www.jerusalemengg.ac.in
JERUSALEM COLLEGE OF ENGINEERING
AN AUTONOMOUS INSTITUTION
Approved by AICTE, New Delhi, Affiliated to Anna University Chennai.
Accredited by NBA, New Delhi and Accredited by NAAC with "A" Grade.
Velachery Main Road, Narayanapuram, Pallikaranai, Chennai, Tamil Nadu - 600100.
DEPARTMENT OF INFORMATION TECHNOLOGY

AY 2024-2025

UNIT I FOUNDATIONS OF SOFTWARE TESTING 9

Why do we test Software?, Software Testing Life Cycle, V-model of Software Testing, Software Testing Principles ,The Tester's Role in a Software Development Organization – Reliability versus Safety, Failures, Errors and Faults , Origins of Defects – Cost of defects – Defect Classes – The Defect Repository and Test Design

1:1Why do we need testing for software?

Software testing is imperative, as it identifies any issues and defects with the written code so they can be fixed before the software product is delivered. Improves product quality. When it comes to customer appeal, delivering a quality product is an important metric to consider.

Software testing

Software testing is the process of evaluating and verifying that a software product or application does what it is supposed to do. The benefits of testing include preventing bugs, reducing development costs and improving performance.

Main reason of testing:

First, testing is about verifying that what was specified is what was delivered: it verifies that the product (system) meets the functional, performance, design, and implementation requirements identified in the procurement specifications.

4 benefits of software testing:

Benefits of Software Testing

- Customer Satisfaction. ...
- Cost Effective. ...
- Quality Product. ...
- Low Failure. ...
- Bug-Free Application. ...
- Security. ...
- Easy Recovery. ...

1:2Software Testing Life Cycle (STLC)

- ❖ The Software Testing Life Cycle (STLC) is a systematic approach to testing a software application to ensure that it meets the requirements and is free of defects. It is a process that follows a series of steps or phases, and each phase has specific objectives and deliverables. The STLC is used to ensure that the software is of high quality, reliable, and meets the needs of the end-users.
- ❖ The main goal of the STLC is to identify and document any defects or issues in the software application as early as possible in the development process. This allows for issues to be addressed and resolved before the software is released to the public.
- ❖ The stages of the STLC include Test Planning, Test Analysis, Test Design, Test Environment Setup, Test Execution, Test Closure, and Defect Retesting. Each of these stages includes specific activities and deliverables that help to ensure that the software is thoroughly tested and meets the requirements of the end users.
- ❖ Overall, the STLC is an important process that helps to ensure the quality of software applications and provides a systematic approach to testing. It allows organizations to release high-quality software that meets the needs of their customers, ultimately leading to customer satisfaction and business success.

Characteristics of STLC

- STLC is a fundamental part of the Software Development Life Cycle (SDLC) but STLC consists of only the testing phases.
- STLC starts as soon as requirements are defined or software requirement document is shared by stakeholders.
- STLC yields a step-by-step process to ensure quality software.

In the initial stages of STLC, while the software product or the application is being developed, the testing team analyzes and defines the scope of testing, entry and exit criteria, and also test cases. It helps to reduce the test cycle time and also enhances product quality. As soon as the development phase is over, the testing team is ready with test cases and starts the execution. This helps in finding bugs in the early phase.

Phases of 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.

The activities that take place during the 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

Creating a requirement traceability matrix (RTM) to map requirements to test cases
At the end of this stage, the testing team should have a clear understanding of the software requirements and should have identified any potential issues that may impact the testing

process. This will help to ensure that the testing process is focused on the most important areas of the software and that the testing team is able to deliver high-quality results.

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.

The activities that take place during the 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

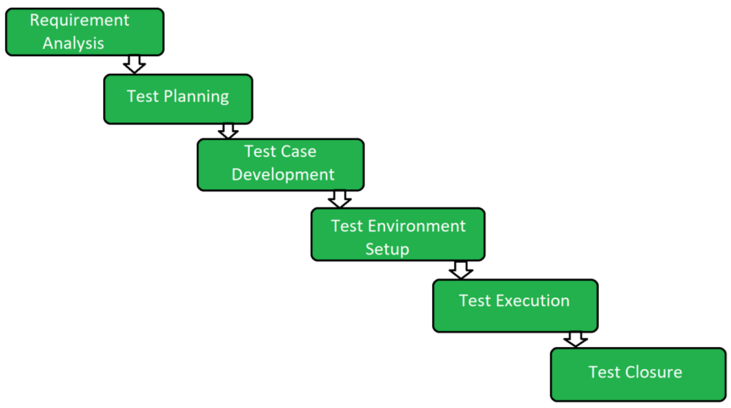
At the end of this stage, the testing team should have a detailed plan for the testing activities that will be performed, and a clear understanding of the testing objectives, scope, and deliverables. This will help to ensure that the testing process is well-organized and that the testing team is able to deliver high-quality results.

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.

The activities that take place during the 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

At the end of this stage, the testing team should have a set of comprehensive and accurate test cases that provide adequate coverage of the software or application. This will help to ensure that the testing process is thorough and that any potential issues are identified and addressed before the software is released



4. Test Environment Setup: Test environment setup is a vital 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: 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
- **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.

It is important to note that test execution is an iterative process and may need to be repeated multiple times until all identified defects are fixed and the software is deemed fit for release.

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. The main activities that take place during the 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

Characteristics of STLC

- STLC is a fundamental part of the Software Development Life Cycle (SDLC) but STLC consists of only the testing phases.
- STLC starts as soon as requirements are defined or software requirement document is shared by stakeholders.
- STLC yields a step-by-step process to ensure quality software.

In the initial stages of STLC, while the software product or the application is being developed, the testing team analyzes and defines the scope of testing, entry and exit criteria, and also test cases. It helps to reduce the test cycle time and also enhances product quality. As soon as the development phase is over, the testing team is ready with test cases and starts the execution. This helps in finding bugs in the early phase.

1:3V-model of Software Testing

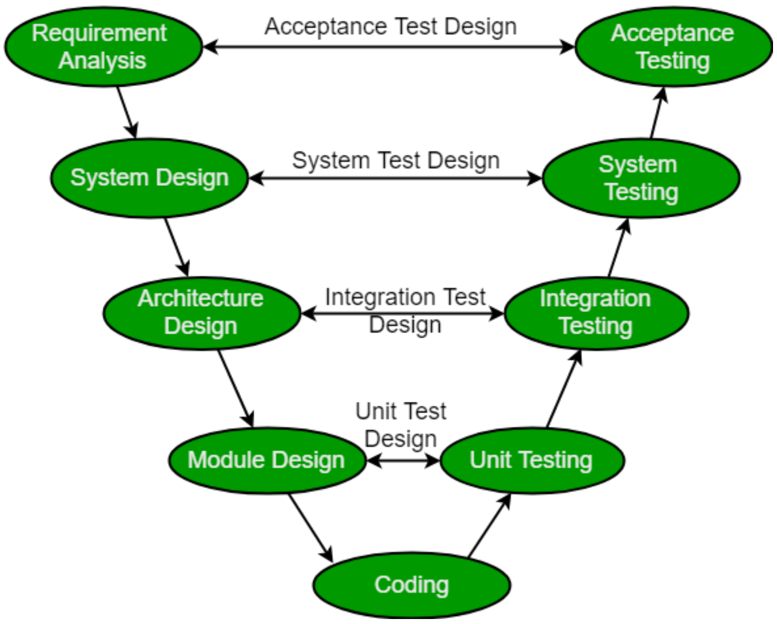
The V-model is a type of SDLC model where process executes in a sequential manner in V-shape. It is also known as Verification and Validation model. It is based on the association of a testing phase for each corresponding development stage. Development of each step directly associated with the testing phase. The next phase starts only after completion of the previous phase i.e. for each development activity, there is a testing activity corresponding to it.

The V-Model is a software development life cycle (SDLC) model that provides a systematic and visual representation of the software development process. It is based on the idea of a “V” shape, with the two legs of the “V” representing the progression of the software development process from requirements gathering and analysis to design, implementation, testing, and maintenance.

The V-Model is a linear and sequential model that consists of the following phases:

1. Requirements Gathering and Analysis: The first phase of the V-Model is the requirements gathering and analysis phase, where the customer’s requirements for the software are gathered and analyzed to determine the scope of the project.
2. Design: In the design phase, the software architecture and design are developed, including the high-level design and detailed design.
3. Implementation: In the implementation phase, the software is actually built based on the design.
4. Testing: In the testing phase, the software is tested to ensure that it meets the customer’s requirements and is of high quality.
5. Deployment: In the deployment phase, the software is deployed and put into use.
6. Maintenance: In the maintenance phase, the software is maintained to ensure that it continues to meet the customer’s needs and expectations.
7. The V-Model is often used in safety-critical systems, such as aerospace and defense systems, because of its emphasis on thorough testing and its ability to clearly define the

steps involved in the software development process.



Verification: It involves static analysis technique (review) done without executing code. It is the process of evaluation of the product development phase to find whether specified requirements meet.
Validation: It involves dynamic analysis technique (functional, non-functional), testing done by executing code. Validation is the process to evaluate the software after the completion of the development phase to determine whether software meets the customer expectations and requirements.
So V-Model contains Verification phases on one side of the Validation phases on the other side. Verification and Validation phases are joined by coding phase in V-shape. Thus it is called V-Model.

Design Phase:

- **Requirement Analysis:** This phase contains detailed communication with the customer to understand their requirements and expectations. This stage is known as Requirement Gathering.
- **System Design:** This phase contains the system design and the complete hardware and communication setup for developing product.
- **Architectural Design:** System design is broken down further into modules taking up different functionalities. The data transfer and communication between the internal modules and with the outside world (other systems) is clearly understood.
- **Module Design:** In this phase the system breaks down into small modules. The detailed design of modules is specified, also known as Low-Level Design (LLD).

Testing Phases:

- **Unit Testing:** Unit Test Plans are developed during module design phase. These Unit Test Plans are executed to eliminate bugs at code or unit level.
- **Integration testing:** After completion of unit testing Integration testing is performed. In integration testing, the modules are integrated and the system is tested. Integration testing is performed on the Architecture design phase. This test verifies the communication of modules among themselves.

- **System Testing:** System testing test the complete application with its functionality, inter dependency, and communication.It tests the functional and non-functional requirements of the developed application.
- **User Acceptance Testing (UAT):** UAT is performed in a user environment that resembles the production environment. UAT verifies that the delivered system meets user's requirement and system is ready for use in real world.

Industrial Challenge: As the industry has evolved, the technologies have become more complex, increasingly faster, and forever changing, however, there remains a set of basic principles and concepts that are as applicable today as when IT was in its infancy.

- Accurately define and refine user requirements.
- Design and build an application according to the authorized user requirements.
- Validate that the application they had built adhered to the authorized business requirements.

Principles of V-Model:

- **Large to Small:** In V-Model, testing is done in a hierarchical perspective, For example, requirements identified by the project team, create High-Level Design, and Detailed Design phases of the project. As each of these phases is completed the requirements, they are defining become more and more refined and detailed.
- **Data/Process Integrity:** This principle states that the successful design of any project requires the incorporation and cohesion of both data and processes. Process elements must be identified at each and every requirements.
- **Scalability:** This principle states that the V-Model concept has the flexibility to accommodate any IT project irrespective of its size, complexity or duration.
- **Cross Referencing:** Direct correlation between requirements and corresponding testing activity is known as cross-referencing.

Tangible Documentation: This principle states that every project needs to create a document. This documentation is required and applied by both the project development team and the support team. Documentation is used to maintaining the application once it is available in a production environment.

Why preferred?

- It is easy to manage due to the rigidity of the model. Each phase of V-Model has specific deliverables and a review process.
- Proactive defect tracking – that is defects are found at early stage.

When to use?

- Where requirements are clearly defined and fixed.
- The V-Model is used when ample technical resources are available with technical expertise.
- Small to medium-sized projects with set and clearly specified needs are recommended to use the V-shaped model.
- Since it is challenging to keep stable needs in large projects, the project should be small.

Advantages:

- This is a highly disciplined model and Phases are completed one at a time.
- V-Model is used for small projects where project requirements are clear.
- Simple and easy to understand and use.
- This model focuses on verification and validation activities early in the life cycle thereby enhancing the probability of building an error-free and good quality product.
- It enables project management to track progress accurately.

- Clear and Structured Process: The V-Model provides a clear and structured process for software development, making it easier to understand and follow.
- Emphasis on Testing: The V-Model places a strong emphasis on testing, which helps to ensure the quality and reliability of the software.
- Improved Traceability: The V-Model provides a clear link between the requirements and the final product, making it easier to trace and manage changes to the software.
- Better Communication: The clear structure of the V-Model helps to improve communication between the customer and the development team.

Disadvantages:

- High risk and uncertainty.
- It is not a good for complex and object-oriented projects.
- It is not suitable for projects where requirements are not clear and contains high risk of changing.
- This model does not support iteration of phases.
- It does not easily handle concurrent events.
- Inflexibility: The V-Model is a linear and sequential model, which can make it difficult to adapt to changing requirements or unexpected events.
- Time-Consuming: The V-Model can be time-consuming, as it requires a lot of documentation and testing.
- Overreliance on Documentation: The V-Model places a strong emphasis on documentation, which can lead to an overreliance on documentation at the expense of actual development work.

1:4 BASIC DEFINATIONS

Goals of Testing:

- Detect faults
- Establish confidence in software
- Evaluate properties of software

– Reliability
– Performance
– Memory Usage
– Security
– Usability

Why Test?

- Devil's Advocate:

Program testing can be used to show the presence of defects, but never their absence!"

- -Dijkstra

We can never be certain that a testing system is correct."

--Manna

•In Defence of Testing:

- Testing is the process of showing the presence of defects.
- There is no absolute notion of \correctness".
- Testing remains the most cost effective approach to building
- confidence within most software systems.

Most Common Software problems:

- Incorrect calculation
- Incorrect data edits & ineffective data edits
- Incorrect matching and merging of data
- Data searches that yields incorrect results
- Incorrect processing of data relationship
- Incorrect coding / implementation of business rules
- Inadequate software performance
- Confusing or misleading data
- Software usability by end users &
- Obsolete Software
- Inconsistent processing
- Unreliable results or performance
- Inadequate support of business needs
- Incorrect or inadequate interfaces
- with other systems
- Inadequate performance and security controls
- Incorrect file handling

What is Software Testing?

Executing software in a simulated or real environment, using inputs selected somehow.

Testing is the process of exercising or evaluating a system orsystem component by manual or automated means to verifythat it satisfies specified requirements, or to identify differencesbetween expected and actual results.

--" IEEE

The process of executing a program or system with the intentof finding errors. -- (Myers 1979)

The measurement of software quality." -- (Hetzel 1983

Errors:
An error is a mistake, misconception, or misunderstanding on the part of a software developer.

Faults (Defects):

A fault (defect) is introduced into the software as the result of an error. It is an anomaly in the software that may cause it to behave incorrectly, and not according to its specification**.**

Failures:

A failure is the inability of a software system or component to perform its required functions within specified performance requirements .

Test cases:

A test case in a practical sense is a test-related item which contains the following information:

1. A set of test inputs.These are data items received from an external source by the code under test. The external source can be hardware, software, or human.

2. Execution conditions.These are conditions required for running the test, for example, a certain state of a database, or a configuration of a hardware device.

3. Expected outputs.These are the specified results to be produced by the code under test.

Test

A test is a group of related test cases, or a group of related test cases and test procedure

Test Oracle

A test oracle is a document, or piece of software that allows testers to determine whether a test has been passed or failed.

Test Bed

A test bed is an environment that contains all the hardware and software needed to test a software component or a software system.

Software Quality
Two concise definitions for quality are found in the *IEEE Standard Glossary of Software Engineering Terminology*

1. Quality relates to the degree to which a system, system component, or process meets specified requirements.
2. Quality relates to the degree to which a system, system component, or process meets customer or user needs, or expectations.

Metric:

- A metric is a quantitative measure of the degree to which a system, system component, or process possesses a given attribute.

- A quality metric is a quantitative measurement of the degree to which an item possesses a given quality attribute

***IEEE Standards for Software Quality Metrics Methodology* and work by Schulmeyer and Grady [6–8]. Some examples of quality attributes with brief explanations are the following:**
correctness—the degree to which the system performs its intended function
reliability—the degree to which the software is expected to perform its required functions under stated conditions for a stated period of time
usability—relates to the degree of effort needed to learn, operate, prepare input, and interpret output of the software
integrity—relates to the system's ability to withstand both intentional and accidental attacks
portability—relates to the ability of the software to be transferred from one environment to another
maintainability—the effort needed to make changes in the software
interoperability—the effort needed to link or couple one system to another.

Review:

A review is a group meeting whose purpose is to evaluate a software artifact or a set of software artifacts

What is an Error?
Error is a situation that happens when the Development team or the developer fails to understand a requirement definition and hence that misunderstanding gets translated into buggy code. This situation is referred to as an Error and is mainly a term coined by the developers.

- Errors are generated due to wrong logic, syntax, or loop that can impact the end-user experience.
- It is calculated by differentiating between the expected results and the actual results.
- It raises due to several reasons like design issues, coding issues, or system specification issues and leads to issues in the application.

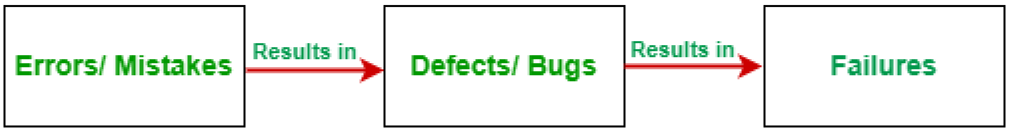
What is a Fault?
Sometimes due to certain factors such as Lack of resources or not following proper steps Fault occurs in software which means that the logic was not incorporated to handle the errors in the application. This is an undesirable situation, but it mainly happens due to invalid documented steps or a lack of data definitions.

- It is an unintended behavior by an application program.
- It causes a warning in the program.
- If a fault is left untreated it may lead to failure in the working of the deployed code.
- A minor fault in some cases may lead to high-end error.
- There are several ways to prevent faults like adopting programming techniques, development methodologies, peer review, and code analysis.

What is a Failure?
Failure is the accumulation of several defects that ultimately lead to Software failure and results in the loss of information in critical modules thereby making the system unresponsive. Generally, such situations happen very rarely because before releasing a product all possible scenarios and test cases for the code are simulated. Failure is detected by end-users once they face a particular issue in the software.

- Failure can happen due to human errors or can also be caused intentionally in the system by an individual.
- It is a term that comes after the production stage of the software.
- It can be identified in the application when the defective part is executed.

A simple diagram depicting Bug vs Defect vs Fault vs Failure:



Bug vs Defect vs Error vs Fault vs Failure

Some of the vital differences between bug, defect, fault, error, and failure are listed in the below table:

Basis	Bug	Defect	Fault	Error	Failure
Definition	A bug refers to defects which means that the software product or the application is not working as per the adhered requirements set	A Defect is a deviation between the actual and expected output	A Fault is a state that causes the software to fail and therefore it does not achieve its necessary function.	An Error is a mistake made in the code due to which compilation or execution fails,	Failure is the accumulation of several defects that ultimately lead to Software failure and results in the loss of information in critical modules thereby making the system unresponsive.
Raised by	Test Engineers	The defect is identified by The Testers And is resolved by developers in the development phase of SDLC.	Human mistakes lead to fault.	Developers and automation test engineers	The failure is found by the test engineer during the development cycle of SDLC
Different types	• Logical bugs • Algorithm	Defects are classified as follows:	• Business Logic Faults	• Syntactic Error • UI screen	NA

Basis	Bug	Defect	Fault	Error	Failure
	ic bugs • Resource bugs	**Based on Priority:** • High • Medium • Low **Based on Severity:** • Critical • Major • Minor • Trivial	• Functional and Logical Faults • Graphical User Interface (GUI) Faults • Performance Faults • Security Faults • Hardware Faults	error • Error handling error • Flow control error • Calculation error • Hardware error	
Reasons behind	• Missing Logic • Erroneous Logic • Redundant codes	• Receiving & providing incorrect input • Coding/Logical Error leading to the breakdown of software	• Wrong design of the data definition processes. • An irregularity in Logic or gaps in the software leads to the non-functioning of the software.	• Error in code. • Inability to compile/execute a program • Ambiguity in code logic • Misunderstanding of requirements • Faulty design and architecture • Logical error	• Environment variables • System Errors • Human Error
Way to prevent the reasons	• Implementing Test-driven development. • Adjusting enhanced development practices	• Implementing Out-of-the-box programming methods. • Proper usage of primary and correct software	• Peer review of the Test documents and requirements. • Verifying the correctne	• Conduct peer reviews and code-reviews • Need for validation of bug fixes and enhancing the overall quality of	

Basis	Bug	Defect	Fault	Error	Failure
	and evaluation of cleanliness of the code.	coding practices.	ss of software design and coding.		

1:5 Software Testing Principles

A principle can be defined as:

1. a general or fundamental, law, doctrine, or assumption;
2. a rule or code of conduct;
3. the laws or facts of nature underlying the working of an artificial device.

Principle 1. Testing is the process of exercising a software component using a selected set of test cases, with the intent of (i) revealing defects, and (ii) evaluating quality.

Principle 2. When the test objective is to detect defects, then a good test case is one that has a high probability of revealing a yet undetected defect(s).

Principle 3. Test results should be inspected meticulously.

Principle 4. A test case must contain the expected output or result.

Principle 5. Test cases should be developed for both valid and invalid input conditions.

Principle 6. The probability of the existence of additional defects in a software component is proportional to the number of defects already detected in that component.

Principle 7. Testing should be carried out by a group that is independent of the development group.

Principle 8. Tests must be repeatable and reusable.

Principle 9. Testing should be planned.

Principle 10. Testing activities should be integrated into the software life cycle

1.6 THE TESTER’S ROLE IN A SOFTWARE DEVELOPMENT ORGANIZATION

Testing is sometimes erroneously viewed as a destructive activity. The tester’s job is to,

- reveal defects,
- find weak points,
- inconsistentbehavior, and
- circumstances where the software does not work as expected.

It is difficult for developers to effectively test their own code (Principles 3 and 8). Developers view their own code as their creation, their “baby,” and they think that nothing could possibly be wrong with it!

A tester requires extensive programming experience in order to understand,

- ❖ how code is constructed, and
- ❖ where, and what kind of, defects are likely to occur.
- ❖ Testers also need to work along side with Requirements Engineers
- ❖ To ensure that requirements are testable, and to plan for system &acceptance test Designers
- ❖ To plan for integration and unit test. Project Manager
- ❖ To develop reasonable test plans,Upper Management
- ❖ To provide input for the development and maintenance of organizational testing standards, polices, and goals.software quality assurance staff and software engineering process group members.

In view of these requirements for multiple working relationships, communication and team working skills are necessary for a successful career as a tester.

Developers, analysts, and marketing staff need to realize that testers add value to a software product in that they detect defects and evaluate quality as early as possible in the software life cycle. This ensures that developers release code with few or no defects, and that marketers can deliver software that satisfies the customers’ requirements, and is reliable, usable, and correct.

1:7 Origins of Defects

The term *defect* and its relationship to the terms *error* and *failure* in the context of the software development.

Defects have detrimental affects on software users, and software engineers work very hard to produce high-quality software with a low number of defects. But even under the best of development circumstances errors are made, resulting in defects being injected in the software during the phases of the software life cycle.

1. ***Education:*** The software engineer did not have the proper educational background to prepare the software artifact. She did not understand how to do something.

    - For example, a software engineer who did not understand the precedence order of operators in a particular programming language could inject a defect in an equation that uses the operators for a calculation.

2. ***Communication:*** The software engineer was not informed about something by a colleague.

    - For example, if engineer 1 and engineer 2 are working on interfacing modules, and engineer 1 does not inform engineer 2 that a no error checking code will appear in the interfacing module he is developing, engineer 2 might make an incorrect assumption relating to the presence/absence of an error check, and a defect will result.

3. ***Oversight:*** The software engineer omitted to do something. For example, a software engineer might omit an initialization statement.

4. ***Transcription:***The software engineer knows what to do, but makes a mistake in doing it. A simple example is a variable name being misspelled when entering the code.

5. ***Process:***The process used by the software engineer misdirected her actions. For example, a development process that did not allow sufficient time for a detailed specification to be developed and reviewed could lead to specification defects

When defects are present due to one or more of these circumstances, the software may fail, and the impact on the user ranges from a minor inconvenience to rendering the software unfit for use. Our goal as testers is to discover these defects preferably before the software is in operation.

One of the ways we do this is by designing test cases that have a high probability of revealing defects.

<u>HYPOTHESIS:</u>

- Test cases are then designed based on the hypotheses. The tests are run and results analyzed to prove, or disprove, the hypotheses.

- Myers has a similar approach to testing. He describes the successful test as one that reveals the presence of a (hypothesized) defect]. He compares the role of a tester to that of a doctor who is in the process of constructing a diagnosis for an ill patient.

- A successful test will reveal the problem and the doctor can begin treatment. Completing the analogy of doctor and ill patient, one could view defective software as the ill patient. Testers as doctors need to have knowledge about possible defects (illnesses) in order to develop defect hypotheses. They use the hypotheses to:

- design test cases;
- design test procedures;
- assemble test sets;
- select the testing levels (unit, integration, etc.) Appropriate for the tests;
- evaluate the results of the tests.

- A successful testing experiment will prove the hypothesis is true—that is, the hypothesized defect was present. Then the software can be repaired (treated).

- A very useful concept related to this discussion of defects, testing, and diagnosis is that of a fault model.

A fault (defect) model can be described as a link between the error made (e.g., amissing requirement, a misunderstood design element, a typographical error), andthe fault/defect in the software.

- Digital system engineers describe similar models that link physical defects in digital components to electrical (logic) effects in the resulting digital system Physical defects in the digital world may be due to manufacturing errors, component wear-out, and/or environmental effects.

- The fault models are often used to generate a fault list or dictionary. From that dictionary faults can be selected, and test inputs developed for digital components.

- The effectiveness of a test can be evaluated in the context of the fault model, and is related to the number of faults as expressed in the model, and those actually revealed by the test.

Topic 9: Defect Classes, the Defect Repository, and Test Design

- Defects can be classified in many ways. It is important for an organization to adapt a single classification scheme and apply it to all projects. No matter which classification scheme is selected, some defects will fit into more than one class or category. Because of this problem, developers, testers, and SQA staff should try to be as consistent as possible when recording defect data.

- The defect types and frequency of occurrence should be used to guide test planning, and test design. Execution-based testing strategies should be selected that have the strongest possibility of detecting particular types of defects.

- It is important that tests for new and modified software be designed to detect the most frequently occurring defects.

- Defects, as described in this text, are assigned to four major classes reflecting their point of origin in the software life cycle—the development phase in which they were injected. These classes are: requirements/ specifications, design, code, and testing defects as summarized.
- It should be noted that these defect classes and associated subclasses focus on defects that are the major focus of attention to execution-based testers.

1 R e q u i r e m e n t s a n d S p e c i f i c a t i o n D e f e c t s

- The beginning of the software life cycle is critical for ensuring high quality in the software being developed. Defects injected in early phases can persist and be very difficult to remove in later phases.

- Since many requirements documents are written using a natural language representation, there are very often occurrences of ambiguous, contradictory, unclear, redundant, and imprecise requirements. Specifications in many organizationsare also developed using natural language representations, and these too are subject to the same types of problems as mentioned above.

1 . Functional Description Defects

The overall description of what the product does, and how it should behave (inputs/outputs), is incorrect, ambiguous, and/or incomplete.

2 . Feature Defects

Features may be described as distinguishing characteristics of a software component or system.

3 . Feature Interaction Defects

- These are due to an incorrect description of how the features should interact. For example, suppose one feature of a software system supports adding a new customer to a customer database.

- This feature interactswithanother feature that categorizes the new customer. The classification feature impacts on where the storage algorithm places the new customer in the database, and also affects another feature that periodically supportssending advertising information to customers in a specific category. When testing we certainly want to focus on the interactions between these features.

4 . Interface Description Defects

- These are defects that occur in the description of how the target software is to interface with external software, hardware, and users. For detecting many functional description defects, black box testing techniques, which are based on functional specifications of the software, offer the best approach.

- Randomtesting and error guessing are also useful for detecting these types of defects. The reader should note that many of these types of defects can be detected early in the life cycle by software reviews.

- Black box–based tests can be planned at the unit, integration, system, and acceptance levels to detect requirements/specification defects.

- Manyfeature interaction and interfaces description defects are detected using black box–based test designs at the integration and system levels.

Topic 9:2 D e s i g n D e f e c t s

Design defects occur when system components, interactions between system components, interactions between the components and outside ware/hardware, or users are incorrectly designed.

This covers defects in the design of algorithms, control, logic, data elements, module interface descriptions, and external software/hardware/user interface descriptions.

When describing these defects we assume that the detailed design description for the software modules is at the pseudo code level with processing steps, data structures, input/output parameters, and major control structures defined.

If module design is not described in such detail then many of the defects types described here may be moved into the coding defects class.

1 . Algorithmic and Processing Defects

These occur when the processing steps in the algorithm as described by the pseudo code are incorrect.

Example: of a defect in this subclass is the omission of error condition checks such as division by zero. In the case of algorithm reuse, a designer may have selected an inappropriate algorithm for this problem

2 . Control, Logic, and Sequence Defects

Control defects occur when logic flow in the pseudo code is not correct.

For **example:** branching to soon, branching to late, or use of an incorrect branching condition

3 . Data Defects

These are associated with incorrect design of data structures. For example, a record may be lacking a field, an incorrect type is assigned to a variable or a field in a record, an array may not have the proper numberof elements assigned, or storage space may be allocated incorrectly.

4 . Module Interface Description Defects

These are defects derived from, for example, using incorrect, and/or inconsistent parameter types, an incorrect number of parameters, or an incorrect ordering of parameters.

5 . Functional Description Defects

The defects in this category include incorrect, missing, and/or unclear design elements.

For example, the design may not properly describe the correct functionality of a module. These defects are best detected during a design review.

6 . External Interface Description Defects

These are derived from incorrect design descriptions for interfaces with COTS components, external software systems, databases, and hardware devices (e.g., I/O devices).

Other examples are user interface descriptiondefects where there are missing or improper commands, improper sequences of commands, lack of proper messages, and/or lack of feedback messages for the user.

Topic 9:3 . C o d i n g D e f e c t s

Coding defects are derived from errors in implementing the code. Coding defects classes are closely related to design defect classes especially if pseudo code has been used for detailed design.

Some coding defects comefrom a failure to understand programming language constructs, and miscommunication with the designers. Others may have transcription or omission origins. At times it may be difficult to classify a defect as a designor as a coding defect.

1 . Algorithmic and Processing Defects
Adding levels of programming detail to design, code-related algorithmic and processing defects would now include unchecked overflow and underflow conditions, comparing inappropriate data types, converting one data type to another, incorrect ordering of arithmetic operators (perhaps due to misunderstanding of the precedence of operators), misuse or omission of parentheses, precision loss, and incorrect use of signs.

2 . Control, Logic and Sequence Defects
On the coding level these would include incorrect expression of case statements, incorrect iteration of loops (loop boundary problems), and missing paths.

3 . Typographical Defects

These are principally syntax errors, for example, incorrect spelling of a variable name, that are usually detected by a compiler, self-reviews, or peer reviews.

4 . I n i t i a l i z a t i o n Defects
These occur when initialization statements are omitted or are incorrect. This may occur because of isunderstandings or lack of communication between programmers, and/or programmers and designers, carelessness,
or misunderstanding of the programming environment.

5 . Data-Flow Defects
There are certain reasonable operational sequences that data should flow through. For example, a variable should be initialized, before it is used in a calculation or a condition. It should not be initialized twice before there is an intermediate use. A variable should not be disregarded before it is used.

Occurrences of these suspicious variable uses in the code may, or may not, cause anomalous behavior. Therefore, in the strictest sense of the definition for the term "defect," they may not be considered as true instances of defects.

6 . Data Defects
These are indicated by incorrect implementation of data structures. For example, the programmer may omit a field in a record, an incorrect type or access is assigned to a file, an array may not be allocated the proper number of elements. Other data defects include flags, indices, and constants set incorrectly.

7 . Module Interface Defects
As in the case of module design elements, interface defects in the code may be due to using incorrect or inconsistent parameter types, an incorrect number of parameters, or improper ordering of the parameters. In addition to defects due to improper design, and improper implementation of design, programmers may implement an incorrect sequence of calls or calls to nonexistent modules.

8 . Code Documentation Defects
When the code documentation does not reflect what the program actually does, or is incomplete or ambiguous, this is called a code documentation defect. Incomplete, unclear, incorrect, and out-of-date code documentationaffects testing efforts. Testers may be misled by documentation defects and thus reuse improper tests or design new tests that are not appropriate for the code. Code reviews are the best tools to detect these types of defects.

9 . External Hardware, Software Interfaces Defects
These defects arise from problems related to system calls, links to databases, input/output sequences, memory usage, resource usage, interrupts and exception handling, data exchanges with hardware, protocols, formats, interfaces with build files, and timing sequences (race conditions may result).

Many initialization, data flow, control, and logic defects that occur in design and code are best addressed by white box testing techniques applied at the unit (single-module) level.

White box testing approaches are dependent on knowledge of the internal structure of the software, in contrast to black box approaches, which are only dependent on behavioral specifications.

For example, application of decision tables is very useful for detecting errors in Boolean expressions. Black box tests as described in applied at the integration and system levels help to reveal external hardware and software interface defects. The author will stress repeatedly throughout the text that a combination of both of these approaches is needed to reveal the many types of defects that are likely to be found in software.

Topic 9:4 T e s t i n g D e f e c t s

Defects are not confined to code and its related artifacts. Test plans, test cases, test harnesses, and test procedures can also contain defects. Defects in test plans are best detected using review techniques.

1 . Test Harness Defects

In order to test software, especially at the unit and integration levels, auxiliary code must be developed. This is called the test harness or scaffolding code. Chapter 6 has a more detailed discussion of the need for this code. The test harness code should be carefully designed, implemented, and tested since it a work product and much of this code can be reused when new releases of the software are developed. Test harnesses are subject to the same types of code and design defects that can be found in all other types of software.

2 . Test Case Design and Test Procedure Defects

These would encompass incorrect, incomplete, missing, inappropriate test cases, and test procedures. These defects are again best detected in test plan reviews.. Sometimes the defects are revealed during the testing process itself by means of a careful analysis of test conditions and test results. Repairs will then have to be made.

Topic :10 Defect Examples: The Coin Problem

The following examples illustrate some instances of the defect classes that were discussed in the previous sections. A simple specification, a detailed design description, and the resulting code are shown, and defects in eachare described. Note that these defects could be injected via one or more of the five defect sources discussed at the beginning of this chapter. Also note that there may be more than one category that fits a given defect.

Sample informal specification for a simple program that calculates the total monetary value of a set of coins.Theprogram could be a component of an interactive cash register system to

support retail store clerks. This simple example shows requirements/ specification defects, functional description defects, and interface description defects.
The functional description defects arise because the functional description is ambiguous and incomplete. It does not state that the input, number_of_coins, and the output, number_of_dollars and number _of_cents, should all have values of zero or greater. The number_of_coins cannot be negative, and the values in dollars and cents cannot be negative in the real-world domain. As a consequence of these ambiguities and specification incompleteness, a checking routine may be omitted from the design, allowing the final program to accept negative values for the input
number_of_coins for each of the denominations, and consequently it may calculate an invalid value for the results.
A more formally stated set of preconditions and postconditions would be helpful here, and would address some of the problems with the specification. These are also useful for designing black box tests.

A precondition is a condition that must be true in order for a software component to operate properly.

In this case a useful precondition would be one that states for example:

Number-of-coins >= 0

A postcondition is a condition that must be true when a software component completes its operation properly.

A useful postcondition would be:

Number-of-dollars, number-of-cents >0.

In addition, the functional description is unclear about the largest number of coins of each denomination allowed, and the largest number of dollars and cents allowed as output values.

A sample specification with defects.
Interface description defects relate to the ambiguous and incomplete description of user–software interaction. It is not clear from the specification how the user interacts with the program to provide input, and how the output is to be reported. Because of ambiguities in the user interaction description the software may be difficult to use.
Likely origins for these types of specification defects lie in the nature of the development process, and lack of proper education and training.
A poor-quality development process may not be allocating the proper time and resources to specification development and review. In addition, software engineers may not have the proper education and training to develop a quality specification. All of these specification defects, if not detected and repaired, will propagate to the design and coding phases.
Black box testing techniques, which we will study in Chapter 4, will help to reveal many of these functional weaknesses.
Figure 3.4 shows the specification transformed in to a design description.
There are numerous design defects, some due to the ambiguous and incomplete nature of the specification; others are newly introduced.

Design defects include the following:

Control, logic, and sequencing defects.

The defect in this subclass arises from an incorrect “while” loop condition (should be less than or equal to six)

Design Description for Program calculate_coin_values:

```
Program calculate_coin_values
number_of_coins is integer
total_coin_value is integer
number_of_dollars is integer
number_of_cents is integer
coin_values is array of six integers representing
each coin value in cents
initialized to: 1,5,10,25,25,100
begin
initializetotal_coin_value to zero
initializeloop_counter to one
whileloop_counter is less then six
begin
output "enter number of coins"
read (number_of_coins )
total_coin_value = total_coin_value +
number_of_coins * coin_value[loop_counter]
incrementloop_counter
end
number_dollars = total_coin_value/100
number_of_cents = total_coin_value - 100 * number_of_dollars
output (number_of_dollars, number_of_cents)
end
```

Algorithmic, and processing defects.

These arise from the lack of error checks for incorrect and/or invalid inputs, lack of a path where users can correct erroneous inputs, lack of a path for recovery from input errors. The lack of an error check could also be counted as a functional design defect since the design does not adequately describe the proper functionality for the program.

Data defects.

This defect relates to an incorrect value for one of the elements of the integer array, coin_values, which should read 1,5,10,25,50,100.

External interface description defects.

These are defects arising from the absence of input messages or prompts that introduce the program to the user and request inputs. The user has no way of knowing in which order the number of coins for each denomination must be input, and when to stop inputting values. There is an absence of help messages, and feedback
for user if he wishes to change an input or learn the correct format and order for inputting the number of coins. The output description and output formatting is incomplete. There is no description of what the outputs means in terms of the problem domain. The user will note that two values are output, but has no clue as to their meaning. The ontrol and logic design defects are best addressed by white box– based tests, (condition/branch testing, loop testing). These other designdefects will need a combination of white and black box testing techniques for detection.

.

Control, logic, and sequence defects.

These include the loop variable increment step which is out of the scope of the loop. Note that incorrect loop condition (*i* _ 6) is carried over from design and should be counted as a design defect.

Algorithmic and processing defects.

The division operator may cause problems if negative values are divided, although this problem could be eliminated with an input check.

Data Flow defects.

The variable total_coin_value is not initialized. It is used before it is defined. (This might also be considered a data defect.)

Data Defects.

The error in initializing the array coin_values is carried over from design and should be counted as a design defect.

External Hardware, Software Interface Defects.

The call to the external function "scanf" is incorrect. The address of the variable must be provided (&number_of_coins).

Code Documentation Defects.

The documentation that accompanies this code is incomplete and ambiguous. It reflects the deficiencies in the external interface description and other defects that occurred during specification and design. Vital information is missing for anyone who will need to repair, maintain or reuse this code. The control, logic, and sequence, data flow defects found in this example could be detected by using a combination of white and black box
testing techniques. Black box tests may work well to reveal the algorithmic and data defects. The code documentation defects require a code review for detection

```
/****************************************************************
programcalculate_coin_values calculates the dollar and cents
value of a set of coins of different dominations input by the user
denominations are pennies, nickels, dimes, quarters, half dollars,
and dollars
*****************************************************************/
main ()
{
inttotal_coin_value;
intnumber_of_coins = 0;
intnumber_of_dollars = 0;
intnumber_of-cents = 0;
intcoin_values = {1,5,10,25,25,100};
{
int i = 1;
while ( i < 6)
{
printf("input number of coins\n");
scanf ("%d", number_of_coins);
total_coin_value = total_coin_value +
(number_of_coins * coin_value{i]);
}
i = i + 1;
number_of_dollars = total_coin_value/100;
number_of_cents = total_coin_value - (100 * number_of_dollars);
printf("%d\n", number_of_dollars);
printf("%d\n", number_of-cents);
}
/*****************************************************************/
```