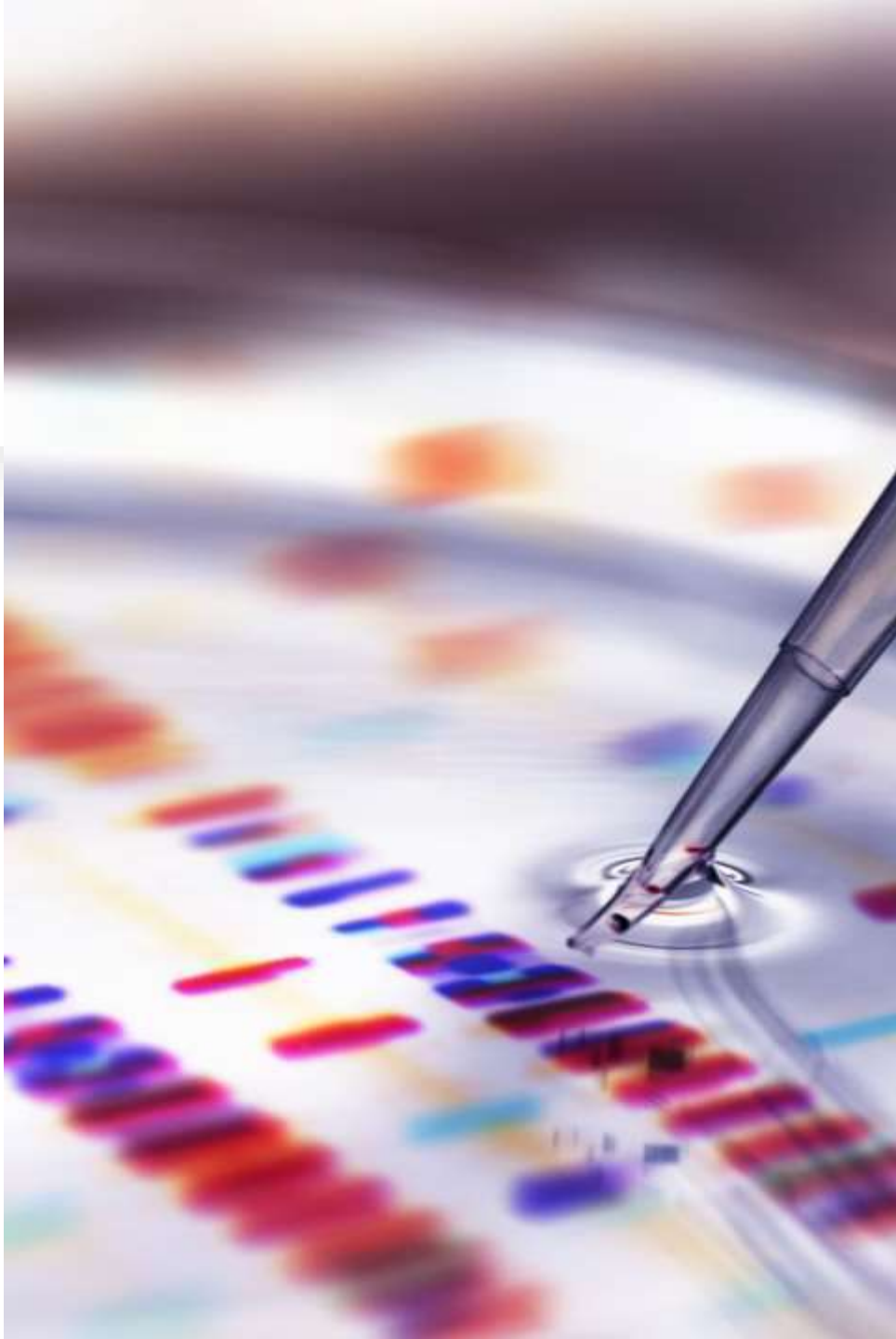
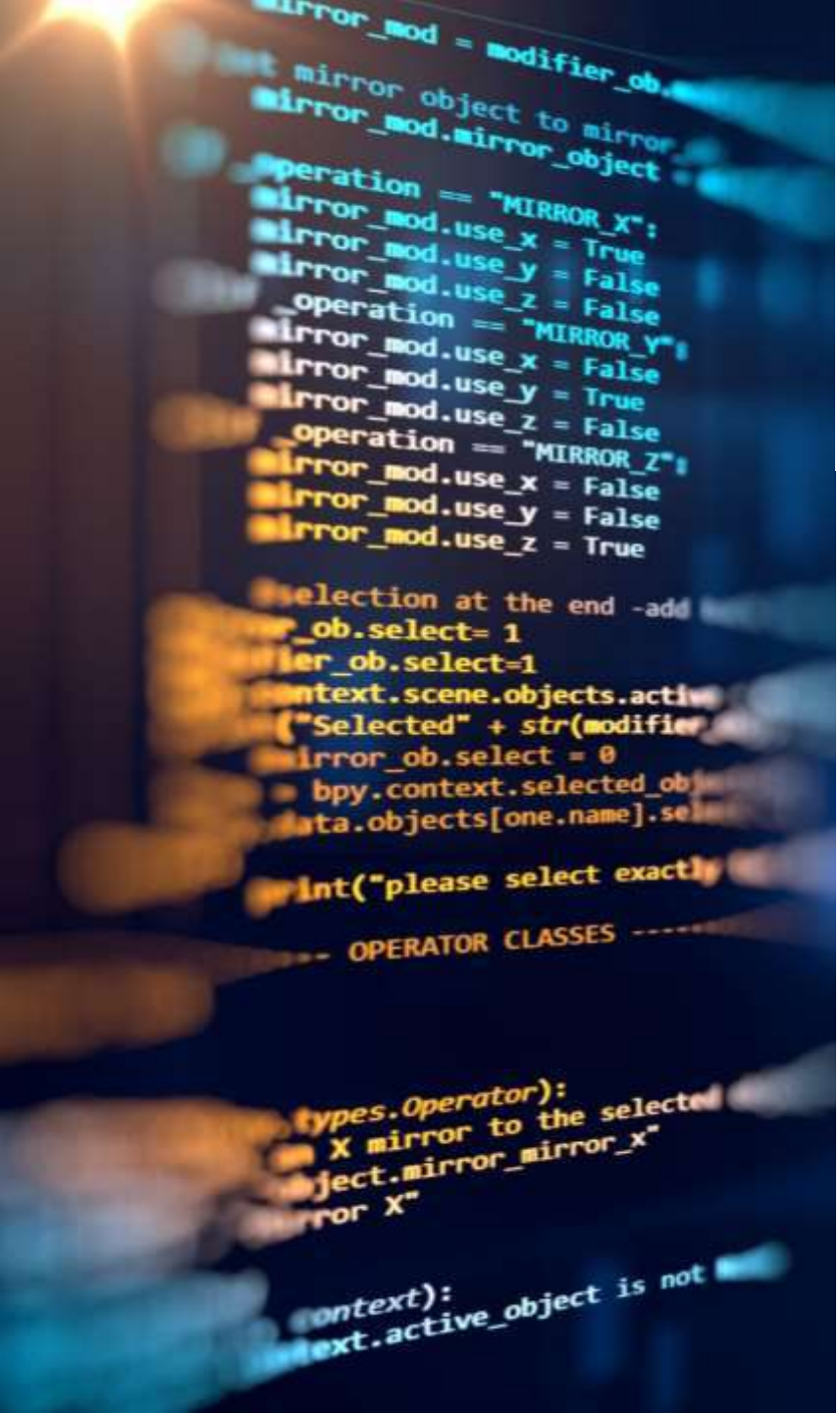
Software Testing

Testing Objectives and Principles of Software Testing

By,
Aditya Upadhyay





Introduction to Software Testing

- **Software Testing** is the **process of evaluating a software application** (or) **system** to **identify defects** and **ensure** that it **meets the specified requirements**. It involves **executing the software under controlled conditions** to **detect errors, gaps,** or **missing functionalities** and to **verify that the product works as intended**.

- **Goals**: Quality assurance, Defect Identification.

Some More Definitions of Testing

- Testing is the process of executing a program with the intent of finding errors – **Myers [2]**
- A successful test is one that uncovers an as-yet undiscovered error – **Myers [2]**
- Testing can show the presence of bugs but never their absence – **Dijkstra [125]**
- Program testing aim is to affirm the quality of software systems by systematically exercising the software in carefully controlled circumstances – **E. Miller [84]**
- Testing is a support function that helps developers look good by finding their mistakes before anyone else does – **James Bach**
- Software testing is an empirical investigation conducted to provide stakeholders with information about the quality of the product or service under test, with respect to the context in which it is intended to operate **– Cem Kaner [85]**
- The underlying motivation of program testing is to affirm software quality with methods that can be economically and effectively applied to both large-scale and small-scale systems – **Miller [126]**
- Testing is a concurrent lifecycle process of engineering, using and maintaining testware (i.e. testing artifacts) in order to measure and improve quality of software being tested – **Craig [117]**

Goals of Software Testing

Goals of software testing may be classified into three major categories:

1. **Immediate Goals**

Bug discovery: The immediate goal of testing is to find errors at any stage of software development. The more the bugs discovered at an early stage, better will be the success rate of software testing.

2. **Long-term Goals** : These goals affect the product quality in the long run, when one cycle of the SDLC is complete. Some of them are:

Quality: Since software is also a product, its quality is primary from the users' point of view. Thorough testing ensures superior quality. Therefore, the first goal of understanding and performing the testing process is to enhance the quality of the software product. Though quality depends on various factors, such as correctness, integrity, efficiency, etc., reliability is the major factor to achieve quality. The software should be passed through a rigorous reliability analysis to attain high quality standards.

Reliability : Measure of confidence that the software will not fail, and this level of confidence increases with rigorous testing.

- **Customer satisfaction**: If we want the customer to be satisfied with the software product, then testing should be complete and thorough. Testing should be complete in the sense that it must satisfy the user for all the specified requirements mentioned in the user manual, as well as for the unspecified requirements, which are otherwise understood. A complete testing process achieves reliability, which enhances the quality, and quality ensures customer satisfaction.

- **Risk management:** Risk is the probability that undesirable events will occur in a system. These undesirable events will prevent the organization from successfully implementing its business initiatives. Risks must be controlled to manage them with ease. Software testing may act as a control, which can help in eliminating or minimizing risks. Thus, managers depend on software testing to assist them in controlling their risks.

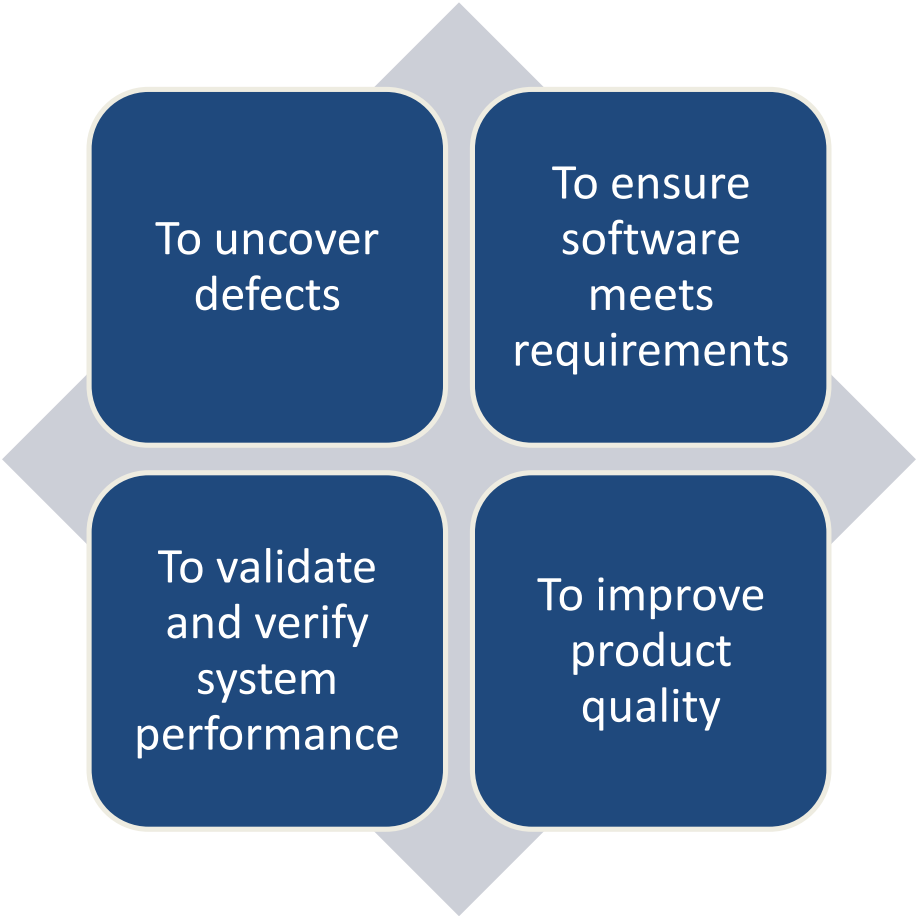
3. Post-implementation Goals : These goals are important after the product is released. Some of them are:

- **Bug prevention**: It is the consequence of testing process. These issues are not static, but, present in all software versions. The testing process for one project may not be successful and there may be scope for improvement. Past test executions, history of issues, test behavior and interpretation of the development team gives feedback for future projects. Thus, bug prevention is a strong goal of testing.

- **Reduced maintenance cost**: The maintenance cost of any software product is due to failure in use. The bugs are difficult to detect. Thus, testing should be done rigorously and effectively, and once the bugs are minimized and, in turn, the maintenance cost is reduced.

- **Improved testing process**: The only importance of any software product is to make it successful. The testing process, if successful, improves how testing can be benefited in future projects. Thus, application of testing knowledge and improvement of the testing process is a long-term goal for software testers.

Testing Objectives



Typical Objectives of Testing

- To evaluate work products such as **requirements**, **user stories**, **design** and **code** by using static testing techniques, such as **reviews**.
- To **verify** whether all specified requirements have been fulfilled, **for example**, in the resulting system.
- To **validate** whether the test objective is complete and works as the users and other stakeholders expect – **for example**, together with user (or) stakeholder groups.
- To build **confidence** in the level of quality of the test objective, such as when those tests considered highest risk pass, and when the failures that are observed in the other tests are considered acceptable.
- To prevent **defects**, such as when early test activities identify defects in requirements specifications that are removed before they cause defects in the design specifications and subsequently the code itself.

Aspect	Verification	Validation
Purpose	Ensures the product is **built correctly** (as per design/specifications).	Ensures the **right product is built** (meets user needs).
Focus	Process-oriented (checking design, documents, code).	Product-oriented (testing the actual software).
Activities	Reviews, inspections, walkthroughs.	Functional testing, system testing, UAT.
Timing	Done **before testing** begins.	Done **after verification** and during testing.
Answered Question	“Are we building the product right?”	“Are we building the right product?”

Benefits of Testing

Early bug detection

Cost reduction

Improved user satisfaction

Increased development efficiency

Types of Testing

Manual Testing

Automated Testing

Functional Testing

Non-Functional Testing

Manual vs Automated Testing

MANUAL: HUMAN INVOLVEMENT, EXPLORATORY TESTING

AUTOMATED: TOOL-BASED, REPETITIVE TESTING TASKS

Aspect	Manual Testing	Automation Testing
Pros	- Best for exploratory, usability, and ad-hoc testing. - No need for coding skills. - Immediate feedback without setup.	- Fast execution of repetitive test cases. - High reusability of test scripts. - Supports continuous testing and CI/CD. - More accurate and less human error.
Cons	- Time-consuming and less efficient for regression tests. - Prone to human error. - Cannot be executed 24/7 automatically.	- High initial cost (tools, scripting effort). - Requires skilled resources. - Not ideal for exploratory or UI/UX testing.

Principles of Software Testing

- Testing shows presence of defects but never their absence
- Exhaustive testing is impossible
- Early testing
- Defect clustering
- Pesticide paradox
- Testing is context dependent
- Absence of errors fallacy

Principle 1 - Testing Shows Presence of Defects

Explanation:

- Software testing can **only show that defects are present** in the application.
- **It can never prove that the software is 100% defect-free**, even if all test cases pass.
- Testing **reduces the probability of undiscovered defects** but cannot guarantee total absence of bugs.

Example: Suppose you test a **login page** with 50 test cases and all pass.

Later, a user tries to log in with a **special character in the username**, and the system crashes. This shows that **testing indicates the presence of defects**, but passing tests didn't guarantee a bug-free system.

Misconception: Testing proves correctness.

Principle 2 - Exhaustive Testing is Impossible

Exhaustive testing means testing **all possible inputs, paths, and combinations** in a software application.

In **real-world projects**, exhaustive testing is **practically impossible** because:

1. **Infinite input combinations** are possible.
2. **Multiple user actions, environments, and configurations** exist.
3. **Time and cost** constraints make it infeasible.

Instead, we use **risk-based**, **boundary value**, **and equivalence partitioning techniques** to **select representative test cases**.

Principle 3 - Early Testing

Early Testing means **starting testing activities as early as possible** in the Software Development Life Cycle (SDLC).

Detecting defects **early** (during requirement analysis or design phase) is **cheaper and easier to fix** than catching them during implementation or after release.

The cost of fixing a defect **increases exponentially** as the project progresses.

In modern approaches (like **V-Model** or **Agile**), testing begins **parallel to development** rather than waiting for coding to complete.

Principle 4 & 5

Defect Clustering means that **a small number of modules** (or) **components contain most of the defects** in the software.

This follows the **Pareto Principle (80/20 rule)** – around **80% of the defects are usually found in 20% of the modules**.

These defect-prone modules are often **complex, newly developed, or frequently modified**.

Testers can **focus more testing effort** on these modules to optimize resources and improve quality.

Pesticide Paradox: Repeating same tests won't find new bugs.

Principle 6 & 7

- **Testing is Context Dependent**: Different methods for different projects.
- **Absence of Errors Fallacy**: Software might be bug-free but still useless if it doesn't solve intended problem or align with requirements.

Summary

- Recap of key points: Testing purpose, objectives, principles
- Importance of systematic and strategic testing

Software Testing –Module 1

Presented by: Rahul Kumar Balyan, Assistant Professor

Department of Computer Science Engineering and Technology

Objective Of Software Testing

1. **Verification and Validation**: It is a verification activity carried out to assure that the product is developed in a way that it is fit for the intended use and expectations of the stakeholders
2. **Identification of Defects/ Bug Detection**: The other basic aim of software testing in the very first stage is to detect as many defects, bugs, and errors in the software as possible.
3. **Defects Prevention**: Systematic software testing and result analysis let the development team identify the causes of defects and possible corrective actions, so as not to repeat its occurrence in the future, ensuring a better quality standard in practice of software development.
4. **Ensuring Quality Attributes in the Product/ Quality Assurance** : Some of the quality attributes tested with software testing include functionality, performance, usability, security, compatibility, and scalability, among others.

Objectives of Software Testing Cont..

- **Risk Management**: Includes assessing security vulnerabilities, performance bottlenecks, and reliability problems in software in such a way that these areas of the software will not expose the product to significant risks on software release.

- **Customer Satisfaction-**

Principle Of software Testing

- The principles are for effective and efficient testing practices.
- These principles guide testers in identifying defects, managing test efforts, and ensuring software quality.
- **Testing Shows the Presence of Defects, Not Their Absence:** Testing can reveal that defects exist, but it cannot definitively prove that no defects are present. **Example:** Running a test case that successfully logs a user into an application demonstrates that the login functionality works, but it does not guarantee that there are no hidden vulnerabilities or edge cases that could cause a failure.
- **Exhaustive Testing is Impossible:** Testing all possible combinations of inputs and conditions is not feasible for most real-world applications. **Example:** For a simple calculator application, testing every possible numerical input and operation combination would be an infinite task. Instead, testers focus on boundary values, valid/invalid inputs, and common scenarios.

Principle Of software Testing

- Pesticide Paradox:
- Early Testing Saves Time and Money: Identifying defects early in the software development lifecycle (SDLC) reduces the cost and effort required for fixes. Example: Finding a design flaw during the requirements phase through a review process is significantly cheaper to fix than discovering it during user acceptance testing, which might require extensive re-coding and re-testing.
- Defects Cluster Together: A small number of modules or components often contain the majority of defects. This is often related to the Pareto Principle (80/20 rule).Example: In an e-commerce application, the payment gateway or inventory management module might consistently exhibit more defects than static content pages due to their complexity and interactions with external systems.

Principle Of software Testing

- Repeating the same tests repeatedly becomes less effective at finding new defects over time. Tests need to be updated and diversified. **Example:** Continuously running the exact same set of regression tests without adding new test cases or exploring different scenarios may miss new defects introduced by recent code changes.

- **Testing is Context-Dependent:** The testing approach and techniques should vary depending on the type of software, its purpose, and its risk profile. **Example:** Testing a safety-critical medical device requires a much more rigorous and formal testing process than testing a simple marketing website.

- **Absence of Errors Fallacy:** Even if a system is found to be free of known defects, it may still not be usable or meet user needs if it doesn't fulfill the intended purpose or user expectations.
  Example: An application might be technically bug-free, but if its user interface is unintuitive or its performance is too slow for practical use, it is still a failure from a user perspective.

Testing vs Debugging

- Testing is the process of evaluating a software system or its components to identify defects, errors, or bugs and verify whether it meets specified requirements and functions as intended. It is a proactive process aimed at finding issues before the software is released to users. Various types of testing exist, including unit testing, integration testing, system testing, and acceptance testing, among others.

- Debugging is the process of identifying, analyzing, and resolving defects or errors found during testing or in production. Once a defect is identified, debugging involves investigating the code to pinpoint the root cause of the issue and then implementing a fix to eliminate the problem. Debugging is a reactive process that focuses on problem-solving and correction.

-

Test Importance of Metrics in Software Testing:

metrics are essential in determining the software's quality and performance. Developers may use the right software testing metrics to improve their productivity.

- **Early Problem Identification:** By measuring metrics such as defect density and defect arrival rate, testing teams can spot trends and patterns early in the development process.
- **Allocation of Resources:** Metrics identify regions where testing efforts are most needed, which helps with resource allocation optimization. By ensuring that testing resources are concentrated on important areas, this enhances the strategy for testing as a whole.
- **Monitoring Progress:** Metrics are useful instruments for monitoring the advancement of testing. They offer insight into the quantity of test cases that have been run, their completion rate, and if the testing effort is proceeding according to plan.
- **Continuous Improvement:** Metrics offer input on the testing procedure, which helps to foster a culture of continuous development.

Types of Software Testing Metrics:

1. **Process Metrics:** A project's characteristics and execution are defined by process metrics. These features are critical to the SDLC process's improvement and maintenance (Software Development Life Cycle).

2. **Product Metrics:** A product's size, design, performance, quality, and complexity are defined by product metrics. Developers can improve the quality of their software development by utilizing these features.

3. **Project Metrics:** Project Metrics are used to assess a project's overall quality. It is used to estimate a project's resources and deliverables, as well as to determine costs, productivity, and flaws.

Software Testing Metrics

1. **Test Case Effectiveness:**

- **Test Case Effectiveness = (Number of defects detected / Number of test cases run) x 100**

2. **Passed Test Cases Percentage:** Test Cases that Passed Coverage is a metric that indicates the percentage of test cases that pass.

- **Passed Test Cases Percentage = (Total number of tests ran / Total number of tests executed) x 100**

3. **Failed Test Cases Percentage:** This metric measures the proportion of all failed test cases.

- **Failed Test Cases Percentage = (Total number of failed test cases / Total number of tests executed) x 100**

4. **Blocked Test Cases Percentage:** During the software testing process, this parameter determines the percentage of test cases that are blocked.

- **Blocked Test Cases Percentage = (Total number of blocked tests / Total number of tests executed) x 100**

5. **Fixed Defects Percentage:** Using this measure, the team may determine the percentage of defects that have been fixed.

- **Fixed Defects Percentage = (Total number of flaws fixed / Number of defects reported) x 100**

Software Testing Metrics

6. Rework Effort Ratio: This measure helps to determine the rework effort ratio.

- **Rework Effort Ratio = (Actual rework efforts spent in that phase/ Total actual efforts spent in that phase) x 100**

7. Accepted Defects Percentage: This measures the percentage of defects that are accepted out of the total accepted defects.

- **Accepted Defects Percentage = (Defects Accepted as Valid by Dev Team / Total Defects Reported) x 100**

8. Defects Deferred Percentage: This measures the percentage of the defects that are deferred for future release.

- **Defects Deferred Percentage = (Defects deferred for future releases / Total Defects Reported) x 100**

Software Testing Metrics Numericals

S No.	Testing Metric	Data retrieved during test case development
1	No. of requirements	5
2	The average number of test cases written per requirement	40
3	Total no. of Test cases written for all requirements	200
4	Total no. of Test cases executed	164
5	No. of Test cases passed	100
6	No. of Test cases failed	60
7	No. of Test cases blocked	4
8	No. of Test cases unexecuted	36
9	Total no. of defects identified	20
10	Defects accepted as valid by the dev team	15
11	Defects deferred for future releases	5
12	Defects fixed	12

Answers--

- ***1. Percentage test cases executed =*** *(No of test cases executed / Total no of test cases written) x 100*
  = (164 / 200) x 100
  = 82
- ***2. Test Case Effectiveness =*** *(Number of defects detected / Number of test cases run) x 100*
  = (20 / 164) x 100
  = 12.2
- ***3. Failed Test Cases Percentage =*** *(Total number of failed test cases / Total number of tests executed) x 100*
  *= (60 / 164) * 100*
  = 36.59
- ***4. Blocked Test Cases Percentage =*** *(Total number of blocked tests / Total number of tests executed) x 100*
  *= (4 / 164) * 100*
  = 2.44
- ***5. Fixed Defects Percentage =*** *(Total number of flaws fixed / Number of defects reported) x 100*
  *= (12 / 20) * 100*
  = 60
- ***6. Accepted Defects Percentage =*** *(Defects Accepted as Valid by Dev Team / Total Defects Reported) x 100*
  *= (15 / 20) * 100*
  = 75
- ***7. Defects Deferred Percentage =*** *(Defects deferred for future releases / Total Defects Reported) x 100*
  *= (5 / 20) * 100*
  = 25

Thank You

- Rahul.kumar@bennett.edu.in

Outline

- Testing Metrics
- Measurements
- Case Study

What is a Metric?

- Metrics can be defined as “STANDARDS OF MEASUREMENT”
- Metric is a unit used for describing or measuring an attribute
- **A Metric is a quantitative measure of the degree to which a system, system component, or process possesses a given attribute.**
- Test metrics are the means by which the software quality can be measured
- Test metrics provides the visibility into the readiness of the product, and gives clear measurement of the quality and completeness of the product. “How many issues are found in a thousand lines of code?”, here the number of issues is one measurement & No. of lines of code is another measurement. Metric is defined from these two measurements.

Testing Metrics

- "We cannot improve what we cannot measure" and Test Metrics helps us to do exactly the same.
- Software Testing Metrics are the quantitative measures used to estimate the progress, quality, productivity and health of the software testing process.
- The goal of software testing metrics is to improve the efficiency and effectiveness in the software testing process and to help make better decisions for further testing process by providing reliable data about the testing process.

Testing Metrics

- Software testing metrics or software test measurement is the quantitative indication of extent, capacity, dimension, amount or size of some attribute of a process or product.
- Example for software test measurement: Total number of defects.

- **Test metrics example:**
- How many defects exist within the module?
- How many test cases are executed per person?
- What is the Test coverage %?

Why we need Metrics ?

- You cannot Control what you cannot measure
- You cannot improve what you cannot measure
- Take decision for next phase of activities
- Evidence of the claim or prediction
- Understand the type of improvement required
- Take decision or process or technology change

Why Test Metrics

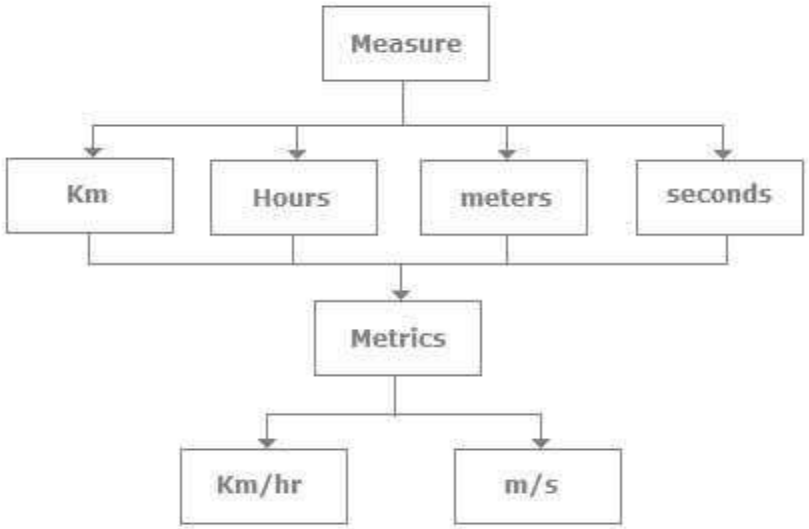
- The aim of collecting **test metrics** is to use the data for **improving the test process.** This includes finding answers to the questions like:

– How long will it take to test? – How much money will it take to test? – How bad are the bugs? – How many bugs found were fixed? reopened? closed? deferred? – How many bugs did the test team did not find? – How much of the software was tested?	– Will testing be done on time? Can the software be shipped on time? – How good were the tests? Are we using low-value test cases? – What is the cost of testing? – Was the test effort adequate? Could we have fit more testing in this release?

Software Test Measurement

- **What is Software Test Measurement**
- **Measurement indicates the extent, amount, dimension, capacity, or size of some attribute of a product or process.**

- **Test Measurement example:** Total number of defects.
- Please refer below diagram for a clear understanding of the difference between Measurement & Metrics.

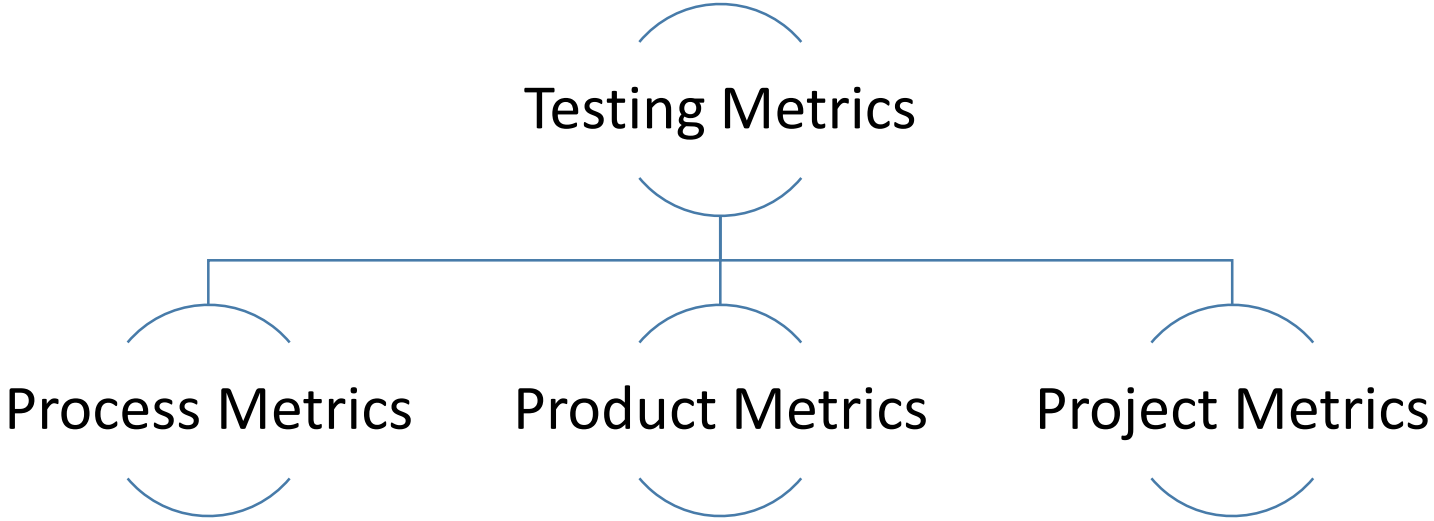


Test Metrics are used to:-

- Decide for the next phase of activities, such as estimating the cost & schedule of future projects.
- Understand the kind of improvement required to succeed in the project.
- Decide on the Process or Technology to be modified, etc.

- **Test Measurement example:** Total number of defects.
- Please refer below diagram for a clear understanding of the difference between Measurement & Metrics.
- **in the Test Report, we can publish:**
- How many test cases have been designed per requirement?
- How many test cases still need to be designed?
- How many test cases are executed?
- How many test cases are passed/failed/blocked?
- How many test cases have not been executed yet?
- How many defects are identified & what is the severity of those defects?
- How many test cases are failed due to one particular defect? etc.

Types of Test Metrics



- **Process Metrics:** used to improve the process efficiency of the SDLC (Software Development Life Cycle)
- **Product Metrics:** deals with the quality of the software product
- **Project Metrics:** used to measure the efficiency of a project team or any testing tools being used by the team members

Identifying Metrics

- Identification of correct testing metrics is very important.
- Few things need to be considered before identifying the test metrics
  - Fix the target audience for the metric preparation
  - Define the goal for metrics
  - Introduce all the relevant metrics based on project needs
  - Analyze the cost benefits aspect of each metrics and the project lifestyle phase in which it results in the maximum output

- **how can we measure the quality of the software by using Metrics**?
- Suppose, if a project does not have any metrics, then how will the quality of the work done by a Test Analyst will be measured?
- **For Example,** A Test Analyst has to,
- Design the test cases for 5 requirements.
- Execute the designed test cases.
- Log the defects & need to fail the related test cases.
- After the defect is resolved, we need to re-test the defect & re-execute the corresponding failed test case.

Measurement Scale

- Nominal Scale:
- This scale categorizes data without any inherent order or ranking.
- Examples include colors (red, blue, green), or types of fruit (apple, banana, orange).
- Numbers can be assigned to categories, but these numbers are just labels and don't indicate any quantitative relationship.

- Ordinal Scale:
- This scale represents data with a meaningful order or ranking, but the intervals between the ranks are not necessarily equal.
- Examples include rankings in a competition (1st, 2nd, 3rd) or levels of satisfaction (very dissatisfied, dissatisfied, neutral, satisfied, very satisfied

- Interval Scale:
- This scale has ordered data with equal intervals between values, but it lacks a true zero point.
- Examples include temperature measured in Celsius or Fahrenheit, where the difference between 20°C and 30°C is the same as the difference between 30°C and 40°C, but 0°C doesn't represent the absence of temperatur

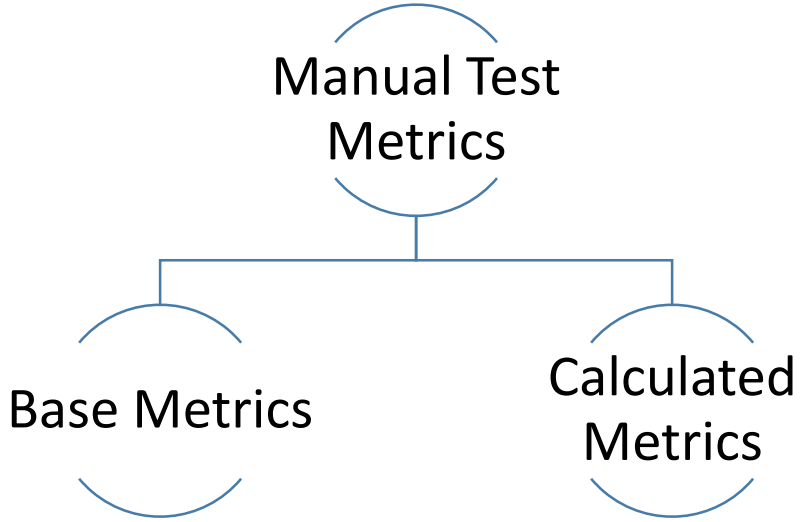
- This scale has ordered data with equal intervals and a true zero point, allowing for meaningful ratios between values.
- Examples include height, weight, age, or income, where a value of zero indicates the absence of the measured attribute.

- **Case Study Question (Basics of Metrics & Measurement)**
- **Scenario:**
  A software team is trying to introduce metrics for the first time in their testing process. They collected the following data during one test cycle of a mobile app:
- Each defect was categorized by **severity**: Critical, Major, Minor.
- Each test case execution result was marked as **Pass** or **Fail**.
- The number of test cases executed per tester per day was recorded.
- The actual effort spent (in hours) was compared with the planned effort.
- The project manager is confused about whether the data collected is **useful, measurable, and actionable**.

- Identify the **measurement scale** (Nominal, Ordinal, Interval, Ratio) for each of the following:
  a) Defect severity (Critical/Major/Minor)
  b) Test case result (Pass/Fail)
  c) Test cases executed per tester per day
  d) Effort spent in hours
- For each metric, comment whether it is **valid, reliable, and actionable**. Justify your answer.
- The manager says: *"More metrics are always better."* — Do you agree or disagree? Provide two reasons.
- Suggest **two additional useful testing metrics** that the team can collect in future, ensuring they follow the qualities of a good metric (valid, reliable, actionable).

Manual Test Metrics

- Manual test metrics are classified into two classes
  - Base Metrics
  - Calculated Metrics



Manual Test Metrics

- Base metrics is the raw data collected by Test Analyst during the test case development and execution (# of test cases executed, # of test cases).
- Calculated metrics are derived from the data collected in base metrics.
- Calculated metrics is usually followed by the test manager for test reporting purpose (% Complete, % Test Coverage).

Testing Metrics

- **Major Base Metrics:**

1. Total number of test cases	7. Number of defects rejected
2. Number of test cases passed	8. Number of defects deferred
3. Number of test cases failed	9. Number of critical defects
4. Number of test cases blocked	10. Number of planned test hours
5. Number of defects found	11. Number of actual test hours
6. Number of defects accepted	12. Number of bugs found after shipping

- Base Metrics are a great starting point, and can then be used to produce calculated metrics

Testing Metrics

- **Major Calculated Metrics:**
  1. **Test Coverage Metrics:**
     - Test coverage metrics help answer, "*How much of the application was tested*?".
     - Two major coverage metrics:
       - Test execution coverage
       - Requirement coverage

Testing Metrics

- **Major Calculated Metrics:**
  1. **Test Coverage Metrics:**

1.1 Executed Tests or Test Execution Coverage Percentage $= \left(\frac{Number\ of\ tests\ run}{Total\ number\ of\ tests\ to\ be\ run}\right) \times 100$

  - This metric gives us an idea of the total tests executed compared to the total number of tests to be run. It is usually presented as a percentage value

Testing Metrics

- **Major Calculated Metrics:**
  1. **Test Coverage Metrics:**

1.2 $$\text{Requirements Coverage} = \left(\frac{\textit{Number of requirements covered}}{\textit{Total number of requirements}}\right) \times 100$$

  - This metric gives us an idea on the percentage of the requirements that have been covered in our testing compared to the total number of requirements

Testing Metrics

- **Major Calculated Metrics:**
  2. **Test Effectiveness Metrics:** Test effectiveness answers, "*How good were the tests?*" or "*Are we running high value test cases?*" It is a measure of the bug-finding ability and quality of a test set. It can be calculated as follow:

2.1 $$\left(\frac{Bugs\ found\ in\ Test}{Total\ bugs\ found(bugs\ found\ in\ test + bugs\ found\ after\ shipping}\right) \times 100$$

  - The higher the test effectiveness percentage, the better the test set is and the lesser the test case maintenance effort will be in the long-term.
  - **Example:** If for a release the test effectiveness is 80%, it means that 20% of the defects got away from the test team.

Testing Metrics

- **Major Calculated Metrics:**
  3. **Test Effort Metrics:**
     - Test effort metrics will answer "*how long, how many, and how much*" questions about your **test effort**.
     - These metrics are great to establish **baselines** for future test planning.
     - Major Effort Metrics include:
       - Number of test runs per time, number of defects per hour, number of bugs per test, and average time to test a bug fix.

Testing Metrics

- **Major Calculated Metrics:**
  3. **Test Effort Metrics:**

3.1 $$\text{Number of tests run per time period} = \frac{\text{Number of tests run}}{\text{Total time}}$$

  – This metric gives us an idea on the number of test runs over a certain period of time. (i.e. 30 tests per one day)

Testing Metrics

- **Major Calculated Metrics:**
  3. **Test Effort Metrics:**

3.2 $$\text{Bug find rate or Number of defects per test hour} = \frac{\text{Total number of defects}}{\text{Total number of test hours}}$$

– This metric provides information about the rate of finding defects by showing the number of detected defects per test hour.

Testing Metrics

- **Major Calculated Metrics:**
  3. **Test Effort Metrics:**

3.3 $$Number\ of\ bugs\ per\ test = \frac{Total\ number\ of\ defects}{Total\ number\ of\ tests}$$

– This metric gives an estimation for the number of defects found in one test. This is calculated by dividing the total number of defects over the total number of conducted tests.

Testing Metrics

- **Major Calculated Metrics:**
  3. **Test Effort Metrics:**

$$3.4 \quad \text{Average time to test a bug fix} = \frac{\text{Total time between defect fix to retest for all defects}}{\text{Total number of defects}}$$

  - This metric presents the average time required to test a bug fix.

  - **Example:**

Average time to test a defect fix= 4/3 = 1.3 defects per day

Defect	Time between fixing the defect and retesting the fix
D1	1 day
D2	1 day
D3	2 days
Total	**4**

Testing Metrics

- **Major Calculated Metrics:**
  4. **Test Tracking and Quality Metrics:**

4.1 Passed Test Cases Percentage $= \left(\frac{Number\ of\ Passed\ Tests}{Total\ number\ of\ tests\ executed} \right) \times 100$

– This metric gives an indication on the quality of the tested application. It shows the percentage of passed test cases in relation to the total number of executed tests.

Testing Metrics

- **Major Calculated Metrics:**
  4. **Test Tracking and Quality Metrics:**

4.2 $$\text{Failed Test Cases Percentage} = \left(\frac{\textit{Number of Failed Tests}}{\textit{Total number of tests executed}}\right) \times 100$$

  - This gives an indication on the quality of the tested application. It shows the percentage of failed test cases in relation to the total number of executed tests. It also gives an indication on the effectiveness of the conducted tests.

Testing Metrics

- **Major Calculated Metrics:**
  4. **Test Tracking and Quality Metrics:**

4.3 $$\text{Critical Defects Percentage} = \left(\frac{\textit{Critical Defects}}{\textit{Total defects reported}}\right) \times 100$$

  - This metric tracks the percentage of the critical defects in relations to the total number of reported defects.

Testing Metrics

- **Major Calculated Metrics:**
  4. **Test Tracking and Quality Metrics:**

4.4 $$\text{Fixed Defects Percentage} = \left(\frac{\textit{Defects Fixed}}{\textit{Defects Reported}}\right) \times 100$$

  - This metric calculates the percentage of the fixed defects in relations to the number of the reported defects. This also gives an indication on the efficiency of testing.

Testing Metrics

- **Major Calculated Metrics:**
  5. **Test Efficiency Metrics:**

5.1 $$\text{Average time to repair defects} = \left(\frac{\textit{Total time taken for bug fixes}}{\textit{Number of bugs}}\right)$$

  - This metric calculates the average time taken to repair a defect by dividing the total time taken to fix all bugs over the total number of bugs. This metric gives an indication on how efficient repairing defects is.

Testing Metrics

- **Major Calculated Metrics:**
  - More Metrics are used to monitor the testing team productivity, Cost of testing products, etc.
  - You can read more about testing metrics in:
    - https://www.qasymphony.com/blog/64-test-metrics/
    - https://softcrylic.com/blogs/top-25-metrics-measure-continuous-testing-process/
    - https://www.getzephyr.com/resources/whitepapers/qa-metrics-value-testing-metrics-within-software-development

- **Conceptual Questions**
- What is the purpose of test coverage metrics in software testing?
- Differentiate between **code coverage** and **requirement coverage**.
- Why are **test effort metrics** important for project managers?
- Explain how defect density is calculated and what it indicates.
- Discuss the difference between **verification metrics** and **validation metrics** with examples.

Q. What is the purpose of test coverage metrics in software testing?

- Purpose of **test coverage metrics**
- Show how much of the target (code, requirements, risks) has been exercised by tests.
- Reveal untested areas → risk visibility & prioritization.
- Help plan additional tests and track progress.

- Differentiate between **code coverage** and **requirement coverage**.

- **Code coverage vs Requirement coverage**
- Code coverage: % of code elements executed (lines/branches/conditions). Technical, white-box.
- Requirement coverage: % of requirements linked to/validated by tests. Business/functional, traceability focused.

Q. Why are **test effort metrics** important for project managers?

- Why **test effort metrics** matter
- Forecast staffing & schedule; detect over/under-utilization.
- Identify bottlenecks (e.g., execution vs setup).
- Support productivity tracking and continuous improvement.

Q. Explain how defect density is calculated and what it indicates.

- **Defect density** (what/why)
- Defects per size unit (e.g., per KLOC, per function point).
- Indicates defect concentration/hotspots; used to compare modules or releases of similar size.

- Q. A project has 500 requirements. 420 of them have been tested. Calculate the **requirement coverage**.

1. **Requirement coverage**

   Formula: $\%RC = \frac{\text{Requirements tested}}{\text{Total requirements}} \times 100$

   Calc: $\frac{420}{500} \times 100 = 84\%$

   Interpretation: 84% of requirements are validated by tests; 16% remain untested.

Q. If 1200 lines of code are written and 900 lines are covered by test cases, compute the **code coverage percentage?**

Cumulative Test Time

- The total amount of time spent actually testing the product measured in test hours
- Provides an indication of product quality
- Is used in computing software reliability growth (the improvement in software reliability that results from correcting faults in the software)

2. **Code coverage**

   Formula: $\%CC = \frac{\text{LOC covered}}{\text{Total LOC}} \times 100$

   Calc: $\frac{900}{1200} \times 100 = 75\%$

   Interpretation: 25% of the code has never been executed by tests.

Test Coverage Metrics

- Code Coverage (How much of the code is being exercised?)
- Requirements coverage (Are all the product's features being tested?)
  - The percentage of requirements covered by at least one test

Quality Metrics

- Defect removal percentage
  - What percentage of known defects is fixed at release?
  - [Number of bugs fixed prior to release/ Number of known bugs prior to release] x 100
- Defects reported in each baseline
  - Can be used to help make decisions regarding process improvements, additional regression testing, and ultimate release of the software
- Defect detection efficiency
  - How well are we performing testing?
  - [Number of unique defects we find / (Number of unique defects we find + Number of unique defects reported by customers)] x 100
  - Can be used to help make decisions regarding release of the final product and the degree to which your testing is similar to actual customer use

Summary

- Fundamental test metrics are a combination of Base Metrics that can then be used to produce Calculated Metrics.
  - Base Metrics → raw collected data
  - Calculated Metrics → calculated from the base metrics

Summary

- Major Base Software Test Metrics
  - Total number of test cases
  - Number of test cases passed
  - Number of test cases failed
  - Number of test cases blocked
  - Number of defects found
  - Number of defects accepted
  - Number of defects rejected
  - Number of defects deferred
  - Number of critical defects
  - Number of planned test hours
  - Number of actual test hours
  - Number of bugs found after shipping
- Major Calculated Software Test Metrics
  - Coverage
    - Test Execution Coverage
    - Requirement Coverage
  - Effectiveness
  - Effort
    - Number of tests run per time
    - Number of defects per hour
    - Number of defects per test
    - Average time to test a bug fix
  - Quality
    - Passed test cases
    - Failed test cases
    - Critical test cases
    - Fixed Defect percentage
  - Efficiency

Fundamental Questions in Testing

- When can we stop testing?
  - ✓Test coverage
- What should we test?
  - ✓Test generation
- Is the observed output correct?
  - ✓Test oracle
- How well did we do?
  - ✓Test efficiency
- Who should test your program?
  - ✓Independent V&V

Interview Questions

- What is difference between QA, QC and Software Testing?
- What is verification and validation?
- Explain Branch Coverage and Decision Coverage.
- What is pair-wise programming and why is it relevant to software testing?
- Why is testing software using concurrent programming hard? What are races and why do they affect system testing.
- Phase in detecting defect: During a software development project two similar requirements defects were detected. One was detected in the requirements phase, and the other during the implementation phase.
- Why do we measure defect rates and what can they tell us?
- What is Static Analysis?

What is difference between QA, QC and Software Testing?

- **Quality Assurance (QA):** QA refers to the planned and systematic way of monitoring the quality of process which is followed to produce a quality product. QA tracks the outcomes and adjusts the process to meet the expectation. QA is **not** just testing.
- **Quality Control (QC):** Concern with the quality of the product. QC finds the defects and suggests improvements. The process set by QA is implemented by QC. The QC is the responsibility of the tester.
- **Software Testing:** is the process of ensuring that product which is developed by the developer meets the user requirement. The motive to perform testing is to find the bugs and make sure that they get fixed.

Verification And Validation

What is verification and validation?

- **Verification:** process of evaluating work-products of a development phase to determine whether they meet the specified requirements for that phase.
- **Validation:** process of evaluating software during or at the end of the development process to determine whether it meets specified requirements.

Branch Coverage and Decision Coverage

Explain Branch Coverage and Decision Coverage.

- **Branch Coverage** is testing performed in order to ensure that every branch of the software is executed at least once. To perform the Branch coverage testing we take the help of the Control Flow Graph.
- **Decision coverage** testing ensures that every decision taking statement is executed at least once.
- Both decision and branch coverage testing is done to ensure the tester that no branch and decision taking statement will lead to failure of the software.
- To Calculate Branch Coverage:
  - **Branch Coverage = Tested Decision Outcomes / Total Decision Outcomes.**

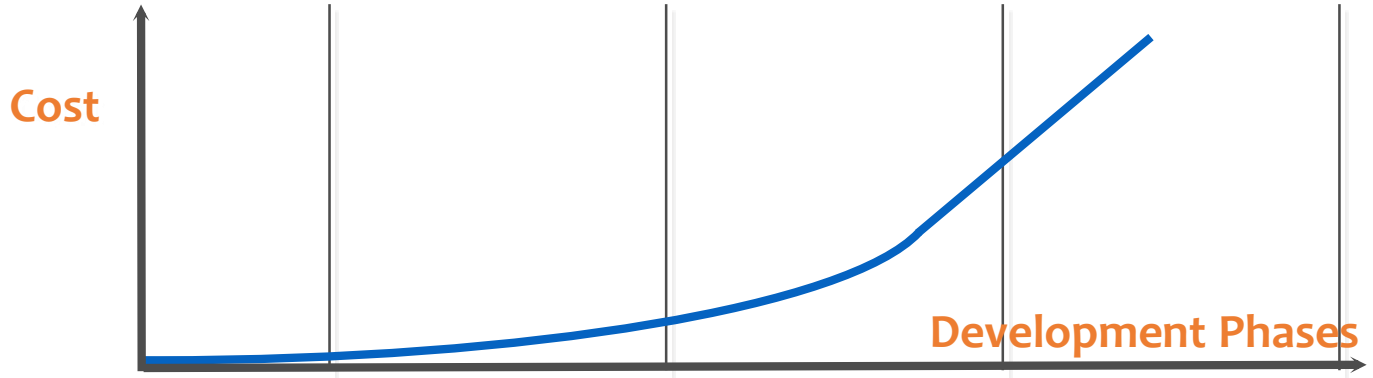
What is Static Analysis?

- The term "static analysis" is conflated, but here we use it to mean a collection of algorithms and techniques used to analyze source code in order to automatically find bugs.
- The idea is similar in spirit to compiler warnings (which can be useful for finding coding errors) but to take that idea a step further and find bugs that are traditionally found using run-time debugging techniques such as testing.
- Static analysis bug-finding tools have evolved over the last several decades from basic syntactic checkers to those that find deep bugs by reasoning about the semantics of code.

Defect Costs

Questions:

- When you find one, how much will it cost to fix?
  - How much depends on when the defect was created vs. when you found it?
- Just how many do you think are in there to start with?!



The cost of fixing a defect rises exponentially by lifecycle phase

- But this is simplistic
  - When were the defects injected?
  - Are all defects treated the same?
  - Do we reduce costs by getting better at fixing or at prevention?

“QA” & Testing

- Testing “Phases”
  - ✓Unit
  - ✓Integration
  - ✓System
  - ✓User Acceptance Testing
- Testing Types
  - ✓Black-box
  - ✓White-box
- Static vs. Dynamic Testing
- Automated Testing
  - ✓Pros and cons
- Integration: 2 types
  - ✓Top down
  - ✓Bottom up

Software Quality Assurance

- **Software quality assurance** (SQA) is a process that ensures that developed **software** meets and complies with defined or standardized **quality** specifications.
- SQA is an ongoing process within the **software** development life cycle (SDLC) that routinely checks the developed **software** to ensure it meets desired **quality** measures.

Software Quality Assurance

- The area of Software Quality Assurance can be broken down into a number of smaller areas such as
  - Quality of planning,
  - Formal technical reviews,
  - Testing

  and
  - Training.

Professional Ethics

- If you can't test it, **don't build it**
- Put **quality first** : Even if you lose the argument, you will gain respect
- Begin test activities **early**
- **Decouple**
  - **Designs** should be independent of language
  - **Programs** should be independent of environment
  - Couplings are **weaknesses** in the software!
- **Don't take shortcuts**
  - If you lose the argument you will **gain respect**
  - **Document** your objections
  - **Vote** with your feet
  - Don't be afraid to be **right**!

Professional Ethics

- **Recognizing the ACM and IEEE code of ethics for engineers:**
- PUBLIC - Software testers shall act consistently with the public interest.
- CLIENT AND EMPLOYER - Software testers shall act in a manner that is in the best interests of their client and employer, consistent with the public interest.
- PRODUCT - Software testers shall ensure that the deliverables they provide (on the products and systems they test) meet the highest professional standards possible.
- JUDGMENT - Software testers shall maintain integrity and independence in their professional judgment.
- MANAGEMENT - Software test managers and leaders shall subscribe to and promote an ethical approach to the management of software testing.
- PROFESSION - Software testers shall advance the integrity and reputation of the profession consistent with the public interest.
- COLLEAGUES - Software testers shall be fair to and supportive of their colleagues, and promote cooperation with software developers.
- SELF - Software testers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Test Metrics & Measurements and Testing vs Debugging

Software Testing - Module I (Remaining Parts)

Testing & Debugging

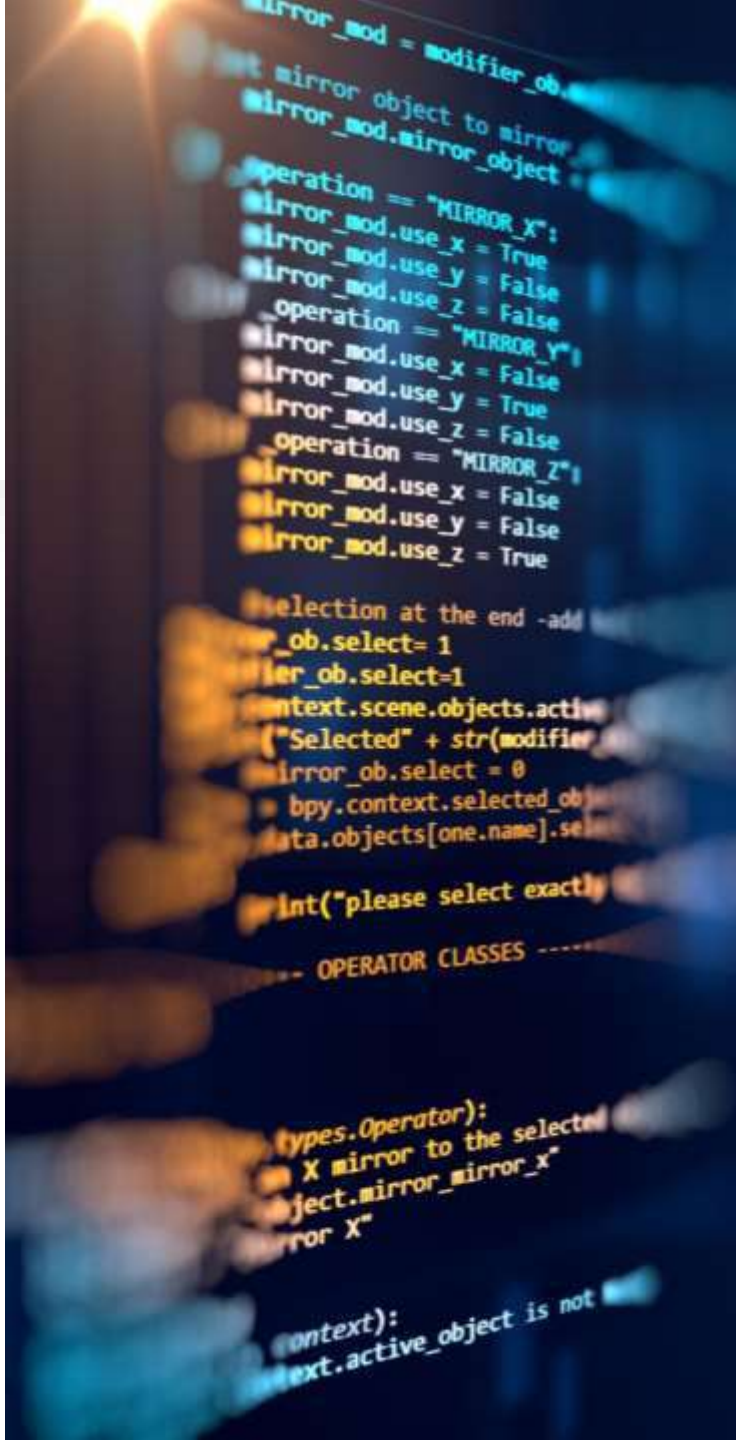
Testing:

- Process of finding defects in a software product.
- Can be manual or automated. Performed by both Tester and Developer.
- Goal: Identify and report defects.

Debugging:

- Process of locating and fixing the defects found during testing.
- Done by developers.

Goal: Remove defects and make software work as intended.



Testing and Debugging

Testing finds defects; Debugging fixes them.

Testing can be done both by testers and developers; Debugging is done by developers.

Testing is planned and systematic; Debugging is ad-hoc.

Testing can be automated; Debugging is manual.

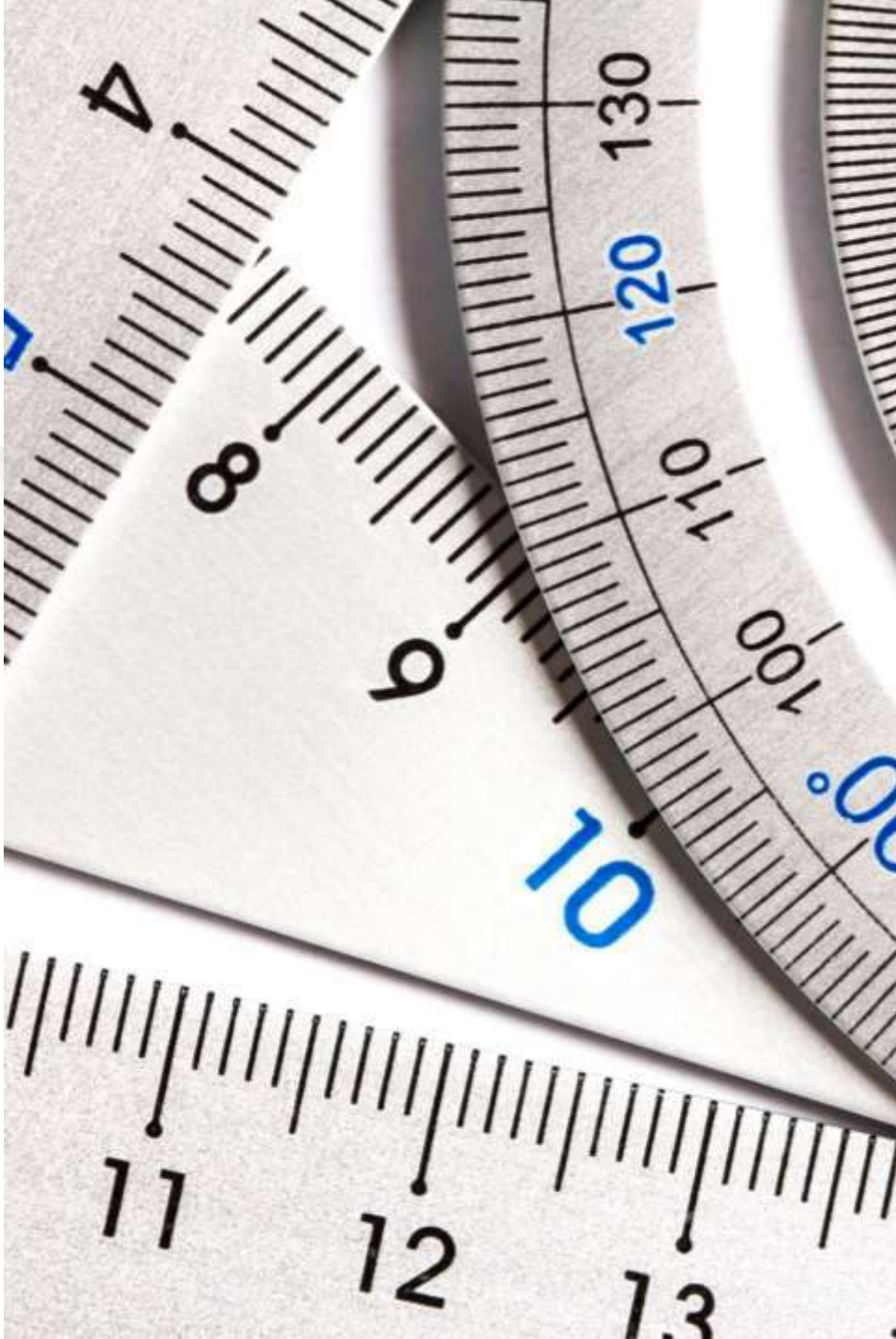
Testing aims at quality assurance; Debugging aims at defect removal.

TESTING VS DEBUGGING

	Testing	Debugging
Objective	Objective of testing is to identify defects in the application.	Purpose of debugging is to fix errors observed in testing.
Process	It is the process to identify the failure of code.	It is the process of fixing mistakes in the code.
Programming Knowledge	Most testing requires no knowledge of the source code.	Debugging requires a proper understanding of the source code.
Performed By	Tester performs testing. Testing can be carried out both by insiders and by outsiders.	Debugging is performed by a programmer. Only an insider does the debugging.
Design Knowledge	Design knowledge is not needed in the testing process.	Design knowledge is not needed but recommended as haphazardly fixing bugs can cause other defects or maintenance problems in the future.
Automation	Testing can be manual or automated.	Debugging is always manual.
Initiation	Testers can only initiate testing after developers write the code.	Debugging begins after a failed test case execution.

Test Metrics & Measurements - Introduction

- Test metrics are quantitative measures to assess the efficiency and effectiveness of testing.
- They help track progress, quality, and performance.
- Measurements are the actual values obtained from metrics.
- **Example**: Defect density, Test coverage.



Types of Test Metrics

Process Metrics: Measure the efficiency of the testing process (e.g., defect removal efficiency).

Product Metrics: Measure the quality of the product (e.g., defect density).

Project Metrics: Measure the overall project health (e.g., test case execution rate).

Common Test Metrics

Defect Density = Number of defects / Size of software.

Test Coverage = (Number of requirements tested / Total requirements) × 100%.

Defect Removal Efficiency = (Defects found in testing / Total defects) × 100%.

Mean Time to Detect (MTTD) and Mean Time to Repair (MTTR).

Benefits of Using Test Metrics

Objective evaluation of testing progress.

Identification of bottlenecks.

Better resource allocation.

Improved decision making.

Helps in process improvement.

Summary

TESTING AND DEBUGGING ARE DISTINCT BUT COMPLEMENTARY ACTIVITIES.

TEST METRICS PROVIDE QUANTITATIVE DATA TO IMPROVE TESTING.

MEASURING THE RIGHT METRICS ENSURES BETTER QUALITY AND PRODUCTIVITY.

Software Testing –Module 2

Presented by: Rahul Kumar Balyan, Assistant Professor

Department of Computer Science Engineering and Technology

Verification and Validation

- **Verification** evaluates software artifacts (such as requirements, design, code, etc.) to ensure they meet the specified requirements and standards. It ensures the software is built according to the needs and design specifications.
- **Validation** evaluates software to meet the user's needs and requirements. It ensures the software fits its intended purpose and meets the user's expectations.

Verification

- It involves reviewing various documents like requirements specification, design blueprints, ER diagrams, database table design, test cases, test scenarios, traceability matrix, etc. to ensure everything aligns with the project's standards and protocols.

- **Code reviews, walkthroughs, inspections, design,** and **specification analysis** are common components of verification testing.

- Verification tests ensure that all development elements (software, hardware, documentation, and human resources) adhere to organizational and team-specific standards and protocols.

Techniques for Verification Testing

Here are some techniques followed for verification testing:

- **Requirement Reviews:** Evaluating requirement documents to ensure they are complete, clear, and testable before development begins. It reduces ambiguity and rework.
- **Design Reviews:** Systematically examining software design artifacts to verify logical correctness, alignment with requirements, and readiness for implementation.
- **Code Reviews:** Peer reviewing source code to catch bugs, enforce coding standards, and ensure code quality before unit testing starts.
- **Static Code Analysis:** Using automated tools to analyze source code for defects, security issues, or maintainability problems without executing the code.

Techniques for Verification Testing

- **Walkthroughs:** Informal sessions where developers or testers explain code, design, or test plans to peers to validate logic and uncover potential issues early.
- **Inspections:** Formal, structured reviews of code or documentation involving checklists and assigned roles to detect issues before dynamic testing begins.
- **Unit Testing:** Writing and executing test cases for individual units or functions to verify they work as expected in isolation.
- **Test Case Review:** Reviewing test cases and test scripts to ensure they cover all scenarios, follow standards, and align with requirements.
- **Traceability Checks:** Verifying that every requirement has corresponding test coverage, ensuring that all intended functionality is accounted for in the test plan.

Advantages of Verification Testing

The below are some of the top advantages of verification testing:

- Early and frequent verification reduces the number of bugs and defects that may show up in later stages.
- By verifying at each stage, devs, product managers, and stakeholders can get more insight into what the product may need to be developed better in the coming stages.
- Even if they can’t solve all bugs immediately, verifying helps QAs estimate the emerging issues and help them better prepare to handle those when they appear.
- Verification helps keep software closely aligned with customers and business requirements at every stage. This ensures that devs have to put in less unnecessary work as development continues.

When to use Verification Testing?

- **Before Code Integration:** When developers complete a module of code, verification testing ensures that this module meets its design specifications and works correctly before integrating it into the larger system.

- **During Design Reviews:** After creating a design blueprint for a new feature, verification testing checks if the design aligns with the initial requirements and design documents.

- **Before User Acceptance Testing:** Before a system is handed over for user acceptance testing, verification testing confirms that the system meets all specified requirements and is free from critical errors.

When to use Verification Testing?

- **After Software Updates:** When a software update or patch is applied, verification testing ensures that the changes have been implemented correctly and that they meet the specified requirements without introducing new issues.
- **During System Integration:** When different system components are integrated, verification testing checks if the integrated system meets all the defined requirements and functions as expected.
- **When Requirements Change:** If project requirements are updated or revised, verification testing ensures that the system still complies with the new requirements.

What is Validation Testing?

Validation is an activity that ensures that an end product stakeholder’s true needs and expectations are met.

- Unlike verification testing, which happens throughout the development process, validation testing typically occurs at the end of a development phase or after the entire system is built.
- Its main goal is to confirm that the final product aligns with what stakeholders and customers wanted.

Techniques for Validation Testing

1. **Functional Testing:** Validates that the software functions according to the specified requirements by executing test cases that cover all features and user interactions.

2. **Integration Testing:** Tests interactions between integrated modules to ensure data is correctly exchanged and workflows behave as expected across components.

3. **System Testing:** Performs end-to-end testing on the fully integrated system to validate that it meets the defined business and technical requirements.

Techniques for Validation Testing

4. User Acceptance Testing (UAT): Executed by end users or clients to verify that the software meets their needs and is ready for deployment in a real-world environment.

5. Regression Testing: Reruns previously executed test cases to ensure that recent code changes have not adversely affected existing functionality.

6. **Smoke Testing:** Performs a basic set of tests to validate that the most critical functionalities of the application work, typically done after a new build is deployed.

Techniques for Validation Testing

7. Sanity Testing: A quick, focused validation of a specific module or functionality after minor changes to ensure that it behaves correctly.

8. **Performance Testing:** Validates the system's responsiveness, stability, and scalability under expected or stress conditions to ensure it meets performance requirements.

9. Security Testing: Checks the software for vulnerabilities, data protection flaws, and access control issues to ensure the application is secure.

10. **Usability Testing:** Assesses how user-friendly and intuitive the software is by observing real users completing tasks, ensuring a positive user experience.

Advantages of Validation Testing

- Any bugs missed during verification will be detected while running validation tests.
- If specifications were incorrect and inadequate, validation tests would reveal their inefficacy. Teams will have to spend time and effort fixing them, but it will prevent a bad product from hitting the market.
- Validation tests ensure that the product matches and adheres to customer demands, preferences, and expectations under different conditions (slow connectivity, low battery, etc.)

When to use Validation Testing?

Some popular scenarios where validation testing is used:

- **After System Completion:** Once the entire system or software is fully developed and all components are integrated, validation testing ensures that the final product meets the requirements and expectations of stakeholders.
- **During User Acceptance Testing (UAT):** When the product is ready for end-users to test, validation testing is used to confirm that it meets their needs and functions correctly in real-world scenarios.
- **Before Product Launch:** Prior to releasing a product to the market, validation testing is conducted to ensure that it satisfies customer requirements and performs as expected under actual usage conditions.

Real-world example

- A real-world example of validation testing is a User Acceptance Testing (UAT) for a new e-commerce website.
- Where actual end-users test the site to ensure it fulfills their needs, such as an intuitive shopping cart, easy checkout, and accurate display of product information, thereby validating that the software meets real-world user expectations rather than just technical specifications

When to use Validation Testing?

- **After Major Changes or Enhancements:** When significant updates or new features are added to a system, validation testing checks that these changes align with stakeholder expectations and do not negatively impact the overall functionality.
- **During Beta Testing:** In beta testing phases, where a product is released to a select group of users for feedback, validation testing is used to gather insights on whether the product meets user needs and expectations.
- **Post-Implementation Review:** After the product is deployed and in use, validation testing can be performed to verify that it continues to meet the stakeholders' needs and functions as intended.

Process for Validation Testing

Unlike verification (which checks if you built it right), validation checks if you built the right thing. Here's the typical process for validation testing:

1. **Requirement Analysis & Validation:** Ensure that all business and functional requirements are well-understood, testable, and aligned with what the end users actually need.

2. **Validate Test Plan:** Define the scope and objectives of validation testing, including types of tests to be performed (for example, system testing, UAT), responsibilities, schedules, and required environments.

3. **Design of Test Scenarios and Cases:** Create test scenarios and test cases that simulate real-world usage, focusing on validating user workflows, critical functionality, and business processes.

4. **Test Environment Setup:** Set up a realistic test environment that mirrors production as closely as possible to validate behaviour under actual operating conditions.

Process for Validation Testing

5. Execution of Validation Tests: Execute validation test cases, including functional testing, system testing, user acceptance testing, and usability testing to assess whether the software performs as intended.

6. Defect Logging and Analysis: Identify and record any issues, deviations, or unmet requirements found during validation. Prioritize and address them in collaboration with development.

7. Re-testing and Final Validation: Re-run failed tests after fixes and perform necessary regression testing to ensure that the software remains stable and fully functional.

Software Quality

What is Software Quality?

- Software Quality shows how good and reliable a product is. To convey an associate degree example, think about functionally correct software. It performs all functions as laid out in the SRS document (software requirement specifications).
- Another example is also that of a product that will have everything that the users need but has an associate degree virtually incomprehensible and not maintainable code. Therefore, the normal construct of quality as “fitness of purpose” for code products isn't satisfactory.
- It has an associate degree virtually unusable program. even though it should be functionally correct, we tend not to think about it to be a high-quality product.

Software Quality

- A product (like software) might do **everything the user needs** (so from the outside it looks perfect),
  BUT the **code inside** could be so messy, hard to read, and unorganized that developers cannot **understand it or maintain it**.
- So, if you only define software quality as **“fitness of purpose”** (meaning it works and serves the user’s needs), that definition is **not enough**. Because true quality also means:
- The code should be **readable**,
- **Maintainable**,
- **Easy to improve or fix in the future**.

Example--Imagine a car that looks great, runs smoothly, and satisfies the driver.

- But inside the engine, all the parts are arranged in a very weird way, no mechanic can repair it easily

Factors of Software Quality

Factors of Software Quality

1. **Portability:** A software is claimed to be transportable, if it may be simply created to figure in several package environments, in several machines, with alternative code products, etc.(In different OS)

2. **Usability:** A software has smart usability if completely different classes of users (i.e. knowledgeable and novice users) will simply invoke the functions of the products.

3. **Reusability:** A software has smart reusability if completely different modules of the products will simply be reused to develop new products.

4. **Correctness:** Software is correct if completely different needs as laid out in the SRS document are properly enforced.

Factors of Software Quality

5. **Maintainability:** A software is reparable, if errors may be simply corrected as and once they show up, new functions may be simply added to the products, and therefore the functionalities of the products may be simply changed, etc

6. **Reliability:** Software is more reliable if it has fewer failures. Since software engineers do not deliberately plan for their software to fail, reliability depends on the number and type of mistakes they make. Designers can improve reliability by ensuring the software is easy to implement and change, by testing it thoroughly, and also by ensuring that if failures occur, the system can handle them or can recover easily.

7. **Efficiency.** The more efficient software is, the less it uses of CPU-time, memory, disk space, network bandwidth, and other resources. This is important to customers in order to reduce their costs of running the software, although with today's powerful computers, CPU time, memory and disk usage are less of a concern than in years gone by.

Software Quality Management System

- Software Quality Management System contains the methods that are used by the authorities to develop products having the desired quality.

Some of the methods are:

- **Managerial Structure:** Quality System is responsible for managing the structure as a whole. Every Organization has a managerial structure.

- **Individual Responsibilities:** Each individual present in the organization must have some responsibilities that should be reviewed by the top management and each individual present in the system must take this seriously.

- **Quality System Activities:** The activities which each quality system must have been

  - Project Auditing.
  - Review of the quality system.
  - It helps in the development of methods and guidelines.

Evolution of Quality Management System

- Quality Systems are basically evolved over the past some years. The evolution of a Quality Management System is a four-step process.

1. **Inspection:** Product inspection task provided an instrument for quality control (QC).

2. **Quality Control:** The main task of quality control is to detect defective devices, and it also helps in finding the cause that leads to the defect. It also helps in the correction of bugs.

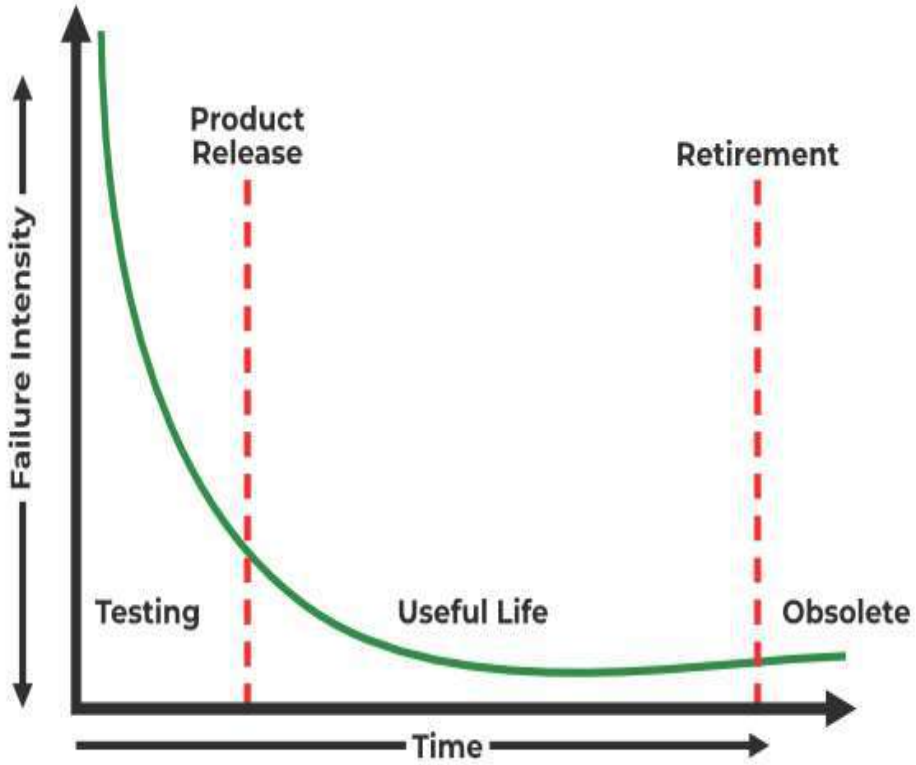
3. **Quality Assurance:** Quality Assurance helps an organization in making good quality products. It also helps in improving the quality of the product by passing the products through security checks.

4. **Total Quality Management (TQM):** Total Quality Management(TQM) checks and assures that all the procedures must be continuously improved regularly through process measurements.

Software Reliability

- Software reliability is the probability that the software will operate failure-free for a specific period of time in a specific environment. It is measured per some unit of time.
- Software Reliability starts with many faults in the system when first created.
- After testing and debugging enter a useful life cycle.
- Useful life includes upgrades made to the system which bring about new faults.
- The system needs to then be tested to reduce faults.
- Software reliability cannot be predicted from any physical basis, since it depends completely on the human factors in design.

Software Reliability

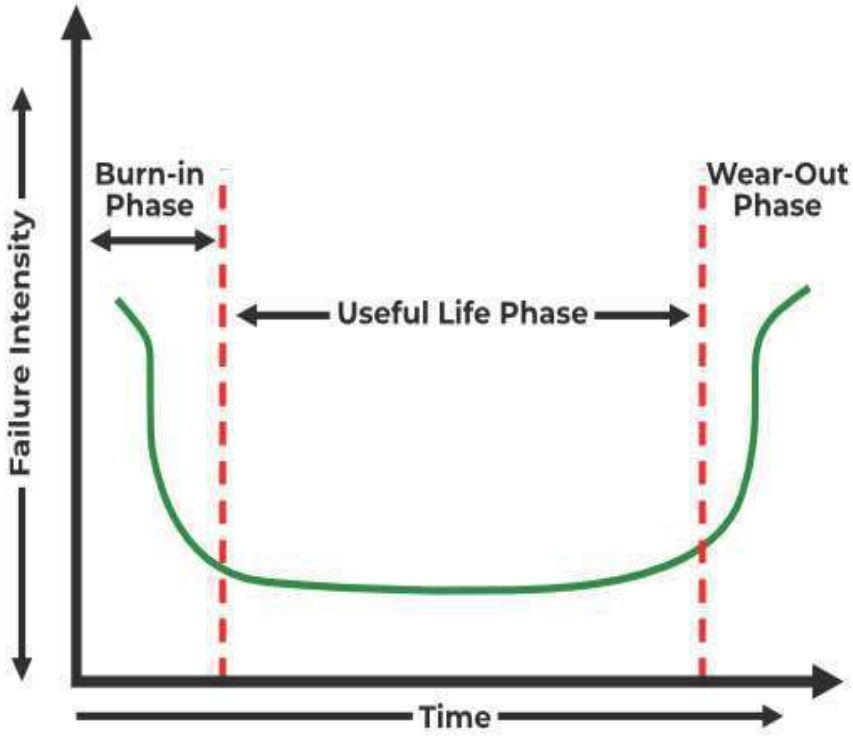


(b) Software Product

Hardware Reliability

- Hardware reliability is the probability that the ability of the hardware to perform its function for some period of time. It may change during certain periods such as initial burn-in or the end of useful life.
- It is expressed as Mean Time Between Failures (MTBF).
- Hardware faults are mostly physical faults.
- Thorough testing of all components cuts down on the number of faults.
- Hardware failures are mostly due to wear and tear.
- It follows the Bathtub curve principle for testing failure.

Hardware Reliability



(a) Hardware Product

Hardware Reliability VS Software Reliability

Features	Hardware Reliability	Software Reliability
Source of Failure	Failures are caused due to defects in design, production, and maintenance.	Failures are caused due to defects in design.
Wear and Tear	Failure occurs due to physical deterioration in wear and tear.	In software reliability, there is no wear and tear.
Deterioration Warning	In this prior deterioration warning about failure.	In this no prior deterioration warning about failure.
Failure Curve	The bathtub curve is used for failure rates apply.	There is no Bathtub curve for failure rates.
Is Failure Time-dependent?	In this failures are time-dependent.	In this failures are not time-dependent.
Reliability Prediction	In this reliability can be predicted from design.	In this reliability can not be predicted from design.
Reliability Complexity	The complexity of hardware reliability is very high.	The complexity of software reliability is low.

MTBF (Mean Time Between Failures)

- MTBF is calculated using an arithmetic mean. Basically, this means taking the data from the period you want to calculate and dividing that period's total operational time by the number of failures.

- So, let's say we're assessing a 24-hour period and there were two hours of downtime in two separate incidents. Our total uptime is 22 hours. Divided by two, that's 11 hours. So our MTBF is 11 hours.

- This Matric is used to track reliability.

- MTBF does not factor in expected down time during scheduled maintenance. Instead, it focuses on unexpected outages and issues.

- MTBF comes to us from the aviation industry, where system failures mean particularly major consequences not only in terms of cost, but human life as well. The initialism has since made its way across a variety of technical and mechanical industries and is used particularly often in manufacturing.

- MTBF is helpful for buyers who want to make sure they get the most reliable product, fly the most reliable airplane, or choose the safest manufacturing equipment for their plant

- MTBF is a metric for failures in **repairable systems**. For failures that require system replacement, typically people use the term MTTF (mean time to failure)

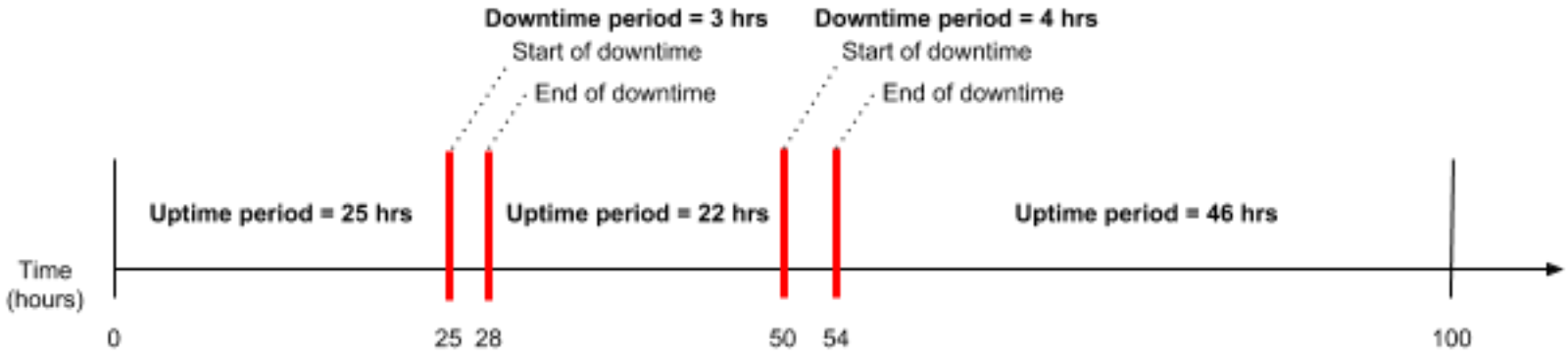
Mean Time Between Failure (MTBF) Example

- Take for example a mechanical mixer designed to operate for 10 hours per day. Suppose the mixer breaks down after normally operating for 5 days. The MTBF for this case is 50 hours as calculated below.

- **MTBF = (10 hours per day * 5 days) / 1 breakdown = 50 hours**
- The MTBF calculation will require more steps when accounting for longer periods of time with increased occurrences of breakdowns.

Mean Time Between Failure (MTBF) Example

- Let's say that the same mechanical mixer, running for 10 hours per day breaks down twice in the span of 10 days. The first breakdown occurred 25 hours from the start time and took 3 hours to repair. The second breakdown occurred 50 hours from the start time and took 4 hours to repair before the mixer was operating normally.



Mean Time Between Failure (MTBF) Example

- **MTBF = (25 hours + 22 hours + 46 hours) / 2 breakdowns = 93 hours / 2 breakdowns = 46.5 hours**

MTTR (Mean Time To Repair)

- **MTTR (mean time to repair) is the average time it takes to repair a system (usually technical or mechanical).** It includes both the repair time and any testing time. The clock doesn't stop on this metric until the system is fully functional again.

- You can calculate MTTR by adding up the total time spent on repairs during any given period and then dividing that time by the number of repairs.

- So, let's say we're looking at repairs over the course of a week. In that time, there were 10 outages and systems were actively being repaired for four hours. Four hours is 240 minutes. 240 divided by 10 is 24. Which means the mean time to repair in this case would be 24 minutes.

Question---

- The mean time between failures of machine is 400 hrs. If the availability of machine is 80% then the mean time to repair of machine in hours?

Solution--

Concept:

Availability,

$$A = \frac{MTBF}{MTBF+MTTR}$$

Where MTBF = Mean time before failure, MTTR = Mean time to repair.

Calculation:

Given:

A = 80% = 0.8, MTBF = 400

Availability,

$$A = \frac{MTBF}{MTBF+MTTR}$$

$$0.8 = \frac{400}{400+MTTR}$$

400 + MTTR = 500

MTTR = 100

Question

1. A product has MTBF of 200 hours and MTTR of 10 hours. What is the availability of product.

2. The following table shows the time between failures for a software:

MTBF???

ERROR NUMBER.	1	2	3	4	5
Time since last failure (hours).	6	4	8	5	6

MTTA: Mean Time To Acknowledge

- **MTTA (mean time to acknowledge) is the average time it takes from when an alert is triggered to when work begins on the issue.** This metric is useful for tracking your team's responsiveness and your alert system's effectiveness.

- To calculate your MTTA, add up the time between alert and acknowledgement, then divide by the number of incidents.

- For example: If you had 10 incidents and there was a total of 40 minutes of time between alert and acknowledgement for all 10, you divide 40 by 10 and come up with an average of four minutes.

- MTTA is useful in tracking responsiveness. Is your team suffering from alert fatigue and taking too long to respond? This metric will help you flag the issue.

MTTA: Mean Time To Acknowledge question

- A monitoring system produced 6 alerts during a week. The times (in minutes) between each alert and when an engineer acknowledged them were: **2, 3, 1.5, 4, 2.5, 3**.
  Find the MTTA.

- A system ran for 10,000 hours in total across several units and during that period experienced 8 failures. The average repair time per failure was 5 hours.
  a) Compute **MTBF**.
  b) Compute **Availability**.

Solution

- MTBF = total operating time ÷ number of failures
  = 10,000 ÷ 8
- Do division stepwise:
  8 × 1250 = 10,000 exactly
- So **MTBF = 1,250 hours**
- **Solution — (b) Availability**
- MTTR = average repair time = 5 hours
  Availability = MTTF / (MTTF + MTTR)
  = 1250 ÷ (1250 + 5)
  Denominator = 1250 + 5 = 1255
- Compute fraction: 1250 / 1255 = 0.996012...
- Rounded → **Availability ≈ 0.9960 (or 99.60%)**

Question

- A critical service has MTBF = 2,000 hours and MTTR = 4 hours.
  a) What is the expected number of failures per year (assume 1 year = 8,760 hours)?
  b) What is the expected total downtime per year (in hours and as a percentage of the year)?

Solution

- Expected failures per year = total hours per year ÷ MTTF
  = 8760 ÷ 2000
- Compute:
  2000 × 4 = 8000; remainder 760
  So 8760 ÷ 2000 = 4 + (760/2000) = 4 + 0.38 = 4.38 failures/year
- We usually interpret as **about 4.38 failures per year** (i.e., ~4–5 failures).
- **Solution — (b) Total downtime per year**
- Each failure causes MTTR = 4 hours downtime
- Total yearly downtime = failures_per_year × MTTR
  = 4.38 × 4 = 17.52 hours/year
- Percentage of year = (17.52 ÷ 8760) × 100%
  First compute fraction: 17.52 ÷ 8760 ≈ 0.0020 (stepwise)
- 8760 × 0.002 = 17.52 exactly, so fraction = 0.002
- Percentage = 0.002 × 100% = **0.2%**
- **Answers: ~4.38 failures/year**; **17.52 hours/year downtime ≈ 0.2% of year**

MTTF: Mean Time To Failure

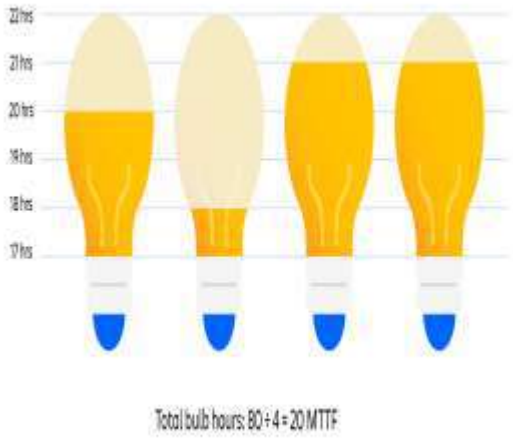
- **MTTF (mean time to failure) is the average time between non-repairable failures of a technology product.** For example, if Brand X's car engines average 500,000 hours before they fail completely and have to be replaced, 500,000 would be the engines' MTTF.

- The calculation is used to understand how long a system will typically last, determine whether a new version of a system is outperforming the old, and give customers information about expected lifetimes and when to schedule check-ups on their system.

- Mean time to failure is an arithmetic average, so you calculate it by adding up the total operating time of the products you're assessing and dividing that total by the number of devices.

- For example: Let's say you're figuring out the MTTF of light bulbs. How long do Brand Y's light bulbs last on average before they burn out? Let's further say you have a sample of four light bulbs to test (if you want statistically significant data, you'll need much more than that, but for the purposes of simple math, let's keep this small).

- Light bulb A lasts 20 hours. Light bulb B lasts 18. Bulb C lasts 21. And bulb D lasts 21 hours. That's a total of 80 bulb hours. Divided by four, the MTTF is 20 hours.



Software Defect

- Any deviation between the real outcome of the software and its expected outcome is called a defect in software testing,
- Usually known as a bug.
- It refers to some developed flaw or error in the software due to which it is doing something unexpected, inaccurate, or returning unintended results.
- The defects may occur in any of the steps involved in the software development lifecycle: requirements gathering, design, coding, testing, or sometimes even deployment.

Examples of Defects:

- An application crashes if a user provides certain data.
- A log-in function is unable to take valid credentials
- A webpage is not getting loaded properly in a few web browsers.

Types of Defects

Awareness of the various kinds of **defects** helps with proper reporting and processing.

1. Functional Defects

- Functional defects are those that concern the failure of software to behave per the functional specifications or requirements. These types of defects directly affect the core functionality of an application.
- **Example:** A payment gateway is unable to process some transactions even though all required fields have been correctly filled.

2. Performance Defects

- Performance **defects** describe the performance of software under particular conditions. It could be related to load, stress, or volume. These are defects that create problems, such as slow response times or crashing under heavy loads.
- **Example:** A website is slow to load if many users hit it all at the same time.

Types of Defects

3. Usability Defects

- Usability defects deal with the user experience and concern issues of ease of use, navigability, and intuitiveness. This class of defects can make users frustrated, hence creating a bad user experience.
- **Example:** A button not being able to be clicked or a form field not clearly labelled.

4. Security Defects

- Security defects are vulnerabilities in the software that may be exploited to result in unauthorized access, data, or operations disruption by the attack. These are critical defects that need urgent action.
- **Example:** A web application is vulnerable to SQL injection attacks because of poor input validation.

Types of Defects

5. Compatibility Defects

- These are defects that allow software to run and execute all its functions but not across all environments, such as on different browsers, operating systems, or devices. This defect affects the accessibility and usability of the software.
- **Example:** The mobile app crashes on a specific version of an operating system.

Common Origins of Defects

A defect can come from a very wide variety of sources. Appreciating the origin of the different defects is key to preventing them in future development cycles:

- **Incomplete Requirements**: Poorly defined or ambiguous requirements are subject to misinterpretation causing a defect in the software.
- **Design Errors**: Problems in the software architecture or design can cause defects.
- **Coding Mistakes**: Various human errors during the coding phase of the software—that is to say, syntax, logic, or algorithm errors—can result whereby defects are introduced.
- **Unidentified Defects**: The testing is inadequate or bad, so that defects are not detected before software delivered.

The Defect Lifecycle

The lifetime of a defect, also known as a bug lifecycle, is the journey taken by a defect from its identification to its closure. Knowledge of this lifecycle helps manage defects efficiently.

- **Defect Detection**: The first stage is identification. Testers come to know of **defects** during testing activities when the software doesn't behave as expected.

- **Logging of Defects**: Once they are known, defects are logged into a defect tracking tool with the detailed description containing steps to reproduce, severity, screenshots, and the comparison of actual and expected results. Practically, defects are reviewed with a priority setting based on its magnitude and effect on the system during triage. This is done in order to decide on which defects should be corrected first.

- **Defect Fixing**: Developers fix the problems according to the priority setting. In this, a defect has been subjected to analysis, the root cause has been found, and the corresponding fix is applied.

- **Retesting of Defects**: Subsequently, the process of retesting the functionality that was worked on begins to verify whether the defect has been rectified and no new defects are introduced with the fix.

- **Defect Closure**: Once the retesting goes through successful bug fixing and it is seen that there is no defect, then it is closed. Else, it is re-opened and further dispatched for the fixing process.

Defect Management Tools

- Defect management tools provide for efficient defect reporting, tracking, and managing. Some of the most widely used defect management tools are:
- **JIRA:** One of the widely used tools for bug management and project development.
- **Bugzilla:** An open-source tool designed for managing software defects.
- **Redmine:** A flexible project management web tool that can be effectively used for tracking any issues and defects.
- **MantisBT:** An open-source issue tracking tool that is easy to use and simple in terms of its interface.

Manual and Automation Testing

What is Manual Testing?

- Manual testing is a type of testing in which we do not use any tools or automation to perform the testing.
- In this testing, testers make test cases for the codes test the software, and give the final report about that software.
- Manual testing is time-consuming testing because humans do it and there is a chance of human errors.
- Manual testing is conducted to discover bugs in the developed software application.
- The tester checks all the essential features of the application.
- The tester executes test cases and generates test reports without any help from the automation tools.
- It is conducted by the experienced tester to accomplish the testing process.

Manual Testing

When to Perform Manual Testing?

- Manual testing is done when automation can't be used or isn't enough.
- **Exploratory Testing**: Discovering issues in new or unclear features by exploring them.
- **Usability Testing**: Checking if the UI is easy to use and looks good.
- **Ad-Hoc Testing**: Doing quick, informal tests after updates or bug fixes.
- **Visual/GUI Testing**: Checking the layout, colors, or how the site looks across different browsers.

Manual and Automation Testing

Manual testing should be conducted when:

- **Flexibility is required:** With manual tests, QA can quickly test and provide fast feedback.

- **Short-term projects:** It is not advisable to invest more money and effort to set up short-term projects that focus on minor features because such setup will require huge effort and money that would be too high for such small projects.

- **When testing end-user usability:** Humans can use their sensibilities to understand the application behavior if the application offers a satisfactory user journey. No machine can perform this task as humans can.

Manual and Automation Testing

Benefits of Manual Testing

- **Easy hiring:** In manual testing, anyone can test so it helps in easy hiring.
- **Fast feedback:** Manual testing helps to provide fast and accurate feedback.
- **Versatile:** Manual test cases can be applied to many test cases.
- **Flexible:** Manual testing is flexible as it can adapt easily to changes in the user interface.
- **Less expensive:** Manual testing is less expensive as one does not need to spend a budget on automation tools and processes.

Automation Testing

What is Automation Testing?

- Automation testing is a type of testing in which we take the help of tools (automation) to perform the testing. It is faster than manual testing because it is done with some automation tools. There is no chance of any human errors.
- It relies entirely on pre-scripted test which runs automatically to compare actual results with expected results.
- Automation testing helps the tester determine whether the application performs as expected or not.
- It allows the execution of repetitive tasks and regression tests.
- Automation requires manual effort to create initial testing scripts.

When is Automation Testing?

When to Perform Automation Testing?

- **When need to run repetitive tasks:** Automated tests are the best option in scenarios where there is a requirement to run repetitive tests. For example, in the case of regression tests must be executed periodically to make sure that the newly added code does not disrupt the existing functionality of the software.

- **When human resources are scarce:** Automated tests are viable and the best option to get tests executed within deadlines when there are only a limited number of dedicated testers.

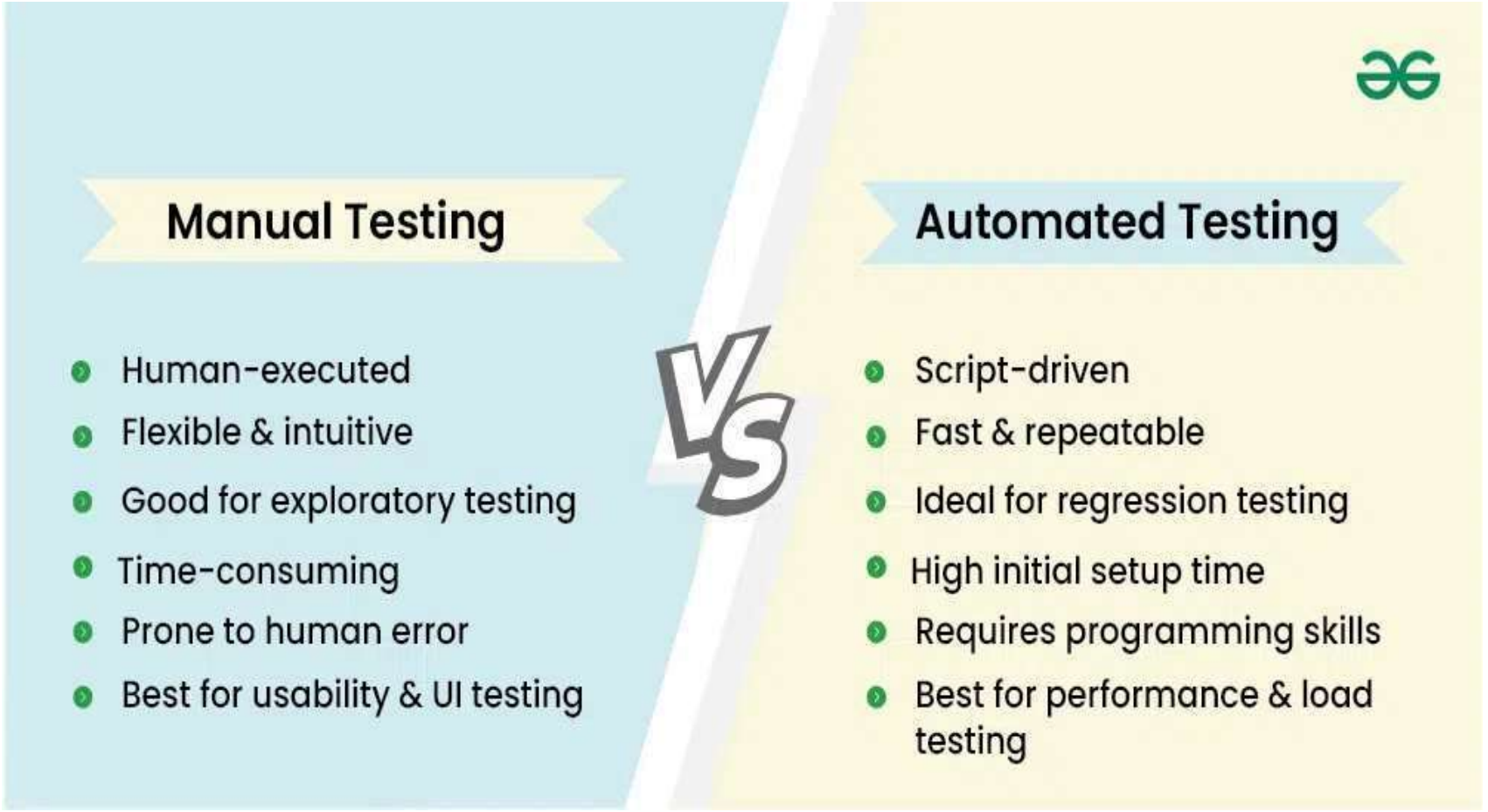
Benefits of Automation Testing

- **Finds more bugs:** Automation testing helps to find more bugs and defects in the software.
- **Reduce time for regression tests:** Automated tests are suitable for regression tests as the tests can be executed in a repetitive manner periodically.
- **The process can be recorded:** This is one of the benefits of using automation tests as these tests can be recorded and thus allows to reuse of the tests.
- **No fatigue:** As automation, tests are executed using software tools so there is no fatigue or tiring factor as in manual testing.
- **Increased test coverage:** Automation tests help to increase the test coverage as using the tool for testing helps to make sure that not even the smallest unit is left for testing.

Limitations of Automation Testing

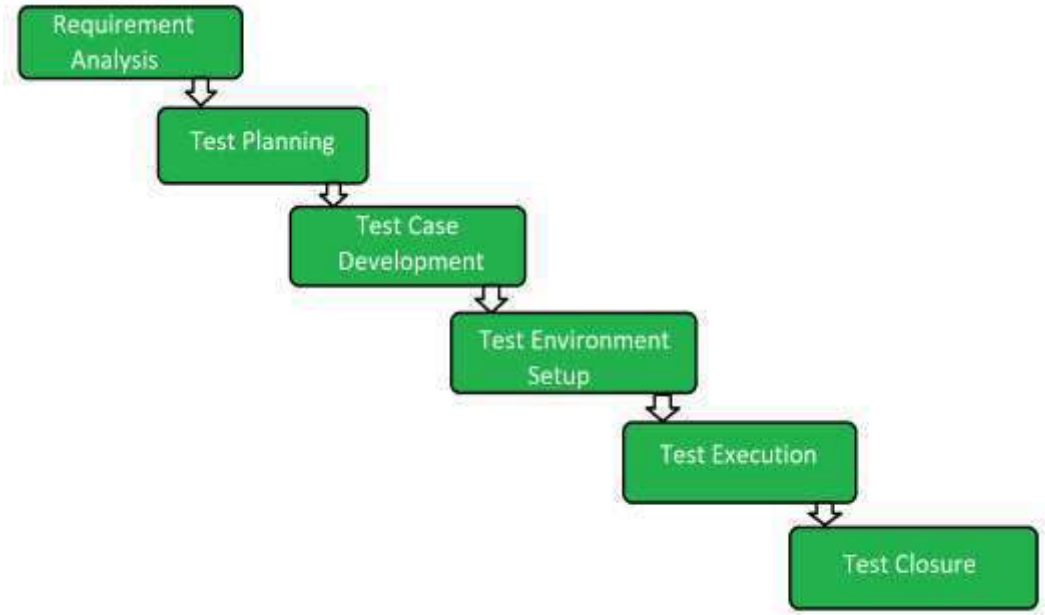
- **Difficult to inspect visual elements:** In automated tests, it is difficult to get insight into the visual elements like color, font size, font type, button sizes, etc. as there is no human intervention.
- **High cost:** Automation tests have a high cost of implementation as tools are required for testing, thus adding the cost to the project budget.
- **Test maintenance is costly:** In automation tests, test maintenance is costly.
- **Not false proof:** Automation tests also have some limitations and mistakes in automated tests can lead to errors and omissions.
- **Trained employees required:** For conducting automated tests, trained employees with knowledge of programming languages and testing knowledge are required.

Manual Testing VS Automated Testing



Software Testing Life Cycle (STLC)

- The **Software Testing Life Cycle (STLC)** is a process that verifies whether the Software Quality meets the expectations or not.
- STLC is an important process that provides a simple approach to testing through the step-by-step process.
- Software Testing Life Cycle (STLC) is a fundamental part of the Software Development Life Cycle (SDLC)



Software Testing Life Cycle (STLC)

1. Requirement Analysis

- Requirement Analysis is the first step of the Software Testing Life Cycle (STLC).
- In this phase quality assurance team understands the requirements like what is to be tested.
- If anything is missing or not understandable then the quality assurance team meets with the stakeholders to better understand the detailed knowledge of requirements

Requirement Analysis stage include:

- Reviewing the software requirements document (SRD) and other related documents
- Interviewing stakeholders to gather additional information
- Identifying any ambiguities or inconsistencies in the requirements
- Identifying any missing or incomplete requirements
- Identifying any potential risks or issues that may impact the testing process

Software Testing Life Cycle (STLC)

2. Test Planning

- Test Planning is the most efficient phase of the software testing life cycle where all testing plans are defined.
- In this phase manager of the testing, team calculates the estimated effort and cost for the testing work. This phase gets started once the requirement-gathering phase is completed.

Test Planning stage include:

- Identifying the testing objectives and scope
- Developing a test strategy: selecting the testing methods and techniques that will be used
- Identifying the testing environment and resources needed
- Identifying the test cases that will be executed and the test data that will be used
- Estimating the time and cost required for testing
- Identifying the test deliverables and milestones
- Assigning roles and responsibilities to the testing team
- Reviewing and approving the test plan

Software Testing Life Cycle (STLC)

3. Test Case Development

- The Test Case Development phase gets started once the test planning phase is completed.
- In this phase testing team notes down the detailed test cases. The testing team also prepares the required test data for the testing.
- When the test cases are prepared then they are reviewed by the quality assurance team.

Test Case Development stage include:

- Identifying the test cases that will be developed
- Writing test cases that are clear, concise, and easy to understand
- Creating test data and test scenarios that will be used in the test cases
- Identifying the expected results for each test case
- Reviewing and validating the test cases
- Updating the requirement traceability matrix (RTM) to map requirements to test cases

Software Testing Life Cycle (STLC)

4. Test Environment Setup

- Test Environment Setup is an important part of the STLC. Basically, the test environment decides the conditions on which software is tested. This is independent activity and can be started along with test case development.
- In this process, the testing team is not involved. either the developer or the customer creates the testing environment.

5. Test Execution

- In Test Execution, after the test case development and test environment setup test execution phase gets started. In this phase testing team starts executing test cases based on prepared test cases in the earlier step.
- The activities that take place during the test execution stage of the Software Testing Life Cycle (STLC) include:
- **Test execution:** The test cases and scripts created in the test design stage are run against the software application to identify any defects or issues.
- **Defect logging:** Any defects or issues that are found during test execution are logged in a defect tracking system, along with details such as the severity, priority, and description of the issue.
- **Test data preparation:** Test data is prepared and loaded into the system for test execution
- **Test environment setup:** The necessary hardware, software, and network configurations are set up for test execution

Software Testing Life Cycle (STLC)

- **Test execution:** The test cases and scripts are run, and the results are collected and analyzed.
- **Test result analysis:** The results of the test execution are analyzed to determine the software's performance and identify any defects or issues.
- **Defect retesting:** Any defects that are identified during test execution are retested to ensure that they have been fixed correctly.
- **Test Reporting:** Test results are documented and reported to the relevant stakeholders.

Software Testing Life Cycle (STLC)

6. Test Closure

- **Test Closure** is the final stage of the Software Testing Life Cycle (STLC) where all testing-related activities are completed and documented.
- The main objective of the test closure stage is to ensure that all testing-related activities have been completed and that the software is ready for release.
- At the end of the test closure stage, the testing team should have a clear understanding of the software's quality and reliability, and any defects or issues that were identified during testing should have been resolved.
- The test closure stage also includes documenting the testing process and any lessons learned so that they can be used to improve future testing processes
- Test closure is the final stage of the Software Testing Life Cycle (STLC) where all testing-related activities are completed and documented.

Test Closure stage include:

- **Test summary report:** A report is created that summarizes the overall testing process, including the number of test cases executed, the number of defects found, and the overall pass/fail rate.
- **Defect tracking:** All defects that were identified during testing are tracked and managed until they are resolved.
- **Test environment clean-up:** The test environment is cleaned up, and all test data and test artifacts are archived.
- **Test closure report:** A report is created that documents all the testing-related activities that took place, including the testing objectives, scope, schedule, and resources used.
- **Knowledge transfer:** Knowledge about the software and testing process is shared with the rest of the team and any stakeholders who may need to maintain or support the software in the future.
- **Feedback and improvements:** Feedback from the testing process is collected and used to improve future testing processes

Test Case Preparation

- **Introduction to Test Case Preparation**
- **Definition:**
  A test case is a set of conditions or inputs used to verify whether a software system works as expected.
- **Purpose:**
  - Validate functionality
  - Detect defects early
  - Ensure software quality
- **Key Components:**
  - Test case ID
  - Test description
  - Preconditions
  - Test steps
  - Expected result

Test Case Preparation

1. **Understand Requirements** – Study SRS, user stories, and acceptance criteria.
2. **Identify Test Scenarios** – Break down requirements into testable conditions.
3. **Design Test Cases** – Write clear, simple, and reusable test cases.
4. **Define Test Data** – Prepare input values, boundary values, invalid data.
5. **Review & Finalize** – Peer review test cases to ensure coverage & accuracy.

Good Test Case Writing Practices

- Keep test cases **clear and simple**
  Use **consistent format & naming convention**
  Cover **positive, negative, boundary, and edge cases**
  Include **expected results** for comparison
  Ensure **traceability** with requirements

Example Test Case (Login Feature)

Field	Details
Test Case ID	TC001
Test Description	Verify login with valid username/password
Preconditions	User must be registered
Test Steps	1. Open login page 2. Enter valid credentials 3. Click login
Expected Result	User is redirected to dashboard

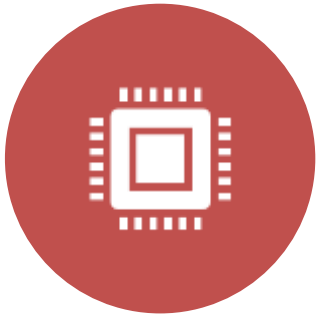
Thank You

- Rahul.kumar@bennett.edu.in

Manual and Automation Testing

Concepts, Differences & Examples

Introduction to Software Testing



SOFTWARE TESTING IS THE PROCESS OF EVALUATING A SYSTEM TO DETECT DIFFERENCES BETWEEN GIVEN INPUT AND EXPECTED OUTPUT.

ENSURES SOFTWARE QUALITY, RELIABILITY, AND USER SATISFACTION.

HELPS IN IDENTIFYING DEFECTS EARLY AND REDUCING DEVELOPMENT COST.

Manual Testing

- Executed by human testers without automation tools.

- Suitable for exploratory, usability, and ad-hoc testing.

- Time-consuming and error-prone but flexible.

- Best for small projects or short-term testing needs.

Automation Testing

- Testing executed with the help of automation tools.
- Suitable for regression, load, and performance testing.
- Provides speed, accuracy, and reusability.
- Requires initial investment in tools and scripting.

Key Differences: Manual vs Automation

- Manual: Time-consuming | Automation: Fast execution
- Manual: No tools needed | Automation: Requires tools/scripts
- Manual: Error-prone | Automation: High accuracy
- Manual: Low initial cost | Automation: High initial setup cost
- Manual: Best for short-term | Automation: Best for long-term projects

Advantages & Disadvantages

Manual Testing

✓ Flexible and good for exploratory testing

✗ Slow and error-prone

Automation Testing

✓ Fast, reusable, accurate

✗ Expensive setup, not ideal for usability testing

Real-Time Examples

Manual Testing Examples:

- Usability testing of a website
- Exploratory testing of a mobile app

Automation Testing Examples:

- Regression testing with Selenium
- Performance testing with JMeter
- Continuous integration testing with Jenkins

Popular Automation Tools

Selenium

JUnit

TestNG

QTP (UFT)

JMeter

Appium

When to Choose What?

Use Manual Testing when:

- Testing small projects
- Performing usability or exploratory testing

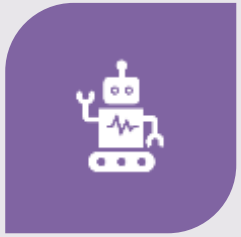
Use Automation Testing when:

- Performing regression, load, or performance testing
- Working on long-term or large-scale projects

Conclusion

MANUAL AND AUTOMATION TESTING COMPLEMENT EACH OTHER.

MANUAL IS BEST FOR EXPLORATORY AND USABILITY TESTING.



AUTOMATION IS BEST FOR REGRESSION AND LARGE-SCALE PROJECTS.

A BALANCED APPROACH ENSURES HIGH-QUALITY SOFTWARE.

Scenario Based Questions on Principles of Software Testing

Principle: Exhaustive Testing is impossible

Q) A banking application has: 5 input fields (each can take 10 possible values) 3 radio buttons (each with 2 possible values).

i) Calculate the total number of possible input combinations if exhaustive testing is attempted.

ii) If a tester can execute 200 test cases per day, estimate how many days exhaustive testing would take.

iii) Relating to the principle, explain why exhaustive testing is not practical here and suggest a better approach.

Defect Clustering

Q) In a large ERP System, testing reports following defect distribution across modules

Payroll	15	120	
HR	10	30	
Finance		20	180
Inventory	25	40	
Sales	30	50	

i) Calculate the defect density for each module

ii) Identify which module shows defect clustering.

iii) Explain how this principle helps in prioritizing testing effort.

Testing shows presence of defects

Q) A software project has an estimated 12,000 lines of code (LOC). Based on industry data, the average defect density for this type of software is 15 defects per 1,000 LOC. The testing team has found and fixed 160 defects. What is the estimated number of remaining defects? If the testing team's efficiency in finding defects is 80%, what is the total number of defects that were originally in the code?

Early Testing

Q) A software project has a total estimated cost of $500,000. The cost to fix a defect at the requirements phase is $100. The cost to fix a defect at the testing phase is $1,000. The cost to fix a defect in production is $10,000. A review of the requirements document found 5 critical defects. A test plan review found 3 critical defects. If these defects were not found until the production phase, how much additional cost would the company incur?

Efficient & Effective Testing

Q) A software development team is preparing for the final quality assurance phase of a new mobile game. The testing team has planned a total of **300 test cases**. The project lead wants to analyze the team's performance by looking at two different testing sessions.

- In **Session A**, the team executed **75 test cases** in **3 hours** and discovered **15 defects**.
- In **Session B**, the team executed **225 test cases** in **10 hours** and discovered **60 defects**.
- It is estimated that the game has a total of **80 defects** before this testing phase.

Tasks:

1. Calculate the **efficiency** of testing in both Session A and Session B (test cases executed per hour).
2. Calculate the **effectiveness** of testing in both Session A and Session B (percentage of defects detected).
3. Compare the results of Session A and Session B in terms of both efficiency and effectiveness.

Consider the following program:

```
void compareNos(int a, int b) {
if (a == b) {
                printf("Equal\n");
}
else if (a > b) {
    if ((a – b) > 10) {
                printf("a much greater than b\n");
} else {
                printf("a slightly greater than b\n");
  }
} else {
   if ((b – a) > 10) {
                printf("b much greater than a\n");
   } else {
                printf("b slightly greater than a\n");
   }
}
printf("End\n");
}
```

Evaluate the following:

1. Calculate the number of **statements**, **decisions**, and **independent paths**.
2. Provide a minimum set of test cases to achieve **100% statement coverage**.
3. Provide a minimum set of test cases to achieve **100% decision coverage**.
4. Provide a minimum set of test cases to achieve **100% path coverage**.

Consider the following scenario:

Updated Scenario (Online Shopping App Testing)

- Test cases designed = **120**
- Test cases executed = **100**
- Test cases passed = **85**
- Test cases failed = **15**
- Defects reported = **20**
- Defects fixed = **15**
- Defects reopened = **5**
- Defects rejected = **2**
- Effort spent = **50 hours**
- Application size = **10 KLOC**

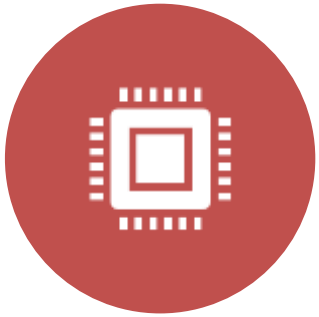
Tasks

1. Test Case Execution Rate
2. Test Case Pass Percentage
3. Defect Density
4. Defect Fix Rate
5. Defect Reopen Rate
6. Productivity (Test cases executed/hour)
7. **Defect Removal Efficiency (DRE)**
8. Interpret the results

Manual and Automation Testing

Concepts, Differences & Examples

Introduction to Software Testing



SOFTWARE TESTING IS THE PROCESS OF EVALUATING A SYSTEM TO DETECT DIFFERENCES BETWEEN GIVEN INPUT AND EXPECTED OUTPUT.

ENSURES SOFTWARE QUALITY, RELIABILITY, AND USER SATISFACTION.

HELPS IN IDENTIFYING DEFECTS EARLY AND REDUCING DEVELOPMENT COST.

Manual Testing

• Executed by human testers without automation tools.

• Suitable for exploratory, usability, and ad-hoc testing.

• Time-consuming and error-prone but flexible.

• Best for small projects or short-term testing needs.

Automation Testing

- Testing executed with the help of automation tools.
- Suitable for regression, load, and performance testing.
- Provides speed, accuracy, and reusability.
- Requires initial investment in tools and scripting.

Key Differences: Manual vs Automation

- Manual: Time-consuming | Automation: Fast execution
- Manual: No tools needed | Automation: Requires tools/scripts
- Manual: Error-prone | Automation: High accuracy
- Manual: Low initial cost | Automation: High initial setup cost
- Manual: Best for short-term | Automation: Best for long-term projects

Advantages & Disadvantages

Manual Testing

✓ Flexible and good for exploratory testing

✗ Slow and error-prone

Automation Testing

✓ Fast, reusable, accurate

✗ Expensive setup, not ideal for usability testing

Real-Time Examples

Manual Testing Examples:

- Usability testing of a website
- Exploratory testing of a mobile app

Automation Testing Examples:

- Regression testing with Selenium
- Performance testing with JMeter
- Continuous integration testing with Jenkins

Popular Automation Tools

Selenium

JUnit

TestNG

QTP (UFT)

JMeter

Appium

When to Choose What?

Use Manual Testing when:

- Testing small projects
- Performing usability or exploratory testing

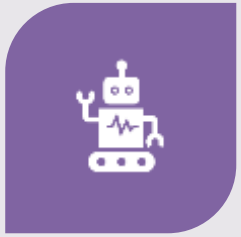
Use Automation Testing when:

- Performing regression, load, or performance testing
- Working on long-term or large-scale projects

Conclusion

MANUAL AND AUTOMATION TESTING COMPLEMENT EACH OTHER.

MANUAL IS BEST FOR EXPLORATORY AND USABILITY TESTING.

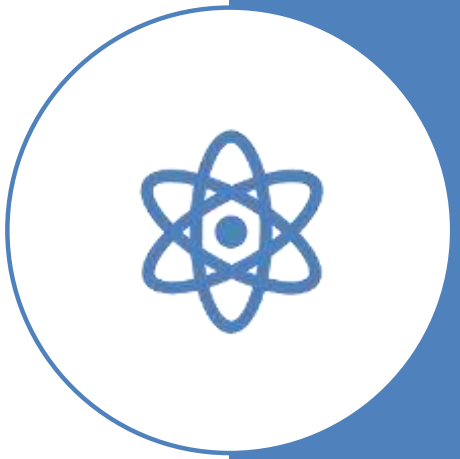


AUTOMATION IS BEST FOR REGRESSION AND LARGE-SCALE PROJECTS.

A BALANCED APPROACH ENSURES HIGH-QUALITY SOFTWARE.

Testing Techniques: An Overview

Software Testing Techniques ensure the system meets business and technical requirements. Two primary techniques are:



- White Box Testing
- Black Box Testing

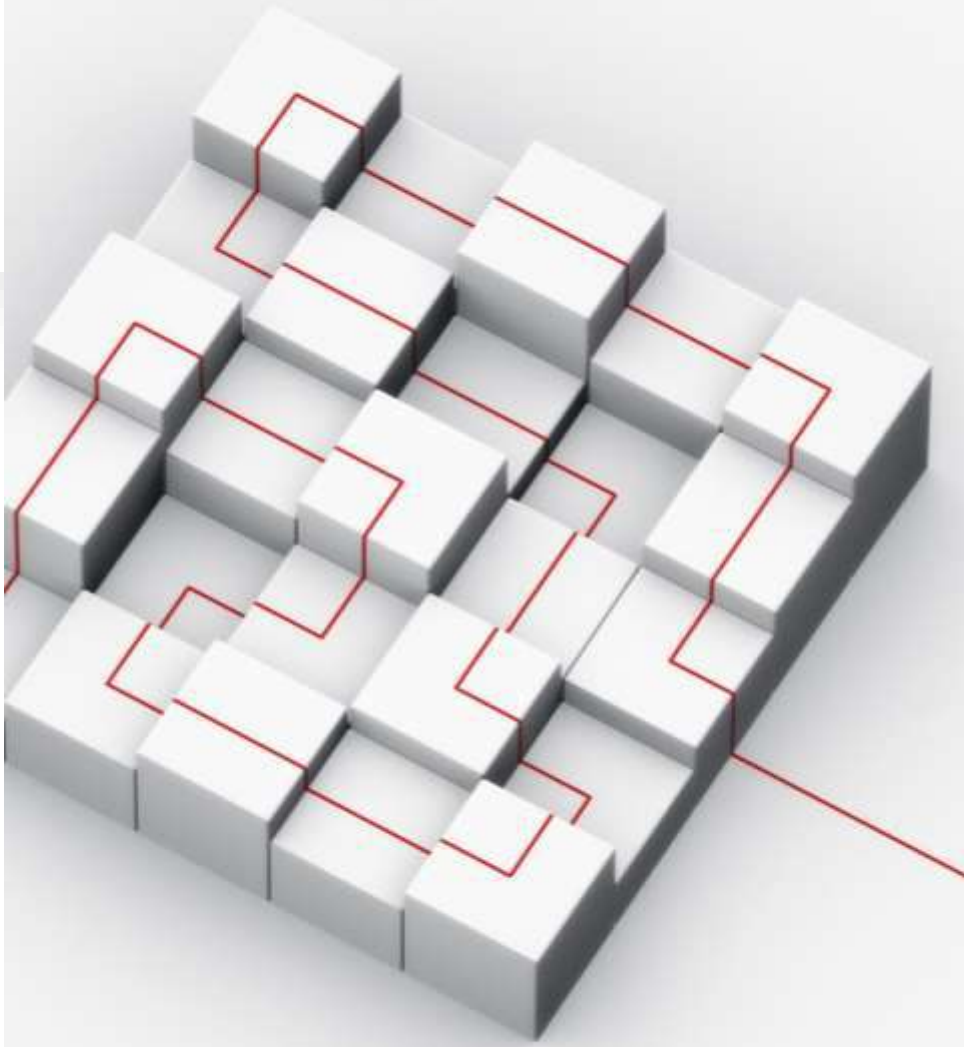
Both are essential for comprehensive software validation and defect prevention.

White Box Testing - **Introduction**

- Also known as **Clear Box**, **Structural**, or **Glass Box Testing**.
- **Focus**: Internal logic, structure, and code implementation.
- **Performed by**: Developers or test engineers with programming knowledge.

Objectives of White Box Testing

- Ensure all independent code paths are executed. (**Path Coverage**)
- Validate loops, conditions, and data flow.
- Verify internal operations conform to design specifications.



White Box Testing Techniques

Common techniques include:

1. Statement Coverage
2. Branch Coverage
3. Condition Coverage
4. Path Coverage
5. Loop Testing

Example: Ensuring each 'if-else' path in login logic executes correctly.

Loop Testing

Test for:

- Zero iteration
- One iteration
- Typical iteration
- Maximum iteration
- One more than maximum iteration (for failure)

Path Coverage Example

If-Else structure with nested conditions.

Example: If (x>0) then If (y>0)...

- Total Paths = Product of independent decisions.
- Used to ensure all possible logical paths are tested.

Numerical Example: Path Coverage

Program has 3 independent conditions.

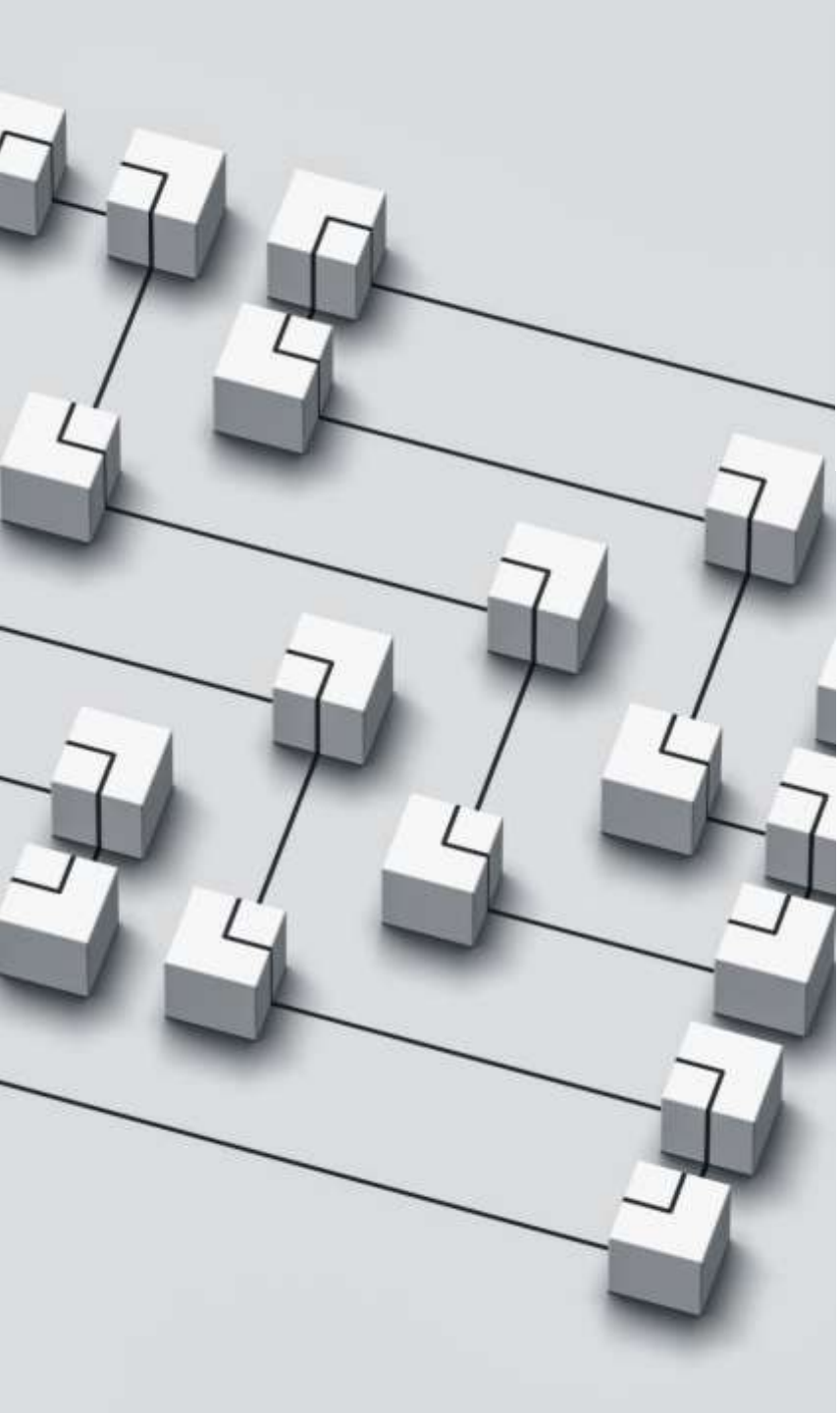
Total independent paths = 2^3 = 8

Hence, 8 test cases needed for full path coverage.

Code Example

```
if A > 10:
    if B > 5:
        print("Path 1")
    else:
        print("Path 2")
else:
    if B > 0:
        print("Path 3")
    else:
        print("Path 4")
```

Q) What are the independent decisions? How many Test Cases are required?



Industrial Example: White Box Testing

Example: Testing payment gateway code at PayPal.

Developers ensure:

- All encryption and API calls are triggered.
- Exception handling logic works correctly.
- Loops managing retries for failed transactions terminate as expected.

Benefits & Limitations - White Box Testing

Benefits:

- Early defect detection. (Early Defect Detection)
- Optimized code coverage.
- Logical errors can be easily detected. (Faster Bug Recovery)

Limitations:

- Time-consuming.
- Requires code-level expertise.
- Not suitable for large-scale testing alone.

Black Box Testing - Introduction

Also known as **Behavioral** or **Functional** Testing.

Focus: External behavior of the software.

Performed by: Testers or QA engineers without code knowledge.

Objectives of Black Box Testing

- Validate software functionality against requirements.
- Identify incorrect or missing functions.
- Ensure proper user interface behavior.

Black Box Testing Techniques

Common techniques include:

1. Equivalence Partitioning
2. Boundary Value Analysis
3. Decision Table Testing
4. State Transition Testing
5. Use Case Testing

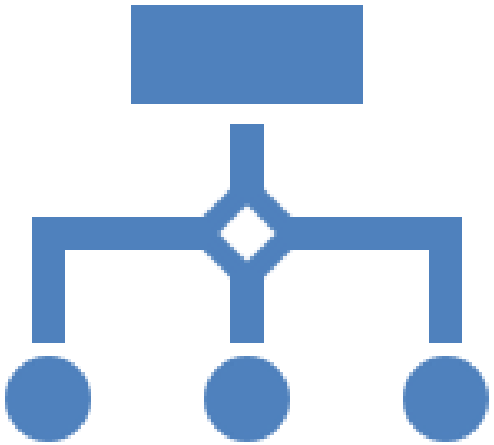
Example: Testing login form for valid and invalid credentials.

Industrial Example: Black Box Testing

Example: E-commerce checkout system at Amazon.

Testers validate:

- User can add/remove items correctly.
- Discounts and coupons apply properly.
- Payment errors display user-friendly messages.
- Backend failures don’t expose sensitive data.



Equivalence Partitioning Example

Example: Age input should be between 18 and 60.

- Valid partition: 18–60
- Invalid partitions: <18, >60

Test cases:

- 17 (invalid)
- 30 (valid)
- 65 (invalid)

Boundary Value Analysis (BVA)

Example: Marks should be between 0 and 100.

- Boundaries: 0, 1, 99, 100
- Test cases: -1, 0, 1, 99, 100, 101

→ Total 6 test cases for BVA

Question

A software system accepts marks between **0 and 40 (inclusive)** as valid input.
If the marks are outside this range, an **“Invalid Marks”** message is displayed.

Design **test cases** using **Equivalence Partitioning** and **Boundary Value Analysis**.

Decision Table Testing

Used when multiple input conditions produce different actions.

Example: ATM withdrawal

- **Conditions**: Card Valid? | Sufficient Balance?
- **Actions**: Allow / Deny

→ 4 possible rules (2x2 combinations).

State Transition Testing

Tests behavior when software changes states.

Example: Login system

- **State 1**: Logged Out → Enter valid credentials → State 2: Logged In
- **State 2**: Logged In → Click Logout → State 1: Logged Out

Comparison: White Box vs Black Box

White Box Testing: Structural testing with code.

- **Focus**: Internal logic
- **Tester**: Developer
- **Basis**: Code structure

Black Box Testing: Functional testing without code.

- **Focus**: System behaviour
- **Tester**: QA/Tester
- **Basis**: Requirements and specifications

Summary & Best Practices

1. Combine both techniques for maximum coverage.
2. Use White Box Testing in unit/integration testing phases.
3. Use Black Box Testing in system/user acceptance phases.
4. Automate repetitive test cases using tools (e.g., Selenium, JUnit).

A Deep-Dive into Testing Types: Unit, Integration, UAT, Alpha & Beta, Smoke, Sanity, Regression

With Technical Definitions, Feynman Explanations, Real-World Analogies & Examples

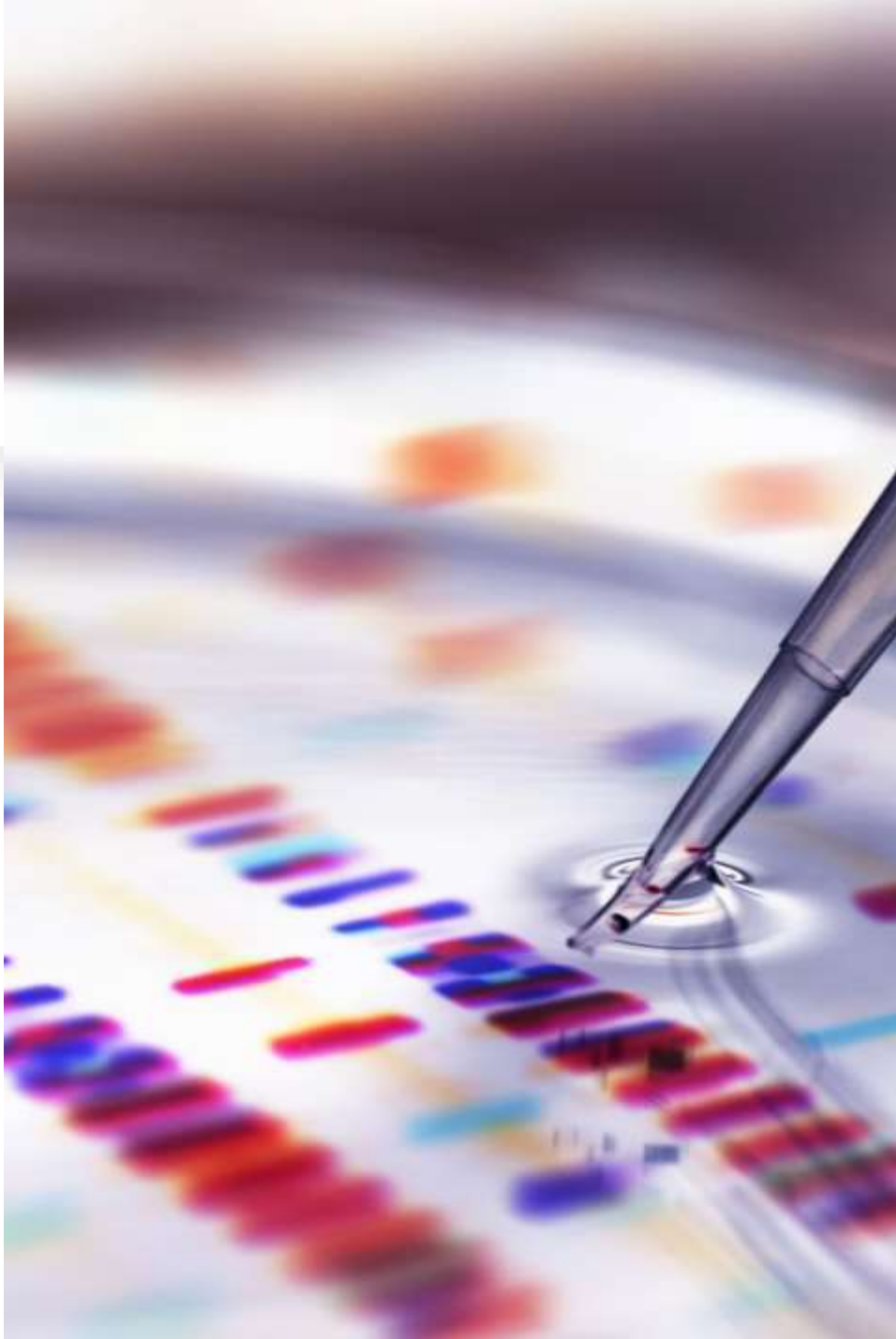
Prepared by: Aditya Upadhyay

Assistant Professor, SCSET, Bennett University

Unit Testing – Technical Definition

Unit Testing is a level of software testing where individual components or functions are tested in isolation to ensure they perform as expected.

Usually performed by developers using frameworks like JUnit or pytest.



Unit Testing – Feynman Explanation

- Imagine your software as a Lego castle. Before building the whole thing, you check if each Lego piece fits perfectly.
- Unit Testing simply asks: 'Does this small piece of code do what it’s supposed to?'

Unit Testing – Real-World Analogy

In a car factory, engineers test the engine, brakes, and headlights separately before assembling the car.

Each part must work independently before integration.

Unit Testing – Software Example

Testing a function **'calculate_discount(price, coupon)'** to ensure it applies the correct discount before integrating into the checkout system.

Unit Testing – Key Takeaway

‘TEST SMALL, FAIL FAST, FIX EARLY.’

UNIT TESTS MAKE DEBUGGING CHEAPER, QUICKER, AND MORE RELIABLE.

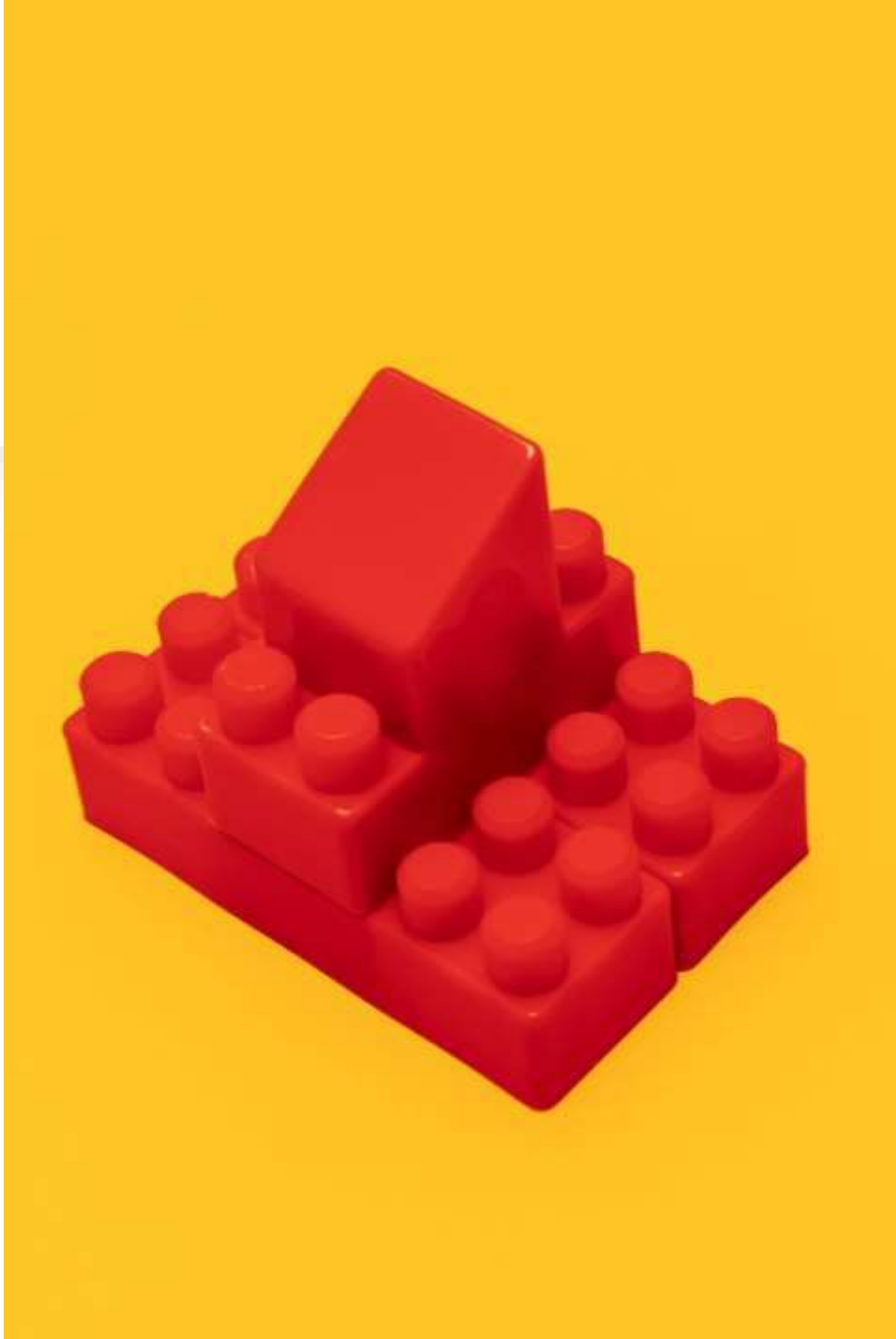
Integration Testing – Technical Definition

Integration Testing verifies the interaction and data flow between connected modules or services after integration.

Goal: detect interface and communication errors.

Integration Testing – Feynman Explanation

You’ve tested all Lego blocks; now you’re checking if they fit together correctly. Even perfect pieces can fail when combined.



Integration Testing – Real-World Analogy

In a restaurant, the chef, waiter, and cashier all work fine alone. But if the waiter forgets to tell the chef your order, the customer remains hungry — that’s an integration bug.

Integration Testing – Software Example

Testing if the Login module correctly passes user credentials to the Dashboard and retrieves correct user data.

Integration Testing – Key Takeaway

Ensures that different parts of the system ‘talk’ to each other correctly and data flows seamlessly.

User Acceptance Testing (UAT) – Technical Definition

UAT is the final phase of testing performed by clients or end users to confirm that the system meets business requirements and is ready for production.

UAT – Feynman Explanation

You’ve built a car — now give it to the customer for a test drive to confirm it feels right. UAT ensures the product solves the user’s real problem.

UAT – Real-World Analogy

A restaurant invites guests for a soft opening before the grand launch to check if dishes meet expectations.

UAT – Software Example

Bank staff test the new loan application workflow to confirm it matches their business process.

UAT – Key Takeaway

User Acceptance Testing validates that the system is ready for real-world use and meets user needs.

Alpha and Beta Testing – Technical Definition

Alpha and Beta Testing are pre-release testing phases:

- **Alpha**: Internal testing in a controlled environment.
- **Beta**: External testing by real users before official launch.

Alpha and Beta – Feynman Explanation

Alpha: You taste the soup yourself before serving.

Beta: You serve it to guests and collect feedback before putting it on the menu.

Alpha and Beta – Real-World Analogy

A game developer plays their game internally (Alpha), then releases a demo version to selected players (Beta) for real feedback.

Alpha and Beta – Software Example

- **Alpha**: QA tests mobile banking UI in house.
- **Beta**: A limited group of users tests the app on their devices and reports usability issues.

Alpha and Beta – Key Takeaway

Alpha finds internal defects; Beta finds real-world usability issues and helps refine product quality.

Smoke Testing – Technical Definition

Smoke Testing is a preliminary test that checks the stability of the software build and ensures that critical features work before detailed testing begins.

Smoke Testing – Feynman Explanation

You turn on a new machine — if it doesn’t emit smoke, it’s safe to test further. Smoke tests confirm the system is alive.

Smoke Testing – Real-World Analogy

Before cooking a full meal, you turn on the stove to make sure the flame lights up.

Smoke Testing – Software Example

After new deployment, QA verifies that login, product search, and checkout modules work before starting functional tests.

Smoke Testing – Key Takeaway

Smoke testing saves hours by identifying unstable builds early in the testing cycle.

Sanity Testing – Technical Definition

Sanity Testing is focused testing after receiving minor code fixes or patches to verify that specific changes work and related features remain unaffected.

Sanity Testing – Feynman Explanation

You repaired your car’s headlights — now you check just the lights, not the entire car again.

Sanity Testing – Real-World Analogy

After fixing a door hinge, you open and close it to verify it works smoothly without rechecking the whole house.

Sanity Testing – Software Example

After fixing a 'coupon not applying' bug, QA verifies that the discount calculation and checkout process work fine.

Sanity Testing – Key Takeaway

A sanity check ensures that bug fixes are logical and effective, avoiding unnecessary full testing.

Regression Testing – Technical Definition

Regression Testing ensures that new changes, updates, or bug fixes have not broken existing functionalities in the system.

Regression Testing – Feynman Explanation

You fixed one loose wire in your TV — now you check volume, color, and brightness again. One change can affect many areas.

Regression Testing – Real-World Analogy

When a restaurant adds a new menu item, the chef still tastes the old dishes to ensure they remain perfect.

Regression Testing – Software Example

After adding a wallet payment feature, QA re-runs all critical flows like login, cart, and checkout to confirm stability.

Regression Testing – Key Takeaway

Regression Testing prevents the ‘new code broke old code’ problem and ensures consistent system reliability.

Case Study – Online Banking Application

Example of applying all testing types:

- • Unit: Test interest and balance calculations.
- • Integration: Verify login → transaction → DB flow.
- • UAT: Bank staff simulate real user transactions.
- • Alpha/Beta: Internal QA & select customers test the app.
- • Smoke/Sanity: Validate build stability and bug fixes.
- • Regression: Ensure new QR payments don't affect transfers.

Summary – All Testing Types

- • Unit → Test smallest code pieces.
- • Integration → Test module connections.
- • UAT → User confirmation.
- • Alpha/Beta → Internal + real-world pre-release checks.
- • Smoke/Sanity → Build stability & fix verification.
- • Regression → Continuous quality maintenance.

Together, they form a complete software quality cycle.