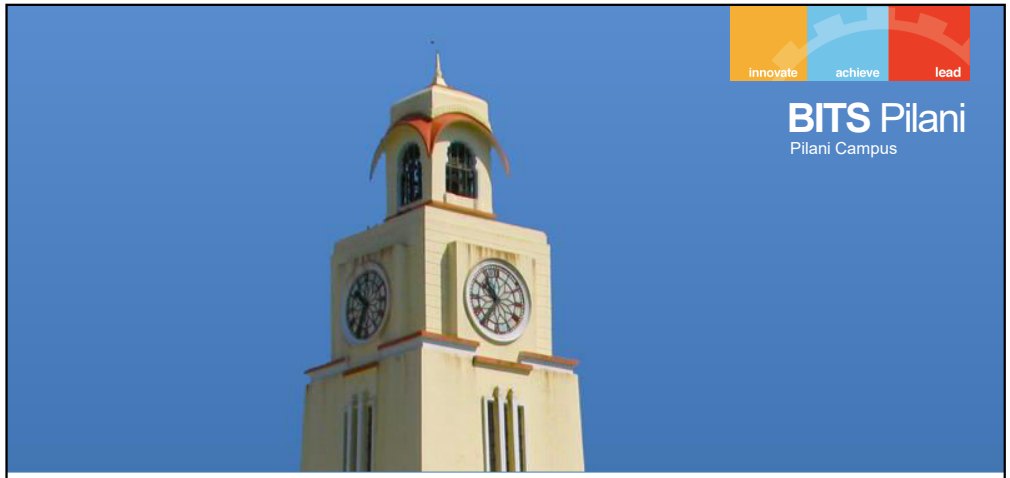
Testing risks

Testing risks" refer to potential **issues or challenges that could arise during the testing process, affecting its accuracy, efficiency, or effectiveness**. These risks could include:

1. **Insufficient Test Coverage** – Missing critical areas in testing.
2. **Defect Leakage** – Bugs **going undetected** and appearing in production.
3. **Limited Resources** – Shortage of testers, tools, or infrastructure.
4. **Tight Deadlines** – Insufficient time to conduct thorough testing.
5. **Unstable Test Environment** – Issues with testing setups affecting reliability.

Poor Test Data – Inaccurate or incomplete data leading to misleading test results





SE ZG501
Software Quality Assurance and Testing
Module – Session 10

Method 2) Three Point Estimation

- Three-Point estimation is one of the techniques that could be used to estimate a task.
- The simplicity of the Three-point estimation makes it a very useful tool for a Project Manager that who wants to estimate.
- In three-point estimation, **three** values are produced initially for every task based on **prior experience** or **best-guesses** as follows:

- a: The best-case estimate
- m: The most likely estimate
- b: The worst case estimate

When estimating a task, the Test Manager needs to provide three values, as specified above.
The three values identified, estimate what happens in an **optimal state**, what is the **most likely**, or what we think it would be the **worst case scenario.**

Example: "**Create the test specification**", for bank website

You can estimate as following

The **best case** to complete this task is **120** man-hours (around 15 days). In this case, you have a talented team, they can finish the task in smallest time.

The **most likely** case to complete this task is **170** man-hours (around 21 days). This is a normal case, you have enough resource and ability to complete the task

The **worst case** to complete this task is **200** man-hours (around 25 days). You need to perform much more work because your team members are not experienced.

Now, assign the value to each parameter as below

$$a = 120 \quad m = 170 \quad b = 200$$

The effort to complete the task can be calculated using **double-triangular distribution** formula as follows-

$$E = (a + 4m + b)/6$$

PERT

$$E = (120 + 4 * 170 + 200)/6$$

$$E = 166.6\ (man - hours)$$

- Parameter E is known as **Weighted Average.**
- It is the estimation of the task "Create the test specification".

In the above estimation, you just determine a **possible** value, and not a **certain** one., we must know about the **probability** that the estimation is correct. You can use the other formula:

Since estimation involves uncertainty, it is essential to understand the **probability** that the estimated effort is correct

$$SD = (b - a)/6$$

$$SD = (200 - 120)/6$$

$$SD = 13.33\ (man - hours)$$

To measure this uncertainty, we use **Standard Deviation (SD)**, which quantifies how much the actual effort may **deviate from the estimated value**..

- Now you can conclude the estimation for the task "Create the test specification"
- To complete the task "Create the test specification" of Bank website, you need **166.6 ± 13.33** Man-hour (153.33 to 179.99 man-hour)

Step 4) Validate the estimation

- Once you create an aggregate estimate for all the tasks mentioned in the WBS, you need to forward it to the **management board**, who will **review** and **approve** it.
- The member of management board could comprise of the CEO, Project Manager & other stakeholders.
- The management board will review and discuss your estimation plan with you. You may explain them your estimation **logically** and **reasonably** so that they can approve your estimation plan.

Test estimation best practices

Add some buffer time
Account Resource planning in estimation
Use the past experience as reference
Stick to your estimation

Test Data Management

The process of planning, creating, and maintaining the datasets used in testing activities, ensuring that they are the right data for the right test case, in the right format, and available at the right time.

Test data is the set of input values used during the testing process of an application (software, web, mobile application, API, etc).

These values represent what a user would enter the system in a real-world scenario.

Testers usually can write a test script to automatically and dynamically identify the right type of values to put into the system and see how it responds to those data.

Example : Test data for the testing of a login page

- **Username** column and a **Password** column.
- A test script or automation testing tool can open the Login page, identify the Username field, the Password field, then input the values
- Can have hundreds to thousands of such credential pairs representing unique test scenarios.

Username	Password
user_123	Pass123!
testuser@email	Secret@321
admin_user	AdminPass#
jane_doe	JaneDoePass

- You can have hundreds to thousands of such credential pairs representing unique test scenarios.
- But having a huge database does not immediately mean all of it is high-quality.

Criteria to evaluate test data quality

Relevance: it makes sense to have test data that accurately reflects the scenario being tested.

Imagine testing the response of the login page when users enter the wrong set of credentials, but the test data being used is actually the correct, stored-in-database credentials. This returns inaccurate results.

Availability: what's the point of having thousands of relevant data points yet you can't retrieve them for testing activities?

Usually QA teams have clearly defined role-based access for test data, so TDM activities are also about assigning the right level of access to the right personnel.

Updated: software constantly changes, bringing with it new complexities and dependencies.

The responsibility of QA teams is to be aware of those updates and make changes to the test data accordingly to ensure that results accurately reflect the current state of the software.

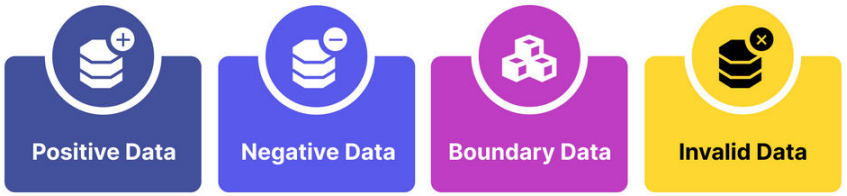
Compliance: aside from the technical aspects, we should never forget compliance requirements.

QA teams sometimes leverage directly production data for testing activities due to its instant availability, but production data is a tricky domain: it may contain confidential information protected by GDPR, HIPAA, PCI, or other data privacy focused policies.

PCI DSS, HIPAA, and GDPR are all security standards that protect sensitive information, but they differ in their scope, purpose, and enforcement.

- **Payment Card Industry Data Security Standard** is a set of security standards that protect cardholder data.
- **Health Insurance Portability and Accountability Act** is a federal law that protects patient health information (PHI)
- **General Data Protection Regulation** is a law that protects the personal data of EU citizens.

Types of Test Data



- **Positive Test Data:** this type of data consists of input values that are **valid and within the expected range** and is designed to test how the system behaves under expected and normal conditions. Examples: a set of valid username and password that allows users to login to their account page on an eCommerce site.

- **Negative Test Data:** in contrast with positive data, negative test data consists of input values that are invalid, unexpected, or outside the specified range. It is designed to test how the system behaves when users do something out of the "correct" pathintended.
- Examples: a set of username and password that is too long.

- **Boundary Test Data:** these are values at the edges or boundaries of acceptable input ranges chosen to assess how the system handles inputs at the upper and lower limits of the allowed range.

- **Invalid Test Data:** these are data that does not accurately represent the real-world scenarios or conditions that the software is expected to encounter. It does not conform to the expected format, structure, or rules within a given context.

Break-Time Brain Buster!

A login page consistently displays a "Login Successful" message, even when incorrect credentials are entered during testing.
Which of the following best explains the likely cause, based on test data quality principles?

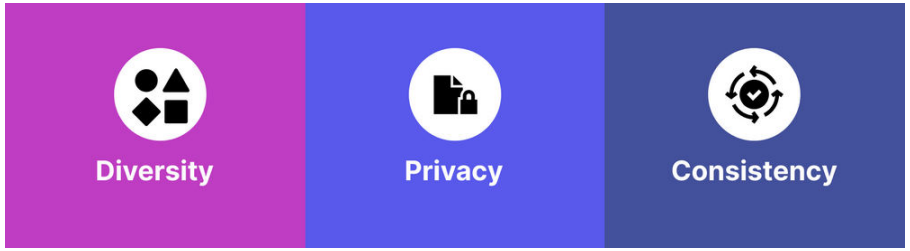
A. The system is malfunctioning and should be shut down.

B. The test data used for invalid credential scenarios contains actual valid credentials, resulting in inaccurate test outcomes.

C. The test data volume is insufficient for executing the test cases.

D. The automation script was not implemented for the login functionality.

Why Test Data Management?



Diversity: high test coverage means testing many different scenarios and having the right data for each one.

A simple registration page, for example, already requires so many datasets to cover all of the possible scenarios that can happen there :

- Valid credentials
- Empty username
- Empty password
- Incorrect username
- SQL injection attempt
- Special characters
- Too long username
- Too long password

Data Privacy: without good TDM practices, testers can risk using PII (personally identifiable information) to test, which is a breach in security.

- There are so many things you can do in TDM to prevent this from happening, such as data anonymization, which is essentially a process to replace real, sensitive data with similar but fictitious data.
- If teams decide to use real data, they can mask (i.e. encrypt) specific sensitive data fields, and use only the most necessary.
- Several teams employ Dynamic Data Masking (DDM) to dynamically mask data fields based on user roles and permissions.

Data Consistency: QA teams also need to ensure that their test data is uniform across the entire systems, adhering to the same format and standards, and even the relationships among the datasets must be continuously maintained over time when the complexity of the system grows.

Test Data Management Techniques

1. **Data Masking**
2. **Data Subsetting**
3. **Synthetic Data Generation**

Data Masking

- Data masking is the technique used to protect sensitive information in non-production environments by replacing, encrypting, or otherwise "masking" confidential data while retaining the original data's format and functionality.
- Data masking creates a sanitized version of the data for testing and development purposes without exposing sensitive information.

Data Masking Technique	Definition + Examples
Substitution	**Definition**: Replace actual sensitive data with fictional or anonymized values. You can leverage Generative AI for this approach; however, note that creating entirely new data is resource-intensive. **Example**: Replace actual names with randomly generated names (e.g., John Doe).
Shuffling	**Definition**: Randomly shuffle the order of data records to break associations between sensitive information and other data elements. This approach is faster and easier to achieve compared to the Substitution. **Example**: Shuffle the order of employee records, disconnecting salary information from individuals.

Encryption	**Definition**: Use encryption algorithms to transform sensitive data into unreadable ciphertext. Only authorized users with decryption keys can access the original data. This is a highly secured approach to take. **Example**: Encrypt credit card numbers, rendering them unreadable without proper decryption.
Tokenization	**Definition**: Replace sensitive data with randomly generated tokens. Tokens map to the original data, allowing reversible access by authorized users. **Example**: Replace social security numbers with unique tokens (e.g., Token123).

Character Masking	**Definition**: Mask specific characters within sensitive data, revealing only a portion of the information. **Example**: Mask all but the last four digits of a social security number (e.g., XXX-XX-1234).
Dynamic Data Masking	**Definition**: Dynamically control and limit the exposure of confidential data in real-time during query execution. In other words, sensitive data is masked at the moment of retrieval, just before being presented to the user (usually the masking logic is based on user roles). **Example**: Mask salary information in query results for users without financial access rights.

Randomization	**Definition**: Introduce randomness to the values of sensitive data for creating diverse test datasets. **Example**: Randomly adjust salary values within a specified percentage range for a group of employees.

Data Sub setting

Data sub setting is a technique to create a smaller yet representative subset of a production database for use in testing and development environments.

Data Subsetting : Benefits

- Reduce data volume, especially in organizations with large datasets. For testing purposes, smaller data volume minimizes resource requirements and therefore reduces maintenance needs.
- Preserve data integrity, as sub setting a dataset does not change the relationship between rows, columns, and any entities within it
- Easily include/exclude data based on specific criteria relevant to the team's testing needs, giving them a higher level of control. At the same time, this translates into improved efficiency in terms of data storage, transmission, and processing.

Synthetic Data Generation

- Synthetic data generation is the process of creating artificial datasets that simulate real-world data without containing any sensitive or confidential information.
- This approach is usually reserved only for when obtaining real data is challenging (i.e. financial, medical, legal data) or risky data (i.e. employee personal information).

- generating entirely new sets of data for testing purposes is a more practical approach.
- These synthetic datasets aim to simulate the original dataset as closely as possible, and that means capturing its statistical properties, patterns, and relationships.

Tools : Test Data Management

- **Informatica-** Data provisioning, data sub setting, data masking and data profiling are all included.
- **Compuware-** It intends to make the extraction, masking and delivery of test data simpler.
- **Delphix-** It can interact with many databases and systems and allows you to build and deliver masked or fake information copies for testing.
- **Microfocus Data Express-** Sensitive data is hidden and portions of production data are created.
- **IBM InfoSphere Optim-** It allows to produce, subset and conceal data for testing while preserving the security and privacy of the data.

Software Configuration Management

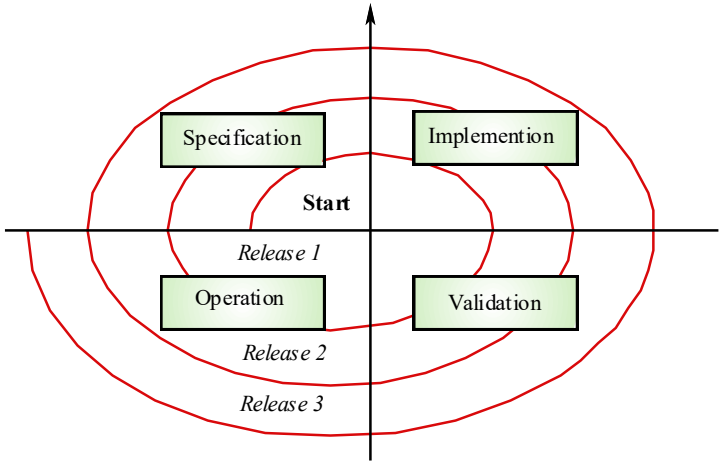
Maintenance is Inevitable

- System requirements may change during development due to environmental factors.
- Systems are tightly coupled to their environment
- When a system is installed, it changes the environment and that can change the system requirements.
- The delivered system may not meet its requirements.
- Systems must be maintained to remain useful in their environment.

Types of Maintenance

- Corrective Maintenance (21%)
  - making changes to repair defects
- Adaptive Maintenance (25%)
  - making changes to adapt software to external environment changes (hardware, business rules, OS, etc.)
- Perfective Maintenance (50%)
  - extending system beyond its original functional requirements
- Preventative Maintenance (4%)
  - Modifying work products so that they are more easily corrected, adapted, or enhanced

Spiral Maintenance Model



Maintenance Costs

- Usually greater than the development costs (2 to 10 times as much in some cases)
- Affected by both technical and non-technical factors
- Increase as software is maintained and system corruption is introduced
- Aging software can have high support costs (e.g. old languages, compilers, etc.)

Maintenance Developer Tasks

- Understand system.
- Locate information in documentation.
- Keep system documentation up to date.
- Extend existing functions.
- Add new functions.
- Find sources of errors.
- Correct system errors.
- Answer operations questions.
- Restructure design and code.
- Delete obsolete design and code.
- Manage changes.

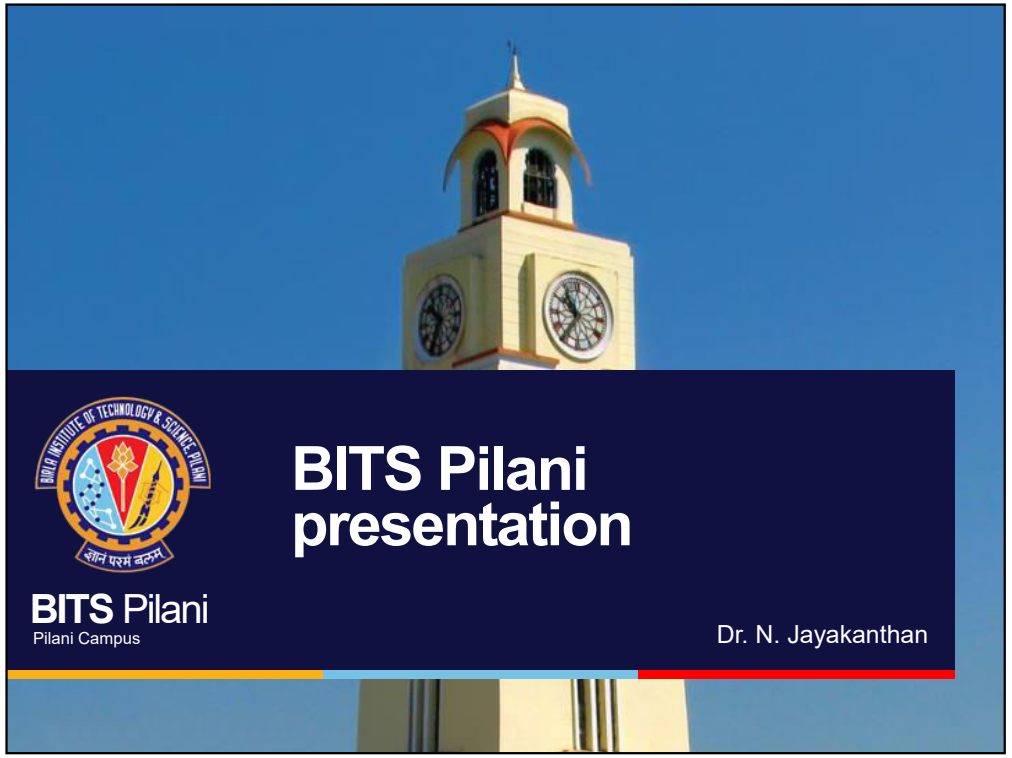
Maintenance can be tough

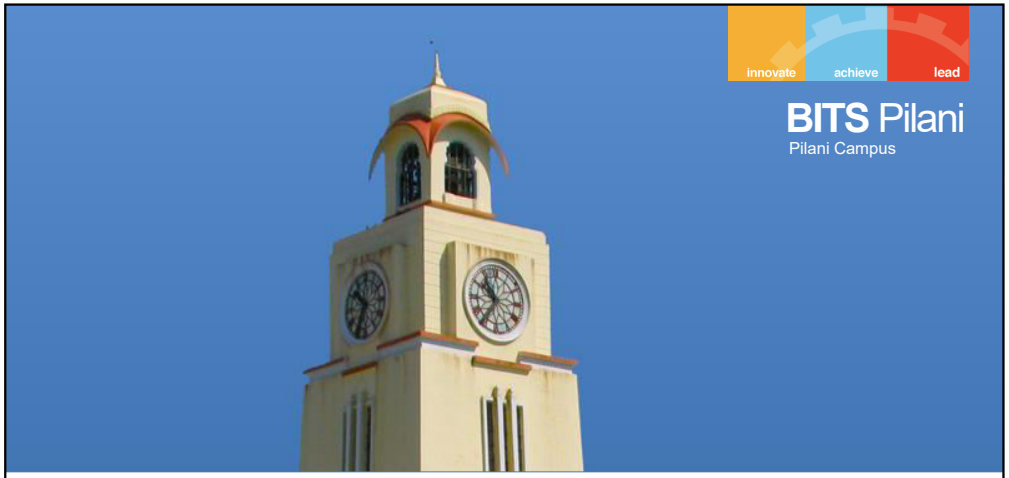
- Limited understanding of hardware and software (maintainer).
- Management priorities (maintenance may be low priority).
- Technical problems.
- Testing difficulties (finding problems).
- Morale problems (maintenance is boring).
- Compromise (decision making problems).

Maintenance Cost Factors

- Staff turnover
  - No turnover usually means lower maintenance costs
- Contractual responsibility
  - Developers may have no contractual obligation to maintain the delivered system and no incentive to design for future change
- Staff skills
  - Maintenance staff are often inexperienced and have limited domain knowledge
- Program age and structure
  - As programs age, their structure degrades, making them harder to understand and modify.

THANK YOU



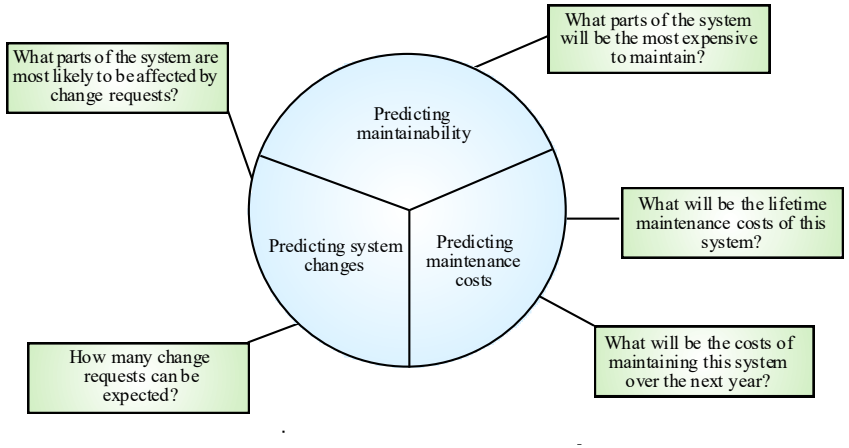


SE ZG501
Software Quality Assurance and Testing
Module 7

Maintenance Prediction

- Identifies **parts of the system** that may cause problems and be costly to maintain.
- The **ease of accepting changes** depends on how maintainable the system is.
- Making changes can make the system harder to maintain over time.
- **More changes lead to higher maintenance costs**.
- The cost of changes depends on how easy the system is to maintain.

Maintenance Prediction



Maintenance Complexity Metrics

- Predictions of maintainability can be made by assessing component complexities.
- Most maintenance efforts only affect a small number of system components.
- Maintenance complexity depends on
  - complexity of control structures
  - complexity of data structures
  - module size

Maintenance Process Metrics

- Maintainability measurements
  - Number of requests for corrective maintenance
  - Average time required for impact analysis
  - Average time to implement a change request
  - Number of outstanding change requests
- If any if these increases it may signal a decline in maintainability

Maintenance Tools

- Text editors (better than punch cards).
- File comparison tools.
- Compilers and linkage editors.
- Debugging tools.
- Cross reference generators.
- Complexity calculators.
- Control Libraries.
- Full life cycle CASE tools.

Configuration Management

- Software changes are inevitable.
- One goal of software engineering is to improve how easy it is to change software.
- Configuration management is all about change control.
- Every software engineer has to be concerned with how changes made to work products are tracked and propagated throughout a project.
- To ensure quality is maintained the change process must be audited.

Software Configuration Items

Computer programs
- source
- executable

Documentation
- technical
- user

Data
- contained within the program
- external data (e.g. files and databases)

Baselines

- A baseline is an approved version of a document or code.
- It shows that something is finished and ready to move forward.
- It marks an important stage in the project.
- After setting a baseline, changes need approval before being made.

Sources of Change

New market conditions dictate changes to product requirements or business rules

New customer needs demand modification of data, functionality, or services

Business reorganization causes changes in project priorities or SE team structure

Budgetary or scheduling constraints require system to be redefined

Change Requests

Requests can come from users, customers, or management

Change requests should be carefully analyzed as part of the maintenance process before they are implemented.

Some changes requests must be implemented urgently due to their nature

- Fault repair
- System environment changes
- Urgently required business changes

Change Prediction

- Predicting the number of changes requires understanding the relationships between a system and its environment
- Tightly coupled systems require changes whenever the
- environment changes
- Factors influencing the system/environment relationship
  - number and complexity of system interfaces
  - number and volatility of system requirements
  - business processes where the system is uses

Configuration Management Tasks

- Identification
  - Tracking changes to multiple SCI versions
- Version control
  - Controlling changes before and after customer release
- Change control
  - Authority to approve and prioritize changes
- Configuration auditing
  - Ensure changes are made properly
- Reporting
  - Tell others about changes made

Version Control Terms

1. Entity

- A complete group of related items that are at the same version.
- Think of it as one full package or unit.

2. Variant

- A different version of the same entity made for a specific purpose.
- It works alongside other variants but with small changes..

3. New Version

- Created when big changes are made to the entity.
- Shows major improvements or added features.

Change Control Process - 1

Change request is submitted and evaluated to assess its technical merit and impact on the other configuration objects and budget.

Change report containing the results of the evaluation is generated.

Change control authority (CCA) makes the final decision on the status and priority of the change based on the change report.

Change Control Process - 2

Engineering change order (ECO) is generated for each change approved.

- ECO describes the change, lists the constraints, and criteria for review and audit

Object to be changed is **checked-out** of the project database subject to access control parameters for the object.

Modified object is subjected to appropriate SQA and testing procedures.

Change Control Process - 3

Modified object is **checked-in** to the project database and version control mechanisms are used to create the next version of the software.

Synchronization control is used to ensure that parallel changes made by different people don't overwrite one another.

Configuration Management Team

Analysts.
Programmers.
Program Librarian.

Change Control Board

Customer representatives.
Some members of the Configuration management team.

Programmer's View - 1

Problem is discovered.
Problem is reported to configuration control board.
The board discusses the problem
- is the problem a failure?
- is it an enhancement?
- who should pay for it?

Assign the problem a priority or severity level, and assign staff to fix it.

Programmer's View - 2

Programmer or analyst
- Locates the source of the problem
- Determines what is needed to fix it

Programmer works with the librarian to control the installation of the changes in the operational system and the documentation.
Programmer files a change report documenting all changes made.

Change Control Issues

Synchronization (when?)
Identification (who?)
Naming (what?)
Authentication (done correctly?)
Authorization (who O.K.'d it?)
Routing (who's informed?)
Cancellation (who can stop it?)
Delegation (responsibility issue)
Valuation (priority issue)

Software Configuration Audit - 1

- Was the change made exactly as described in the Engineering Change Order (ECO)?
- Was a Formal Technical Review (FTR) conducted to ensure the change is technically correct?
- Did the team follow the proper software development process and apply all required software engineering standards?

Software Configuration Audit - 2

- Do the updated files or components clearly show the change that was made?
- Were the standard rules for recording and reporting the change (SCM standards) followed?
- Were all related documents and software items (SCI = Software Configuration Items) correctly updated?

Brain Break Challenge: Who's the Real Fixer?"

Question:

A team is consistently facing issues where updates made by different developers overwrite each other's changes. This has resulted in version mismatches and lost work.

Based on the configuration management concepts, which specific task or process should be reviewed and improved to address this issue?

A. Baseline approval
B. Synchronization control
C. Change request evaluation
D. Configuration item identification

Configuration Status Report

What happened?
Who did it?
When did it happen?
What else will be affected by the change?

Test Process Improvement

- TPI (Test Process Improvement) is a structured and systematic approach used to enhance the quality and effectiveness of software testing within an organization.
- TPI focuses on **identifying weaknesses and areas for improvement** in the **testing process, tools, and resources**, with the ultimate goal of improving the overall testing capability and software quality.
- Improvement of the test process from various **standpoints.**

Standpoints

- **Identifying defects**: Testing is about finding and fixing defects, such as bugs, missing requirements, and incorrect functionality.
- **Focusing on high-risk areas**: It's not possible to test every combination of scenarios in a software application, so testers should focus on the **most critical areas**.
- **Testing early**: It's more cost-effective to identify and fix defects early in the development process.
- **Verifying requirements**: Testing ensures that the product meets the functional, performance, design, and implementation requirements.
- **Providing information**: Testing can help determine if a product is ready for market.

- **Increasing reliability**: Testing can help identify and fix bugs, making the software more dependable.
- **Improving performance**: Testing can highlight areas for improvement and performance enhancements.
- **Ensuring user-focus**: Testing can help align the software with evolving user needs and feedback.
- **Enabling compatibility**: Testing can ensure that the software performs well across different platforms, devices, and environments.

- An improved process is not only more capable of finding better bugs but also testing the application in ways that uplift its quality and standards.

When to Perform Test Process Improvement?

- **Significant rise in bugs**
- **Increase in complexity (test management)**
- **Increase in involved resources**
- **Increase in time of testing**
- **Increase in testing costs**
- **Newer methods have arrived**
- **not retrospected for a long time**

Benefits of Test Process Improvement

- **Testing improvements**
- **Good quality software**
- **Align testing with other phases**
- **Enhanced Value of Testing**
- **Reduced Downtime and Minimized Mistakes**
- **Adherence to Industry Standards**
- **Sustainable Cost Savings**
- **Accelerated Project Schedules**
- **Effective Evaluation of Improvement Efforts**

Implement Test Process Improvement

- Test process improvement is a process of analysis and acting upon our observation.
- Implementation process can be divided into four parts.
  - **Diagnose the test improvement**
  - **Initiate the process/planning phase**
  - **Acting on the plan**
  - **Verify, Report, and Learn**

Diagnose the test improvement

Diagnose the situation :

- Determining why we need to perform test improvement
- Report specific problems in the current test process.
- Once we report our findings, we need to dig a little deeper into those areas to explore specific findings that can point us to the exact problems.
- For example, if our concern is cost management, then, we need to document what the costs were earlier, how it has grown over time, graphs depicting a clear view of the growth, what has impacted this high cost in the recent past, any significant change in the resource, etc. Consider this document as the sole evidence of moving further with test process improvement.

Initiate the process/planning phase

- Plan the steps you'll take to improve a process.
- Write down (document) what you want to achieve, the steps to follow, and how much of the process you'll cover.
- Find any possible problems (risks) that could slow down or stop the process, and guess how much delay each risk might cause.
- Set a deadline—when you expect the process improvement to be finished and when to check the results.

Acting on the plan

- With all the blueprints in our hand, we start the actual method of test process improvement.
- This should **follow the guidelines and expectations** described during the planning phase and **focus only on defects pointed out during the diagnosis phase**.
- The **senior testers** are required to monitor and guide this process through their expertise.
- They should also ensure that **delivery dates are not impacted and the process remains on schedule**.

Verify, Report, and Learn

At last, we verify the actions we performed with the results we got and match them with the expectations we set during the planning phase.

Next, we report all our results (including metrics) on the report for approval and sign-off from senior testers and higher management.

Lastly, we retrospect on the test improvement process from phases 1 to 4 and document our learnings throughout the process.

Measure Testing Process Improvement Impact

Define Clear Objectives: Clearly define the objectives of the testing process improvement.

Establish Baseline Metrics: Before implementing improvements, establish baseline metrics to understand the current state of the testing process.

Implement Changes Gradually: Introduce process improvements gradually rather than all at once. This allows for a more controlled assessment of the impact of each change on the overall testing process.

Monitor Key Performance Indicators (KPIs): Identify key performance indicators that directly align with the objectives of the testing process improvement.

Compare Metrics Before and After Implementation Compare the baseline metrics with data collected after the implementation of testing process improvements.

Calculate Return on Investment (ROI): Evaluate the return on investment by comparing the benefits gained from testing process improvements against the costs incurred in implementing those changes.

Utilize Surveys and Feedback: Administer surveys or conduct feedback sessions with the testing team to gauge their perceptions of the impact of process improvements.

Software Testing Process Improvement Models

- Similar to software development models (such as Agile and Waterfall), we rely on many test process improvement models to follow a set standard and improve our testing process.
- These models are often divided into "maturities"
- which means a certain stage that progresses toward better arrangement, organization, and completion.

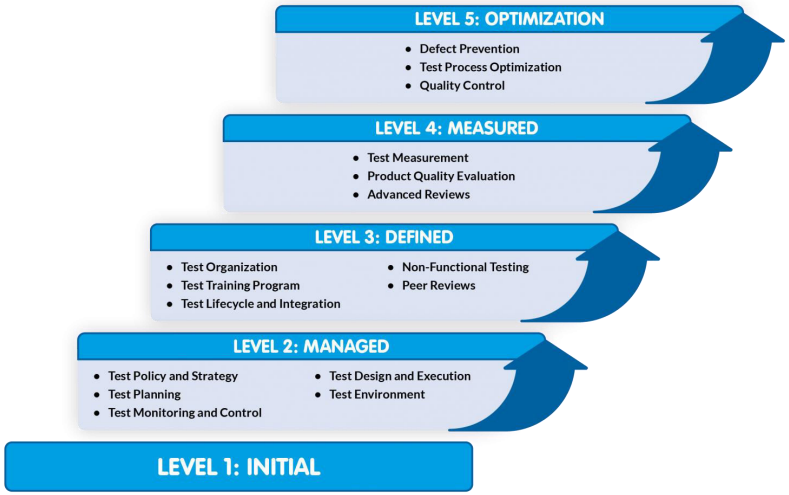
Testing Maturity Model integration (TMMi)

- TMMi is a "*maturity model*" which means it focuses on the maturity levels of each process to effectively implement the improvement in the testing lifecycle.
- Through TMMi, we can determine the maturity level of the testing process in an organization and improve it to achieve higher maturity.
- It was introduced in 2005 as a response to the ineffectiveness of the **Capability Maturity Model (CMM)** in the testing domain.
- TMMi tries to involve the organization structure and all the testers to progress together in the improvement process.

The benefits of using TMMi models are:

- Covers test-related activities extensively.
- Covers the best practices from the existing models.
- Covers the requirements for current trends and needs.
- Covers all aspects of test quality as described by industry experts.
- Provides a common standard for use.
- Can be applied to all phases of the software development lifecycle.

TMMi is divided into five maturity levels.



Maturity Level 1 – Initial

Maturity level 1 is not defined by the TMMi model because it presumes that all organizations start at this basic level. So, whatever you currently do and whatever process you follow, you can assume it to be of maturity level 1. This includes debugging and chaotic (unorganized) processes.

Maturity Level 2 – Managed

If the organization follows even a basic test approach towards testing and manages the same approach, it comes to maturity level 2. At this level, the organization should have a foundational structure in place that includes **test planning, test policy, monitoring, and setting up of a test environment**. This, however, **depends deeply on the project as a lot of things change when a new project is introduced**.

Maturity Level 3 – Defined

Maturity level 3 standardizes the testing process across the organization and expects **each project to follow the same process**. With this level, testing is now integrated very early into the development making it part of the development cycle.

Teams are required to be trained in testing and each member has a specific job for testing i.e. teams are more organized than before.

Maturity level 3 also expects that non-functional testing be planned and executed accordingly with reviews.

innovate achieve lead

Maturity Level 4 – Measured

When the outcomes and parameter values are applied to all the projects to ensure a bug-free application, we reach maturity level 4. The review practices introduced in the maturity level 3 are now more advanced and thorough in nature.

Maturity Level 5 – Optimization

At maturity level 5, the organization has a series of methods set up for optimization of the processes followed in testing up to maturity level 4. Continuous optimization leads to a bug-free application.

innovate achieve lead

THANK YOU

BITS Pilani, Pilani Campus

Change Control Issues (Challenges in Managing Changes)

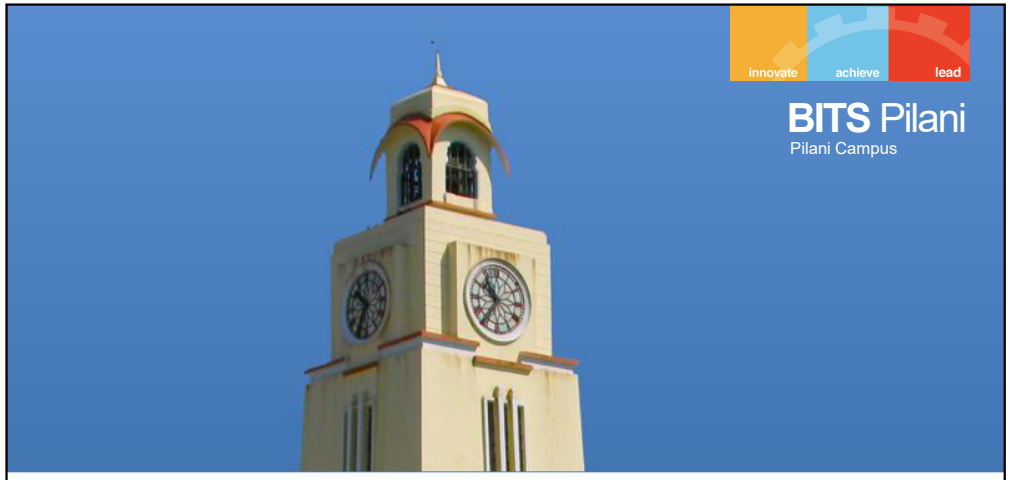
1. **Synchronization (When?)**
   - Ensuring that changes happen **at the right time** without disrupting ongoing work.
   - Example: Updating software only during maintenance hours.
2. **Identification (Who?)**
   - Knowing **who requested or made the change** for accountability.
   - Example: Tracking the developer responsible for code modifications.
3. **Naming (What?)**
   - Properly naming files or versions to **avoid confusion**.
   - Example: Using version numbers like **v1.0, v1.1, v2.0**.
4. **Authentication (Done correctly?)**
   - Ensuring only **authorized users** can make changes.
   - Example: Requiring login credentials before modifying system settings.
5. **Authorization (Who approved it?)**
   - Making sure changes are **officially approved** before implementation.
   - Example: A project manager must approve critical updates.
6. **Routing (Who's informed?)**
   - Making sure the **right people** know about the change.
   - Example: Informing all team members about an updated feature.
7. **Cancellation (Who can stop it?)**
   - Defining **who has the authority** to cancel an unnecessary or risky change.
   - Example: A security team stopping a software release due to a security flaw.
8. **Delegation (Responsibility issue)**
   - Assigning **who is responsible** for implementing the change.
   - Example: Assigning a senior developer to handle a critical bug fix.
9. **Valuation (Priority issue)**
   - Determining **how important** a change is compared to others.
   - Example: Fixing a security vulnerability before adding new features.

When to Perform Test Process Improvement?

1. **Significant rise in bugs**
   - If the number of defects found in production or late testing stages is increasing, it indicates flaws in the current test process that need addressing.
2. **Increase in complexity (test management)**
   - When projects become more complex (e.g., multiple platforms, integrations), test management becomes harder. This complexity requires a more structured and efficient test process.
3. **Increase in involved resources**
   - As more people get involved in testing, coordination issues may arise. Improving the process ensures better collaboration and role clarity.
4. **Increase in time of testing**
   - If testing is taking longer than expected or delaying releases, it's a sign the process might be inefficient and needs optimization.
5. **Increase in testing costs**
   - Rising costs can be due to inefficiencies, redundancy, or lack of automation. Process improvement can reduce unnecessary expenses.
6. **Newer methods have arrived**
   - Adopting modern testing practices (like automated testing, CI/CD, exploratory testing) can improve efficiency and quality.
7. **Not retrospected for a long time**
   - Regular retrospection helps identify gaps and improvement areas. If it's been a while since the last review, it's time to assess and update the test process.

Benefits of Test Process Improvement (Elaborated)

1. **Testing Improvements**
   - Enhances test coverage, accuracy, and efficiency through refined processes and updated tools.
2. **Good Quality Software**
   - Fewer bugs and better performance, leading to a reliable and user-friendly product.
3. **Align Testing with Other Phases**
   - Ensures testing is well-integrated with development, design, and deployment for smoother workflows.
4. **Enhanced Value of Testing**
   - Shows stakeholders the importance of testing by delivering measurable results and quality assurance.
5. **Reduced Downtime and Minimized Mistakes**
   - Helps detect and fix issues early, reducing production outages and costly rework.
6. **Adherence to Industry Standards**
   - Follows best practices and compliance requirements (e.g., ISO, IEEE), increasing credibility and trust.
7. **Sustainable Cost Savings**
   - Saves money by avoiding late-stage defect fixes and reducing waste in resources and time.
8. **Accelerated Project Schedules**
   - Speeds up the overall project by making testing faster, more automated, and less error-prone.
9. **Effective Evaluation of Improvement Efforts**
   - Enables tracking of testing metrics, allowing continuous monitoring and better decision-making.



SE ZG501
Software Quality Assurance and Testing
session No. 12

Test Process Improvement (TPI) Next

- The Test Process Improvement Next model, also called TPI Next, is the next generation of its predecessor TPI (Test Process Improvement) model.
- TPI Next aims to strongly establish its predecessor's strengths while keeping in focus the business goals of the organization.
- Since TPI Next aims at the business-related key areas, it can be referred to for activities beyond testing and is a generic model to implement.

TPI Next model describes four maturity levels:

Maturity Level 1 – Initial

At this level, no formal process is followed.The testing process is chaotic, unstructured, and highly inconsistent .There are no rules or standard practices, and testing is usually adhoc. The organization operates with whatever approach is currently in place, often leading to high defect rates and inefficiency..

Maturity Level 2 – Controlled

A basic structure is introduced to make the process more organized and manageable. Stakeholders begin to participate actively in streamlining testing activities. This level helps in reducing bugs but is still far from optimal. The process is controlled but not yet optimized, and many bugs may still exist.

Maturity Level 3 – Efficient

At this stage, the focus is on achieving **efficiency in existing testing processes**.

- Redundant or overlapping testing steps are identified and **eliminated**.
- Efforts are made to **streamline** the workflow and **reduce time wastage**.
- Processes are **integrated into a unified and structured testing cycle**.
- Goal: Minimize inefficiencies and ensure testing is lean and well-coordinated.

Maturity Level 4 – Optimizing

This is the **highest level of maturity** in the TPI Next model.

- A team of coordinators or experts **continuously evaluates** the testing process.
- Insights from **past projects (bugs, delays, challenges)** are used to make improvements.
- The aim is **continuous optimization** for future projects.
- Testing is now **adaptive, preventive, and driven by feedback and learning**.

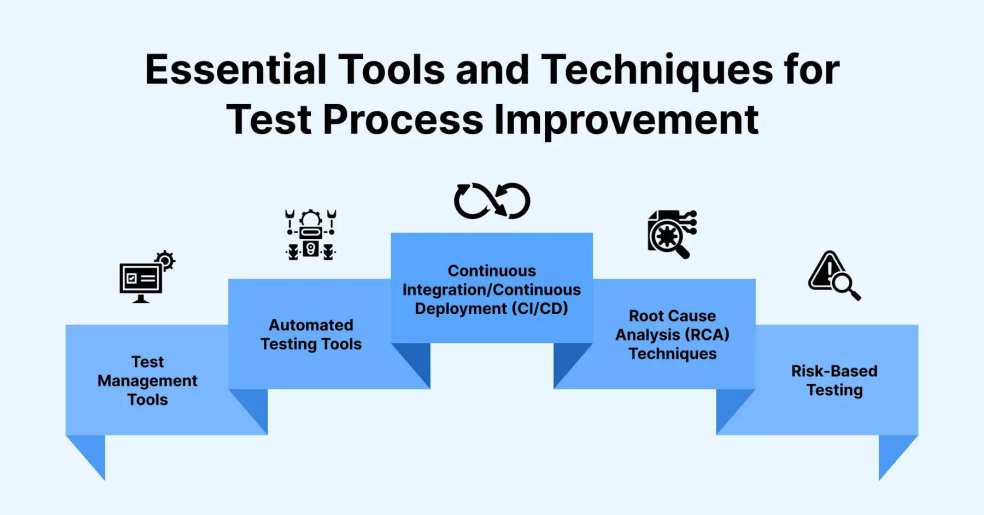
Systematic Test and Evaluation Process (STEP)

The STEP model focuses on testing software requirements systematically from the beginning of the Software Development Life Cycle (SDLC). It ensures better quality by evaluating results early and optimizing the process before coding begins.

Characteristics of STEP

- STEP focuses on requirement-driven testing
- STEP aims at performing testing first and then starting coding.
- STEP helps in early defect detection due to the “testing first” approach.

Tools and Techniques for Test Process Improvement



Test Management Tools :

- To successfully improve the test process, test management must be effective.
- Teams can effectively plan, carry out, and monitor testing operations using test management solutions like Jira, TestRail, and HP ALM (Application Lifecycle Management).

Automated Testing Tools:

- Teams may automate time-consuming and repetitive test cases using tools like Selenium, Appium, and JUnit, freeing up testers to concentrate on more important scenarios.
- Automation broadens the scope of tests, quickens the testing process, and improves the accuracy of test results. Additionally, it makes early bug discovery easier, which lowers the expense and work needed to correct flaws.

Continuous Integration/Continuous Deployment (CI/CD):

- Implementing CI/CD pipelines guarantees that software be released more quickly and frequently without sacrificing quality.
- The processes for building, integrating, and deploying software are automated by tools like Jenkins, GitLab CI/CD, and CircleCI.
- Organizations may do continuous testing, identify issues early, and establish a smooth and reliable deployment process by integrating automated tests into the CI/CD pipeline.

Root Cause Analysis (RCA) Techniques:

RCA is a method used to **find out the underlying reason** for defects or problems in the testing process.
It helps prevent the same issues from happening again by **fixing the real cause**, not just the symptoms.
Example: If frequent bugs are found in one module, RCA might reveal that unclear requirements are the actual issue.

Risk-Based Testing

Risk-Based Testing prioritizes testing efforts based on **potential risk and impact**.
Features or components that are **more likely to fail** or cause **serious problems** are tested first.
Example: In a banking app, login and payment systems are tested more thoroughly than the user profile section.

CMMi

- The Capability Maturity Model Integration (CMMI) is a model that aids in identifying the strengths and weaknesses of an organization's current processes and shows the way to improvement.
- CMMI's primary goal is to create high-quality software.

- CMMI is a technology offered by SEI that assists businesses in standardizing **software development**, **testing**, and **deployment** to improve the product's quality.
- The implementation of standards that will assist raise the quality of the software products is supported by CMMI.
- According to CMMI, a complete model consisting of five "Maturity Levels" that is essential to creating excellent software.
- CMMI is a more advanced model of its CMM predecessor.

What is CMM?

CMM: Capability Maturity Model

Developed by the **Software Engineering Institute of the Carnegie Mellon University**

Framework that **describes the key elements of an effective software process**.

What is CMM?

Describes an **evolutionary improvement path** for software organizations from an ad hoc, immature process to a mature, disciplined one.

Provides guidance on how to gain control of processes for developing and maintaining software and how to evolve toward a culture of software engineering and management excellence.

Process Maturity Concepts

Software Process

- Set of **activities, methods, practices, and transformations that people use to develop and maintain software and the associated products (e.g., project plans, design documents, code, test cases, user manuals)**

Software Process Capability

- Describes the range of expected results **that can be achieved by following a software process**
- Means of predicting the most likely outcomes **to be expected from the next software project the organization undertakes**

Process Maturity Concepts

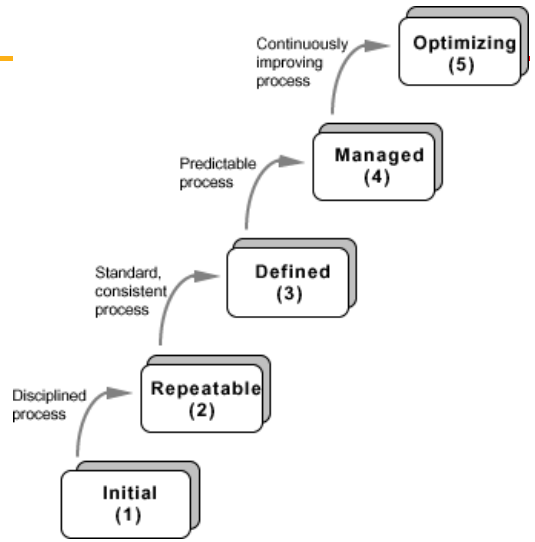
- **Software Process Performance**
  - Actual results achieved **by following a software process**
- **Software Process Maturity**
  - Refers to how well a process is **clearly defined, managed, measured, and controlled** to ensure its effectiveness.
  - It reflects the organization's ability to **grow and improve** and shows how **consistently and thoroughly** the process is applied across all projects.

What are the CMM Levels?

(The five levels of software process maturity)

Maturity level indicates level of process capability:

- Initial
- Repeatable
- Defined
- Managed
- Optimizing



Level 1: Initial

- The software process is ad hoc and sometimes even chaotic. There are very few defined processes, and success depends mostly on individual effort.
- At this level, the team often struggles to make commitments in an orderly manner.
- Products are usually completed over budget and behind schedule. There are large variations in cost, quality, schedule, and functionality. The capability resides in individuals rather than in the organization as a whole.

Level 2: Repeatable

Basic process management practices are established to **monitor cost, schedule, and functionality**. A consistent process discipline allows teams to repeat successful outcomes from previous similar projects.

Realistic project commitments are made based on past experiences. Software standards are defined and followed. Processes may vary between projects but are carried out in a disciplined manner. Successes from earlier projects can be reliably repeated.

Level 3: Defined

- The software process for both management and engineering activities is **documented, standardized, and integrated** into a defined software process across the organization.
- All projects follow an **approved and tailored version** of this standard process, ensuring consistency in software development and maintenance practices.

Level 4: Managed

- At this level, detailed measurements of the software process and product quality are collected. Both are quantitatively understood and controlled.
- Process variations are kept within acceptable limits.
- Corrective actions are taken when performance deviates from expected bounds.
- The process is quantifiable, predictable, and can be used to forecast trends in process and product quality.

Level 5: Optimizing

- Continuous improvement is driven by quantitative feedback from the process and by piloting innovative ideas and technologies.
- The focus is on preventing defects through causal analysis and proactive measures.
- Process effectiveness data is used for cost-benefit analysis of new technologies and to support proposed process changes, ensuring ongoing optimization and innovation.

Break-time Brain Teaser:

An IT company has reached a high level of process maturity. It uses defect trend data to identify recurring issues, applies root cause analysis to prevent future defects, and runs pilot projects to evaluate the benefits of new development tools before adopting them company-wide.

Which of the following practices indicate the organization is operating at CMMI Level 5 – Optimizing?
(Select all that apply)

a) Using root cause analysis to prevent defects
b) Conducting pilot studies for new technologies
c) Following basic project management practices
d) Applying data-driven feedback for continuous improvement
e) Relying only on historical practices without change

Break-time Brain Teaser:

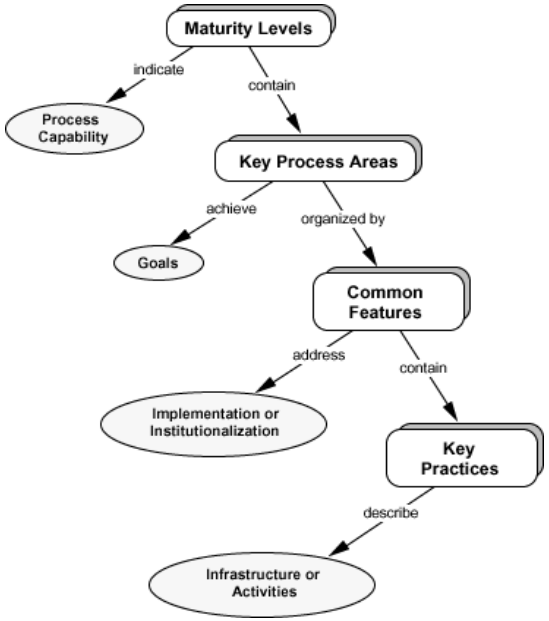
innovate achieve lead

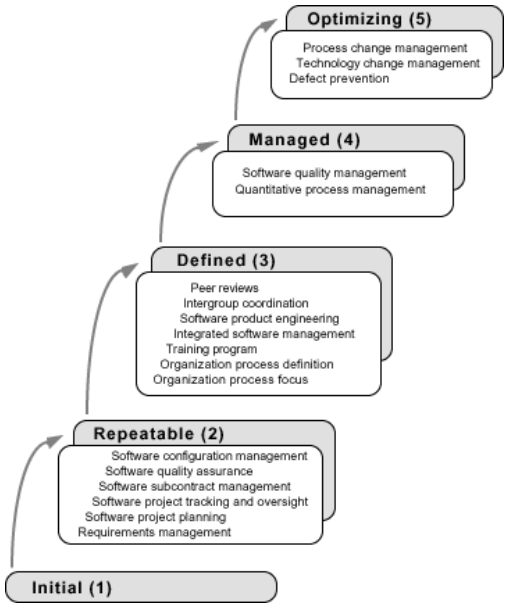
Answer:

a) Using root cause analysis to prevent defects
b) Conducting pilot studies for new technologies
d) Applying data-driven feedback for continuous improvement

Internal Structure to Maturity Levels

- Except for level 1, each level is decomposed into key process areas (KPA)
- Each KPA identifies a cluster of related activities that, when performed collectively, achieve a set of goals considered important for enhancing software capability.
  - commitment
  - ability
  - activity
  - measurement
  - verification





The Key Process Areas by Maturity Level

Level 2 KPAs

Requirements Management
- Establish common understanding of customer requirements between the customer and the software project
- Requirements is basis for planning and managing the software project
- Not working backwards from a given release date!

Software Project Planning
- Establish reasonable plans for performing the software engineering activities and for managing the software project.

Level 2 KPAs

Software Project Tracking and Oversight
- Establish adequate visibility into actual progress
- Take effective actions when project's performance deviates significantly from planned

Software Subcontract Management
- Manage projects outsourced to subcontractors

Software Quality Assurance
- Ensure visibility into the process and the quality of work products.

Level 2 KPAs

Software Configuration Management
- Track and control changes to software and documents.
- Maintain the integrity of all work products.
- Use product baselines as approved reference versions.
- Allow changes only through authorized baseline control.

Level 3 KPAs

1. Organization Process Focus
- Assign teams to improve and manage software processes.
- Goal: Strengthen the organization's overall development process.

2. Organization Process Definition
- Create and maintain a standard set of process guidelines.

- **Provides a stable base for:**
  - Consistent use across projects.
  - Collecting useful data for improving processes.

Level 3 KPAs

Training Program:

- Ensure team members have the right skills and knowledge.
- Training is based on project needs and managed by the organization.

Integrated Software Management

- Combine engineering and management activities.
- Use standard processes, customized for specific project and business needs.

Level 3 KPAs

Software Product Engineering

•All technical work (like in the Software Development Life Cycle) is clearly planned.

•Work results are accurate and consistent.

Intergroup Coordination

•Software teams work well and actively with other departments.

Peer Reviews

•Helps find and fix problems early.

•Improves understanding of the product.

•Done using reviews like inspections and walkthrough

Level 4 KPAs

Quantitative Process Management

•Use numbers and data to measure how well the process is working.

•Compare actual results with expected results.

•Find and fix problems that cause unexpected changes in the process.

•**Software Quality Management**

•Use data to understand and improve software quality.

•Focus on both:

- •the **product** (the software itself)
- •the **process** (how the software is developed)

Level 5 KPAs

1. **Process Change Management**
   1. Keep improving processes to:
      1. make better quality products
      2. work faster
      3. increase productivity
2. **Technology Change Management**
   1. Find and use helpful new tools, methods, or processes to improve work.
3. **Defect Prevention**
   1. Understand why defects happen and stop them from happening again.

What are the benefits ?

- Helps forge a shared vision of what software process improvement means for the organization
- Defines set of priorities for addressing software problems
- Supports measurement of process by providing framework for performing reliable and consistent appraisals
- Provides framework for consistency of processes and product

Why measure software and software process?

Obtain data that helps us to better control:

- Schedule
- Cost
- Quality of software products

Consistent measurement provide data for:

- Define clear goals, requirements, and success criteria
- Track progress and detect problems early
- Make smart decisions about resource allocation
- Accurately estimate time, cost, and quality

Measurements

1. **Historical** – Data from past projects (used for comparison and learning).
2. **Plan** – What we expect to happen (like timelines and costs).
3. **Actual** – What really happened during the project.
4. **Projections** – Forecasts for future progress based on current trends.

SEI Core Measures

Unit of Measure	Characteristics Addressed
Physical source lines of code Logical source lines of code	Size, reuse, rework
Staff hours	Effort, cost, resource allocations
Calendar dates for process milestones Calendar dates for deliverables	Schedule, progress
Problems and defects	Quality, improvement trends, rework, readiness for delivery

Examples of measurements for size of work products

Estimated number of requirements
Actual number of requirements
Estimated source lines of code (SLOC)
Actual SLOC
Estimated number of test cases
Actual number of test cases

Example of measurements of effort

- Estimated man-hours to design/code a given module
- Actual man-hours expended for designing /coding the module
- Estimated number of hours to run builds for a given release
- Actual number of hours spent running builds for the release

Examples of measurements of quality of the work product

- Number of issues raised at requirements inspection
- Number of requirements issues open
- Number of requirements issues closed
- Number of issues raised during code inspection
- Number of defects opened during unit testing

Examples of measurements of quality of the work product

- Number of defects opened during system testing
- Number of defects opened during UAT
- Number of defects still open
- Number of defects closed
- Defect age

Examples of measurements of quality of the work product

- Total number of build failures
- Total number of defects fixed for a given
- release
- Total number of defects verified and accepted
- Total number of defects verified and rejected

innovate achieve lead

Thank You

Scenario: Building a Mobile Banking App

Level 1: Initial (Ad Hoc)

The company decides to build a banking app.

- Developers start coding without planning.
- No timelines or quality checks.
- Features change often; bugs appear late.
- The project is delayed and over budget.

Level 2: Repeatable (Basic Project Management)

After struggling with the first version, the team creates a basic checklist.

- They use past project timelines to plan the next version.
- Some steps (e.g., testing and deployment) are reused from previous success.
- Delivery is more controlled, but only for similar projects.

Level 3: Defined (Standardized Process)

The company documents a full app development lifecycle.

- All teams use a defined process: planning → design → coding → testing → review.
- Everyone follows the same process with some tailoring for specific needs.

Level 4: Managed (Quantitative Control)

The company now tracks key metrics:

- Bug count, response time, test coverage, customer feedback.
- They set quality thresholds (e.g., < 5 bugs per release).
- Deviations trigger corrective action before release.

Level 5: Optimizing (Continuous Improvement)

The team reviews all project data.

- They notice manual testing is slow → they test automated tools.
- A pilot project shows 30% faster testing and fewer bugs.

- Automation is adopted company-wide to optimize quality and speed.
- They regularly experiment with new ideas based on data.

■ CMMI Structure (Based on Diagram Flow)

1. **Maturity Levels**
   → Indicate the **overall process capability** of an organization.
2. **Key Process Areas (KPAs)**
   → Each maturity level **contains** KPAs — groups of related activities that help improve processes.
3. **Goals**
   → Each KPA is designed to **achieve specific goals** that are important for process improvement.
4. **Common Features**
   → KPAs are **organized by common features** (like commitment, ability, and measurement).
5. **Implementation or Institutionalization**
   → Common features **address how to make processes part of daily work**, ensuring they are followed consistently.
6. **Key Practices**
   → Common features **contain key practices**, which are the **actual steps** or actions to perform.
7. **Infrastructure or Activities**
   → Key practices **describe the tools, training, or actions** needed to carry out the process effectively.

Key Process Areas (KPAs) in CMMI:

◆ 1. Commitment

The organization shows strong **intent and responsibility** to perform the process.

✅ *Example:* Management agrees to provide resources and support to implement quality standards.

◆ 2. Ability

The organization has the **resources, skills, and tools** needed to carry out the process.

✅ *Example:* The team has trained staff and access to testing tools to conduct quality assurance.

◆ 3. Activity

The specific **steps or tasks** that must be carried out to implement the process.

✅ *Example:* Conducting code reviews or documenting software requirements.

◆ 4. Measurement

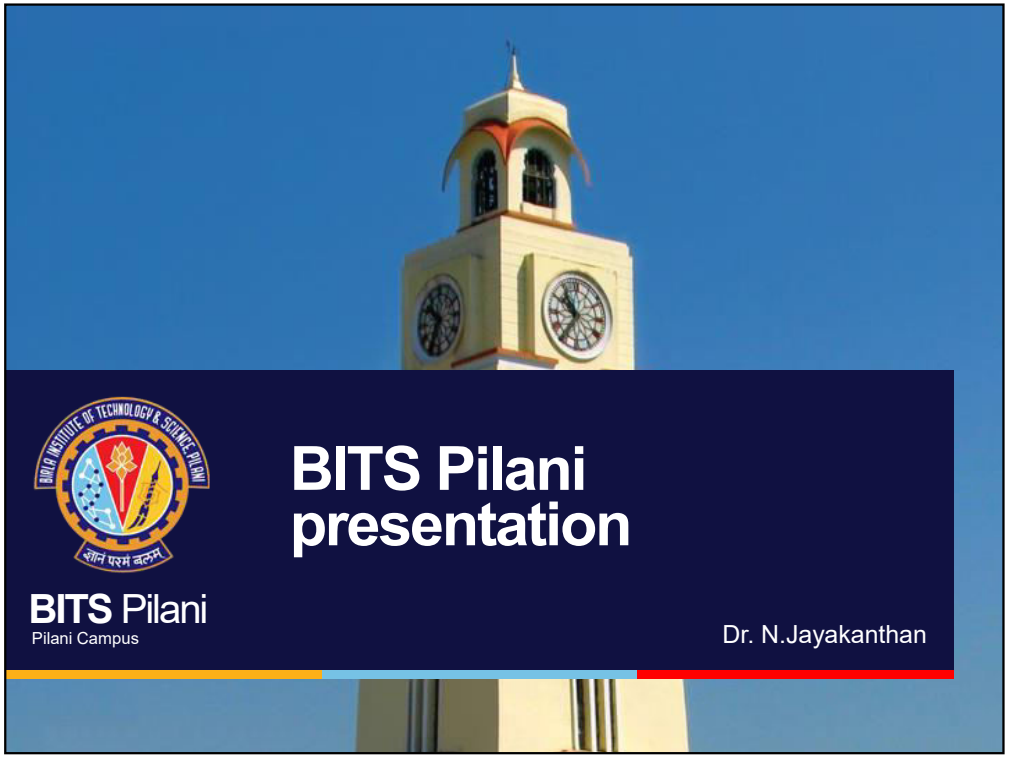
The process is tracked using **metrics** to check performance and progress.

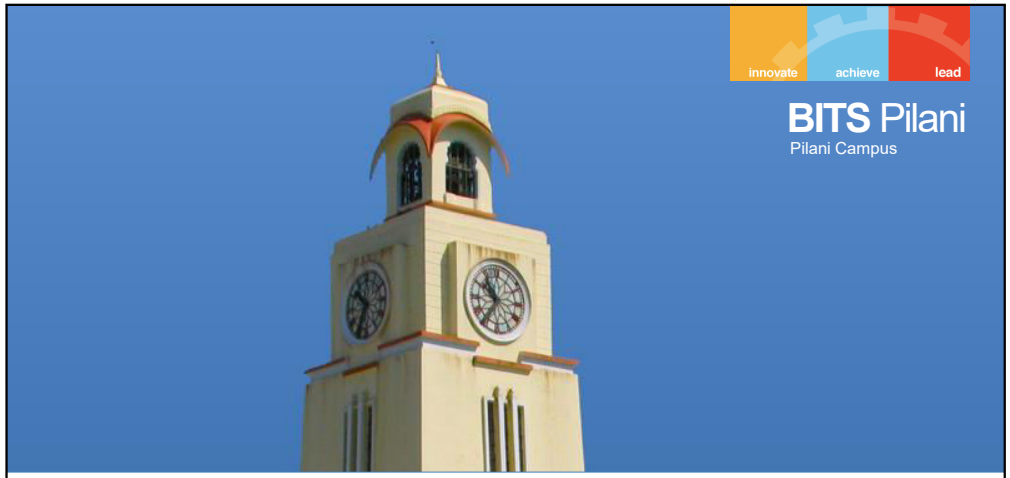
✅ *Example:* Measuring the number of defects per module or average resolution time.

◆ 5. Verification

Ensuring that the **process** is followed correctly and meets its goals.

✅ *Example:* Auditing project documents or conducting peer reviews to confirm process adherence.





SE ZG501
Software Quality Assurance and Testing
Lecture No. 13

Examples of measurements of quality of the work product

- Number of defects opened during system testing
- Number of defects opened during UAT
- Number of defects still open
- Number of defects closed
- Defect age

Examples of measurements of quality of the work product

- Total number of build failures
- Total number of defects fixed for a given release
- Total number of defects verified and accepted
- Total number of defects verified and rejected

levels of CMMI

An organization receives one of two ratings during a Class A appraisal: a maturity level rating or a Capability level rating.

Maturity levels range from 1 to 5, with level 5 as the highest grade and the target businesses aim towards.

1. Initial Level

The processes at this CMMI level tend to be erratic and reactive. The organization is at its worst at this point due to the unpredictability of the environment and the likelihood of errors and ineptitude.

2. Managed processes

- At Maturity Level 2, an organization has achieved the basic goals for its important process areas.
- This means they now plan their work, carry it out as per plan, track progress, and keep control over cost, time, and quality.
- The processes are repeatable and can be managed on a project-by-project basis.

Standards implemented at this level are usually as follows:

Requirements Management, or REQM
Project Planning (PP)
Configuration Management, or CM
Measurement and Analysis (MA)
Process and Product Quality Assurance (PPQA)
Project Monitoring and Control, or PMC
Supplier Agreement Management (SAM).

3. Defined processes

Organizations take a more preventative approach than a reactive one at this level.
Managers are now aware of the flaws and how to fix them to enhance their operations. There are several KPAs of which helps to offer direction across projects, departments, and business units.:

- Decision Analysis and Resolution, or DAR.
- Organizational Process Focus (OPF)
- Integrated Project Management (IPM) plus IPPD (Integrated Product and Process Development)
- OPD stands for organizational process definition, while OT stands for executive training.
- Product Integration (PI)
- Risk Management: RSKM
- Validation, or VAL
- Technical Solution (TS)
- Verification, VER

4. Managed quantitatively

The company has reached a high maturity level and relies on predictable methods based on the stakeholders' needs.

- Processes are well-structured, consistent, and precisely executed.
- The organization actively identifies risks and applies quantitative strategies to manage and improve processes.
- Decisions are based on measured performance data, not
- assumption ,
- This is reflected in the implementation of the following Key Process Areas (KPAs):
  - OPP – Organizational Process Performance
  - QPM – Quantitative Project Management

5. Optimizing

At **Maturity Level 5**, the organization operates in a **stable yet flexible environment** that supports continuous improvement:

- The focus is on **agility, innovation, and proactive problem-solving**.
- The company continuously seeks new opportunities for **process improvement** and **performance enhancement**.
- Improvement is driven by **quantitative feedback** and **innovative practices**, not just by reacting to issues.

Organizations at this level demonstrate a **high degree of maturity**, consistently evolving to meet the changing expectations of **clients, stakeholders, and market demands**.

Six Sigma in Software Engineering

- Six Sigma is a methodology that helps organizations in making their process better and more efficient by identifying and removing errors and variations.
- Variations in processes can lead to errors, these errors can lead to product defects and product defects can lead to poor customer satisfaction.
- By reducing variation and errors Six Sigma can reduce process costs and increase customer satisfaction.

It is a statistical concept that aims to define the variation found in any process.

Six Sigma is a process of producing high and improved quality output. This can be done in two phases – identification and elimination.

The cause of defects is identified and appropriate elimination is done, which reduces variation in whole processes.

Six Sigma processes have a failure rate of only 3.4 per million opportunities i.e. 99.99966 percent of Six Sigma products are free from defect, while Five Sigma processes have a failure rate of only 233 errors per million opportunities.

Characteristics of Six Sigma

Statistical Quality Control: Six Sigma is derived from the Greek Letter ? which denote Standard Deviation in statistics. Standard Deviation is used for measuring the quality of output.

Methodical Approach: The Six Sigma is a systematic approach of application in DMAIC and DMADV which can be used to improve the quality of production. **DMAIC** means for Design-Measure- Analyze-Improve-Control. While DMADV stands for Design-Measure-Analyze-Design-Verify.

Fact and Data-Based Approach: The statistical and methodical method shows the scientific basis of the technique.

1. **Project and Objective-Based Focus:**
   The Six Sigma process is implemented to address specific requirements and conditions through well-defined projects and goals.
2. **Customer Focus:**
   Customer focus is a core element of the Six Sigma methodology. Quality improvements and control standards are tailored to meet the specific needs of customers.
3. **Teamwork Approach to Quality Management:**
   Six Sigma emphasizes collaborative efforts within the organization to systematically enhance quality and efficiency.

The 6 Key Principals of Six Sigma

Customer Centric Improvement

Customer Centric Improvement

The primary principal of six sigma methodology is to focus on customer.

Voice of the Customer (VoC) and methods for determining what the customer truly want from a product or process. Organizations can boost customer happiness by combining that knowledge with measurements, analytics, and process improvement approaches, resulting in **higher profits, client retention, and loyalty.**

Continuous Process Improvement

The Six Sigma approach requires constant process improvement. An organization that fully implements the Six Sigma technique never stops improving.

It continuously discovers and priorities opportunities. Once one area has been improved, the organization will move on to another.

The organization continuously find ways to increase the sigma level because the goal is to achieve the level of 99.99966 accuracy for all processes inside an organization while also making sure other essentials like financial stability.

Reduce Variation

To improve a process, it's important to reduce variation. Every process has some natural variation, but too much variation can lead to errors. These errors may cause product defects, which result in low customer satisfaction.

By reducing variation and mistakes, Six Sigma helps lower costs and improve customer satisfaction.

For example, in software development, each developer may write code differently due to their experience, style, and environment. This causes variation.

To reduce this, organizations can:

- Use coding standards and guidelines
- Conduct code reviews
- Automate testing
- Maintain proper documentation

These steps help make processes more consistent and reduce errors.

Eliminating Waste

Waste is a major problem in the six sigma methodology.

Eliminating waste means removing items, procedures or people that are not required for the process's outcome or **removing anything that does not add value to customer**.

Eliminating waste **can reduce processing time, errors in process and lowers overall costs.**

Empowering Employees

Until organizations provide employees with the tools they need to monitor and sustain improvements, implementing improved processes is only a temporary solution.

Process improvement usually involves two approaches in most organizations.

An improvement is first defined, planned, and carried out by a process improvement team consisting up of project managers, methodology specialists, and subject matter experts.

The employees that deal with the process on a daily basis are then equipped by that team to supervise and handle it in its improved condition.

Controlling the Process

Six Sigma improvements are frequently used to handle uncontrolled processes. Out-of-control processes meet certain statistical conditions.

The purpose of improvement is to bring a process back under statistical control.

Then, after the improvements are implemented, measurements, statistics, and other Six Sigma tools are utilized to keep the process under control. Implementing controls and training people on how to apply them is a key component of continuous improvement.

The Six Sigma Methodology

The Two Six Sigma methodologies used in the Six Sigma projects are DMAIC and DMADV.

Six Sigma teams usually use DMAIC or DMADV approaches to achieve process improvements and establish process control.

Question: An organization is working to improve its process accuracy. Initially, it has a Five Sigma quality level (233 defects per million). After implementing Six Sigma techniques, it achieves 3.4 defects per million. Which of the following BEST describes what the organization has achieved?

a) Increased the process variation and reduced customer satisfaction.
b) Reduced the process variation and improved product quality.
c) Increased the number of defects but reduced costs.
d) Reduced defects but ignored continuous improvement..

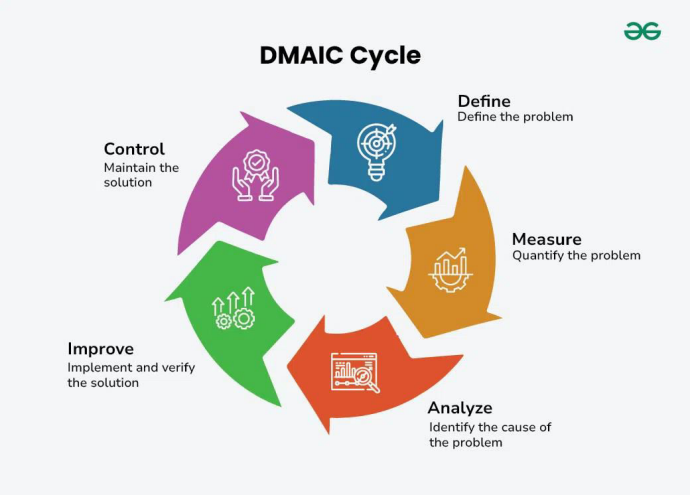
Answer: b) Reduced the process variation and improved product quality

DMAIC Six Sigma Methodology

DMAIC is used to **enhance an existing business process**.

A DMAIC project involves identifying important problem that are creating the problem, verifying those problem, brainstorming solutions, implementing them, and designing a control plan to maintain the improved state.

The DMAIC methodology is designed for the team who are responsible for improving a project.



The DMAIC project methodology has five phases:

- Define
- Measure
- Analyze
- Improve
- Control

Define

The Define phase of a DMAIC project involves identifying problems, establishing project requirements, and setting success goals.

Six Sigma leaders use specific tools during this phase to adapt the approach for different projects, based on leadership input, available resources, and budget.

Measure

In the **Measure** phase of DMAIC, the goal is to:

- **Collect data** to understand the current state of the process.
- **Use data to confirm or reject assumptions** about the problem.

This phase is important because reliable data is needed for effective analysis.

🔍 **Key Activities:**

- Gather and organize relevant data.
- Build tools or use software to extract information.
- Use filters or manual checks to process large data sets.
- Without good data, it's hard to measure or improve anything accurately.

Analyze

Analyze phase is a critical stage where the root causes of problems or inefficiencies within a process are identified and understood.

During the Analyze phase of a DMAIC project, teams develop predictions about relationships between inputs and outputs, use statistical analysis and data to validate the prediction and assumptions they've made thus far.

In a DMAIC project, the Analyze phase leads to the Improve phase, where hypothesis testing can confirm assumptions and potential solutions.

Improve

- Teams **develop and test solutions** based on what they learned during the **Analyze** phase.
- They use **statistics and real-life observations** to check if their ideas work.
- As they start applying the solutions, they keep doing **hypothesis testing** to make sure the changes are effective.

☞ The goal is to **fix the root cause** and make the process **better and more efficient**.

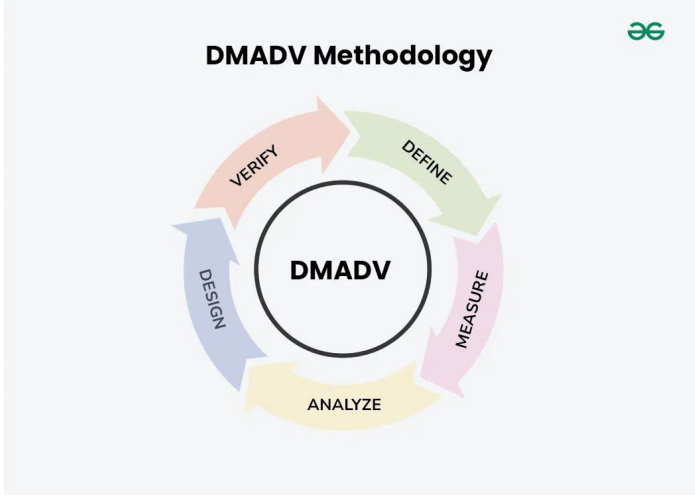
Control

- This is the **final step** of the DMAIC process.
- The goal is to **make sure the improvements last** over time.
- Teams create **rules, tools, and standards** to keep the process running smoothly.
- They also **train the people** (called **process owners**) who will manage the improved process every day.

DMADV Six Sigma Methodology

DMADV is used to create new product designs or process designs. Six Sigma teams use DMADV in the following scenario:

- The organization wants to launch a new service or product.
- Business leaders decide to replace a process to meet upgrade requirements or to align business processes, machinery, or workers with future goals.
- A Six Sigma team learns that **upgrading an existing process is unlikely to achieve the expected outcomes**, and a new design is necessary to meet quality and performance standards..



The DMADV project methodology also has **five** phases:

- Define
- Measure
- Analyze
- Design
- Verify

Define

In the **Define** stage of a DMADV project, the focus is on understanding the problem clearly and setting project goals.

- Teams **identify the problem** or opportunity.
- Requirements are defined **based on customer needs** and expectations.
- If a **change management program** is already in place in the organization, its needs must be integrated into this stage.
- This phase sets the **foundation** for all future design work, making it more structured and clear than in DMAIC.
- Define stage is slightly more strict than in DMAIC.

Measure

During the DMADV Measure phase, teams use data to validate assumptions about the process and problem. Validation of assumptions also makes it into the analysis step. **The measurement phase focuses on collecting and arranging data for analysis**.

Analyze

Analyze phase is a critical stage where the root causes of problems or inefficiencies within a process are identified and understood. They priorities identifying best practices and standards for measuring and designing new processes.

Design

The **Design** phase is where DMADV begins to diverge significantly from DMAIC.

- In this stage, teams **create a detailed blueprint** of the new process or product.
- The focus is on building a solution from the ground up, guided by insights gained in earlier phases.
- The design process includes:
  - **Solution testing and prototyping**
  - **Process mapping and workflow planning**
  - **Infrastructure and system development**
  - **User experience and operational efficiency considerations**

☞ **Key Focus:** Designing a robust, scalable solution that meets customer and business needs.

Verify Phase

The Verify phase in DMADV checks if the designed solutions work as intended, measuring their success against initial goals, ensuring improvements are effective and sustainable.

Difference between DMAIC and DMADV

The primary difference between **DMAIC** and **DMADV** lies in their **team goals** and **project outcomes**:

- **DMAIC** is used to **improve existing processes**.
- **DMADV** is used to **design new processes or products** from the ground up.

Both methodologies aim to:

- Deliver **better quality**
- Improve **efficiency and productivity**
- Increase **profits**
- Ensure **customer satisfaction**

Conclusion- six sigma

Six Sigma is a structured methodology used by organization to improve processes by reducing inherit variation and defects.

Six Sigma helps the organization in improving the efficiency, quality and customer satisfaction by reducing variation and defects in processes.

Six Sigma consists of two methodology DMAIC and DMADV.

DMAIC stands for "D-Define", "M-Measure", "A-Analysis", "I-Improve", "C-Control" . DMAIC is used to enhance an existing business process.

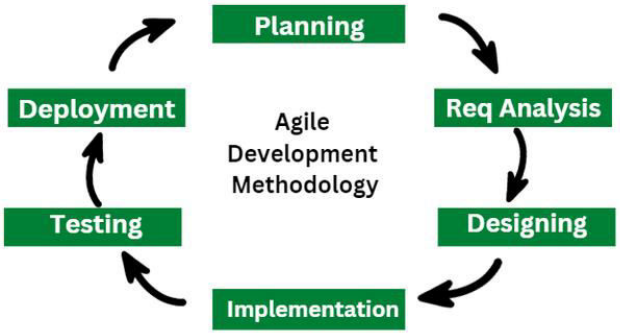
DMADV stands for "D-Define", "M-Measure", "A-Analysis", "D-Design", "V-Verify". DMADV is used to create new product designs or process designs.

Introduction to Agile Methodology and Testing

Agile is a project management and software development approach that aims to be more effective.

- It focuses on delivering smaller pieces of work regularly instead of one big launch.
- This allows teams to adapt to changes quickly and provide customer value faster.

Life cycle of Agile Methodology



Agile Software Testing

innovate achieve lead

Agile Testing is a type of software testing that follows the principles of agile software development to test the software application.

All members of the project team along with the special experts and testers are involved in agile testing.

Agile testing is not a separate phase and it is carried out with all the development phases i.e. requirements, design and coding, and test case generation.

Agile testing takes place simultaneously throughout the Development Life Cycle.

innovate achieve lead

- Agile testers are involved throughout the entire software development life cycle. They collaborate closely with developers to ensure that the software meets customer requirements. This leads to better design and more reliable code.
- The **testing team and development team function as one unified team** with a shared goal of delivering high-quality software.
- Agile Testing follows short time frames called iterations or loops.
- This approach is also known as delivery-driven, as it **enables faster feedback and better predictions** on working software within shorter time spans.

innovate achieve lead

Thank You

These levels describe how mature the overall process of an organization is.

Maturity Level	Name	Description
Level 1	**Initial**	No structured process. Everything is ad hoc and chaotic.
Level 2	**Managed**	Projects follow basic project management practices.
Level 3	**Defined**	Organization-wide standards are used in all projects.
Level 4	**Quantitatively Managed**	Uses data and metrics to manage and control processes.
Level 5	**Optimizing**	Focus on continuous improvement using innovative ideas and feedback.

Capability Level in CMMI (0–5)

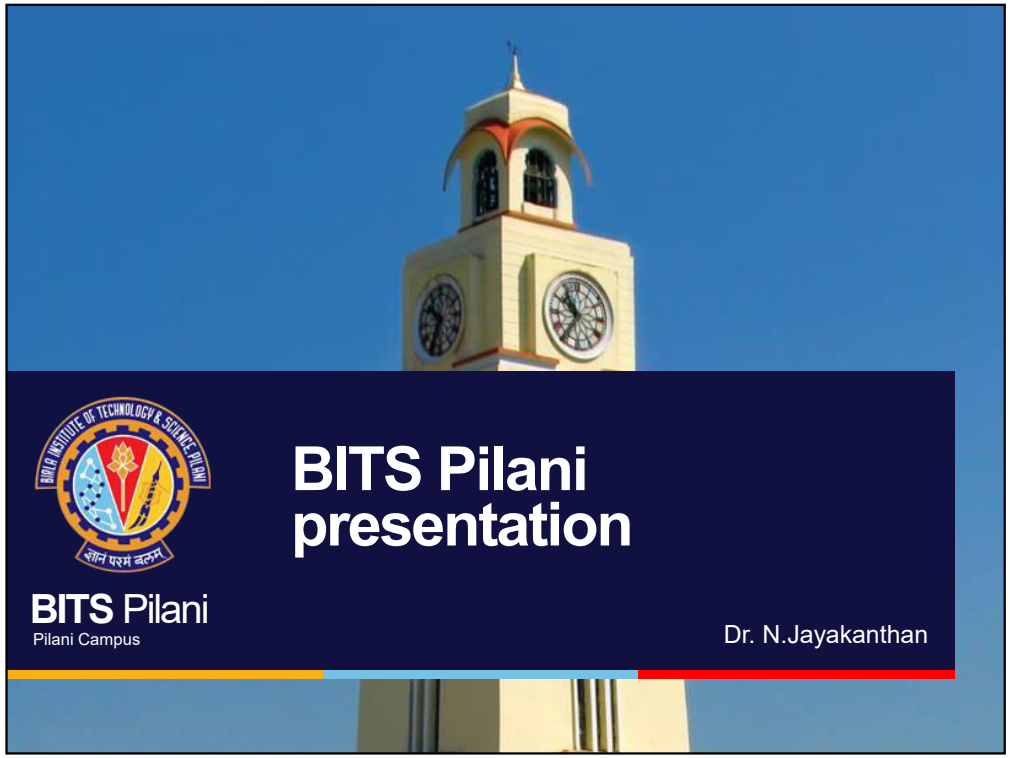
Capability Levels are used in the **Continuous Representation** of CMMI to assess **how well a specific process area** is implemented.

Levels of Capability (0 to 5):

Level	Name	Description
0	**Incomplete**	Process is not performed or only partially done.
1	**Performed**	Process is executed and delivers basic work products.
2	**Managed**	Process is planned, tracked, and has basic project management.
3	**Defined**	Process is standardized and tailored from organizational standards.
4	**Quantitatively Managed**	Process is controlled using quantitative data.
5	**Optimizing**	Focus on continuous improvement using feedback and innovation.

Difference from Maturity Levels:

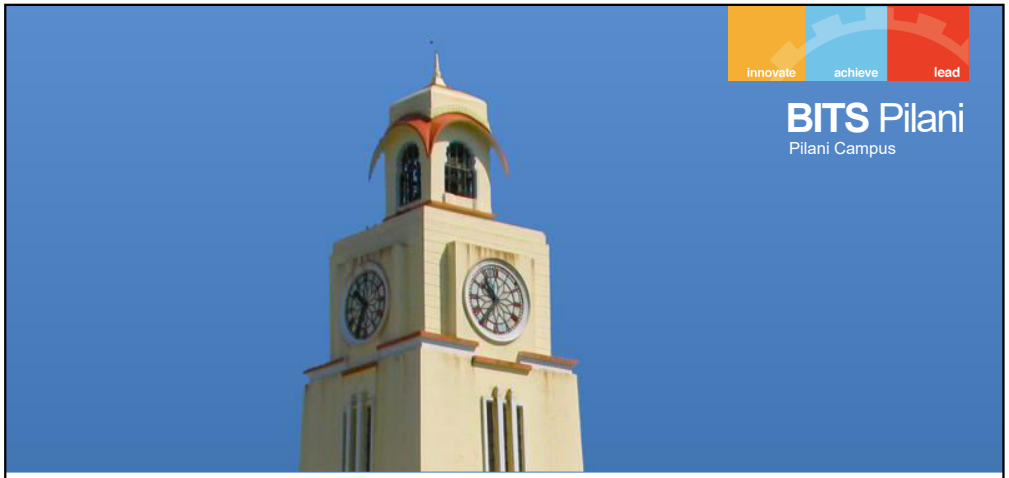
- **Capability Levels** → Focus on **individual process areas**
- **Maturity Levels** → Reflect the **organization as a whole**



BITS Pilani presentation

BITS Pilani
Pilani Campus

Dr. N.Jayakanthan



innovate achieve lead

BITS Pilani
Pilani Campus

SE ZG501
Software Quality Assurance and Testing
Session 14

- In Agile, testing is an ongoing activity rather than a distinct phase, focusing on collaboration between testers, developers, and stakeholders.
- This approach helps detect and fix issues early, aligning with Agile's iterative development cycle.

Agile Testing Principles

Shortening feedback iteration: In Agile Testing, the testing team gets to know the product development and its quality for each and every iteration. Thus continuous feedback minimizes the feedback response time and the fixing cost is also reduced.

Testing is performed alongside Agile testing is not a different phase. It is performed alongside the development phase. It ensures that the features implemented during that iteration are actually done. Testing is not kept pending for a later phase.

Involvement of all members: Agile testing involves each and every member of the development team and the testing team. It includes various developers and experts.

Documentation is weightless: In Instead of creating large, detailed documents, agile testers rely on simple and reusable checklists to plan tests. They focus on the core purpose of testing, avoiding unnecessary details, and use lightweight documentation tools to stay efficient

Clean code: The defects that are detected are fixed within the same iteration. This ensures clean code at any stage of development.

Constant response: Agile testing helps to deliver responses or feedback on an ongoing basis. Thus, the product can meet the business needs.

Customer satisfaction: In agile testing, customers are exposed to the product throughout the development process. Throughout the development process, the customer can modify the requirements, and update the requirements and the tests can also be changed as per the changed requirements.

Test-driven: In agile testing, testing is performed **together with development** to speed up the overall process. In contrast, in traditional methods, testing usually starts **only after** the software has been fully developed.

Agile Testing Methodologies

Test-Driven Development (TDD): TDD is the software development process relying on creating unit test cases before developing the actual code of the software. It is an iterative approach that combines 3 operations, programming, creation of unit tests, and refactoring.

Behavior Driven Development (BDD): BDD is agile software testing that aims to document and develop the application around the user behavior a user expects to experience when interacting with the application. It encourages collaboration among the developer, quality experts, and customer representatives.

Exploratory Testing: In exploratory testing, the tester has the freedom to explore the code and create effective and efficient software. It helps to discover the unknown risks and explore each aspect of the software functionality.

Acceptance Test-Driven Development (ATDD): ATDD is a collaborative process where customer representatives, developers, and testers come together to discuss the **requirements, and potential pitfalls** and thus **reduce the chance of errors before coding begins.**

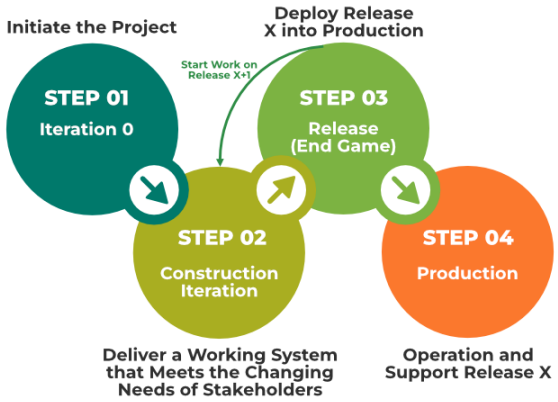
Extreme Programming (XP): Extreme programming is a customer-oriented methodology that helps to deliver a good quality product that meets customer expectations and requirements.

Session-Based Testing: It is a structured and time-based approach that involves the progress of exploratory testing in multiple sessions. This involves uninterrupted testing sessions that are time-boxed with a duration varying from 45 to 90 minutes. During the session, the tester creates a document called a **charter document** that includes various information about their testing.

Dynamic Software Development Method (DSDM): DSDM is an agile project delivery framework that provides a framework for building and maintaining systems. It can be used by users, developers, and testers.

Crystal Methodologies: This methodology focuses on people and their interactions when working on the project instead of processes and tools. The suitability of the crystal method depends on three dimensions, team size, criticality, and priority of the project.

Agile Testing Strategies



Iteration 0

Iteration 0:

- This is the **first stage** of the testing process where the **initial setup** is done.
- **The testing environment is created** in this phase.

Key activities:

- Find people for testing, set up the usability testing lab, and prepare necessary resources.
- Verify the project's business case, project boundaries, and scope.
- Summarize important requirements and use cases.
- Plan the project's initial costs and valuation.
- Identify potential risks.
- Create one or more possible design options for the project.

Construction Iteration

innovate achieve lead

It is the second phase of the testing process and the main phase where most of the work happens.

It includes multiple smaller iterations to gradually build and improve the solution.

Confirmatory testing: This type of testing concentrates on Focuses on **checking if the system meets stakeholder requirements** as shared with the team so far. It is performed by the development team and has two types:

Agile acceptance testing: It is the combination of acceptance testing and functional testing. It can be executed by the development team and the stakeholders.

Developer testing: It is the combination of unit testing and integration testing and verifies both the application code and database schema.

innovate achieve lead

Investigative Testing:

- This testing **finds problems that were missed** during confirmatory testing.
- Testers look for **hidden or unexpected issues** by creating **defect stories.**

Focus areas:

- Integration testing
- Load testing
- Security testing
- Stress testing

BITS Pilani, Pilani Campus

Release End Game

innovate achieve lead

Release End Game (Transition Phase):

- This is the **final phase** before product release.
- It includes **full system testing** and **acceptance testing**.
- The product is **tested very thoroughly** to find and fix any remaining issues (defect stories).

Key activities in this phase:

- Train the end-users.
- Support operational and service people.
- Plan and execute product marketing.
- Set up backup and recovery systems.
- Finalize system and user documentation.

BITS Pilani, Pilani Campus

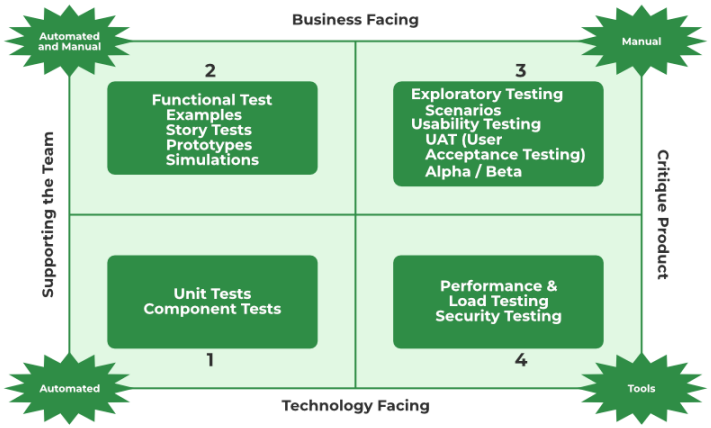
Production

innovate achieve lead

It is the last phase of agile testing. The product is finalized in this stage after the removal of all defects and issues raised.

BITS Pilani, Pilani Campus

Agile Testing Quadrants



Quadrant 1 (Automated)

The first agile quadrat focuses on the internal quality of code which contains the test cases and test components that are executed by the test engineers.

All test cases are technology-driven and used for automation testing. All through the agile first quadrant of testing, the following testing can be executed:

- Unit testing.
- Component testing.

Quadrant 2 (Manual and Automated)

The second agile quadrant focuses on the customer requirements that are provided to the testing team before and throughout the testing process. The test cases in this quadrant are **business-driven** and are used for manual and automated functional testing. The following testing will be executed in this quadrant:

- Pair testing.
- Testing scenarios and workflow.
- Testing user stories and experiences like prototypes.

Quadrant 3 (Manual)

Quadrant 3 (Manual Testing):

- This quadrant gives **feedback** to Quadrant 1 (internal quality) and Quadrant 2 (business needs).
- It involves **multiple rounds of manual testing** to **review, strengthen, and improve the code.**
- The feedback collected here is used to **prepare better automation tests** later.

Main types of testing done:

- **Usability Testing:** Check how easy and friendly the system is for users.
- **Collaborative Testing:** Testers and others (like developers or customers) work together to find issues.
- **User Acceptance Testing (UAT):** Verify if the system meets customer requirements.
- **Pair Testing with Customers:** Testers and customers test together to explore the system.

Quadrant 4 (Tools)

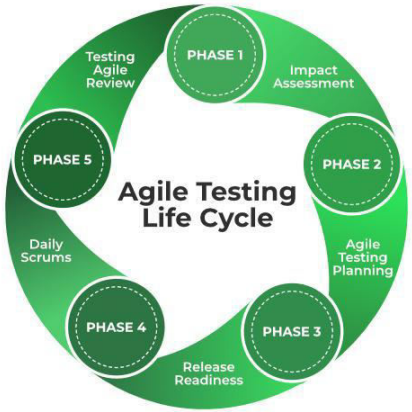
Quadrant 4 (Tools):

- Focuses on the **non-functional requirements** of the product like **performance**, **security**, and **stability**.
- The goal is to ensure the system is **strong, secure, and scalable**.
- This quadrant uses **special testing tools** to measure system qualities and expected performance.

Main types of testing done:

- **Non-functional Testing:** Stress testing, load testing, performance testing.
- **Security Testing:** Check for vulnerabilities and risks.
- **Scalability Testing:** Ensure the system can handle growth (more users, data).
- **Infrastructure Testing:** Check server, network, cloud setup.
- **Data Migration Testing:** Ensure data moves safely between systems.

Agile Testing Life Cycle



Agile Testing Life Cycle :
5 different phases:

Impact Assessment: This is the first phase of the agile testing life cycle also known as the feedback phase where the inputs and responses are collected from the users and stakeholders. This phase supports the test engineers to set the objective for the next phase in the cycle.

Agile Testing Planning: In this phase, the developers, customers, test engineers, and stakeholders team up to plan the testing process schedules, regular meetings, and deliverables.

Release Readiness: This is the third phase in the agile testing lifecycle where the test engineers review the features which have been created entirely and test if the features are ready to go live or not and the features that need to be sent again to the previous development phase.

Daily Scrums: This phase involves the daily morning meetings to check on testing and determine the objectives for the day. The goals are set daily to enable test engineers to understand the status of testing.

Test Agility Review: This is the last phase of the agile testing lifecycle that includes weekly meetings with the stakeholders to evaluate and assess the progress against the goals.

Brain Teaser Break

During an Agile sprint, a team is preparing to validate new features before release. The team involves a tester and developer working side-by-side, discusses how a user will complete a purchase flow, and simulates this flow using early interface designs.

Which of the following testing practices from **Quadrant 2** are being applied in this scenario?
(Select all that apply)

A) Pair Testing
B) Exploratory Testing
C) Workflow and Scenario Testing
D) Prototype-based Testing

Correct Answers: A, C, D

Agile Test Plan

1. **Purpose**:
    1. In Agile development, a test plan is essential for every iteration or release.
2. **What It Includes**:
    1. **Types of Testing**: Specifies the kinds of tests (e.g., unit, integration, regression) that will be conducted.
    2. **Test Data Requirements**: Defines what test data is needed.
    3. **Test Environments**: Describes where the tests will be executed (e.g., staging servers, test labs).
    4. **Test Results**: Captures and evaluates the outcomes of the testing activities.
3. **Frequency**:
    1. A new **test plan is created and updated for every release** to reflect changes in features or code.

The Agile test plan includes the following:

- Test Scope.
- Testing instruments.
- Data and settings are to be used for the test.
- Approaches and strategies used to test.
- Skills required to test.
- New functionalities are being tested.
- Levels or Types of testing based on the complexity of the features.
- Resourcing.
- Deliverables and Milestones.
- Infrastructure Consideration.
- Load or Performance Testing.
- Mitigation or Risks Plan.

Benefits of Agile Testing

Saves time: Implementing agile testing helps to make cost estimates more transparent and thus helps to save time and money.

Reduces documentation: It requires less documentation to execute agile testing.

Enhances software productivity: Agile testing helps to reduce errors, improve product quality, and enhance software productivity.

Higher efficiency: In agile software testing the work is divided into small parts thus developer can focus more easily and complete one part first and then move on to the next part. This approach helps to identify minor inconsistencies and higher efficiency.

Improve product quality: In agile testing, regular feedback is obtained from the user and other stakeholders, which helps to enhance the software product quality.

Limitations of Agile Testing

Project failure: In agile testing, if one or more members leave the job then there are chances for the project failure.

Limited documentation: In agile testing, there is no or less documentation which makes it difficult to predict the expected results as there are explicit conditions and requirements.

Introduce new bugs: In agile software testing, bug fixes, modifications, and releases happen repeatedly which may sometimes result in the introduction of new bugs in the system.

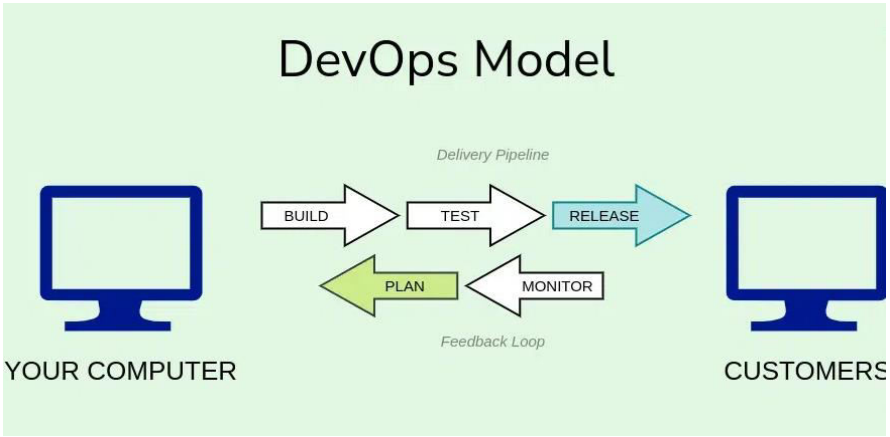
Poor planning: In agile testing, the team is not exactly aware of the end result from day one, so it becomes challenging to predict factors like cost, time, and resources required at the beginning of the project.

No finite end: Agile testing requires minimal planning at the beginning so it becomes easy to get sidetracked while delivering the new product. There is no finite end and there is no clear vision of what the final product will look like.

DEVOPS

- As technology grows and changes fast, there is a **huge need** for **faster and more reliable software delivery**.
- **DevOps** is introduced to **connect software development and IT operations**, making the process **smoother, faster, and more efficient**.

- DevOps is a **culture and practice** that **brings software development (Dev) and IT operations (Ops) together**.
- It encourages **teamwork** and **uses automation tools** to make software releases **faster, smoother, and more reliable**.
- DevOps focuses on **strong communication and collaboration** between developers and operations teams.
- The goal is to **speed up software delivery** and **improve the quality and reliability** of the final product.



Delivery Pipeline

The pipeline represents the different stages that software goes through before it is released to production. These stages might typically include:

Build: The stage where the software code is compiled and packaged into a deployable unit.

Test: The stage where the software is rigorously tested to ensure it functions as expected and identifies any bugs.

Release: The stage where the software is deployed to production for end users.

Feedback Loop

The loop indicates that information and learnings from the production environment are fed back into the earlier stages of the pipeline. This feedback can be used to improve the software development process and future releases.

How DevOps Works?

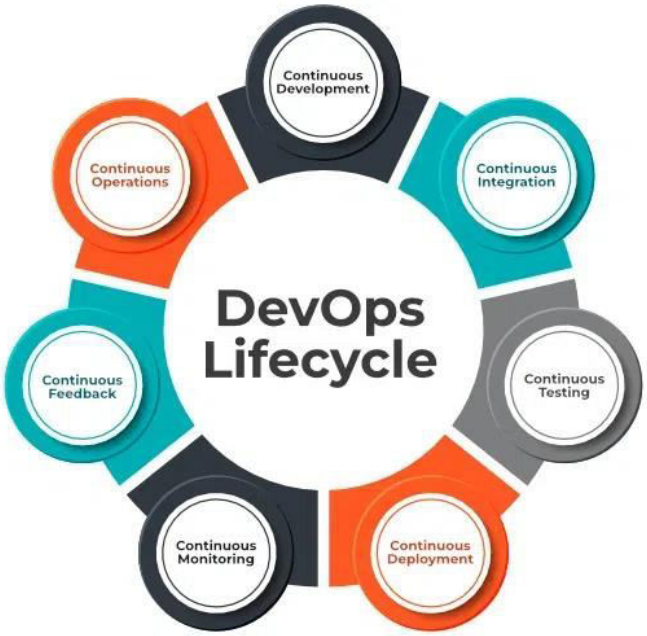
- DevOps will remove the Isolation conditions between the development team and operations team.
- In many cases these two teams will work together for the entire application lifecycle, from development and test to deployment to operations, and develop a range of skills not limited to a single function.

- Teams in charge of security and quality assurance may also integrate more closely with development and operations over the course of an application's lifecycle under various DevOps models.
- DevSecOps is the term used when security is a top priority for all members of a DevOps team.

DevOps Life Cycle

- DevOps is a practice that enables a single team to handle the whole application lifecycle, including development, testing, release, deployment, operation, display, and planning. It is a mix of the terms “Dev” (for development) and “Ops” (for operations).
- We can speed up the delivery of applications and services by a business with the aid of DevOps.
- Amazon, Netflix, and other businesses have all effectively embraced DevOps to improve their customer experience.

- **DevOps Lifecycle** is a series of steps where development and operations teams work together to deliver software faster.
- DevOps uses important steps like:
  coding, building, testing, releasing, deploying, operating, monitoring (displaying), and planning.
- DevOps lifecycle focuses on doing these activities continuously, such as:
  continuous development, integration, testing, monitoring, and feedback.



7 Cs of DevOps

- Continuous Development
- Continuous Integration
- Continuous Testing
- Continuous Deployment/Continuous Delivery
- Continuous Monitoring
- Continuous Feedback
- Continuous Operations

Benefits of DevOps

Faster Delivery: DevOps enables organizations to release new products and updates faster and more frequently, which can lead to a competitive advantage.

Improved Collaboration: DevOps promotes collaboration between development and operations teams, resulting in better communication, increased efficiency, and reduced friction.

Improved Quality: DevOps emphasizes automated testing and continuous integration, which helps to catch bugs early in the development process and improve the overall quality of software.

Increased Automation: DevOps enables organizations to automate many manual processes, freeing up time for more strategic work and reducing the risk of human error.

Better Scalability: DevOps enables organizations to quickly and efficiently scale their infrastructure to meet changing demands, improving the ability to respond to business needs.

Increased Customer Satisfaction: DevOps helps organizations to deliver new features and updates more quickly, which can result in increased customer satisfaction and loyalty.

Improved Security: DevOps promotes security best practices, such as **continuous testing and monitoring**, which can help to reduce the risk of security breaches and improve the overall security of an organization's systems.

Better Resource Utilization: DevOps enables organizations to optimize their use of resources, including hardware, software, and personnel, which can result in cost savings and improved efficiency

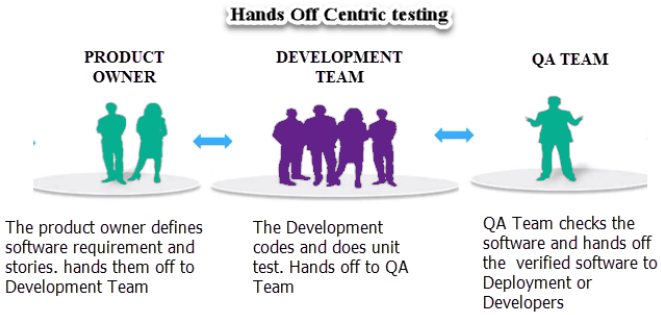
Continuous Testing in DevOps

- Fixing a software bug after a product is launched can be up to 100 times more expensive than addressing it during the development phase.
- Continuous Testing is a vital approach in modern software development, where testing is integrated at every stage of the process.
- This helps identify errors early, reducing both the risk of defects and the costs that come with fixing them later.

- **Continuous Testing** refers to running automated tests every time code changes are made, providing fast feedback as part of the software delivery pipeline.
- It was introduced to help developers quickly identify, notify, and fix issues. The goal is to test more frequently, starting with individual components and later testing the entire codebase.
- Continuous Testing is a key part of the Continuous Integration (CI) process within Agile and DevOps pipelines, enabling faster and more efficient software delivery.

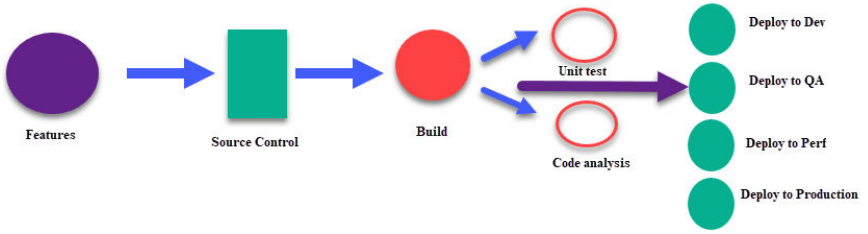
How is Continuous Testing different?

In Traditional (Hands-off) Testing:



- Continuous means undisrupted testing done on a continuous basis.
- In a Continuous DevOps process, a software change (release candidate) is continuously moving from Development to Testing to Deployment.
- The code is continuously developed, delivered, tested and deployed.

Continuous Testing



For Example,

- Whenever a developer uploads the code in the Source Code Server like Jenkins automated set of unit tests are executed in the continuous process.
- If the tests fail, the build is rejected, and the developer is notified. If the build passes the test, it is deployed to performance, QA servers for exhaustive functional and load tests.
- The tests are run in parallel. If the tests pass, the software is deployed in production.

Thank You

The Agile Test Plan includes the following:

- **Test Scope:** What will be tested.
- **Testing Instruments:** Tools used for testing.
- **Data and Settings:** Test data and environment setup.
- **Approaches and Strategies:** How the testing will be done.
- **Skills Required:** Skills needed for testing.
- **New Functionalities:** New features that need testing.
- **Levels or Types of Testing:** Based on how complex the features are (e.g., unit test, system test).
- **Resourcing:** People and time needed for testing.
- **Deliverables and Milestones:** What outputs are expected and when.
- **Infrastructure Consideration:** Systems, servers, and setup needed.
- **Load or Performance Testing:** How the system handles heavy usage.
- **Mitigation or Risks Plan:** Backup plans for any risks or issues found.

7 Cs of DevOps (Simplified)

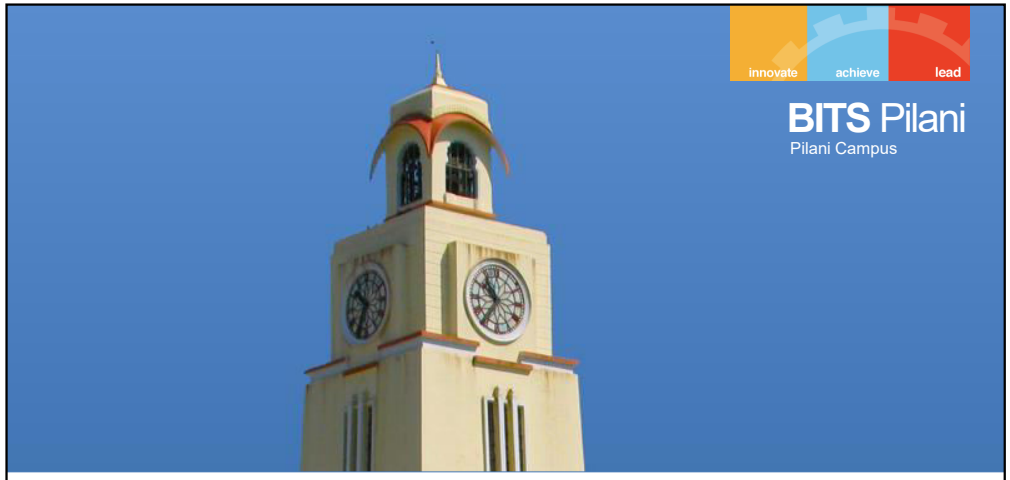
1. **Continuous Development**
   → Writing and improving code constantly.
2. **Continuous Integration**
   → Regularly merging code updates into a shared system.
3. **Continuous Testing**
   → Testing the code frequently to catch errors early.
4. **Continuous Deployment / Delivery**
   → Automatically delivering or releasing new changes to users.
5. **Continuous Monitoring**
   → Always checking how the software is performing after release.
6. **Continuous Feedback**
   → Collecting feedback from users and systems to improve quickly.
7. **Continuous Operations**
   → Keeping the software running smoothly without interruptions.

Continuous Testing Flow

1. **Features**
   → New features are developed.
2. **Source Control**
   → Code is saved in a shared storage (like GitHub, GitLab).
3. **Build**
   → Code is compiled into a working application.
4. **Unit Test & Code Analysis**
   →
   - **Unit Test:** Small tests check if each part of the code works correctly.
   - **Code Analysis:** Automatic checks for coding mistakes and quality.

Deployment Stages
After the code passes testing, it moves automatically through these stages:

1. **Dev Environment**
   → Software is given to **developers** to check if basic functions work.
2. **QA Environment**
   → Software is given to the **testing team (QA)** to test for bugs and issues.
3. **Performance (Perf) Environment**
   → Software is tested for **speed**, **stability**, and **how much load it can handle**.
4. **Production Environment**
   → Final stage. Software is **released to real users**.



SE ZG501
Software Quality Assurance and Testing
Module 9 ,10

How Is Continuous Testing Different from Test Automation?

innovate achieve lead

Parameter	Test Automation	Continuous Testing
Definition	Test automation is a process where tool or software is used for automating tasks.	It is a software testing methodology which focuses on achieving continuous quality & improvement.
Purpose	A set of similar or repetitive tasks, a machine can execute, faster, with a fewer mistake.	The continuous testing process helps to find the risk, address them and improve the quality of the product.
Prerequisite	Automation in testing possible without integrating continuous testing.	Continuous testing can not be implemented without test automation.

Time	Software release can take a month to years.	Software release may be released weekly to hourly.
Feedback	Regular feedback after testing each release.	Feedback at each stage needs to be instant.
History	Automated testing has been done for decades to make the testing process faster.	Continuous testing is a relatively newer concept.

Continuous Testing Tools

1) QuerySurge

QuerySurge is the smart data testing solution that is the first-of-its-kind full DevOps solution for continuous data testing. Key features include Robust API with 60+ calls, detailed data intelligence & data analytics, seamless integration into the DevOps pipeline for continuous testing, and verifies large amounts of data quickly.

2) Jenkins

Jenkins is a Continuous Integration tool which is written using Java language. This tool can be configured via GUI interface or console commands.

3) Travis

Travis is continuous testing tool hosted on the GitHub. It offers hosted and on-premises variants. It provides a variety of different languages and a good documentation.

4) Selenium

Selenium is open-source software testing tool. It supports all the leading browsers like Firefox, Chrome, IE, and Safari. Selenium WebDriver is used to automate web application testing.

Best Practices for SQA Implementation

Core Principles for Effective SQA

Define Clear Quality Objectives:

- Set measurable goals like defect density thresholds, code coverage percentages, and customer satisfaction scores.
- Align objectives with project requirements and expectations.

Adopt a Comprehensive SQA Plan:

- Document scope, schedule, risk assessment, and metrics.
- Continuously update the plan to reflect lifecycle changes

Early QA Involvement and Standardization

Involve QA Early in Development:

- Apply the shift-left approach to integrate QA in requirements and design phases.
- Conduct requirements validation and design reviews.

Use Standardized Processes and Guidelines:

- Implement coding standards, design principles, and testing protocols.
- Follow standards like ISO 9001, CMMI, or IEEE

Automation and Continuous Testing

Automate Testing Where Possible:

- Leverage tools like Selenium, JUnit, or TestNG for unit, regression, and performance testing.
- Integrate testing automation into CI/CD pipelines.

Embrace Continuous Testing:

- Include testing in CI/CD workflows using tools like Jenkins or GitLab CI.
- Execute automated tests after every code commit to detect issues early.

Quality Assurance in Different Development Methodologies

Waterfall Model

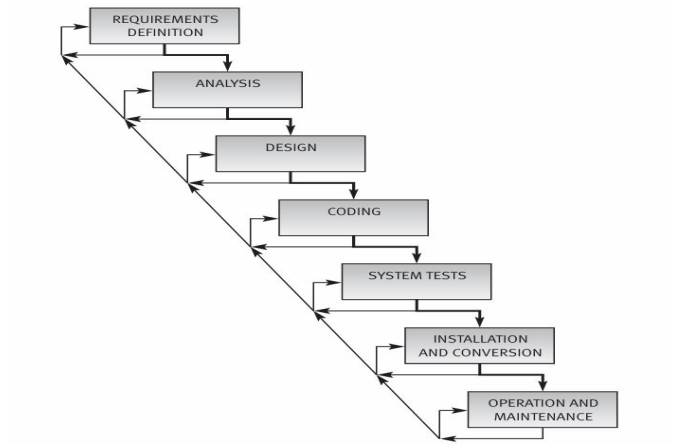


Figure 7.1: The waterfall model
Source: After Boehm (1981) and Royce (1970) (© 1970 IEEE)

1. **Design**: Define outputs, inputs, processing procedures, data structures, databases, and software architecture in detail.
2. **Coding**: Translate the design into code with quality assurance activities like inspections, unit tests, and integration tests.
3. **System Tests**: Conduct comprehensive tests to identify and correct errors. Developers perform these tests, followed by customer acceptance tests or joint system tests to ensure commitments are met.
4. **Installation and Conversion**: Install the approved system as part of a larger setup or replace an existing system, ensuring uninterrupted operations during conversion.
5. **Regular Operation and Maintenance**: Begin system operation, providing:
   - **Corrective Maintenance**: Fix user-reported faults.
   - **Adaptive Maintenance**: Adjust to new requirements.
   - **Perfective Maintenance**: Enhance system performance with minor features.

Prototyping Model

Iterative development using prototypes for user feedback.

Evolved prototypes lead to a refined system or complete software.

Advantages: High user involvement and reduced failure risks.

Limitations: Reduced developer flexibility.

Spiral Model

Combines iterative development, risk analysis, and user feedback.

Activities in each iteration:

1.Planning and defining objectives.
2.Risk analysis and resolution.
3.Engineering activities (design, coding, testing).
4.Customer evaluation and feedback.

Object-Oriented Model

The **object-oriented model** focuses on **reusing software components** (objects or components) through a **software component library** to streamline development (Figure 7.5).

Development Process:

1. Conduct object-oriented **analysis and design**.
2. Acquire **reusable components** from the library, if available; otherwise, proceed with regular development.
3. Stock newly developed components in the library for future reuse.

Benefits:

- **Economy**: Reusing components is more cost-effective than developing new ones.
- **Improved Quality**: Reused components are more reliable due to prior defect detection by other users.

Importance of a Quality Culture

📌 **Why Quality Culture Matters:**

- Ensures **high-quality software** and **customer satisfaction**.
- Encourages **proactive quality management** rather than reactive fixes.
- Fosters a **collaborative environment** where quality is a shared responsibility.

📉 **Consequences of Poor Quality Culture:**

- Increased **software defects, security risks, and customer dissatisfaction**.
- Higher **failure costs** due to post-release issues and rework.
- Lack of accountability leads to **inefficiency and inconsistent product quality**.

✅ **Solution:** Establish a strong **quality-first mindset** across the organization.

Building a quality culture in organizations

🔹 **1. Leadership Commitment**

- Leaders must **prioritize quality** and **set measurable goals**.
- Provide resources and create an **accountability-driven environment**.

🔹 **2. Shared Responsibility & Continuous Improvement**

- Quality is **not just QA's job**—developers, testers, and stakeholders must contribute.
- Regular **evaluations, retrospectives, and root cause analysis** ensure improvement.

🔹 **3. Employee Empowerment & Communication**

- Train and equip teams with **tools and autonomy** for better decision-making.
- Foster **open communication channels** (e.g., Slack, MS Teams) for collaboration.

🔹 **4. Customer-Centric Approach & Standardization**

- Align quality efforts with **customer expectations and feedback**.
- Follow **best practices and industry standards** (ISO, CMMI) to ensure consistency.

Case Studies of Successful SQA Implementations

Introduction

- SQA ensures software meets defined quality standards and user expectations.
- Tailored strategies improve efficiency, reduce risks, and address project-specific challenges.
- Explore six case studies demonstrating successful SQA implementations across various industries.

NASA – Rigorous SQA for Space Missions

Context: Precision required for space missions to avoid catastrophic outcomes.

SQA Strategies:

- Comprehensive Testing: From requirements to deployment. Automated
- Tools: Simulators to mimic space conditions . Peer Reviews: Regular code inspections.
- Continuous Monitoring: Post-deployment performance tracking.

Outcome: High-quality, reliable software for missions like Mars Rover; benchmark for high-stakes industries.

Google: Automated Testing for Scalable Systems

Context: Reliability and scalability for millions of global users.

SQA Strategies:

- Test-Driven Development (TDD): Tests written before coding.
- Continuous Testing: Automated unit, integration, and regression tests.
- Simulations: Stress tests for high user loads.
- Defect Tracking: Centralized tools for quick resolution.

Outcome: Faster delivery cycles and scalable, reliable applications like Search and YouTube.

Lessons Learned from Successful Software Quality Assurance (SQA) Projects

Implementing SQA effectively leads to improved software reliability, reduced risks, and higher customer satisfaction.

Here are key lessons derived from successful SQA projects across industries:

Emerging Technologies and Their Impact on SQA

An Overview of Transformations in Software Quality Assurance

Introduction

Emerging technologies have significantly transformed Software Quality Assurance (SQA), introducing new methodologies, tools, and challenges.

Artificial Intelligence (AI) and Machine Learning (ML)

Impact:

- Test Automation: AI generates test cases, identifies defects, and automates repetitive tasks.
- Predictive Analytics: ML predicts defect-prone areas to prioritize testing.
- Dynamic Testing: AI adapts test cases in real-time.

Challenges:

- High initial setup costs and expertise requirements.
- Difficulty in testing AI systems due to non-deterministic behavior.

Cloud Computing

impact:

- Scalable Testing Environments: Reduces infrastructure costs.
- Cross-Platform Testing: Covers diverse platforms and configurations.
- Disaster Recovery: Cloud-based backups ensure robust data retention.

Challenges:

- Security and compliance concerns in cloud environments.

Internet of Things (IoT)

Impact:

- Complex Scenarios: QA validates diverse device interactions and real-time data.
- Performance Testing: Ensures reliability under high device loads.
- Security Testing: Identifies vulnerabilities in devices and data transmission.

Challenges:

- Managing vast combinations of devices, protocols, and platforms.

Blockchain Technology

Impact:

- Smart Contract Testing: Ensures correctness and security of blockchain systems.
- Decentralized Testing: Requires unique test strategies.

Challenges:

- Debugging is complex due to immutability.
- Limited expertise in blockchain testing.

Big Data and Analytics

Impact:

- Data Validation: Ensures accuracy and integrity of large datasets.
- Performance Testing: Evaluates handling of high-volume, high-velocity data.
- Testing Insights: Analytics identify patterns and root causes of defects.

Challenges:

- Managing and processing large-scale data for testing.

Automation and Robotics

Impact:

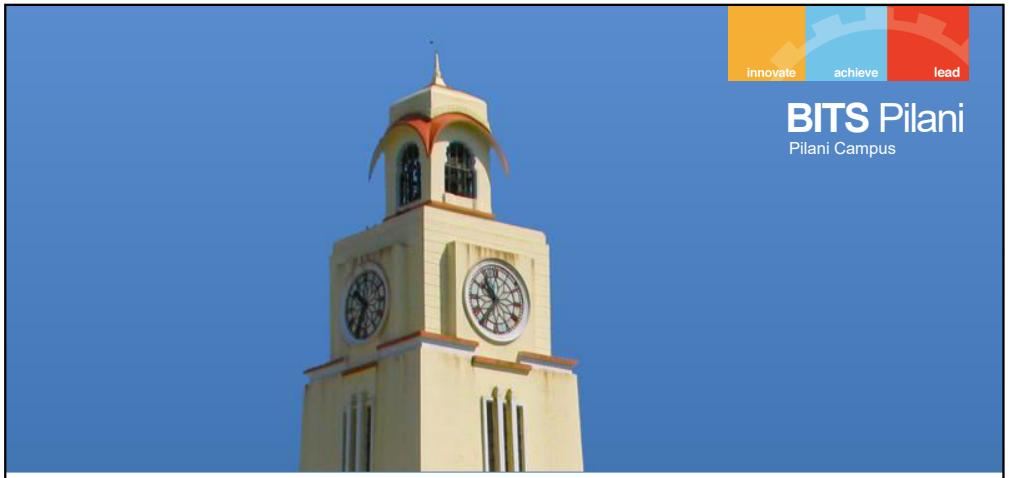
- Robotic Process Automation (RPA): Automates repetitive tasks.
- Hardware-in-the-Loop Testing: Integrates hardware and software testing.

Challenges:

- Testing robotic systems demands specialized environments and expertise.

Conclusion

- Emerging technologies have revolutionized SQA by enabling automation and addressing challenges.
- Adoption requires overcoming skill gaps, high investments, and adapting to rapid advancements.
- SQA professionals must stay updated to ensure software quality in an evolving landscape.



Artificial Intelligence and Machine Learning in Quality Assurance

Introduction

- AI and ML are revolutionizing Quality Assurance (QA).
- Enhances testing efficiency, early defect detection, and predictive analytics.
- Shifts QA from manual testing to intelligent, automated approaches.

Key Concepts

Artificial Intelligence (AI)

- Simulates human intelligence to make decisions and solve problems.
- Used in QA for test automation, defect prediction, and anomaly detection.

Machine Learning (ML)

- Subset of AI where algorithms learn from data to make predictions or decisions.
- Helps in identifying patterns, predicting failures, and improving test coverag

Automated Testing

How It Works:

- AI automates repetitive tasks like test case generation, execution, and reporting.
- ML optimizes test suites by identifying redundant tests.

Benefits:

- Faster test execution.
- Improved accuracy and reduced human errors.

Tools: Selenium, Testim, Applitools.

Defect Prediction and Prevention

How It Works:

- ML analyzes historical data to predict defect-prone areas.
- AI identifies potential failure points before they occur.

Benefits:

- Reduces debugging time.
- Enhances software reliability.

Anomaly Detection

How It Works:

- AI analyzes logs, system behavior, and performance metrics.
- ML flags unusual patterns indicating defects.

Benefits:

- Early detection of performance bottlenecks.
- Ensures system stability.

Test Case Prioritization

How It Works:

- ML ranks test cases based on defect probability, impact, or execution history.
- Focuses on critical areas for testing.

Benefits:

- Saves resources by testing high-risk areas first.
- Improves test efficiency.

Predictive Analytics

How It Works:

- ML analyzes past trends to predict defect patterns and testing timelines.

Benefits:

- Helps in resource allocation.
- Reduces risks in project planning.

Visual Testing

How It Works:

- AI compares screenshots of applications to detect UI inconsistencies.

Benefits:

- Ensures visual consistency across platforms.
- Speeds up regression testing

Natural Language Processing (NLP)

How It Works:

- NLP analyzes requirements and generates test cases automatically.
- Chatbots assist in real-time test result analysis.

Benefits:

- Simplifies test case creation and management.
- Enhances team communication

Self-Healing Test Scripts

Title: Self-Healing Test Scripts

How It Works:

- AI detects changes in application code or UI and updates test scripts automatically.

Benefits:

- Reduces test maintenance efforts.
- Ensures continuous testing in dynamic environments

Challenges in AI/ML Adoption for QA

- **Data Dependency:** Requires large datasets for training ML models.
- **Complexity:** AI/ML implementation can be technically challenging.
- **Cost:** High initial investment in tools and infrastructure.
- **Lack of Expertise:** Requires specialized skills and training.

Best Practices for Using AI/ML in QA

•**Start Small**: Implement AI/ML in specific areas before scaling.
•**Leverage Open-Source Tools**: Use TensorFlow, PyTorch, Selenium AI plugins.
•**Collaborate Across Teams**: Encourage communication between QA, development, and data science teams.
•**Monitor and Improve Models**: Regularly update ML models with new data.
•**Focus on High-Risk Areas**: Automate repetitive and critical testing tasks.

Benefits of AI/ML in QA

- **Efficiency**: Faster testing cycles with higher accuracy.
- **Scalability**: Can handle complex, large-scale projects.
- **Cost Reduction**: Minimizes manual effort and resource usage.
- **Improved Quality**: Early defect detection improves reliability.
- **Continuous Learning**: ML models improve over time.

Conclusion

- AI and ML are transforming QA with automation, efficiency, and predictive insights.
- While challenges exist, strategic adoption improves software quality and delivery timelines.
- Organizations leveraging AI/ML effectively will stay competitive in modern software development.

innovate achieve lead

THANK YOU

Spiral Model

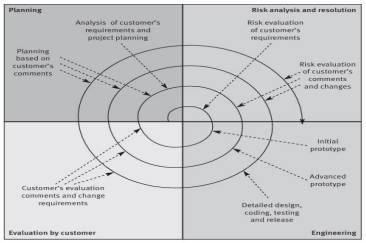


Figure 7.3: The spiral model (Boehm, 1988)
Source: After Boehm (1988) © 1988 IEEE

The image illustrates the **Spiral Model** of software development proposed by Barry Boehm in 1988. It is a risk-driven process model that combines elements of both design and prototyping in stages. Each loop in the spiral represents a phase in the software development lifecycle, emphasizing risk assessment and customer feedback.

Key Quadrants and Activities:

1. **Top-Left: Planning**
   - **"Analysis of customer's requirements and project planning"**
     This step involves understanding what the customer wants and planning the project accordingly.
   - **"Planning based on customer's comments"**
     Customer feedback is integrated into future planning.
2. **Top-Right: Risk Analysis and Resolution**
   - **"Risk evaluation of customer's requirements"**
     Risks related to the initial requirements are assessed.
   - **"Risk evaluation of customer's comments and changes"**
     As feedback and new changes come in, associated risks are re-evaluated.

3. **Bottom-Right: Engineering**
   - **"Initial prototype"**
     A basic version of the software is developed to demonstrate the concept.
   - **"Advanced prototype"**
     A more complete version is developed based on feedback.
   - **"Detailed design, coding, testing and release"**
     The final product is built, tested, and deployed.
4. **Bottom-Left: Evaluation by Customer**
   - **"Customer's evaluation, comments, and change requirements"**
     After each phase, the customer reviews the output and provides feedback.
   - This input feeds into the next loop of the spiral, improving the product iteratively.

How the Spiral Model Works:

- Each cycle (loop) in the spiral goes through **4 main stages**: Planning → Risk Analysis → Engineering → Evaluation.
- After each cycle, the product evolves with more features and improvements.
- This model supports **incremental development**, **prototyping**, and **continuous customer feedback**.

Advantages:

- Strong focus on **risk analysis**.
- Ideal for **large, complex, and high-risk projects**.
- Promotes **early detection of problems** and **continuous customer involvement**.

WIN WIN SPIRAL MODEL

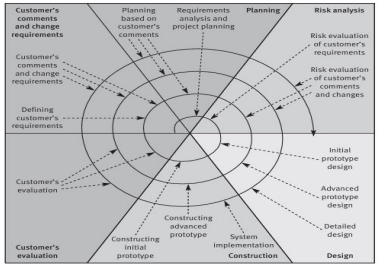


Figure 7.4: The advanced spiral model (Boehm, 1998)

Advanced Spiral Model (Boehm, 1998) – Full Explanation

Overview

The **Advanced Spiral Model** is an iterative and risk-driven software development model that expands on Boehm's original 1988 spiral model. It emphasizes **continuous customer involvement**, **prototyping**, **risk analysis**, and **incremental delivery** through iterative cycles.

Each loop of the spiral represents one phase of development, and the loops expand outward as the project progresses.

Structure of the Spiral

The model is divided into **four main quadrants**, each representing a key activity phase within one development cycle:

1. Customer's Comments and Change Requirements (Top-Left Quadrant)

This quadrant deals with **gathering and analyzing customer feedback**, which drives the next iteration.

Key Activities:

- **Defining Customer's Requirements**
  Initial gathering of what the customer wants from the system.
- **Customer's Comments and Change Requirements**
  Customer feedback after seeing a prototype or release. May include new features, modifications, or clarifications.
- **Planning Based on Customer's Comments**
  The project plan is updated in response to this feedback. Priorities may shift.

◆ 2. Planning (Top-Centre Quadrant)

This quadrant focuses on **organizing the work** and **setting goals** for the next cycle.

Key Activities:

- **Requirements Analysis**
  Understand what the system must do, refine user needs.
- **Project Planning**
  Define scope, tasks, responsibilities, schedule, and cost.
- **Planning Based on Feedback**
  Adapt timelines and features based on customer suggestions from the previous cycle.

◆ 3. Risk Analysis (Top-Right Quadrant)

This quadrant evaluates potential **risks** that might hinder development success.

Key Activities:

- **Risk Evaluation of Requirements**
  Analyze technical, cost, and schedule risks of implementing features.
- **Risk Evaluation of Comments and Changes**
  If customers suggest changes, assess risks before integrating them.

◆ 4. Design and Construction (Bottom-Right and Bottom-Left Quadrants)

These two quadrants are about **building and delivering** the product in iterations.

◆ Design (Bottom-Right Quadrant)

This quadrant focuses on defining the system architecture and interface.

Key Activities:

- **Initial Prototype Design** – Design the first, rough version of the system.
- **Advanced Prototype Design** – Add more features and detail to the next version.
- **Detailed Design** – Technical design with implementation details, architecture, and interface specifications.

◆ Construction (Bottom-Left Quadrant)

This quadrant focuses on **development**, **testing**, and **delivery**.

Key Activities:

- **Constructing Initial Prototype** – Basic implementation of the design.
- **Constructing Advanced Prototype** – Improved and expanded version.
- **System Implementation** – Final product is developed, tested, and deployed.
- **Customer Evaluation** – Customer uses and reviews the product.

🔁 The process then loops again with new feedback.

🔄 How the Spiral Works

Each cycle of the spiral includes:

1. Understanding customer needs
2. Planning the next steps
3. Evaluating and mitigating risks
4. Designing and building prototypes
5. Getting customer feedback

This loop repeats with increasing refinement until the system is complete.

Test suite

A **test suite** is a **collection of test cases** that are grouped together to test a software application.

Simple Example:

If you're testing a **login feature**, the test suite might include:

- Test with correct username and password ✅
- Test with wrong password ❌
- Test with empty fields ❌
- Test with special characters ❌

All these tests together form the **"Login Test Suite."**

So, a **test suite** = group of related tests run together to check if a feature works properly.

JUNIT

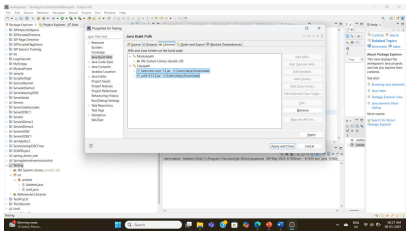
Step 1: Download the JAR Files

- **junit-4.13.2.jar**
  📌 ***Purpose:*** **Provides core testing features (@Test, assertEquals(), etc.)**
- **hamcrest-core-1.3.jar**

  📌 ***Purpose:*** **Supports assertThat() by supplying readable matchers like is(), equalTo()**
- **Unit** uses **Hamcrest** internally to support advanced assertions.
- **JUnit 4.13.2 requires hamcrest-core-1.3.jar** as a companion JAR for full functionality.

Step 2: Add JARs to Your Eclipse Project

1. **Open Eclipse → Right-click your project → Build Path → Configure Build Path**
2. **Click the Libraries tab → Click Add External JARs**
3. **Select both downloaded JARs**
4. **Click Apply and Close**



```
public class Junit {
        public int Add(int one, int two)
```

```
        {
          return one + two;
        }

}
```

```
import static org.junit.Assert.*;
import org.junit.Test;

public class Addtest {
        @Test
        public void testadd()
        {
                Junit test1 = new Junit();
                int result = test1.Add(5, 5);
                assertEquals(10,result);
        }

}
```

import static org.junit.Assert.*;

- **Purpose**: Gives access to JUnit **assertion methods** like:
    - assertEquals()
    - assertTrue()
    - assertFalse()
- Used to **verify test results**.

import org.junit.Test;

- **Purpose**: Allows you to use the @Test annotation.
- @Test tells JUnit to **run the method as a test case**.

Run the method as a test case" means:

JUnit will **automatically execute** the method and **check if it passes or fails** based on the assertions inside.

```
public class Addtest {
        @Test
        public void testadd()
        {
                Junit test1 = new Junit();
                int result = test1.Add(5, 5);
                assertEquals(10,result);
        }

}
```

■ **Explanation:**

- **@Test**: Marks testadd() as a JUnit test case — JUnit will run this method during testing.
- **Junit test1 = new Junit();**: Creates an object of the class being tested.
- **test1.Add(5, 5)**: Calls the Add() method with values 5 and 5.
- **assertEquals(10, result);**: Checks if the method returns 10.
  ✔ Passes if true, ✕ fails if not.

📌

■ **assertEquals(expected, actual)**

Checks if two values are equal.

✔ Test passes if both values match.
✕ Test fails if they are different.

Example:

assertEquals(10, result); // Passes if result is 10