Software Testing and Quality Assurance Notes

Sem 7

Module 1: Testing Methodology

1.1

- *Goals and Models for software testing:*

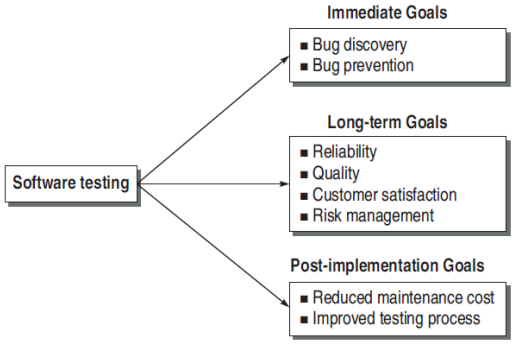


Figure 1.2 Software testing goals

➔ Short-term or immediate goals: These goals are the immediate results after performing testing. These goals may be set in the individual phases of SDLC. Some of them are discussed below.

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

Bug prevention It is the consequent action of bug discovery. From the behavior and interpretation of bugs discovered, everyone in the software development team gets to learn how to code safely such that the bugs discovered should not be repeated in later stages or future projects.

➔ Long-term goals: These goals affect the product quality in the long run, when one cycle of the SDLC is over. Some of them are discussed here.

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.

Reliability: is a matter of confidence that the software will not fail, and this level of confidence increases with rigorous testing. The confidence in reliability, in turn, increases the quality.

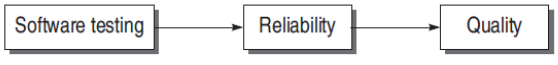


Figure 1.3 Testing produces reliability and quality

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.

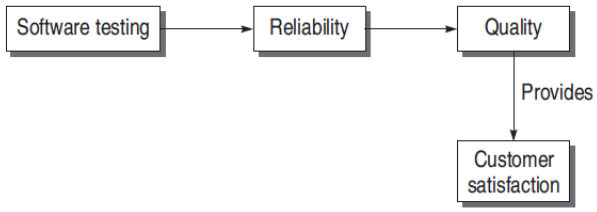


Figure 1.4 Quality leads to customer satisfaction

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.

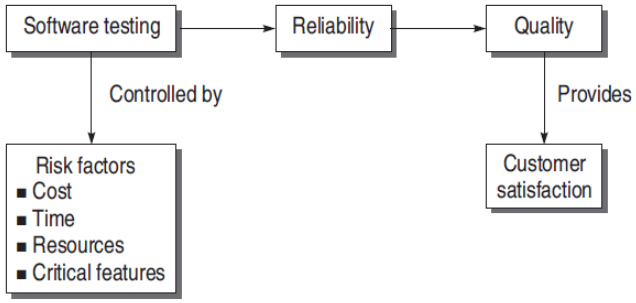


Figure 1.5 Testing controlled by risk factors

➔ Post-implementation goals: These goals are important after the product is released. Some of them are discussed here.

Reduced maintenance cost: Post-release errors are costly to fix, as they are difficult to detect. Thus, if testing has been done rigorously and effectively, then the chances of failure are minimized and in turn, the maintenance cost is reduced.

Improved software testing process: A testing process for one project may not be successful and there may be scope for improvement. Therefore, the bug history and post-implementation results can be analyzed to find snags in the present testing process, which can be rectified in future projects.

➔ The developer develops the software in the prescribed system environment considering the testability of the software. Testability is a major issue for the developer while developing the software, as a badly written software may be difficult to test. Testers are supposed to get on with their tasks as soon as the requirements are specified. Testers work on the basis of a bug model which classifies the bugs based on the criticality or the SDLC phase in which the testing is to be performed. Based on the software type and the bug model, testers decide a testing methodology which guides how the testing will be performed.

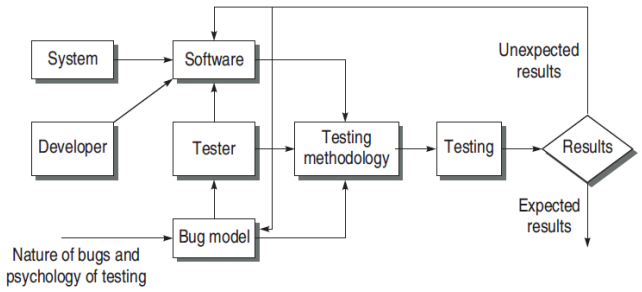


Figure 1.6 Software testing model

➔ Software and Software Model:
In this model of testing, our aim is to concentrate on the testing process, therefore the software under consideration should not be so complex such that it would not be tested. They should design and code the software such that it is testable at every point.

➔ Bug Model:
Bug model provides a perception of the kind of bugs expected. Considering the nature of all types of bugs, a bug model can be prepared that may help in deciding a testing strategy.

➔ Testing methodology and Testing:
Testers can develop a testing methodology that incorporates both testing strategy and testing tactics. Testing strategy is the roadmap that gives us well-defined steps for the overall testing process. It prepares the planned steps based on the risk factors and the testing phase. Once the planned steps of the testing process are prepared, software testing techniques and testing tools can be applied within these steps. Thus, testing is performed on this methodology.

- *Software testing terminology and methods:*
Definitions:

➔ Failure When the software is tested, failure is the first term being used. It means the inability of a system or component to perform a required function according to its specification.

➔ Fault/Defect/Bug Failure is the term which is used to describe the problems in a system on the output side. Fault is a condition that actually causes a system to produce failure. Fault is synonymous with the words defect or bug.

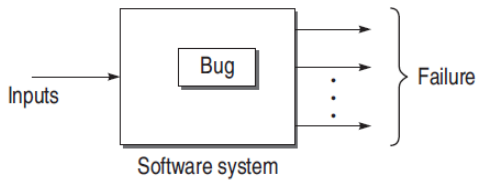


Figure 2.1 Testing terminology

➔ Error Error is a very general term used for human mistakes.

➔ Test case Test case is a well-documented procedure designed to test the functionality of a feature in the system. A test case has an identity and is associated with a program behavior. The primary purpose of designing a test case is to find errors in the system.

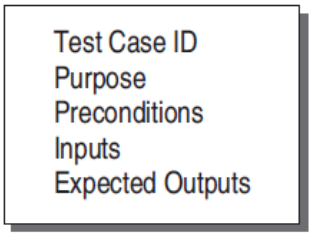


Figure 2.3 Test case template

➔ Testware The documents created during testing activities are known as testware. Taking the analogy from software and hardware as a product, testware are the documents that a test engineer produces.

➔ Incident An incident is the symptom(s) associated with a failure that alerts the user about the occurrence of a failure.

➔ Test oracle An oracle is the means to judge the success or failure of a test, i.e. to judge the correctness of the system for some test

Life Cycle of a Bug:

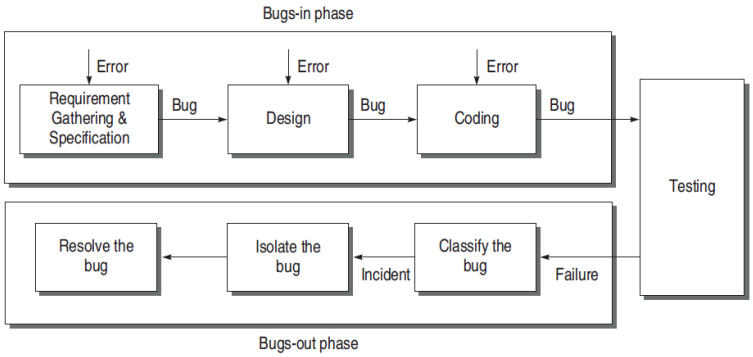


Figure 2.4 Life cycle of a bug

➔ Bugs-In Phase

This phase is where the errors and bugs are introduced in the software. Whenever we commit a mistake, it creates errors on a specific location of the software and consequently, when this error goes unnoticed, it causes some conditions to fail, leading to a bug in the software. This bug is carried out to the subsequent phases of SDLC, if not detected.

➔ Bugs-Out Phase

If failures occur while testing a software product, we come to the conclusion that it is affected by bugs. In this phase, when we observe failures, the following activities are performed to get rid of the bugs.

A. Bug classification: A bug can be critical or catastrophic in nature or it may have no adverse effect on the output behavior of the software. In this way, we classify all the failures.
B. Bug isolation: Bug isolation is the activity by which we locate the module in which the bug appears. Incidents observed in failures help in this activity.
C. Bug resolution: Once we have isolated the bug, we back-trace the design to pinpoint the location of the error.

State Of A Bug:

➔ New: The state is new when the bug is reported for the first time by a tester.
➔ Open: The new state does not verify that the bug is genuine. When the test leader approves that the bug is genuine, its state becomes open.
➔ Assign: If the bug is valid, a developer is assigned the job to fix it and the state of the bug now is 'ASSIGN'.
➔ Deferred: If the priority of the reported bug is not high or there is not sufficient time to test it or the bug does not have any adverse effect on the software, then the bug is changed to deferred state which implies the bug is expected to be fixed in next releases.
➔ Rejected: It may be possible that the developer rejects the bug after checking its validity, as it is not a genuine one
➔ Test: Before releasing to the testing team, the developer changes the bug's state to 'TEST'. It specifies that the bug has been fixed by the development team but not tested and is released to the testing team.
➔ Verified/fixed: The tester tests the software and verifies whether the reported bug is fixed or not. After verifying, the developer approves that the bug is fixed and changes the status to 'VERIFIED'.
➔ Reopened: If the bug is still there even after fixing it, the tester changes its status to 'REOPENED'. The bug traverses the life cycle once again. In another case, a bug which has been closed earlier may be reopened if it appears again. In this case, the status will be REOPENED instead of OPEN.
➔ Closed: Once the tester and other team members are confirmed that the bug is completely eliminated, they change its status to 'CLOSED'.

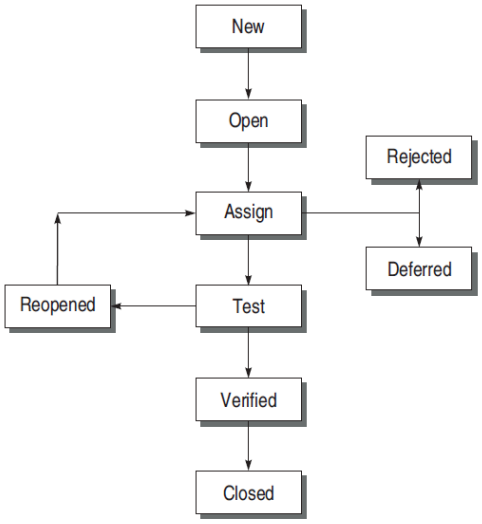


Figure 2.5 States of a bug

Why Do Bugs Occur:

➔ To Err is Human

➔ Bugs in Earlier Stages go Undetected and Propagate

Bugs Affect Economics Of Software Testing:

The cost of a bug is equal to Detection Cost + Correction Cost.

Bugs Classification Based on Criticality:

➔ Critical Bugs

➔ Major Bug

➔ Medium Bugs

➔ Minor Bugs

Bugs Classification Based on SDLC:

➔ Requirements and Specifications Bugs Requirement gathering and specification is a difficult phase in the sense that requirements gathered from the customer are to be converted into a requirement specification which will become the base for design.

➔ Design Bugs Design bugs may be the bugs from the previous phase and in addition those errors which are introduced in the present phase.

A. Control flow bugs
B. Logic bugs
C. Processing bugs
D. Data flow bugs
E. Error handling bugs
F. Race condition bugs
G. Boundary-related bugs (max aur min ke bahar jayega tho kya hoga)

H. User interface bugs

➔ Interface and Integration Bugs
➔ System Bugs
➔ Testing Bugs

Testing Principles:

➔ Effective testing, not exhaustive testing
➔ Testing is not a single phase performed in SDLC
➔ Destructive approach for constructive testing
➔ Early testing is the best policy
➔ Probability of existence of an error in a section of a program is proportional to the number of errors already found in that section
➔ Testing strategy should start at the smallest module level and expand towards the whole program
➔ Testing should also be performed by an independent team
➔ Everything must be recorded in software testing
➔ Invalid inputs and unexpected behavior have a high probability of finding an error
➔ Testers must participate in specification and design reviews

Software Testing Methodologies:

➔ Software testing methodology is the organization of software testing by means of which the test strategy and test tactics are achieved.

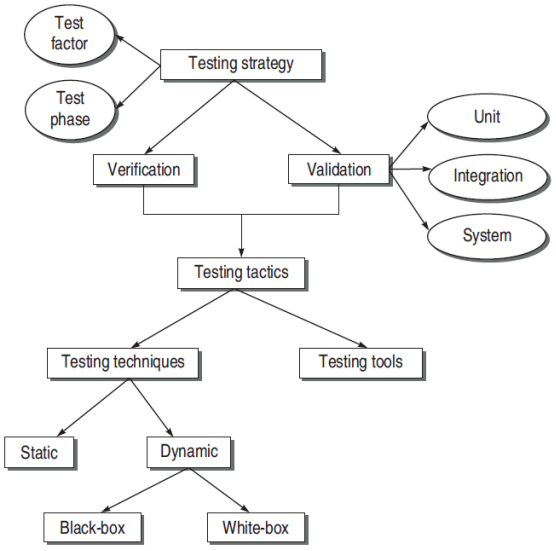


Figure 2.11 Testing methodology

- *Software Testing Life Cycle (STLC):*

➔ The testing process divided into a well-defined sequence of steps is termed as software testing life cycle (STLC). The major contribution of STLC is to involve the testers at early stages of development.

➔ This has a significant benefit in the project schedule and cost. The STLC also helps the management in measuring specific milestones.

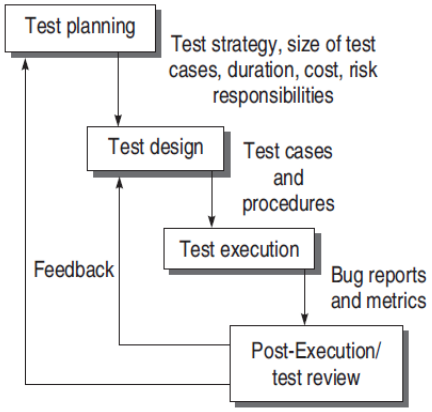


Figure 2.8 Software testing life cycle

A. Test Planning:

- Defining the test strategy.
- Estimate the number of test cases, their duration, and cost.
- Plan the resources like the manpower to test, tools required, documents required.
- Identifying areas of risks.
- Defining the test completion criteria.
- Identification of methodologies, techniques, and tools for various test cases.
- Identifying reporting procedures, bug classification, databases for testing, bug severity levels, and project metrics.

After analyzing the issues, the following activities are performed:

- Develop a test case format.
- Develop test case plans according to every phase of SDLC.
- Identify test cases to be automated (if applicable).
- Prioritize the test cases according to their importance and criticality.
- Define areas of stress and performance testing.
- Plan the test cycles required for regression testing.

B. Test Design:

- Determining the test objectives and their prioritization
- Preparing list of items to be tested
- Mapping items to test cases

After making a list of items to be tested, there is a need to identify the test cases. A matrix can be created for this purpose, identifying which test case will be

covered by which item. The existing test cases can also be used for this mapping. Thus it permits reusing the test cases. This matrix will help in:

(a) Identifying the major test scenarios.

(b) Identifying and reducing the redundant test cases.

(c) Identifying the absence of a test case for a particular objective and as a result, creating them.

Some attributes of a good test case are given below:

(a) A good test case is one that has been designed keeping in view the criticality and high-risk requirements in order to place a greater priority upon, and provide added depth for testing the most important functions

(b) A good test case should be designed such that there is a high probability of finding an error.

(c) Test cases should not overlap or be redundant. Each test case should address a unique functionality, thereby not wasting time and resources.

(d) Although it is sometimes possible to combine a series of tests into one test case, a good test case should be designed with a modular approach so that there is no complexity and it can be reused and recombined to execute various functional paths. It also avoids masking of errors and duplication of test-creation efforts.

(e) A successful test case is one that has the highest probability of detecting an as-yet-undiscovered error.

- Selection of test case design techniques
- Creating test cases and test data
- Setting up the test environment and supporting tools
- Creating test procedure specification

C. Test Execution:

In this phase, all test cases are executed including verification and validation. Verification test cases are started at the end of each phase of SDLC. Validation test cases are started after the completion of a module. It is the decision of the test team to opt for automation or manual execution. Test results are documented in the test incident reports, test logs, testing status, and test summary reports.

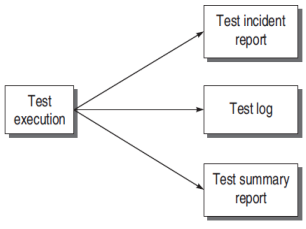


Figure 2.10 Documents in test execution

Table 2.1 Testing level vs responsibility

Test Execution Level	Person Responsible
Unit	Developer of the module
Integration	Testers and Developers
System	Testers, Developers, End-users
Acceptance	Testers, End-users

D. Post-Execution/ Test Review:

As soon as the developer gets the bug report, he performs the following activities:

- Understanding the bug

 Reproducing the bug
 Analyzing the nature and cause of the bug

After this, the results from manual and automated testing can be collected. The final bug report and associated metrics are reviewed and analyzed for the overall testing process. The following activities can be done:
 Reliability analysis can be performed to establish whether the software meets the predefined reliability goals or not.
 Coverage analysis can be used as an alternative criterion to stop testing.
 Overall defect analysis can identify risk areas and help focus our efforts on quality improvement.

1.2

- *Verification:*

➔ If verification is not performed at early stages, there is always a chance of mismatch between the required product and the delivered product.
➔ Verification exposes more errors.
➔ Early verification decreases the cost of fixing bugs.
➔ Early verification enhances the quality of software.
➔ Goals of verification:
   A. Everything Must be Verified
   B. Results of Verification May Not be Binary
   C. Even Implicit Qualities Must be Verified

- *Verification of Requirements:*

➔ An acceptance criterion defines the goals and requirements of the proposed system and acceptance limits for each of the goals and requirements. The acceptance criteria matter the most in case of real-time systems where performance is a critical issue in certain events. Acceptance criteria for a requirement must be defined by the designers of the system and should not be overlooked, as they can create problems while testing the system.
➔ The tester works in parallel by performing the following two tasks:
   A. The tester reviews the acceptance criteria in terms of its completeness, clarity, and testability.
   B. The tester prepares the Acceptance Test Plan.
➔ <u>Verification of Objectives:</u> Two parallel activities are performed by the tester:
   A. The tester verifies all the objectives mentioned in SRS. The purpose of this verification is to ensure that the user's needs are properly understood before proceeding with the project.
   B. The tester also prepares the System Test Plan which is based on SRS.

➔ Verifying Requirements and Objectives: Requirement and objectives verification has a high potential of detecting bugs. Therefore, requirements must be verified.

A. Correctness:

 Testers should refer to other documentations or applicable standards and compare the specified requirement with them.

 Testers can interact with customers or users, if requirements are not well-understood.

 Testers should check the correctness in the sense of realistic requirement. If the tester feels that a requirement cannot be realized using existing hardware and software technology, it means that it is unrealistic. In that case, the requirement should either be updated or removed from SRS.

B. Unambiguous:

 Every requirement has only one interpretation.

 Each characteristic of the final product is described using a single unique term.

C. Consistent:

 Real-world objects conflict, for example, one specification recommends mouse for input, another recommends joystick.

 Logical conflict between two specified actions, e.g. one specification requires the function to perform square root, while another specification requires the same function to perform square operation.

 Conflicts in terminology should also be verified. For example, at one place, the term process is used while at another place, it has been termed as task or module.

D. Completeness:

 Verify that all significant requirements such as functionality, performance, design constraints, attribute, or external interfaces are complete.

 Check whether responses of every possible input (valid & invalid) to the software have been defined.

 Check whether figures and tables have been labeled and referenced completely

E. Updation:

 If the specification is a new one, then all the above mentioned steps and their feasibility should be verified.

 If the specification is a change in an already mentioned specification, then we must verify that this change can be implemented in the current design.

F. Traceability:

The traceability of requirements must also be verified such that the origin of each requirement is clear and also whether it facilitates referencing in future development or enhancement documentation.

 Backward traceabilityCheck that requirement references in previous documents

 Forward traceabilityCheck that each requirement has a unique name or reference number in all the documents.

- *Verification of High Level Design:*

➔ The tester verifies the high-level design. Since the system has been decomposed in a number of subsystems or components, the tester should verify the functionality of these components. Since the system is considered a black box with no low-level details considered here, the stress is also on how the system will interface with the outside world. So all the interfaces and interactions of user/customer (or any person who is interfacing with the system) are specified in this phase. The tester verified that all the components and their interfaces are in tune with requirements of the user. Every requirement in SRS should map the design.

➔ The tester also prepares a Function Test Plan which is based on the SRS.

➔ Verifying High Level Design:

A. Data Design:
   It creates a model of data and/or information that is represented at a high level of abstraction. The design of data structures and the associated algorithms required to manipulate them is essential to create high-quality applications.

B. Architectural Design:
   Focuses on the representation of the structure of software components, their properties, and interactions.

C. Interface Design:
   Creates an effective communication medium between the interfaces of different software modules, interfaces between the software system and any other external entities, and interfaces between a user and the software system.

D. Verification of Data Design:
   - Check whether the sizes of data structure have been estimated appropriately.
   - Check the provisions of overflow in a data structure.
   - Check the consistency of data formats with the requirements.
   - Check whether data usage is consistent with its declaration.
   - Check the relationships among data objects in data dictionary.
   - Check the consistency of databases and data warehouses with the requirements specified in SRS.

E. Verification of Architectural Design:
   - Check that every functional requirement in the SRS has been take care of in this design.
   - Check whether all exceptions handling conditions have been taken care of.
   - Verify the process of transform mapping and transaction mapping, used for the transition from requirement model to architectural design.
   - Since architectural design deals with the classification of a system into subsystems or modules, check the functionality of each module according to the requirements specified.
   - Check the inter-dependence and interface between the modules

 In the modular approach of architectural design, there are two issues with modularity— Module Coupling and Module Cohesion. A good design will have low coupling and high cohesion. Testers should verify these factors, otherwise they will affect the reliability and maintainability of the system which are non-functional requirements of the system.

F. Verification of User-Interface Design:

 Check all the interfaces between modules according to the architecture design.

 Check all the interfaces between software and other non-human producer and consumer of information.

 Check all the interfaces between human and computer.

 Check all the above interfaces for their consistency.

 Check the response time for all the interfaces are within required ranges. It is very essential for the projects related to real-time systems where response time is very crucial.

 For a Help Facility, verify the following:

(i) The representation of Help in its desired manner

(ii) The user returns to the normal interaction from Help

 For error messages and warnings, verify the following:

(i) Whether the message clarifi es the problem

(ii) Whether the message provides constructive advice for recovering from the error

 For typed command interaction, check the mapping between every menu option and their corresponding commands.

- *Verification of Low Level Design:*

➔ In LLD, a detailed design of modules and data are prepared such that an operational software is ready.

➔ Testers also perform the following parallel activities in this phase:

A. The tester verifies the LLD. The details and logic of each module is verified such that the high-level and low-level abstractions are consistent.
B. The tester also prepares the Unit Test Plan.

➔ Verifying Low-Level Design:

A. Verify the SRS of each module.
B. Verify the SDD of each module.
C. In LLD, data structures, interfaces, and algorithms are represented by design notations; verify the consistency of every item with their design notations.

➔ Organizations can build a two-way traceability matrix between the SRS and design (both HLD and LLD) such that at the time of verification of design, each requirement mentioned in the SRS is verified.

- *Verification of Code:*

➔ Check that every design specification in HLD and LLD has been coded using a traceability matrix.

➔ Examine the code against a language specification checklist.

➔ Some points against which the code can be verified are:

A. Misunderstood or incorrect arithmetic precedence
B. Mixed mode operations
C. Incorrect initialization
D. Precision inaccuracy
E. Incorrect symbolic representation of an expression
F. Different data types
G. Improper or non-existent loop termination
H. Failure to exit

➔ Static testing techniques It considers only static analysis of the code or some form of conceptual execution of the code.

➔ Dynamic testing techniques It executes the code on some test data. The developer is the key person in this process who can verify the code of his module by using the dynamic testing technique.

1.3

- *Validation:*

➔ Validation is a set of activities that ensures the software under consideration has been built right and is traceable to customer requirements. Validation testing is performed after the coding is over.

➔ Why to perform?

A. To determine whether the product satisfies the users' requirements, as stated in the requirement specification.
B. To determine whether the product's actual behavior matches the desired behavior, as described in the functional design specification.
C. It is not always certain that all the stages till coding are bug-free. The bugs that are still present in the software after the coding phase need to be uncovered.
D. Validation testing provides the last chance to discover bugs, otherwise these bugs will move to the final product released to the customer.
E. Validation enhances the quality of software.

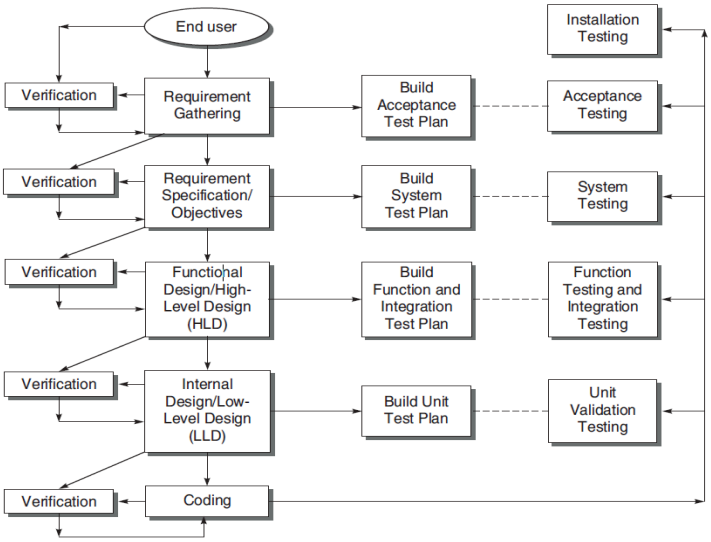


Figure 7.1 V&V activities

- *Unit Validation Testing:*

➔ The testing strategy is to first focus on the smaller building blocks of the full system. One unit or module is the basic building block of the whole software that can be tested for all its interfaces and functionality. This type of testing is largely based on black-box techniques.

➔ Thus, unit testing is a process of testing the individual components of a system. A unit or module must be validated before integrating it with other modules.

➔ Unit validation is the first validation activity after the coding of one module is over.

➔ The motivation for unit validation as compared to the whole system are

➔ as follows:

A. Since the developer has his attention focused on a smaller building block of the software, i.e. unit or module, it is quite natural to test the unit first.
B. If the whole software is tested at once, then it is very difficult to trace the bug. Thus, debugging becomes easy with unit testing.
C. Sometimes, some modules are sent to other organizations for development. This requires parallelism in software development. If we did not have the concept of modules, this type of parallelism would not have existed. Thus, every module can be developed and tested independently.

➔ Driver: Suppose a module is to be tested, wherein some inputs are to be received from another module and this module which passes inputs to the module to be tested is not ready and under development. In such a situation, we need to simulate the inputs required in the module to be tested. For this purpose, a main program is prepared, wherein the required inputs are either hard-coded or entered by the user and passed on to the module under test. This module where the required inputs for the module under test are simulated for the purpose of module or unit testing is known as a driver module. The driver module

may print or interpret the results produced by the module under testing. A test driver provides the following facilities to a unit to be tested:

A. Initializes the environment desired for testing
B. Provides simulated inputs in the required format to the units to be tested.

➔ The module under testing may also call some other module, which is not ready at the time of testing. Therefore, these modules need to be simulated for testing. In most cases, dummy modules instead of actual modules, which are not ready, are prepared for these subordinate modules. These dummy modules are called stubs.Thus, a stub can be defined as a piece of software that works similar to a unit which is referenced by the unit being tested, but it is much simpler than the actual unit. A stub works as a 'stand-in' for the subordinate unit and provides the minimum required behavior for that unit.

➔ The benefits of designing subs and divers are:

A. Stubs allow the programmer to call a method in the code being developed, even if the method doesn't have the desired behavior yet.
B. By using stubs and drivers effectively, we can cut down total debugging and testing time by testing small parts of a program individually, helping us to narrow down problems before they expand.
C. Stubs and drivers can also be an effective tool for demonstrating progress in a business environment.

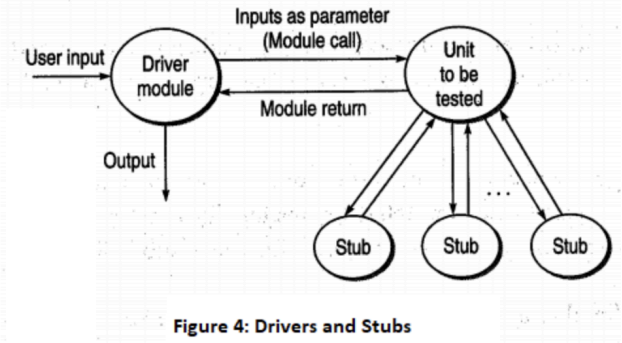


Figure 4: Drivers and Stubs

- *Integration Testing:*

➔ It is the process of combining and testing multiple components or modules together. The individual tested modules, when combined with other modules, are not tested for their interfaces.

➔ Therefore, they may contain bugs in an integrated environment. Thus, the intention here is to uncover the bugs that are present when unit tested modules are integrated.

- *Function Testing:*

➔ When the integrated system has been tested, all the specified functions and their external interfaces are tested on the software. Every functionality of the system specified in the functions is tested according to its external specifications.

➔ The objective of the function test is to measure the quality of the functional (business) components of the system.

➔ The primary processes/deliverables for requirements based function test are discussed below.

A. Test planning
B. Partitioning/functional decomposition
C. Requirement definition
D. Test case design:
E. Traceability matrix formation: Test cases need to be traced/mapped back to the appropriate requirement. A function coverage matrix is prepared. This matrix is a table, listing specific functions to be tested, the priority for testing each function, and test cases that contain tests for each function. Once all the aspects of a function have been tested by one or more test cases, then the test design activity for that function can be considered complete.
F. Test case execution

➔ The function test must determine if each component or business event:

A. performs in accordance to the specifications,
B. responds correctly to all the conditions that may be presented by incoming events/data,
C. moves the data correctly from one business event to the next (including data stores),
D. is initiated in the order required to meet the business objectives of the system.

- *System Testing:*

➔ System testing is actually a series of different tests whose primary purpose is to fully exercise a computer-based system. System testing does not aim to test the specified function, but its intention is to test the whole system on various grounds where bugs may occur.

➔ System Testing Process: System Testing is performed in the following steps:

A. Test Environment Setup: Create testing environment for better quality testing.
B. Create Test Case: Generate test case for the testing process.
C. Create Test Data: Generate the data that is to be tested.
D. Execute Test Case: After the generation of the test case and the test data, test cases are executed.
E. Defect Reporting: Defects in the system are detected.
F. Regression Testing: It is carried out to test the side effects of the testing process.
G. Log Defects: Defects are fixed in this step.
H. Retest: If the test is not successful then again the test is performed.

➔ Types of System Testing:

A. Performance Testing: Performance Testing is a type of software testing that is carried out to test the speed, scalability, stability and reliability of the software product or application.
B. Load Testing: Load Testing is a type of software Testing which is carried out to determine the behavior of a system or software product under extreme load.
C. Stress Testing: Stress Testing is a type of software testing performed to check the robustness of the system under the varying loads.
D. Scalability Testing: Scalability Testing is a type of software testing which is carried out to check the performance of a software application or system in terms of its capability to scale up or scale down the number of user request load.

- *Acceptance Testing:*

➔ Acceptance criteria for the system to be developed are mentioned in one contract by the customer. When the system is ready, it can be tested against the acceptance criteria contracted with the customer. This is called acceptance testing. Thus, acceptance testing can be defined as the process of comparing the final system with the needs of the customer as agreed on the acceptance criteria.

➔ Thus, acceptance testing is designed to:
A. Determine whether the software is fit for the user.
B. Making users confident about the product.
C. Determine whether a software system satisfies its acceptance criteria.
D. Enable the buyer to determine whether to accept the system or not.

➔ The development team and the customer should work together and make sure that they:
A. Identify interim and final products for acceptance, acceptance criteria, and schedule.
B. Plan how and by whom each acceptance activities will be performed.
C. Schedule adequate time for the customer to examine and review the product.
D. Prepare the acceptance plan.
E. Perform formal acceptance testing at delivery.
F. Make a decision based on the results of acceptance testing.

➔ Entry Criteria
A. System testing is complete and defects identified are either fixed or documented.
B. Acceptance plan is prepared and resources have been identified.
C. Test environment for the acceptance testing is available.

➔ Exit Criteria
A. Acceptance decisions are made for the software.
B. In case of any warning, the development team is notified.

Module 2: Testing Techniques

2.1

- *Static Testing:*

➔ Static testing reveals the errors which are not shown by dynamic testing.
➔ Static testing has proved to be a cost-effective technique of error detection.
➔ Benefits of adopting a static testing approach:

A. As defects are found and fixed, the quality of the product increases.
B. A more technically correct base is available for each new phase of development.
C. The overall software life cycle cost is lower, since defects are found early and are easier and less expensive to fix.
D. The effectiveness of the dynamic test activity is increased and less time needs to be devoted for testing the product.
E. Immediate evaluation and feedback to the author from his/her peers which will bring about improvements in the quality of future products.

➔ The objectives of static testing can be summarized as follows:

A. To identify errors in any phase of SDLC as early as possible
B. To verify that the components of software are in conformance with its requirements
C. To provide information for project monitoring
D. To improve the software quality and increase productivity

- *Inspections:*

➔ Inspection is a formal review where people external to the testing team may be involved as inspectors. They are subject matter experts who review the work product.
➔ In this:

A. Thorough preparation is required before an inspection/review
B. Enlisting multiple diverse views.
C. Assigning specific roles to the multiple participants
D. Going sequentially through the code in a structured manner.

➔ Inspection Team:

A. Author/Owner/Producer: A programmer or designer responsible for producing the program or document.
B. Inspector: He finds errors, omissions, and inconsistencies in programs and documents.
C. Moderator: A team member who manages the whole inspection process. He schedules, leads, and controls the inspection session.
D. Recorder: One who records all the results of the inspection meeting.

➔ Inspection Process:

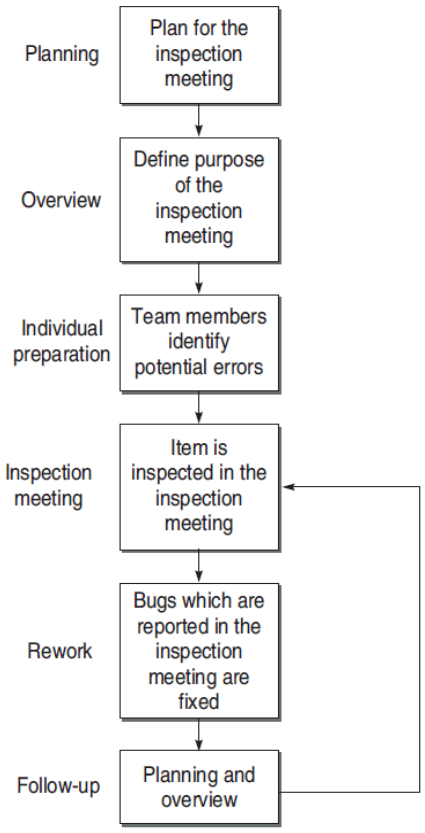


Figure 6.1 Inspection process

A. Planning:

 The product to be inspected is identified.

 A moderator is assigned.

 The objective of the inspection is stated, i.e. whether the inspection is to be conducted for defect detection or something else.

Moderator performs the following activities:

 Assures that the product is ready for inspection

 Selects the inspection team and assigns their roles

 Schedules the meeting venue and time

 Distributes the inspection material like the item to be inspected, checklists, etc.

B. (Rest understand from diagram)

➔ Inspection Process:

A. Bug reduction
B. Bug prevention

C. Productivity
D. Real-time feedback to software engineers
E. Reduction in development resource
F. Quality improvement
G. Project management
H. Checking coupling and cohesion
I. Learning through inspection
J. Process improvement :Finding most error-prone modules, Distribution of error-types

➔ Effectiveness Of Inspection Process:

The effectiveness of the inspection process lies in its rate of inspection. The rate of inspection refers to how much evaluation of an item has been done by the team.

$$\text{Error detection efficiency} = \frac{\text{Error found by an inspection}}{\text{Total errors in the item before inspection}} \times 100$$

➔ Variants of Inspection Process:

A. Active Design Reviews:

 Several reviews are conducted targeting a particular type of bugs and conducted by the reviewers who are experts in that area.

 It covers all the sections of the design document based on several small reviews instead of only one inspection.

Figure 6.2 Active design reviews process

B. Formal Technical Asynchronous Review Method (FTArm):

 Inspection process is carried out without really having a meeting of the members. This is a type of asynchronous inspection in which the inspectors never have to simultaneously meet.

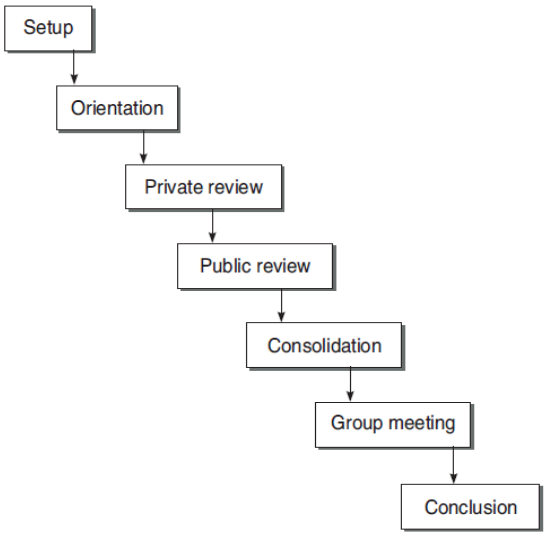


Figure 6.3 Asynchronous method

C. Gilb Inspection:

Three different roles are defined in this type of inspection:

 Leader is responsible for planning and running the inspection.

 Author of the document

 Checker is responsible for finding and reporting the defects in the document.

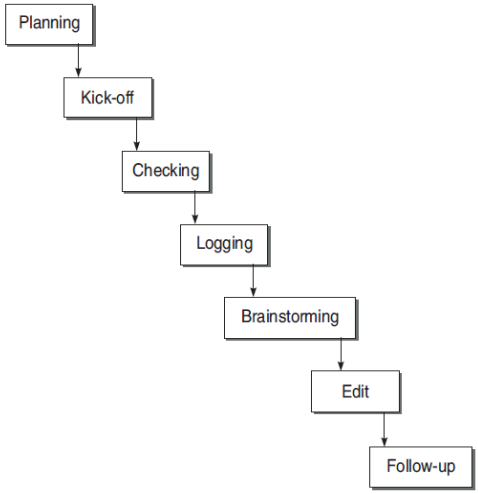


Figure 6.4 Gilb inspection process

D. Humphrey's Inspection Process:

 Preparation phase emphasizes the finding and logging of bugs, unlike Fagan inspections. It also includes an analysis phase wherein individual logs areanalyzed and combined into a single list.

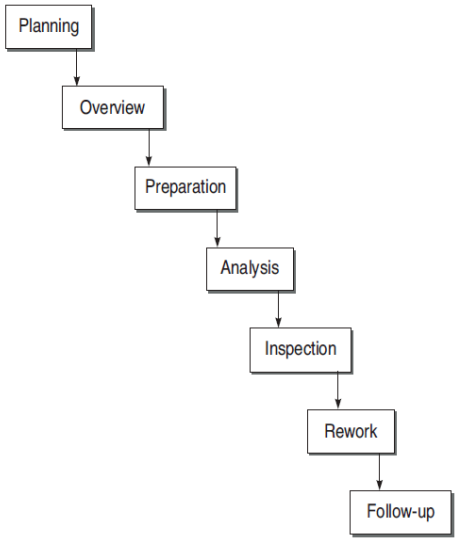


Figure 6.5 Humphrey's process

E. N-Fold inspections:

 Inspection process's effectiveness can be increased by replicating it by having multiple inspection teams.

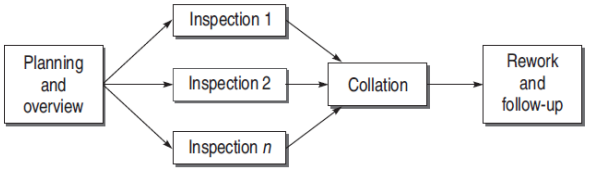


Figure 6.6 N-fold inspection

F. Phased Inspection:

 Phased inspections are designed to verify the product in a particular domain by experts in that domain only.

Single inspector: In this phase, a rigorous checklist is used by a single inspector to verify whether the features specified are there in the item to be inspected.

Multiple inspector: There are many inspectors who are distributed the required documents for verification of an item. Each inspector examines this information and develops a list of questions of their own based on a particular feature.

G. Structured Walkthrough:

 Described by Yourdon. Less formal and rigorous than formal inspections.Roles are coordinator, scribe, presenter, reviewers, maintenance oracle, standard bearer, user representative. Process steps are Organization, Preparation, Walkthrough, and Rework. Lacks data collection requirements of formal inspections.

➔ Reading Techniques:

A. Ad hoc method: The word ad hoc only refers to the fact that no technical support on how to detect defects in a software artifact is given to them.
B. Checklists: A checklist is a list of items that focus the inspector's attention on specific topics, such as common defects or organizational rules, while reviewing a software document
C. Scenario-based reading: Scenario-based reading is another reading technique which stresses on finding different kinds of defects.

 Perspective-based reading: The perspectives mainly depend upon the roles people have within the software development or maintenance process. For each perspective, either one or multiple scenarios are defined, consisting of repeatable activities an inspector has to perform, and the questions an inspector has to answer

 Usage-based reading: use-cases are the basis of inspection, the focus is on finding functional defects, which are relevant from the users' point of view.

 Abstraction driven reading: the inspector creates an abstraction level specification based on the code under inspection to ensure that the inspector has really understood the code.

 Task-driven reading: the inspector has to create a data dictionary, a complete description of the logic and a cross-reference between the code and the specifications.

 Function-point based scenarios: FPA defines a software system in terms of its inputs, files, inquiries, and outputs. The scenarios designed around function-points are known as function-point scenarios.

➔ Checklists for Inspections Process:

Checklists can be prepared with the points mentioned in the verification of each phase. Checklists should be prepared in consultation with experienced staff and regularly updated as more experience is gained by the inspection process.

- *Technical Reviews*:

➔ A technical review is intended to evaluate the software in the light of development standards, guidelines, and specifications and to provide the management with evidence that the development process is being carried out according to the stated objectives.

➔ A technical review team is generally composed of management-level representatives and project management. Review agendas should focus less on technical issues and more on oversight than an inspection. The purpose is to evaluate the system relative to specifications and standards, recording defects and deficiencies. The moderator should

gather and distribute the documentation to all team members for examination before the review.

➔ Set of indicators to measure the following points:

A. Appropriateness of the problem definition and requirements
B. Adequacy of all underlying assumptions
C. Adherence to standards
D. Consistency
E. Completeness
F. Documentation

- *Structured Walkthroughs*:

A walkthrough is less formal, has fewer steps and does not use a checklist to guide or a written report to document the team's work. Rather than simply reading the program or using error checklists, the participants 'play computer'. The person designated as a tester comes to the meeting armed with a small set of paper test cases—representative sets of inputs and expected outputs for the program or module. During the meeting, each test case is mentally executed.

A typical structured walkthrough team consists of the following members:

A. Coordinator: Organizes, moderates, and follows up the walkthrough activities.
B. Presenter/Developer: Introduces the item to be inspected. This member is optional.
C. Scribe/Recorder: Notes down the defects found and suggestions proposed by the members.
D. Reviewer/Tester: Finds the defects in the item.
E. Maintenance Oracle: Focuses on long-term implications and future maintenance of the project.
F. Standards Bearer: Assesses adherence to standards.
G. User Representative/Accreditation Agent: Reflects the needs and concerns of the user

2.2

- *Dynamic Testing:*

➔ Dynamic testing is a software testing technique that involves the execution of the software to evaluate its behavior and performance while it is running. Unlike static testing, which focuses on examining the code or documentation without executing the program, dynamic testing involves interacting with the software in a live environment. This type of testing is essential for assessing various aspects of software quality, including functionality, performance, reliability, and security.

➔ Benefits of Dynamic Testing:

A. Identifying Bugs: Helps in finding and fixing defects and errors in the software.
B. Validation: Verifies that the software meets its intended requirements and functions correctly.

C. Quality Assurance: Ensures that the software is of high quality and meets user expectations.
D. Performance Assessment: Evaluates how the software performs under different conditions.
E. Security Assessment: Identifies vulnerabilities and security weaknesses in the software.

➔ Challenges of Dynamic Testing:

A. Test Data: Generating realistic test data can be challenging, especially for complex systems.
B. Environment Setup: Creating and configuring test environments that mimic the production environment accurately.
C. Cost and Time: Dynamic testing can be time-consuming and costly, especially when extensive testing is required.
D. Test Coverage: Ensuring that all critical parts of the software are adequately tested can be a challenge.

- *White Box Testing:*

➔ White-box testing is another effective testing technique in dynamic testing. It is also known as glass-box testing, as everything that is required to implement the software is visible.

➔ The entire design, structure, and code of the software have to be studied for this type of testing. It is obvious that the developer is very close to this type of testing. Often, developers use white-box testing techniques to test their own design and code. This testing is also known as structural or development testing.

➔ Need for white box testing:

A. white-box testing techniques are used for testing the module for initial stage testing.
B. Since white-box testing is complementary to black-box testing, there are categories of bugs which can be revealed by white-box testing, but not through black-box testing
C. Errors which have come from the design phase will also be reflected in the code, therefore we must execute white-box test cases for verification of code
D. We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis. White-box testing explores these paths too.
E. Some typographical errors are not observed and go undetected and are not covered by black-box testing techniques. White-box testing techniques help detect these errors

- *Basis Path Testing:*

➔ Basis path testing is the technique of selecting the paths that provide a basis set of execution paths through the program. The guidelines for effectiveness of path testing are:

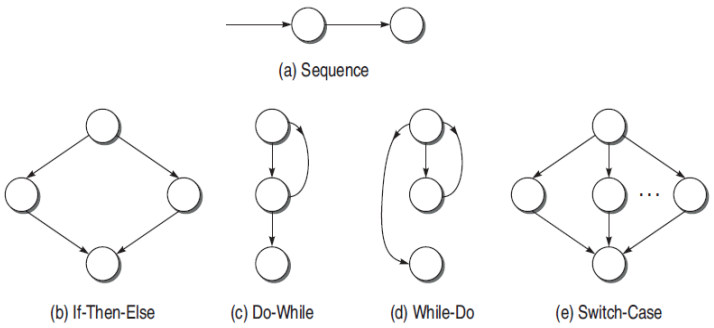
A. Path testing is based on the control structure of the program for which flow graph is prepared.
B. Path testing requires complete knowledge of the program's structure.

C. Path testing is closer to the developer and used by him to test hismodule.
D. The effectiveness of path testing gets reduced with the increase in size of software under test.
E. Choose enough paths in a program such that maximum logic coverage is achieved.

➔ Control Flow Graph: graphical representation of control structure of a program. Flow graphs can be prepared as a directed graph. A directed graph (V, E) consists of a set of vertices V and a set of edges E that are ordered pairs of elements of V.

A. Node: It represents one or more procedural statements. The nodes are denoted by a circle. These are numbered or labeled.
B. Edges or links: They represent the flow of control in a program. This is denoted by an arrow on the edge. An edge must terminate at a node.
C. Decision node: A node with more than one arrow leaving it is called a decision node.
D. Junction node: A node with more than one arrow entering it is called a junction.
E. Regions: Areas bounded by edges and nodes are called regions. When counting the regions, the area outside the graph is also considered a region.

➔ Flow Graph Notations for Different PRogramming Constructs:



➔ Path Testing Terminology:

A. Path: A path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same, junction, decision, or exit.
B. Segment: Paths consist of segments. The smallest segment is a link, that is, a single process that lies between two nodes A direct connection between two nodes, as in an unconditional GOTO, is also called a process by convention, even though no actual processing takes place.
C. Path segment: A path segment is a succession of consecutive links that belongs to some path.
D. Length of a path: The length of a path is measured by the number of links in it and not by the number of instructions or statements executed along the path.
E. Independent path: An independent path is any path through the graph that introduces at least one new set of processing statements or new conditions. An independent path must move along at least one edge that has not been traversed before the path is defined.

➔ Cyclomatic Complexity:

Cyclomatic complexity number can be derived through any of the following three formulae

1. $V(G) = e - n + 2p$
   where e is number of edges, n is the number of nodes in the graph, and p is number of components in the whole graph.
2. $V(G) = d + p$
   where d is the number of decision nodes in the graph.
3. $V(G)$ = number of regions in the graph

Calculating the number of decision nodes for Switch-Case/Multiple If-Else When a decision node has exactly two arrows leaving it, then we count it as a single decision node. However, switch-case and multiple if-else statements have more than two arrows leaving a decision node, and in these cases, the formula to calculate the number of nodes is

$d = k - 1$, where k is the number of arrows leaving the node.

Calculating the cyclomatic complexity number of the program having many connected components Let us say that a program P has three components: X, Y, and Z. Then we prepare the flow graph for P and for components, X, Y, and Z. The complexity number of the whole program is

$$V(G) = V(P) + V(X) + V(Y) + V(Z)$$

We can also calculate the cyclomatic complexity number of the full program with the first formula by counting the number of nodes and edges in all the components of the program collectively and then applying the formula

$$V(G) = e - n + 2P$$

A. The following steps should be followed for designing test cases using path testing:

   - Draw the flow graph using the code provided for which we have to write test cases.
   - Determine the cyclomatic complexity of the flow graph.
   - Cyclomatic complexity provides the number of independent paths. Determine a basis set of independent paths through the program control structure.
   - The basis set is in fact the base for designing the test cases. Based on every independent path, choose the data such that this path is executed.

- *Loop Testing*

➔ Loop Testing is a type of software testing type that is performed to validate the loops. It is one type of Control Structure Testing. Loop testing is a white box testing technique and is used to test loops in the program.

➔ The objective of Loop Testing is:

   A. To fix the infinite loop repetition problem.
   B. To know the performance.
   C. To identify the loop initialization problems.
   D. To determine the uninitialized variables.

➔ Types of Loop testing:

A. Simple Loop Testing: Testing performed in a simple loop is known as Simple loop testing. Simple loop is basically a normal “for”, “while” or “do-while” in which a condition is given and the loop runs and terminates according to the true and false occurrence of the condition respectively. This type of testing is performed basically to test the condition of the loop whether the condition is sufficient to terminate the loop after some point of time.
B. Nested Loop testing: Testing performed in a nested loop is known as Nested loop testing. Nested loop is basically one loop inside another loop. In a nested loop there can be a finite number of loops inside a loop and there a nest is made. It may be either of any of three loops i.e., for, while or do-while.
C. Concatenated Loop Testing: Testing performed in a concatenated loop is known as Concatenated loop testing. It is performed on the concatenated loops. Concatenated loops are loops after the loop. It is a series of loops. Difference between nested and concatenated is that in the nested loop is inside the loop but here the loop is after the loop.
D. Unstructured Loop Testing: Testing performed in an unstructured loop is known as Unstructured loop testing. Unstructured loop is the combination of nested and concatenated loops. It is basically a group of loops that are in no order.

- *Data Flow Testing*

➔ Data flow testing is a white-box testing technique that can be used to detect improper use of data values due to coding errors.

➔ State of data object:

A. Defined (d): A data object is called defined when it is initialized, i.e. when it is on the left side of an assignment statement.
B. Killed/Undefined/Released (k): When the data has been reinitialized or the scope of a loop control variable finishes, i.e. exiting the loop or memory is released dynamically or a file has been closed.
C. Usage (u): When the data object is on the right side of assignment or used as a control variable in a loop, or in an expression used to evaluate the control flow of a case statement, or as a pointer to an object, etc. In general, we say that the usage is either computational use (c-use) or predicate use (p-use).

➔ Data flow Anomalies:

Table 5.1 Two-character data-flow anomalies

Anomaly	Explanation	Effect of Anomaly
du	Define-use	Allowed. Normal case.
dk	**Define-kill**	**Potential bug. Data is killed without use after definition.**
ud	Use-define	Data is used and then redefined. Allowed. Usually not a bug because the language permits reassignment at almost any time.
uk	Use-kill	Allowed. Normal situation.
ku	**Kill-use**	**Serious bug because the data is used after being killed.**
kd	Kill-define	Data is killed and then redefined. Allowed.
dd	**Define-define**	**Redefining a variable without using it. Harmless bug, but not allowed.**
uu	Use-use	Allowed. Normal case.
kk	**Kill-kill**	**Harmless bug, but not allowed.**

~x : indicates all prior actions are not of interest to x.

x~ : indicates all post actions are not of interest to x.

Table 5.2 Single-character data-flow anomalies

Anomaly	Explanation	Effect of Anomaly
~d	First definition	Normal situation. Allowed.
~u	**First Use**	**Data is used without defining it. Potential bug.**
~k	**First Kill**	**Data is killed before defining it. Potential bug.**
D~	**Define last**	**Potential bug.**
U~	Use last	Normal case. Allowed.
K~	Kill last	Normal case. Allowed.

➔ Terminology used in Data flow testing:

A. Definition node: Defining a variable means assigning value to a variable for the very first time in a program.
B. Usage node: It means the variable has been used in some statement of the program. Node n that belongs to G(P) is a usage node of variable v, if the value of variable v is used at the statement corresponding to node n.Two types of usage nodes: Predicate Usage Node, Computation Usage Node
C. Loop-free path segment: It is a path segment for which every node is visited once at most.
D. Simple path segment: It is a path segment in which at most one node is visited twice. A simple path segment is either loop-free or if there is a loop, only one node is involved.
E. Definition-use path (du-path): A du-path with respect to a variable v is a path between the definition node and the usage node of that variable. Usage node can either be a p-usage or a c-usage node
F. Definition-clear path(dc-path): A dc-path with respect to a variable v is a path between the definition node and the usage node such that no other node in the path is a defining node of variable v.

➔ Static Data Flow Testing:

➔ Dynamic Data Flow Testing:

A. Dynamic data flow testing is performed with the intention to uncover possible bugs in data usage during the execution of the code.
B. Strategies:

 All-du Paths (ADUP): It states that every du-path from every definition of every variable to every use of that definition should be exercised under some test. It is the strongest data flow testing strategy, since it is a superset of all other data flow testing strategies.

 All-uses (AU): This states that for every use of the variable, there is a path from the definition of that variable (nearest to the use in backward direction) to the use.

 All-p-uses/Some-c-uses (APU + C): This strategy states that for every variable and every definition of that variable, include at least one dc-path from the definition to every predicate use.

 All-c-uses/Some-p-uses (ACU + P): This strategy states that for every variable and every definition of that variable, include at least one dc-path from the definition to every computational use.

 All-Predicate-Uses (APU): It is derived from the APU+C strategy and states that for every variable, there is a path from every definition to every p-use of that definition.

 All-Computational-Uses (ACU): It is derived from the ACU+P strategy and states that for every variable, there is a path from every definition to every c-use of that definition.

 All-Definition (AD): It states that every definition of every variable should be covered by at least one use of that variable, be that a computational use or a predicate use.

- *Mutation Testing:*

➔ Construct the mutants of a test program.
➔ Add test cases to the mutation system and check the output of the program on each test case to see if it is correct.
➔ If the output is incorrect, a fault has been found and the program must be modified and the process restarted.
➔ If the output is correct, that test case is executed against each live mutant.
➔ If the output of a mutant differs from that of the original program on the same test case, the mutant is assumed to be incorrect and is killed.
➔ After each test case has been executed against each live mutant, each remaining mutant falls into one of the following two categories.

A. One, the mutant is functionally equivalent to the original program. An equivalent mutant always produces the same output as the original program, so no test case can kill it.

B. Two, the mutant is killable, but the set of test cases is insufficient to kill it. In this case, new test cases need to be created, and the process iterates until the test set is strong enough to satisfy the tester.
C. The mutation score for a set of test data is the percentage of non-equivalent mutants killed by that data. If the mutation score is 100%, then the test data is called mutation-adequate.

- *Black Box Testing:*

➔ Black-box technique is one of the major techniques in dynamic testing for designing effective test cases. This technique considers only the functional requirements of the software or module. In other words, the structure or logic of the software is not considered. Therefore, this is also known as functional testing. The software system is considered as a black box, taking no notice of its internal structure, so it is also called as black-box testing technique.

➔ Black-box testing attempts to find errors in the following categories:

A. To test the modules independently
B. To test the functional validity of the software so that incorrect or missing functions can be recognized
C. To look for interface errors
D. To test the system behavior and check its performance
E. To test the maximum load or stress on the system
F. To test the software such that the user/customer accepts the system within defined acceptable limits

- *Boundary Value Analysis:*

➔ BVA is considered a technique that uncovers the bugs at the boundary of input values. Here, boundary means the maximum or minimum value taken by the input domain.

➔ Boundary Value Checking:(4n + 1)

The variable at its extreme value can be selected at:

A. Minimum value (Min)
B. Value just above the minimum value (Min+)
C. Maximum value (Max)
D. Value just below the maximum value (Max−)

➔ Robustness Testing Method:(6n + 1)

The idea of BVC can be extended such that boundary values are exceeded as:

A. A value just greater than the Maximum value (Max+)
B. A value just less than Minimum value (Min−)

When test cases are designed considering the above points in addition to BVC, it is called robustness testing.

➔ Worst-Case Testing Method:

We can again extend the concept of BVC by assuming more than one variable on the boundary. It is called the worst-case testing method.

It can be generalized that for n input variables in a module, 5n test cases can be designed with worst-case testing.

(DO EXAMPLES)

- *Equivalence Class Testing:*

➔ Equivalence partitioning is a method for deriving test cases wherein classes of input conditions called equivalence classes are identified such that each member of the class causes the same kind of processing and output to occur. Thus, instead of testing every input, only one test case from each partitioned class can be executed. It means only one test case in the equivalence class will be sufficient to find errors.

➔ Equivalence partitioning method for designing test cases has the following goals:

A. Completeness Without executing all the test cases, we strive to touch the completeness of the testing domain.
B. Non-redundancy When the test cases are executed having inputs from the same class, then there is redundancy in executing the test cases.

➔ Identification of Equivalent test cases:

A. Valid equivalence classes
B. Invalid equivalence classes
C. Guidelines:
    - ☐ Range Splitting: If there's reason to believe that different parts of the input range will be treated differently, split the range into two or more equivalence classes.
    - ☐ Handling Each Valid Input Differently: If the program handles each valid input differently, define one valid equivalence class per valid input.
    - ☐ Boundary Value Analysis (BVA): Use BVA to identify equivalence classes, including valid ranges and values that cross minimum and maximum boundaries.
    - ☐ Multiple Categories: If an input variable can belong to multiple categories, create equivalence classes for each category.
    - ☐ Range Constraints: If the requirements specify a range for the number of inputs, define valid and invalid classes accordingly, considering cases within, below, and above the specified range.
    - ☐ 'Must Be' Conditions: For conditions that specify a requirement, create valid and invalid equivalence classes based on whether the condition is met.
    - ☐ Output Equivalence Classes: Equivalence classes can also be defined for the desired output of the program. Generate test cases to ensure the output falls within the defined output equivalence class.
    - ☐ Membership in a Set or Group: If an input must be a member of a set or group, define valid and invalid classes based on membership.

- ☐ Multiple Elements Handling: When different elements within an equivalence class are handled differently, split the class into subclasses to represent these variations.

D. Identifying Test case:

- *Decision Table Based Testing:*

➔ Decision table is another useful method to represent the information in a tabular method. It has the specialty to consider complex combinations of input conditions and resulting actions. Decision tables obtain their power from logical expressions. Each operand or variable in a logical expression takes on the value, TRUE or FALSE.

➔ Formation of decision table:

A. Condition stub It is a list of input conditions for which the complex combination is made.
B. Action stub It is a list of resulting actions which will be performed if a combination of input conditions is satisfied.
C. Condition entry It is a specific entry in the table corresponding to input conditions mentioned in the condition stub. When we enter TRUE or FALSE for all input conditions for a particular combination, then it is called a Rule.
D. Action entry It is the entry in the table for the resulting action to be performed when one rule (which is a combination of input conditions) is satisfied. 'X' denotes the action entry in the table.

➔ Test Case design using Decision Table:

A. Interpret condition stubs as the inputs for the test case.
B. Interpret action stubs as the expected output for the test case.
C. Rule, which is the combination of input conditions, becomes the test case itself.
D. If there are k rules over n binary conditions, there are at least k test cases and at the most 2n test cases.

Table 4.3 Decision table structure

		ENTRY				
		Rule 1	**Rule 2**	**Rule 3**	**Rule 4**	...
Condition Stub	C1	True	True	False	I	
	C2	False	True	False	True	
	C3	True	True	True	I	
Action Stub	A1		X			
	A2	X			X	
	A3			X		

➔ Expanding the Immaterial Cases in Decision Table:

- *Usability and Accessibility Testing:*

Usability Testing

This type of system testing is related to a system's presentation rather than its functionality. System testing can be performed to find human-factors or usability problems on the system. The idea is to adapt the software to users' actual work styles rather than forcing the users to adapt according to the software. Thus, the goal of usability testing is to verify that intended users of the system are able to interact properly with the system while having a positive and convenient experience. *Usability testing identifies discrepancies between the user interfaces of a product and the human engineering requirements of its potential users.*

What the user wants or expects from the system can be determined using several ways as proposed by Dustin [12]:

Area experts The usability problems or expectations can be best understood by the subject or area experts who have worked for years in the same area. They analyse the system's specific requirements from the user's perspective and provide valuable suggestions.

Group meetings Group meeting is an interesting idea to elicit usability requirements from the user. These meetings result in potential customers' comments on what they would like to see in an interface.

Surveys Surveys are another medium to interact with the user. It can also yield valuable information about how potential customers would use a software product to accomplish their tasks.

Analyse similar products We can also analyse the experiences in similar kinds of projects done previously and use it in the current project. This study will also give us clues regarding usability requirements.

Usability characteristics against which testing is conducted are discussed below:

Ease of use The users enter, navigate, and exit with relative ease. Each user interface must be tailored to the intelligence, educational background, and environmental pressures of the end-user.

Interface steps User interface steps should not be misleading. The steps should also not be very complex to understand either.

Response time The time taken in responding to the user should not be so high that the user is frustrated or will move to some other option in the interface.

Help system A good user interface provides help to the user at every step. The help documentation should not be redundant; it should be very precise and easily understood by every user of the system.

Error messages For every exception in the system, there must be an error message in text form so that users can understand what has happened in the system. Error messages should be clear and meaningful.

An effective tool in the development of a usable application is the user-interface prototype. This kind of prototype allows interaction between potential users, requirements personnel, and developers to determine the best approach to the system's interface. Prototypes are far superior because they are interactive and provide a more realistic preview of what the system will look like.

➔ Accessibility Testing is one of the Software Testing, in which the process of testing the degree of ease of use of a software application for individuals with certain disabilities. It is performed to ensure that any new component can easily be accessible by physically disabled individuals despite any respective handicaps. Accessibility testing is part of the system testing process and is somehow similar to usability testing. In the accessibility testing process, the tester uses the system or component as it would be used by individuals with disabilities. Individuals can have disabilities like visual disability, hearing disability, learning disability, or non-functional organs. Accessibility testing is a subset of usability testing where the users under consideration are specific people with disabilities.

2.3

- *Regression Testing:*

➔ Regression testing is the selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.

➔ Regression testability refers to the property of a program, modification, or test suite that lets it be effectively and efficiently regression-tested.

- *Objectives:*

➔ It tests to check that the bug has been addressed: The first objective in bugfix testing is to check whether the bug-fixing has worked or not

➔ If finds other related bugs: It may be possible that the developer has fixed only the symptoms of the reported bugs without fixing the underlying bug.

➔ It tests to check the effect on other parts of the program: It may be possible that bug-fixing has unwanted consequences on other parts of a program.

- *Needs and Types:*

➔ Software Maintenance
   A. Corrective maintenance Changes made to correct a system after a failure has been observed (usually after general release).
   B. Adaptive maintenance Changes made to achieve continuing compatibility with the target environment or other systems.
   C. Perfective maintenance Changes designed to improve or add capabilities
   D. Preventive maintenance Changes made to increase robustness, maintainability, portability, and other features.

➔ Rapid Iterative Development: The extreme programming approach requires that a test be developed for each class and that this test be re-run every time the class changes.

➔ First Step of Integration: Re-running accumulated test suites, as new components are added to successive test configurations, builds the regression suite incrementally and reveals regression bugs.

➔ Compatibility Assessment and Benchmarking: Some test suites are designed to be run on a wide range of platforms and applications to establish conformance with a standard or to evaluate time and space performance.

➔ Types:
   A. Bug-Fix regression This testing is performed after a bug has been reported and fixed. Its goal is to repeat the test cases that expose the problem in the first place.
   B. Side-Effect regression/Stability regression It involves retesting a substantial part of the product. The goal is to prove that the change has no detrimental effect on something that was earlier in order. It tests the overall integrity of the program, not the success of software fixes.

Module 3: Testing Metrics

3.1

- Measurement Objectives:
  A measurement program will be more successful, if it is designed with the goals of the project in mind. For this purpose, we can take the help of a goal question metric (GQM). The GQM approach is based on the fact that the objectives of measurement should be clear, much before the data collection begins. According to this approach, the objectives of a measurement process should be identified first. The GQM approach, with reference to test process measurement, provides the following framework:
  - Lists the major goals of the test process.
  - Derives from each goal, the questions that must be answered to determine if the goals are being met.
  - Decides what must be measured in order to answer the questions adequately.

- Attributes and Corresponding metrics:
  An organization needs to have a reference set of measurable attributes and corresponding metrics that can be applied at various stages during the course of execution of an organization-wide testing strategy. The attributes to be measured depend on the following factors:
  - Time and phase in the software testing life cycle.
  - New business needs.
  - Ultimate goal of the project.

Table 11.1 Attribute categories

Category	Attributes to be Measured
Progress	• Scope of testing • Test progress • Defect backlog • Staff productivity • Suspension criteria • Exit criteria
Cost	• Testing cost estimation • Duration of testing • Resource requirements • Training needs of testing group and tool requirement • Cost-effectiveness of automated tool
Quality	• Effectiveness of test cases • Effectiveness of smoke tests • Quality of test plan • Test completeness
Size	• Estimation of test cases • Number of regression tests • Tests to automate

PROGRESS:

1. Scope of testing:

❖ It helps in estimating the overall amount of work involved, by documenting which parts of the software are to be tested thereby estimating the overall testing effort required.

❖ It also helps in identifying the testing types that are to be used for covering the features under test.

2. Test progress:

❖ For measuring the test progress, well-planned test plans are prepared wherein testing milestones are planned. Each milestone is scheduled for completion during a certain time period in the test plan. These milestones can be used by the test manager to monitor how the testing efforts are progressing.

❖ Measurements need to be available for comparing the planned and actual progress towards achieving the testing milestones.

3. Defect backlog:

❖ The defect backlog metric is to be measured for each release of a software product for release-to-release comparisons. One way of tracking it is to use defect removal percentage for a prior release so that decisions regarding additional testing can be wisely taken in the current release.

4. Staff productivity:

❖ Measurement of testing staff productivity helps to improve the contributions made by the testing professionals towards a quality testing process.

❖ Time spent in test planning.

- Time spent in test case design.
- Number of test cases developed.
- Number of test cases developed per unit time.

5. Suspension criteria:

❖ The suspension criteria of testing describes in advance that the occurrence of certain events/conditions will cause the testing to stop temporarily. With the help of suspension metrics, we can save precious testing time by raising the conditions that suspend testing.

6. Exit criteria:

❖ The exit criteria determine the termination of testing effort which must be communicated to the development team prior to the approval of the test plan.

❖ The following metrics are studied for the exit criteria of testing:

- Rate of fault discovery in regression tests.
- Frequency of failing fault fixes.
- Fault detection rate.

BAAKI BAATE UNDERSTOOD HAI

QUALITY:

There are several measures based on faults to check the effectiveness of test cases. Some of them are discussed here:

1. Number of faults found in testing.
2. Number of failures observed by the customer which can be used as a reflection of the effectiveness of test cases.
3. Defect-removal efficiency is another powerful metric for test-effectiveness, which is defined as the ratio of the number of faults actually found in testing and the number of faults that could have been found in testing. There are potential issues that must be taken into account while measuring the defect-removal efficiency. For example, the severity of bugs and an estimate of time by which the customers would have discovered most of the failures are to be established. This metric is more helpful in establishing the test effectiveness in the long run as compared to the current project.
4. Defect age is another metric that can be used to measure the test effectiveness, which assigns a numerical value to the fault, depending on the phase in which it is discovered. Defect age is used in another metric called defect spoilage to measure the effectiveness of defect-removal activities. Defect spoilage is calculated as:

$$\text{Spoilage} = \frac{\text{Sum of (Number of defects Defect age)}}{\text{Total number of defects}}$$

Dimension	Rating		
	Excellent	**Average**	**Poor**
Theory of Objective	Identification of realistic test objectives to have the most efficient test cases.	Describes somewhat credible test objectives.	No objectives or irrelevant objectives.
Theory of Scope	Specific and unambiguous test scope.	Some scope of testing understood by only some people.	Wrong scope assumptions.
Theory of Coverage	The test coverage is completely relative to the test scope.	The test coverage is somewhat related to the test scope.	Coverage is not related to test scope or objective.
Theory of Risk	Identifies possible testing risks.	Risks have inappropriate priorities.	No understanding of project risk.
Theory of Data	Efficient method to generate enough valid and invalid test data.	Some method to generate test data.	There is no method of capturing data.

Theory of Originality	High information in test plan.	A test plan template has some original content.	Template has not been filled in.
Theory of Communication	Multiple modes to communicate test plan to, and receive feedback from appropriate stakeholders.	Less effective feedback.	No distribution, no opportunity for feedback
Theory of Usefulness	Effective test plan such that it discovers critical bugs early.	Critical bugs are not discovered earlier.	Test plans are not useful to the organization.
Theory of Completeness	Enough testing through the plan.	Most test items have been identified.	Enough testing criteria has not been defined.
Theory of Insightfulness	Understanding of what is interesting and challenging in testing this specific project.	The test plan is effective, but weak.	No insight through the plan.

❖ Measuring test completeness:

The advantages of measuring test coverage are that it provides the ability to design new test cases and improve existing ones. There are two issues: (i) whether the test cases cover all possible output states and (ii) the adequate number of test cases to achieve test coverage. The relationship between code coverage and the number of test cases is described by the following expression:

$$C(x) = 1 - e^{-(p/N) * x}$$

where C(x) is the coverage after executing x number of test cases, N is the number of blocks in the program, and p is the average number of blocks covered by a test case during the function test.

- Estimation Models for estimating matrices:
    1. Halstead Metrics:
       The program volume describes the number of volumes of information in bits required to specify a program. The program level is a measure of software complexity. Using these definitions, Halstead effort e can be computed as:

       $$PL = 1/[(n1/2) \text{ X } (N\ 2/n\ 2)]$$

       $$e = V/PL$$

       The percentage of overall testing effort to be allocated to a module k can be estimated using the following relationship:

       $$\text{Percentage of testing effort (k)} = e(k)/\Sigma e(i)$$

       where e(k) is the effort required for module k and $\Sigma e(i)$ is the sum of Halstead effort across all modules of the system.

    2. Development Ratio Method:
       Developer to tester ratio: The results of applying this method is dependent on numerous factors including the type and complexity of the software being developed, testing level, scope of testing, test-effectiveness during testing, error tolerance level for testing, and available budget.
       Tester to developer ratio: collects data on various effects like developer-efficiency at removing defects before testing, developer-efficiency at inserting defects, defects found per person, and the value of defects found. After that, an initial estimate is made to calculate the number of testers based upon the ratio of the baseline project. The initial estimate is adjusted using professional experience to show how the above mentioned effects affect the current project and the baseline project.

    3. Project-Staff Ratio Method:
       Project-staff ratio method makes use of historical metrics by calculating the percentage of testing personnel from the overall allocated resources planned for the project.

Project type	Total number of project staff	Test team size %	Number of testers
Embedded system	100	23	23
Application development	100	8	8

    4. Test Procedure Method:
       This model is based on the number of test procedures planned. The number of test procedures decides the number of testers required and the testing time. Thus, the baseline of estimation here is the quantity of test procedures. But you have to do some preparation before actually estimating the resources.

	Number of test procedures (NTP)	Number of person-hours consumed for testing (PH)	Number of hours per test procedure = PH/ NTP	Total period in which testing is to be done (TP)	Number of testers = PH/TP
Historical Average Record	840	6000	7.14	10 months (1600 hrs)	3.7
New Project Estimate	1000	7140	7.14	1856 hrs	3.8

5. Task Planning Method:
   The baseline for estimation is the historical records of the number of personnel hours expended to perform testing tasks. The historical records collect data related to the work break-down structure and the time required for each task so that the records match the testing tasks.
   Refer page 330 and 331 of naresh chauhan.

- Architecture Design Metric Used for Testing:
  1. Structural Complexity:
     It is defined as
     $S(m) = f^{2out} \times (m)$
     where S is the structural complexity and $f_{out}(m)$ is the fan-out of module m.
     This metric gives us the number of stubs required for unit testing of the module m. Thus, it can be used in unit testing.

  2. Data Complexity:
     This metric measures the complexity in the internal interface for a module m and is defined as
     $D(m) = v(m) / [f_{out}(m) + 1]$
     where v(m) is the number of input and output variables that are passed to and from module m.
     This metric indicates the probability of errors in module m. As the data complexity increases, the probability of errors in module m also increases

  3. System Complexity:
     $SC(m) = S(m) + D(m)$
     overall architectural complexity of the system (which is the sum total of system complexities of all the modules) increases with the increase in each module's complexity.
- Information Flow Metrics Used for Testing:
  - **Local direct flow** exists if
    (i) a module invokes a second module and passes information to it.

(ii) the invoked module returns a result to the caller.

 Local indirect flow exists if the invoked module returns information that is subsequently passed to a second invoked module.

 Global flow exists if information flows from one module to another via a global data structure.

The two particular attributes of the information flow can be described as follows:

(i) **Fan-in** of a module m is the number of local flows that terminate at m, plus the number of data structures from which information is retrieved by m.

(ii) **Fan-out** of a module m is the number of local flows that emanate from m, plus the number of data structures that are updated by m.

Henry and Kafura Design Metric:

$IFC(m) = length(m) \times ((fan\text{-}in(m) \times fan\text{-}out(m))^2$

- Cyclomatic Complexity Measures for Testing:
    1. Since the cyclomatic number measures the number of linearly independent paths through flow graphs, it can be used as the set of minimum number of test cases. If the number of test cases is lesser than the cyclomatic number, then you have to search for the missing test cases. In this way, it becomes a thumb rule that a cyclomatic number provides the minimum number of test cases for effective branch coverage.
    2. McCabe has suggested that ideally, the cyclomatic number should be less than or equal to 10. This number provides a quantitative measure of testing difficulty. If the cyclomatic number is more than 10, then the testing effort increases due to the following reasons:

       1. Number of errors increases.
       2. Time required to detect and correct the errors increase.

       Thus, the cyclomatic number is an important indication of the ultimate reliability. The amount of test design and test effort is better approximated by the cyclomatic number as compared to the lines of code (LOC) measure.
- Function Point Metrics for Testing:

  The function point (FP) metric is used effectively for measuring the size of a software system.

  Can be used as a predictor for overall testing effort.

  Few FP measures:

  1. Number of hours required for testing per FP.
  2. Number of FPs tested per person-month.
  3. Total cost of testing per FP.
  4. Defect density measures the number of defects identified across one or more phases of the development project lifecycle and compares that value with the total size of the

system. It can be used to compare the density levels across different lifecycle phases or across different development efforts. It is calculated as:

Number of defects (by phase or in total) / Total number of FPs

5. Test case coverage measures the number of test cases that are necessary to adequately support thorough testing of a development project.
This measure is calculated as

Number of test cases / Total number of FPs

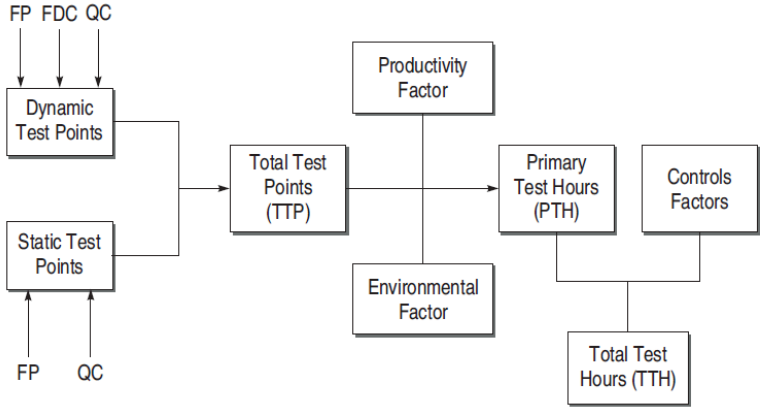
6. Number of test cases in a system can be determined by the function points estimated for the corresponding effort. Formula is:

$$\text{Number of test cases} = (\text{function points})^{1.2}$$

7. Function points can also be used to measure the acceptance test cases. The formula is
Number of test cases = (function points) × 1.2

- Test Point Analysis:
  Test point analysis is a technique to measure the black-box test effort estimation.
  As the test points of the functions are measured, the tester can test the important functionalities first, thereby predicting the risk in testing.
  1. Procedure for Calculating TPA:
     - Dynamic test points are the number of test points which are based on dynamic measurable quality characteristics of functions in the system. Dynamic test points are calculated for every function. To calculate a dynamic function point, we need the following:
       - Function points (FPs) assigned to the function.
       - Function dependent factors (FDC), such as complexity, interfacing, function-importance, etc.
       - Quality characteristics (QC).
     - Static test points are the number of test points which are based on static quality characteristics of the system. It is calculated based on the following:
       - Function points (FPs) assigned to the system.
       - Quality requirements or test strategy for static quality characteristics (QC).
     - After this, the dynamic test point is added to the static test point to get the **total test points (TTP)** for the system. This **total test point is used for calculating primary test hours (PTH)**. PTH is the effort estimation for primary testing activities, such as preparation, specification, and execution. PTH is calculated based on the environmental factors and productivity factors. Environmental factors include development environment, testing environment, testing tools,

etc. Productivity factor is the measure of experience, knowledge, and skills of the testing team. Secondary testing activities include management activities like controlling the testing activities. **Total test hours (TTH) is calculated by adding some allowances to secondary activities and PTH**. Thus, TTH is the final effort estimation for the testing activities.



2. Calculating Dynamic Test Point:
   The number of dynamic test points for each function in the system is calculated as

$$DTP = FP \times FDC_w \times QC_{dw}$$

   where DTP = number of dynamic test points
   FP = function point assigned to function.
   FDC_w = weight-assigned function-dependent factors
   QC_{dw} = quality characteristic factor, wherein weights are assigned to dynamic quality characteristics
   FDC_w is calculated as

$$FDC_w = ((FI_w + UIN_w + I + C) / 20) \times U$$

   where
   FI_w = function importance rated by users
   UIN_w = weights given to usage intensity of the function, i.e. how frequently the function is being used
   I = weights given to the function for interfacing with other functions, i.e. if there is a change in function, how many functions in the system will be affected
   C = weights given to the complexity of function, i.e. how many conditions are in the algorithm of function
   U = uniformity factor

Factor/ Rating	Function Importance (FI)	Function usage Intensity (UIN)	Interfacing (I)	Complexity (C)	Uniformity Factor
Low	3	2	2	3	0.6 for the function, wherein test specifications are largely re-used such as in clone function or dummy function. Otherwise, it is 1.
Normal	6	4	4	6	
High	12	12	8	12	

QC_{dw} is calculated based on four dynamic quality characteristics, namely suitability, security, usability, and efficiency. First, the rating to every quality characteristic is given and then, a weight to every QC is provided. Based on the rating and weights, QC_{dw} is calculated.

$$QC_{dw} = \Sigma \text{ (rating of QC / 4) x weight factor of QC}$$

Characteristic/ Rating	Not Important (0)	Relatively Unimportant (3)	Medium Importance (4)	Very Important (5)	Extremely Important (6)
Suitability	0.75	0.75	0.75	0.75	0.75
Security	0.05	0.05	0.05	0.05	0.05
Usability	0.10	0.10	0.10	0.10	0.10
Efficiency	0.10	0.10	0.10	0.10	0.10

3. Calculating Static Test Points:

The number of static test points for the system is calculated as

$$STP = FP \text{ x } \Sigma QC_{sw} / 500$$

where,

STP = static test point

FP = total function point assigned to the system

QC_{sw} = quality characteristic factor, wherein weights are assigned to static quality characteristics

Static quality characteristic refers to what can be tested with a checklist.

QC_{sw} is assigned the value 16 for each quality characteristic which can be tested statically using the checklist.

$$\text{Total test points (TTP)} = DTP + STP$$

4. Calculating Primary Test Hours:

$$PTH = TTP \times \text{productivity factor} \times \text{environmental factor}$$

Productivity factor It is an indication of the number of test hours for one test point. It is measured with factors like experience, knowledge, and skill set of the testing team. Its value ranges between 0.7 to 2.0. But explicit weightage for testing team experience, knowledge, and skills have not been considered in this metric.

Environmental factor Primary test hours also depend on many environmental factors of the project. Depending on these factors, it is calculated as Environmental factor = weights of (test tools + development testing + test basis + development environment + testing environment + testware)/21

Test tools It indicates the use of automated test tools in the system.

1	Highly automated test tools are used.
2	Normal automated test tools are used.
4	No test tools are used.

Development testing It indicates the earlier efforts made on development testing before system testing or acceptance testing for which the estimate is being done.

2	Development test plan is available and test team is aware about the test cases and their results.
4	Development test plan is available.
8	No development test plan is available.

Test basis It indicates the quality of test documentation being used in the system.

3	Verification as well as validation documentation are available.
6	Validation documentation is available.
12	Documentation is not developed according to standards.

Development environment It indicates the development platforms, such as operating systems, languages, etc. for the system.

2	Development using recent platform.
4	Development using recent and old platform.
8	Development using old platform.

Test environment It indicates whether the test platform is a new one or has been used many times on the systems.

1	Test platform has been used many times.
2	Test platform is new but similar to others already in use.
4	Test platform is new.

Testware It indicates how much testware is available in the system.

1	Testware is available along with detailed test cases.
2	Testware is available without test cases.
4	No testware is available.

5. Calculating Total Test Hours:
   Primary test hours is the estimation of primary testing activities. If we also include test planning and control activities, then we can calculate the total test hours.

   Total test hours = PTH + Planning and control allowance

Team size

3	≤ 4 team members
6	5–10 team members
12	>10 team members

Planning and control tools

2	Both planning and controlling tools are available.
4	Planning tools are available.
8	No management tools are available.

Planning and control allowance (%)
= weights of (team size + planning and control tools)

Planning and control allowance (hours)
= planning and control allowance (%) × PTH

Thus, the total test hours is the total time taken for the whole system. TTH can also be distributed for various test phases, as given in Table 11.16.

Table 11.16 TTH distribution

Testing phase	% of TTH
Plan	10%
Specification	40%
Execution	45%
Completion	5%

- Testing Progress Metrics:

  Test Procedure Execution Status

  It is defined as:

  Test Proc Exec. Status (%) = Number of executed test cases / Total number of test cases

  This metric ascertains the number of percentage of test cases remaining to be executed.

  Defect aging

  Is the turnaround time for a defect to be corrected. This metric is defined as

  Defect aging = Closing date of bug – Start date when bug was opened

  This metric data for various bugs is used for a trend analysis such that it can be known that n number of defects per day can be fixed by the test team.

  Defect Fix Time to Retest

  It is defined as Defect fix time to retest

  = Date of fixing the bug and releasing in new build – Date of retesting the bug

Defect Trend Analysis

It is defined as the trend in the number of defects found as the testing life cycle progresses. For example,

Number of defects of each type detected in unit test per hour

Number of defects of each type detected in integration test per hour

Recurrence Ratio

This metric indicates the quality of bug-fixes. The quality of bug-fixes is good if it does not introduce any new bug in the previous working functionality of the software and the bug does not re-occur.

Formula: Number of bugs remaining per fix

Defect Density

This metric is defined as

$$\text{Defect density} = \frac{\text{Total number of defects found for a requirement}}{\text{Number of test cases executed for that requirement}}$$

Coverage Measures

Coverage goals can be planned, against which the actual ones can be measured. For white-box testing, a combination of following is recommended: degree of statement, branch, data flow, basis path, etc. coverage (planned, actual). In this way, testers can use the following metric:

Actual degree of coverage / Planned degree of coverage

Tester productivity Measures

The following are some useful metrics in measuring the testers' productivity:

1. Time spent in test planning
2. Time spent in test case design
3. Time spent in test execution
4. Time spent in test reporting
5. Number of test cases developed
6. Number of test cases executed

Budget and Resource Monitoring Measures

Don't know

Test case Effectiveness Metric

$$TCE = \frac{Number\ of\ defects\ found\ by\ the\ test\ cases}{Total\ number\ of\ defects} \times 100$$

Module 4: Test Automation and Testing for Specialized Environment

4.1

- *Need of Automation*

  When selecting a testing tool, organizations should consider the following benefits of automation:

  1. Reduction of Testing Effort: Test cases can number in the hundreds of thousands or more, making manual execution time-consuming. Automated testing tools significantly reduce the time required for test execution.

  2. Reduced Tester Involvement: Automating test case execution allows testers to focus on other tasks, increasing parallelism in testing efforts.

  3. Facilitates Regression Testing: Automation simplifies the time-consuming process of regression testing, reducing both testing effort and time compared to manual testing.

  4. Avoids Human Errors: Manual test case execution can introduce errors and biases. Testing tools minimize these issues associated with manual testing.

  5. Reduces Overall Software Cost: Longer testing times increase software costs. Testing tools streamline test case production and execution, resulting in cost savings.

  6. Simulated Testing: Testing tools can simulate real-life scenarios, such as load performance testing, by creating virtual users or data in the test environment. This ensures thorough testing before product release.

  7. Internal Testing: Automated tools expedite tasks like memory leakage testing and coverage checking, which can be cumbersome, inaccurate, and time-consuming when done manually.

  8. Test Enablers: Automation tools provide stubs or drivers to prepare data, simulate environments, make calls, and verify results when some modules for testing are not ready. This reduces effort in such cases.

  9. Test Case Design: Automation tools can be used to design test cases, offering better coverage compared to manual methods.

These benefits highlight the advantages of using testing automation tools to improve efficiency, reduce costs, and enhance the overall quality of the software testing process.

- *Guidelines for Automated Testing*
  The guidelines for automation testing are:

  1. Consider building a tool instead of buying one, if possible: When the requirement is small and resources are available, consider building a tool instead of buying it. Ensure that management commits to budget and resource approvals for this endeavor.

  2. Test the tool on an application prototype: Verify the tool's compatibility with the system under development. If the system isn't available, consider creating a system prototype for tool evaluation.

  3. Not all tests should be automated: While automated testing enhances manual testing, not all tests can be automated. Decide which aspects require automation and prioritize them. Some tests, like verifying printouts, may remain manual.

  4. Select the tools according to organizational needs: Choose automation tools based on your organization's specific needs, resources, and budget, rather than just following trends or competing with other organizations.

  5. Use proven test-script development techniques:
  - Read data values from spreadsheets or tool-provided data pools, avoiding hard-coding data into test-case scripts. This enhances reusability.
  - Implement modular script development for better maintainability and readability.
  - Develop a library of reusable functions by separating common actions into a shared script library.
  - Store all test scripts in a version control tool for better management.

  6. Automate the regression tests whenever feasible: Regression testing can be time-consuming. Automation tools can significantly reduce testing time. Whenever possible, automate regression test cases.

- *Categorization of Testing Tools*
  A variety of testing tools are available to cater to different needs and users. These tools can be categorized as follows:

    - Static and Dynamic Testing Tools:

These tools are classified based on the type of execution of test cases—static and dynamic.

Static Testing Tools: These tools perform static analysis by scanning the source program and identifying possible faults and anomalies. They include static program analyzers that analyze program text, control flow, data use, interface, and paths within the program. They perform different types of static analysis such as:

- Control Flow Analysis: This analysis detects loops with multiple exits and entry points and unreachable code.

- Data Use Analysis: It detects all types of data faults.

- Interface Analysis: It detects all interface faults. It also detects functions that are never declared or called, and function results that are never used.

- Path Analysis: It identifies all possible paths through the program and unravels the program's control.

Dynamic Testing Tools: These tools support dynamic testing activities. They are used to capture the state of events during program execution, provide coverage information, and generate statistics on test cases. It monitors the following activities:

- List the number of times a component is called or a line of code is executed. This information about the statement or path coverage of their test cases is used by testers.

- Report on whether a decision point has branched in all directions, thereby providing information about branch coverage.

- Provide summary statistics that offer a high-level view of the percentage of statements, paths, and branches that have been covered by the collective set of test cases run. This information is important when test objectives are stated in terms of coverage.

- Testing Activity Tools:

These tools are categorized based on the testing activities or tasks during different phases of the software development life cycle (SDLC). Testing activities are categorized as following:

- Tools for Review and Inspections: These tools are designed for static analysis of items. They work with specifications and code, including complexity analysis tools and tools for code comprehension.

- Tools for Test Planning: These tools help in test planning and include templates for test plan documentation, test schedule and staffing estimates, and complexity analyzers.

- Tools for Test Design and Development: These tools are used for test design and development, including test data generators and test case generators.

- Tools for Test Execution and Evaluation: These tools are employed during test execution and evaluation phases. They include capture/playback tools, coverage analysis tools, memory testing tools, test management tools, network testing tools, and performance testing tools.

These tools cater to different aspects of the testing process, from static analysis and planning to dynamic testing activities, helping organizations improve the quality and efficiency of their testing efforts.

- *Selection of Testing Tools*

  The big question is how to select a testing tool. It may depend on several factors. What are the needs of the organization; what is the project environment; what is the current testing methodology; all these factors should be considered when choosing testing tools. Some guidelines to be followed by selecting a testing tool are given below.

  - Match the tool to its appropriate use: Understand the tool's specific use and features before selection.

  - Select the tool for its appropriate SDLC phase: Choose tools that align with the testing phase in the software development life cycle.

  - Select the tool based on the skill of the tester: Pick tools that match the tester's skill level to avoid unnecessary complexity.

- Select an affordable tool: Stick to a tool within the project budget to prevent unnecessary cost increases.

- Determine how many tools are required for testing the system: Assess test requirements and select multiple tools if necessary.

- Select the tool after examining the schedule of testing: Ensure there's enough time for learning and using the tool effectively within the testing schedule.

- *Data Driven Testing*
  Data-Driven Testing (DDT) is a software testing technique in which test scenarios are designed based on various sets of input data, and the same test logic is executed with each set of data. The main idea behind DDT is to separate the test script from the test data, allowing for more extensive testing of an application with minimal changes to the test script. Here are some key points about Data-Driven Testing:

  1. Input Data Separation: Data-Driven Testing (DDT) separates test data from test scripts.
  2. Multiple Test Scenarios: DDT creates various test scenarios by using different data sets.
  3. Reusability: Test scripts remain consistent, allowing for the reuse of the same logic with various data sets.
  4. Scalability: Additional test cases are easily accommodated by adding more data sets.
  5. Automation: DDT is commonly used in automated testing with testing frameworks and tools.
  6. Data Variation: Ideal for testing the same function with various data values.
  7. Data-Related Issues: DDT reveals problems related to data input, validation, and transformations.
  8. Reporting: Test reports include details on data sets used and their pass/fail status.
  9. Maintenance: Updates or new test scenarios can be added by modifying test data, not scripts.
  10. Parameterization: Often combined with parameterization for versatile automation testing.

- *Keyword Driven Testing*
  Keyword-Driven Testing (KDT) is an automated testing approach that separates test case design from its implementation. It involves using keywords and a test script to specify test actions and data. Here are key points about Keyword-Driven Testing:

  1. Keyword-Based Approach: Tests are designed using keywords for actions and data.

2. Abstraction of Test Cases: Testers create and edit test cases without deep scripting knowledge.
3. Modular Structure: Test scripts are organized into reusable modules.
4. Separation of Concerns: Test design is separated from execution.
5. Reusable Keywords: Predefined keywords for test actions are reused.
6. Parameterization: Test data can be varied through parameters.
7. Central Repository: Stores mappings between keywords and functions.
8. Test Data Management: Test data is managed separately.
9. Automation Frameworks: Can be integrated into automation frameworks.
10. Maintenance and Extensibility: Minimizes script changes and allows for new test cases.
11. Customization: Custom keywords can be defined.
12. Reporting and Logging: Detailed reports aid issue identification.
13. Simplifying Complex Tests: Useful for complex applications.
14. Collaboration: Enables collaboration among teams.
15. Code-Free Test Creation: Accessible without coding expertise.
16. Scalability: Handles a wide range of test scenarios.

Keyword-Driven Testing streamlines test case creation, enhances collaboration, and simplifies test maintenance.

4.2

- *Testing Web Based Systems*

Testing web-based systems involves evaluating software programs that operate on the internet through web servers, networks, HTTP, and web browsers, interacting with users via web pages. Key terms related to web-based systems include:

1. Web Page: Information displayed in a single browser window.

2. Website: A collection of web pages and associated software components linked by content and syntax. Websites can be dynamic and interactive.

3. Web Application: A program that runs on web servers and can be accessed by users through a website. Web applications may require one or more web servers, and they are also known as web-based applications. Some applications operate independently of servers, relying on operating system services; these are termed web-enabled applications.

In today's technology landscape, the boundary between web-based and web-enabled applications is becoming increasingly blurred, so both are collectively referred to as web

applications. Testing web-based systems ensures their functionality, security, and performance, while considering the complex interplay of web servers, networks, and browsers.

- *Challenges*

  Testing web-based software presents unique challenges and quality issues due to the distinct characteristics of web applications. Some of these challenges include:

  1. Diversity and Complexity: Web applications involve various components running on diverse hardware and software platforms. They are written in different languages and use various programming approaches. The client side features browsers, HTML, scripting languages, and applets, while the server side incorporates technologies like CGI, Java Server Pages (JSPs), Java Servlets, and .NET. These components interact with a wide range of back-end engines and servers, adding complexity to testing.

  2. Dynamic Environment: Web applications are inherently dynamic, with behavior influenced by user input, changing application requirements, evolving web technologies, and other factors. The dynamic nature of web software makes it challenging to analyze, test, and maintain. It's difficult to statically determine control flow due to its dependence on user input and behavior trends over time or location.

  3. Short Development Time: Clients often demand quick development cycles for web-based systems, leading to shorter project timelines compared to traditional software projects. For instance, e-business systems and sports websites require rapid development.

  4. Continuous Evolution: Web applications frequently experience demands for increased functionality and capacity, even after deployment. Scalability becomes a significant concern as the system evolves.

  5. Compatibility and Interoperability: Web testing is complicated by compatibility and interoperability issues. Incompatibility can arise on both the client and server sides. Server components may run on different operating systems, and clients may use various browser versions on different operating systems. Testing graphics and objects across multiple browsers is essential. Code executed by browsers, written in different versions of HTML with varying tags, must be thoroughly tested to ensure consistency.

  Addressing these challenges requires a comprehensive testing strategy that considers the dynamic and diverse nature of web-based systems, ensuring compatibility, interoperability, and robust performance across different platforms and evolving requirements.

- *Security*

  Security testing is vital for web applications due to the increasing amount of sensitive data and transactions on the web. It aims to protect web applications from unauthorized access and malicious activities.

  Key Security Challenges:

  1. Multifaceted Attack Risk: Web applications are susceptible to attacks on multiple fronts, including the web software, client-side environment, network communications, and server-side components.
  2. Accessibility to Many Users: Web applications are accessible from various locations, systems, and browsers, exposing them to different security issues and external attacks.

  Security Test Plan:

  - Security testing involves two categories: infrastructure security (firewalls and port scans) and web application vulnerability testing (user authentication, encrypted use of cookies, etc.).
  - Threat models can help in planning security tests and identifying potential security issues during design.
  - Testing should focus on component interfaces, as most security vulnerabilities are found at these points.
  - High-priority interfaces should be thoroughly tested, injecting mutated data to check security.
  - While performing security testing, testers must avoid making changes to the application's configuration, server settings, services, and existing user data.

  Threat Types and Corresponding Test Cases:

  1. Unauthorized User/Fake Identity/Password Cracking:
     - Test to ensure unauthorized users cannot access software or view its contents.
  2. Cross-Site Scripting (XSS):
     - Test for cross-site scripting vulnerabilities where attackers insert malicious scripts.
  3. Buffer Overflows:
     - Verify memory allocation handling to prevent buffer overflow attacks.
  4. URL Manipulation:
     - Test HTTP GET requests to prevent URL manipulation and data corruption.
  5. SQL Injection:
     - Protect against SQL injection attacks by escaping special characters in user inputs.
  6. Denial of Service:
     - Test the application's resilience to heavy loads, distorted data, memory overloads, and other factors that can lead to denial of service.

Security testing ensures web applications maintain data confidentiality and that users can perform only authorized tasks. Protecting web applications from external threats is a critical aspect of ensuring their security and reliability.

- *Navigation and Performance Testing*

Navigation Testing:

In navigation testing, we focus on ensuring that the sequence of navigations in a web application is correct. The following elements are covered in this testing:

1. Types of Navigations:
   - Internal Links
   - External Links
   - Redirected Links
   - Navigation for searching within the web application

2. Error Checks:
   - Verify that links are not broken for any reason.
   - Ensure that redirected links display proper messages to users.
   - Check the availability and relevance of all possible navigation paths.
   - Test the functionality of the back and forward buttons to ensure proper operation.

By conducting navigation testing, we aim to confirm that users can smoothly navigate through the web application without encountering broken links or unexpected behaviors.

Performance Testing:

In today's internet-centric environment, web application performance is a significant concern. Users expect quick and responsive access to information without delays, making performance testing a crucial aspect of web application testing.

Importance of Performance Testing:

- Performance testing evaluates metrics like response time, throughput, and resource utilization.
- It helps identify bottlenecks within the system and enables developers to make necessary improvements.
- Performance testing assesses the system's ability to handle high loads and ensures it responds in a timely manner.

Performance Parameters:

- Resource Utilization: Measures the percentage of time that resources like CPU, memory, I/O, peripherals, and network are busy.
- Throughput: Indicates the number of event responses completed within a given time frame.
- Response Time: Measures the time between a request and its response.
- Database Load: Evaluates how often the database is accessed by the web application.
- Scalability: Assesses the application's ability to handle additional workload without degrading performance.
- Round-Trip Time: Measures the time taken for a complete user-requested transaction, including connection and processing time.

Types of Performance Testing:

Performance tests fall into the following categories:

1. Load Testing:
   - Determines whether the system can handle peak loads.
   - Evaluates the system's performance under various simultaneous user requests, large data inputs, connections to databases, and more.
   - Types of load testing include capacity testing (determining the maximum load before failure) and scalability testing (assessing how the system expands to accommodate increased loads).

2. Stress Testing:
   - Involves pushing the system beyond its specified limits.
   - Assesses the system's reaction to stress and how it recovers from crashes.
   - Focuses on performance under extreme conditions, like limited memory, insufficient disk space, or server failures.
   - Aims to determine when and how the application fails and how gracefully it can recover from failures.

Performance testing is essential to ensure web applications can deliver satisfactory performance under real-world conditions and handle unexpected loads and stress without compromising user experience.

4.3

- *Testing Agile Based Software*

Agile methodologies have transformed the software development process by emphasizing collaboration, flexibility, and customer-centricity. Testing in Agile is an integral part of

the development cycle, ensuring continuous improvement and high-quality software delivery. Here are key aspects of testing in Agile-based software development:
Certainly, here's a more concise version of testing Agile-based software:

1. Iterative Testing:
   - Continuous testing in short development cycles.
   - Incremental testing for early defect detection.

2. User Story Testing:
   - Testing aligned with user stories.
   - Acceptance criteria define completeness.
3. Test-Driven Development (TDD):
   - Test creation before code development.
   - Ensures code meets requirements.
4. Automation:
   - Vital for frequent testing.
   - Supports rapid feedback and regression tests.
5. Cross-Functional Teams:
   - Collaborative teams with various skills.
   - Shared understanding of user stories.
6. Continuous Integration (CI):
   - Frequent code integration.
   - Automated tests for early issue detection.
7. Exploratory Testing:
   - Unscripted testing for usability and edge cases.
8. Adaptability and Flexibility:
   - Embraces change and customer feedback.
   - Responsive to evolving requirements.
9. Continuous Feedback:
   - Frequent stakeholder feedback.
   - Drives software improvement.
10. Test Metrics:
    - Burn-down charts, velocity, defect density.
    - Data-driven decisions and process refinement.
11. Shift-Left Testing:
    - Early testing to minimize defects.
    - Cost-effective issue prevention.
12. Test Reporting:
    - Regular reporting for transparency.
    - Tracking progress and sprint goals.

Agile testing ensures quality software delivery through adaptability, collaboration, and customer focus.

- *Mobile Application Testing*

  Mobile app testing is crucial in the highly dynamic world of mobile technology. Here are key aspects to consider:

  1. Platform Compatibility:
     - Test on multiple platforms (iOS, Android).
     - Consider various devices, screen sizes, and OS versions.
  2. Functionality Testing:
     - Verify core functions work as intended.
     - Cover navigation, input, and app features.
  3. Usability and User Experience:
     - Evaluate user-friendliness.
     - Test user flows and interface design.
  4. Performance Testing:
     - Check app speed, responsiveness.
     - Assess resource usage and battery impact.
  5. Security and Privacy:
     - Protect user data and app code.
     - Guard against vulnerabilities and data leaks.
  6. Network Conditions:
     - Test under different network speeds.
     - Handle offline and poor connectivity scenarios.
  7. Installation and Updates:
     - Ensure smooth installation and updates.
     - Check for conflicts with other apps.
  8. Localization and Internationalization:
     - Validate app in multiple languages.
     - Verify currency, date formats, etc.
  9. Device Features:
     - Test features like camera, GPS, and sensors.
     - Verify integration with device capabilities.
  10. Integration and Compatibility:
      - Assess third-party integrations.
      - Ensure compatibility with APIs and services.
  11. Security Compliance:
      - Comply with platform-specific guidelines.
      - Verify data encryption and user privacy.

12. Accessibility:
    - Ensure the app is usable for all, including people with disabilities.
    - Test with screen readers and other assistive tech.
13. App Store Guidelines:
    - Follow app store submission rules.
    - Prepare for reviews and approvals.
14. Regression Testing:
    - Continuously test as updates are made.
    - Avoid new issues in previously working features.
15. Beta Testing:
    - Gather user feedback in real-world scenarios.
    - Fix issues and refine the app.
16. Automated Testing:
    - Utilize test automation tools.
    - Speed up testing and increase coverage.
17. Performance Metrics:
    - Monitor app performance in the field.
    - Collect crash reports, usage analytics.

Mobile app testing ensures a seamless and secure user experience across various devices and scenarios. Adaptability and thorough testing are key in this fast-paced domain.

Module 5: Software Quality Management

5.1

- *Software Quality*

  Software quality is a complex and multi-dimensional concept. Unlike physical products, software quality is not easily defined. It encompasses various aspects beyond simply the absence of defects or bugs. While the classical definition of quality is "conformance to requirements," software quality goes beyond this narrow view.

  Here are key points about software quality:

  1. Defects as a Component: Software quality is often associated with the absence of defects or bugs. If software has functional issues, it may not perform its intended functions correctly. Defects are quantified using metrics like defect density, which measures the number of defects per unit size (e.g., lines of code or function points).

  2. Multi-Dimensional: Software quality is a multi-dimensional concept. It encompasses various aspects, including reliability, performance, security, usability, and maintainability. High-quality software should excel in these different dimensions.

  3. Responsibility of the Team: Achieving software quality is a collective responsibility within a software project. Every team member plays a role in ensuring the quality of the product. Quality, along with cost and schedule, is a fundamental concern for everyone involved.

  4. Minimizing Defects: The ultimate goal of software quality is to minimize defects. This results in higher quality software that meets user expectations and performs reliably. The fewer defects in the final product, the higher the overall quality.

  In summary, software quality is a multi-faceted concept that goes beyond mere bug-free software. It encompasses various dimensions of quality, and achieving it is a shared responsibility among all team members in a software project. The primary objective is to deliver software that meets user expectations with minimal defects, ensuring high-quality outcomes.

- *Five Views of Software Quality*

The "Five Views of Software Quality" represent different perspectives or approaches to understanding and assessing the quality of software. Each view offers a unique lens through which quality is evaluated

1. Transcendental View: The transcendental view defines software quality as an abstract concept that is challenging to quantify or measure objectively. Quality in this view is subjective and depends on individual perceptions. It's often associated with a user's intuitive judgment of a software product's quality, based on their experience and expectations.

2. User View: The user view emphasizes that software quality is determined by how well the software meets the needs and expectations of its end-users. Quality, from this perspective, is closely tied to user satisfaction and usability. It focuses on user experiences, user interface design, and user feedback.

3. Manufacturing View: The manufacturing view treats software development as an industrial process. Quality in this view is assessed by the absence of defects and adherence to predefined specifications and standards. It involves techniques like defect prevention, process control, and adherence to best practices to produce a high-quality product.

4. Product View: The product view focuses on evaluating software quality based on product attributes and characteristics. Quality is measured through metrics, such as reliability, performance, maintainability, and security. This view considers software as a tangible product with quantifiable qualities.

5. Value-Based View: The value-based view assesses software quality by considering its value to the stakeholders, which includes users, customers, and the organization. It takes into account cost-effectiveness, return on investment, and the alignment of the software with business goals and requirements. Quality is measured by the value it delivers to the stakeholders.

- *McCall's Quality Factors and Criteria*

  McCall's quality model, developed by John McCall in the 1970s, provides a structured framework for assessing software quality. It defines 11 quality factors and associated criteria to evaluate software. These factors are grouped into three main categories: product revision, product transition, and product operation. Here are McCall's quality factors and their criteria:

Product Revision:

These factors focus on the software's internal qualities and its ability to be maintained and enhanced.

1. Correctness:
   - Criteria: Absence of defects, accurate and consistent results, adherence to requirements.
2. Reliability:
   - Criteria: Consistent performance, minimal failure rate, fault tolerance, and error recovery capabilities.
3. Efficiency:
   - Criteria: Optimal use of system resources (e.g., memory, CPU), acceptable response times, and minimal resource wastage.
4. Integrity:
   - Criteria: Data security, protection against unauthorized access or tampering, and data consistency.
5. Usability:
   - Criteria: User-friendliness, ease of use, and effectiveness in achieving user goals.
6. Maintainability:
   - Criteria: Ease of modification, addition of new features, and bug fixing without causing regression or introducing new issues.

Product Transition:

These factors assess the software's readiness for deployment and its adaptability to new environments.

7. Portability:
   - Criteria: Ability to run on various platforms and operating systems with minimal adaptation.
8. Reusability:
   - Criteria: Extent to which software components can be reused in other applications or contexts.

Product Operation:

These factors relate to the software's performance and behavior in its operational environment.

9. Operational Efficiency:
   - Criteria: Efficient resource utilization during system operation, including minimal system downtime.
10. Behavior:

- Criteria: Consistency in system behavior under different circumstances and with varying input conditions.

11. Testability:

- Criteria: Ease of testing, identification of test cases, and coverage of testing scenarios.

McCall's model provides a structured way to assess and improve software quality. It emphasizes the importance of both internal and external qualities and considers factors related to maintenance, portability, and the operational environment. By evaluating software using these quality factors and criteria, organizations can identify areas for improvement and deliver higher-quality software products.

- *Software Quality Metrics*

Software quality metrics are essential in evaluating and improving the quality of software products, processes, and projects. These metrics encompass various aspects of software development and maintenance. They can be categorized into three groups, each addressing a different stage in the software lifecycle:

1. Product Quality Metrics:Product quality metrics are crucial for assessing the quality of a software product from the end-user's perspective. They focus on factors that directly impact user satisfaction and overall product quality. Key product quality metrics include:

1. Mean-time to Failure (MTTF): This metric estimates the average time until the first failure of a software product. It is often used in safety-critical systems.

2. Defect Density Metrics: Defect density measures the number of defects in relation to the size of the software product. It's important for estimating maintenance costs and resources.

$$Defect\ density = Number\ of\ defects\ /\ Size\ of\ product$$

3. Customer Problem Metrics: These metrics evaluate the problems customers encounter while using the software, including defects and usability issues. The goal is to reduce the number of problems per user month (PUM) by improving various aspects of the product.

$$PUM = \frac{Total\ problems\ reported\ by\ the\ customer\ for\ a\ time\ period}{Total\ number\ of\ licensed\ months\ of\ the\ software\ during\ the\ period}$$

4. Customer Satisfaction Metrics: These metrics assess customer satisfaction through surveys. They may include percentages of completely satisfied or satisfied customers.

2. In-Process Quality Metrics: In-process quality metrics focus on assessing the quality of the software during its development and testing phases. These metrics are less formally defined than end-product metrics and help identify issues and areas for improvement throughout the development process. Key in-process quality metrics include:

1. Defect Density During Testing: This metric measures the rate of defects discovered during testing. A higher defect rate during testing suggests that more errors were introduced during development. It is valuable for monitoring product quality during testing and for tracking quality improvements in subsequent releases.

2. Defect Arrival Pattern During Testing: Analyzing the pattern of when defects are discovered provides insights into the quality of the software. Stable or infrequent defect arrivals are indicative of higher quality.

3. Defect Removal Efficiency (DRE): DRE quantifies how efficiently defects are removed during a specific time frame.

$$DRE = \frac{Defects\ removed\ during\ the\ month}{Number\ of\ \ problem\ arrivals\ during\ the\ month} \times 100$$

3. Software Maintenance Metrics: During the software maintenance phase, metrics are used to monitor and improve the process of resolving defects and issues. Key metrics include:

1. Fix Backlog and Backlog Management Index (BMI):
- Fix backlog: The count of open issues or defects remaining at the end of a defined time period.
- Backlog Management Index (BMI): A metric to manage the backlog, where a BMI greater than 100 indicates backlog reduction, while a BMI less than 100 signifies an increase. The goal is to maintain a BMI greater than 100.

$$BMI = \frac{Number\ of\ \ problems\ closed\ during\ the\ month}{Number\ of\ \ problem\ arrivals\ during\ the\ month} \times 100$$

2. Fix Response Time and Fix Responsiveness:
- Fix response time: The time it takes to resolve high-severity defects from their reporting to closure.
- Fix responsiveness: An agreed-upon time limit set by customers for resolving defects. Achieving high fix responsiveness is essential for customer satisfaction.

3. Percent Delinquent Fixes:

- Measures the percentage of fixes that exceed the required response time, indicating delays in issue resolution.

$$Percent\ delinquent\ fixes = \frac{Number\ of\ \ fixes\ that\ exceeded\ the\ response\ time\ criteria\ by\ severity\ level}{Number\ of\ \ fixes\ delivered\ in\ \ a\ specified\ time} \times 100$$

4. Fix Quality:

- Evaluates the effectiveness of defect resolution. A fix is considered defective if it doesn't address reported issues or introduces new defects.

- This metric is critical for real-time software and customer satisfaction, measured as a percentage of defective fixes.

By categorizing software quality metrics into these three groups, organizations can comprehensively assess and improve software quality at various stages of development and maintenance. This holistic approach ensures that the software meets both user expectations and industry standards.

5.2

- *PDCA Cycle*

The PDCA (Plan-Do-Check-Act) cycle, also known as the Deming Cycle or the Shewhart Cycle, is a four-step iterative problem-solving and continuous improvement process. It was developed by Walter A. Shewhart, popularized by W. Edwards Deming, and has been widely used in quality management and process improvement. The PDCA cycle consists of the following stages:

1. Plan (P):
   - Identify the problem or opportunity for improvement.
   - Set specific and measurable objectives.
   - Develop a detailed plan to achieve the objectives.
   - Define the resources, responsibilities, and timelines.

2. Do (D):
   - Implement the plan that was created in the "Plan" phase.
   - Execute the planned actions and processes.
   - Collect data and information during the execution.

3. Check (C):
   - Compare the actual results and outcomes with the expected outcomes.
   - Analyze the data and information collected during the "Do" phase.

- Determine whether the objectives were achieved.
- Identify any discrepancies, variations, or problems.

4. Act (A):
- Based on the analysis in the "Check" phase, make decisions and take actions to:
  - Address and correct any problems or discrepancies.
  - Standardize the improvements made.
  - Update the processes, procedures, and plans based on the lessons learned.
- Implement the necessary changes and improvements.
- Repeat the cycle to continue the process of continuous improvement.

The PDCA cycle is a systematic approach to problem-solving and quality improvement. It encourages a culture of continuous learning and adaptation by regularly revisiting and refining processes to achieve better results. It is widely used in various fields, including quality management, project management, and process improvement.

- *Quality Plan*

1. Introduction:
- Overview and scope of the project.
- Purpose of the quality plan.

2. Quality Objectives:
- Clear and measurable quality goals.
- Desired quality standards.

3. Quality Standards and Criteria:
- Specific quality benchmarks.
- Alignment with best practices and regulations.

4. Quality Assurance and Control:
- Processes to ensure and monitor quality.
- Frequency and criteria for quality checks.

5. Roles and Responsibilities:
- Key roles in quality management.
- Responsibility for quality assurance and control.

6. Training and Competence:
- Training and competency requirements.

- Ensuring team readiness for quality goals.

7. Documentation and Communication:
   - Procedures for quality documentation.
   - Communication channels and reporting.

8. Risk Management:
   - Identification of quality risks.
   - Mitigation and contingency plans.

9. Monitoring and Measurement:
   - Quality performance metrics.
   - Key performance indicators (KPIs).

10. Audit and Continuous Improvement:
    - Quality audit and review process.
    - Actions for continuous quality enhancement.

11. Approval and Sign-off:
    - Authorization and approval process.
    - Sign-off authority.

- *Assurance*

  Purpose: Quality Assurance, is more process-oriented. Its main purpose is to verify that the applicable procedures and standards are being followed throughout the software development process.

  Nature: QA activities are related to the management of the software development process. It involves auditing and reporting functions to ensure that established processes, standards, and procedures are adhered to.

  Activities: QA activities include measuring both the software products and the development processes, auditing the quality of each step in the Software Development Life Cycle (SDLC), and verifying the adoption of specified procedures and standards.

  Management: QA activities are primarily within the management's domain. These activities aim to inform management about quality-related issues, provide relevant data, and allow them to address and rectify problems.

  It includes:

„ Measuring the products and processes.
„ Auditing the quality of every step in SDLC and the end-product.
„ Verifying that the adopted procedures and standards are being followed

Quality Assurance, focuses on the process of developing the software, ensuring that established procedures and standards are followed. It includes activities like measurement, auditing, and reporting, and it operates within the management domain.

- *Control and Methods*
  Quality Control (QC):

  Purpose: Quality Control primarily focuses on finding and removing defects in a software product. Its central goal is to ensure that the final software output meets the desired specifications and requirements.

  Nature: QC is product-oriented. It involves activities that aim to maintain minimal variation from specified quality standards. It checks for defects and discrepancies in the software product at various stages of development.

  Activities: QC activities may include static and dynamic testing, such as code reviews, manual testing, and automated testing using verification and validation (V&V) tools.

  Measurement: In QC, work products should have defined, measurable specifications to enable comparisons between the actual output and the desired quality standards.

  Quality Control is concerned with the product itself, striving to minimize variations and ensure it meets specified standards. It involves activities such as testing and reviews.

  Methods:
  1. Testing: This includes various forms of testing, such as unit testing, integration testing, system testing, regression testing, user acceptance testing (UAT), and more.
  2. Code Reviews: Systematic examination of source code by peers or experienced developers to identify and rectify issues.
  3. Static Code Analysis: The use of tools to automatically scan source code for potential defects, security vulnerabilities, and code quality issues.
  4. Dynamic Code Analysis: Analyzing the software's behavior during runtime to identify performance bottlenecks, memory leaks, and other runtime issues.
  5. Defect Tracking and Management: Implementing systems to capture, categorize, prioritize, and manage defects and issues in the software.

6. Configuration Management: Ensuring that the correct versions of software components are used in different environments, preventing configuration mismatches.
7. Process Control: Implementing well-defined development processes with checkpoints and milestones to ensure adherence to established quality standards.
8. Quality Metrics: Measuring and analyzing various quality attributes, such as code complexity, code coverage, and defect density, to assess and control software quality.
9. Inspections: Formal technical inspections involving team reviews of software artifacts like requirements, design documents, and code to identify and rectify issues.
10. Continuous Integration (CI): Implementing automated CI systems that regularly build, test, and integrate code changes into a shared repository to identify and rectify integration issues early.

- *Quality Cost and Benefits*

1. Quality Efforts in Products and Processes: The passage emphasizes that maintaining software quality requires efforts in both the product (the software itself) and the processes (the procedures and practices used in its development). Quality evaluation programs are necessary to ensure that the desired level of quality is achieved.

2. Cost of Quality Categories:

a. Prevention Costs: These are costs associated with activities aimed at identifying the causes of defects and preventing them. Examples include quality planning, formal technical reviews, testing, training, and defect causal analysis. Prevention costs are incurred to stop defects from occurring in the first place.

b. Appraisal Costs: These costs involve evaluating the quality of software products at different levels. Activities include testing, in-process and inter-process inspection, and the implementation of software metrics programs (SMP). Appraisal costs are incurred to assess and ensure quality.

c. Failure Costs: These costs are related to addressing failures in the software. Failure costs can be further divided into two categories:

- External Failure Costs: These occur when failures are discovered at the customer's site after the product has been released. Examples include regression testing to address any bugs, handling customer complaints, product returns, and replacements.

- Internal Failure Costs: These are costs incurred within the developer's site before the product is released. Examples include regression testing, bug isolation and repair, and criticality analysis of bugs.

3. Cost Increase with Development Stage: The passage mentions that the cost of finding and debugging a bug increases tenfold as the software development lifecycle (SDLC) progresses. This underlines the importance of early prevention and appraisal activities to catch and address issues in the early stages when they are less costly to fix. Failing to implement these activities can lead to higher costs for detecting and debugging failures, both internally and externally.

Investing in prevention and appraisal activities can help reduce the overall cost of quality by addressing issues early in the development process and preventing failures that may arise at later stages, both internally and externally.

Benefits:
- „ Customer is satisfied, as the end-product is of high quality.
- „ Productivity increases due to shorter SDLC cycle.
- „ Failures and failure costs reduce.
- „ Rework and cost of quality reduc

- *Defect Prevention and Root Cause Analysis*

  Defect prevention and root cause analysis are critical components of quality management in software development and other industries. They aim to identify, mitigate, and eliminate defects and underlying issues that can lead to quality problems.

  Defect Prevention:

  Defect prevention refers to the proactive strategies and practices implemented throughout the software development process to reduce or eliminate defects before they occur. The primary goal is to improve the quality of the product by addressing the root causes of potential issues. Key aspects of defect prevention include:

  1. Process Improvement: Evaluating and refining development processes to make them more efficient, effective, and error-resistant.
  2. Standardization: Establishing and adhering to coding standards, design guidelines, and best practices to prevent common mistakes and defects.
  3. Training and Education: Ensuring that team members are well-trained and aware of quality standards and processes, reducing the likelihood of errors.
  4. Code Reviews: Conducting regular code reviews to identify and rectify issues early in the development process.
  5. Automated Testing: Implementing automated testing (unit tests, integration tests, etc.) to catch defects as soon as they are introduced.

6. Root Cause Analysis: Investigating the underlying reasons for defects to prevent their recurrence.
7. Risk Management: Identifying potential risks and taking proactive measures to mitigate them.

Root Cause Analysis:

Root cause analysis (RCA) is a systematic process used to identify the fundamental causes of defects, incidents, or issues in order to prevent their recurrence. RCA is not limited to software development; it's used in various industries. Key elements of root cause analysis include:

1. Problem Identification: Clearly defining the problem or defect and its impact on the project or product.
2. Data Collection: Gathering relevant data, including incident reports, logs, and other sources, to understand the issue.
3. Analysis: Investigating the problem, often using techniques like the "5 Whys" or Fishbone diagrams to dig deeper into potential causes.
4. Identifying Root Causes: Pinpointing the underlying reasons that led to the defect or problem.
5. Action Planning: Developing an action plan to address and correct the root causes.
6. Implementation: Executing the action plan to prevent the defect from recurring.
7. Monitoring and Verification: Continuously monitoring the situation to ensure the defect does not reappear.
8. Documentation: Recording the findings and actions taken for future reference.

Root cause analysis is a valuable tool for not only fixing defects but also preventing similar issues from happening in the future. It promotes a culture of continuous improvement and learning from mistakes.

Both defect prevention and root cause analysis are essential for maintaining high-quality products and processes. By identifying and addressing issues at their core, organizations can save time, resources, and ensure the delivery of reliable and error-free software.

5.3

- *Software Quality Tools*
  The common software quality tools are:

1. Integrated Development Environments (IDEs): IDEs like Visual Studio, Eclipse, and IntelliJ IDEA provide essential features for code editing, debugging, and testing, which are crucial for maintaining code quality.
2. Version Control Systems (VCS): Tools like Git, Subversion (SVN), and Mercurial are essential for version control, collaboration, and ensuring code quality by managing source code versions.
3. Static Code Analysis Tools: These tools, such as SonarQube, Checkmarx, and ESLint, analyze source code without execution to identify coding issues and potential defects.
4. Code Review Tools: Platforms like GitHub, GitLab, and Bitbucket offer features for peer code reviews, ensuring adherence to best practices and code quality.
5. Continuous Integration and Continuous Delivery (CI/CD) Tools: CI/CD tools like Jenkins, Travis CI, and CircleCI automate building, testing, and deployment processes, improving software quality by streamlining development workflows.
6. Test Management Tools: Tools like TestRail, Zephyr, and qTest help manage test cases, execute tests, and track defects, ensuring comprehensive testing and quality assurance.
7. Defect Tracking and Issue Management Tools: Issue tracking tools like Jira, Redmine, and Bugzilla are vital for managing, prioritizing, and tracking defects and other issues to resolution.

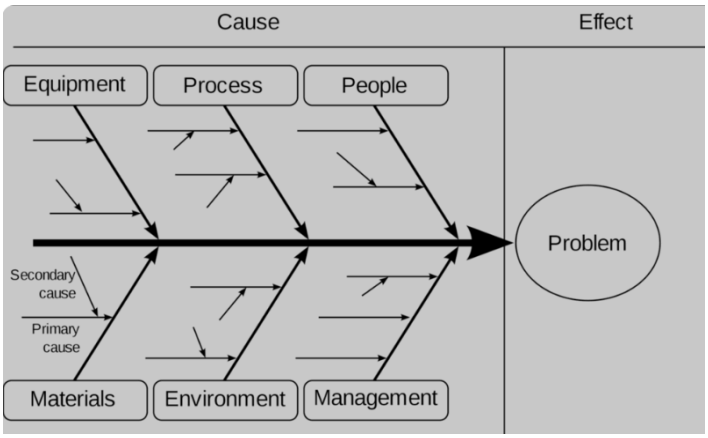
These tools cover key aspects of software quality management, from code quality analysis and testing to collaboration, version control, and defect tracking. The specific tools you choose will depend on your project's requirements and your team's preferences.

- *Ishikawa Diagram*
  The Ishikawa diagram, also known as a fishbone diagram or cause-and-effect diagram, is a visual tool used for problem-solving and identifying the root causes of an issue. It was developed by Japanese quality control expert Kaoru Ishikawa in the 1960s. The diagram is called a fishbone diagram because of its appearance, resembling the skeleton of a fish.

  Key components of an Ishikawa diagram include:

  1. Problem Statement: At the head of the diagram, you write the specific problem or issue you want to address. This problem statement is the "fish's head."
  2. Main Categories: Spreading out from the problem statement, you draw a horizontal line (the "spine" of the fish) with several diagonal lines extending from it. Each diagonal line represents a main category of factors that could contribute to the problem. These main categories are often related to people, processes, equipment, materials, and environment (the 5 Ms).

3. Causes: Along each diagonal line, you list the specific factors or causes that could be contributing to the problem. These causes can be further broken down into sub-causes or factors.
4. Analysis: The Ishikawa diagram helps teams analyze and visualize potential causes and their relationships. By identifying root causes, teams can develop solutions to address the problem effectively.
The Ishikawa diagram is a valuable tool for problem-solving and continuous improvement in various fields, including manufacturing, quality control, and process improvement. It encourages a structured and systematic approach to identifying and addressing issues.



- *Check List*

In quality assurance (QA), checklists are a valuable tool for ensuring that quality standards and processes are consistently followed. Checklists help QA teams identify issues, verify compliance with quality guidelines, and track progress. Here are some common uses of checklists in quality assurance:

1. Quality Inspections: Ensuring products, services, or processes meet quality criteria.
2. Audits and Assessments: Evaluating compliance with quality standards and policies.
3. Software Testing: Planning and executing tests to verify software functionality.
4. Documentation and Records Review: Ensuring accurate and complete documentation.
5. Process Compliance: Verifying adherence to standardized procedures.
6. Supplier Quality Checks: Assessing supplier product quality and performance.
7. Regulatory Compliance: Confirming adherence to legal and regulatory requirements.
8. Training and Onboarding: Facilitating employee training and onboarding.
9. Continuous Improvement: Identifying areas for improvement and tracking progress.
10. Safety Checks: Ensuring safety procedures and precautions are in place.

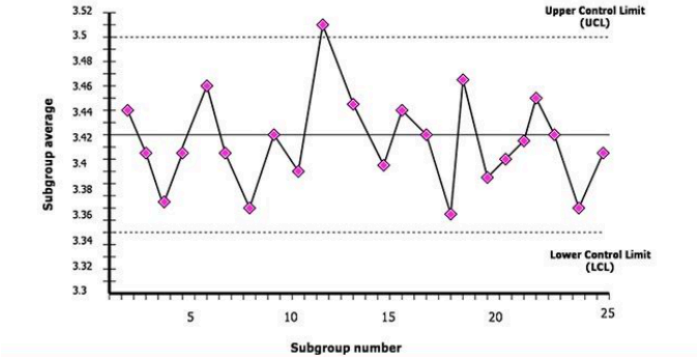
These are key applications of checklists in quality assurance.

- *Control Chart*

  A Control Chart is a graphical tool used in quality control to monitor and visualize process data over time. It helps in assessing whether a process is within acceptable control limits or if it exhibits variations that require attention. Control Charts typically consist of a central line representing the process mean and upper and lower control limits that indicate acceptable variation boundaries.

  Control Charts are designed to detect both common cause variations (inherent to the process) and special cause variations (external factors or anomalies). By regularly plotting data points on the chart, practitioners can quickly identify trends, shifts, or abnormal patterns that may require investigation and corrective action.

  There are various types of Control Charts, including X-bar and R charts for continuous data and p-charts and c-charts for attribute data. These charts are essential in maintaining process stability and ensuring that products or services consistently meet quality standards.



- *Pareto Chart*

  A Pareto Chart is a visual tool used for quality control and problem-solving. It's named after the Italian economist Vilfredo Pareto, who is known for the Pareto Principle or the "80/20 rule," which suggests that approximately 80% of effects come from 20% of causes. The Pareto Chart is a graphical representation of this principle and helps prioritize issues or factors that have the most significant impact on a problem.

  Key features of a Pareto Chart include:

  1. Categories or Factors: The chart displays various categories or factors that contribute to a specific problem or issue. These categories are usually listed on the horizontal axis.

2. Frequency or Impact: The vertical axis represents the frequency, occurrences, or impact of each category or factor. This is typically measured using counts, percentages, or other relevant metrics.

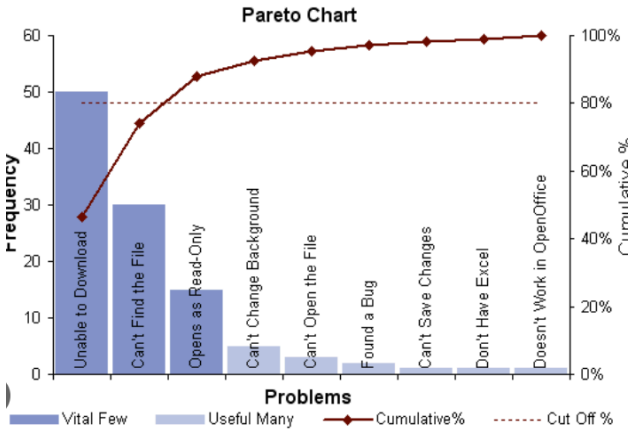
3. Bars: A bar graph represents each category's frequency or impact. The bars are arranged in descending order, from the most significant contributor (the "vital few") to the least significant (the "trivial many"). This arrangement highlights the top contributors.

4. Cumulative Percentage: A line graph (often a cumulative percentage line) may be added to the chart, showing the cumulative contribution of the categories. This helps identify the point where the "vital few" categories contribute significantly to the total.

Pareto Charts are commonly used in quality control, process improvement, and decision-making to:

- Identify the most critical issues or factors that require attention.
- Prioritize efforts by focusing on the significant contributors.
- Visualize data to make it easier to understand and communicate within a team.
- Track the impact of improvement initiatives over time.

Pareto Charts are particularly helpful when there are multiple factors contributing to a problem, and resources need to be allocated efficiently to address the most influential factors first.



5.4

- *SQA Models*

  Software Quality Assurance (SQA) models provide structured frameworks for ensuring software quality throughout the development process. These models help in planning, implementing, and monitoring software quality. They can be customized to fit specific project requirements, and their main goal is to improve the quality of software products.

- *Software Total Quality Management*

  Total Quality Management (TQM) is a comprehensive approach to quality improvement that originated in the manufacturing sector. TQM focuses on creating a quality-centered culture within the entire organization. The key principles of TQM can be summarized as follows:

  1. Customer Focus: TQM aims to achieve total customer satisfaction by understanding and meeting customer needs and requirements. Quality is defined and judged by the customer, making customer satisfaction the ultimate goal.

  2. Process Improvement: TQM emphasizes continuous process improvement, both in business processes and product development processes. This focus on process enhancement leads to improved product quality.

  3. Human-Side of Quality: TQM seeks to establish a company-wide quality culture. It involves elements such as management commitment, total participation, employee empowerment, and other human factors that contribute to quality.

  4. Measurement and Analysis: TQM relies on goal-oriented measurement systems to drive continuous improvement in all quality parameters.

  TQM offers both short-term and long-term advantages. In the long term, organizations can expect benefits like higher productivity, increased morale, reduced costs, and greater customer commitment. TQM, originally rooted in manufacturing, has found successful application in various types of organizations, including software development.

  In the context of software, the adoption of TQM principles is known as Software Total Quality Management (STQM). STQM aligns with the components of TQM and focuses on improving software quality through the following elements:

  1. Customer Focus in Software Development: Gathering and understanding customer needs is crucial in software development, particularly in addressing requirement-related issues.

  2. Process, Technology, and Development Quality: Developing software effectively and ensuring it meets customer requirements require mature processes and up-to-date technologies.

3. Human-Side of Software Quality: In software development, STQM involves integrating project, process, and quality management while actively involving every team member in maintaining in-process quality.

4. Measurement and Analysis: Software metrics play a significant role in monitoring and controlling software development, enabling product delivery within specified timeframes and budgets.

By incorporating TQM principles into software development, organizations can enhance the quality of their software products, maintain customer satisfaction, and achieve operational excellence in the long run.

- *Six Sigma*

  Six Sigma is a quality model initially developed for manufacturing processes, notably by Motorola. It is grounded in statistical principles, with "sigma" representing the standard deviation for a statistical population. The term "Six Sigma" implies that if there are six standard deviations between the mean of a process and the nearest specification limit, there will be virtually no items that fail to meet the specifications. The main objective of this model is to achieve a process quality level that meets this standard. Over time, Six Sigma has been adapted for non-manufacturing processes, and it is now applicable to a wide range of fields, including services, healthcare, insurance procedures, call centers, and software development.

  The primary goal of Six Sigma is to deliver strategic business results by reducing costs and minimizing defects. Six Sigma processes are expected to generate fewer than 3.4 defects per million opportunities. This goal is achieved through a well-defined methodology known as DMAIC, consisting of the following steps:

  1. Define opportunities.
  2. Measure performance.
  3. Analyze opportunities.
  4. Improve performance.
  5. Control performance.

  The DMAIC methodology focuses on reviewing and enhancing existing business processes continually.

  While Six Sigma is often associated with statistical analysis, it can be effectively applied to address software-related issues that impact quality. When applied to software development, Six Sigma is used to analyze customer requirements and align them with

business goals. This approach differs from typical software deployment methods that begin by gathering customer requirements. Six Sigma tools assist in identifying and prioritizing the functionalities to be delivered in software projects.

- *Test Maturtiy Model (TMM)*
  The Test Maturity Model (TMM) is a framework for evaluating and improving an organization's software testing processes. It was developed in 1996 at the Illinois Institute of Technology. TMM consists of two main components:

  1. A maturity model
  2. An assessment model

  Maturity Model
  1. Maturity Levels: TMM defines five maturity levels, each representing a specific stage in the evolution of a matured testing process. The higher levels build upon the practices of the lower levels. The five maturity levels are:
  - Initial
  - Phase Definition
  - Integration
  - Management and Measurement
  - Optimization/Defect Prevention and Quality Control

  2. Maturity Goals: To progress to a particular maturity level, an organization must address specific maturity goals. These goals are similar to key process areas in CMM. Each level, except level 1, has defined maturity goals.

  3. Maturity Sub-Goals: For each maturity goal, there are more specific sub-goals that outline the scope, boundaries, and achievements for that level.

  4. Activities, Tasks, and Responsibilities (ATRs): To implement the sub-goals and achieve maturity goals, a set of activities and responsibilities is defined. These activities and tasks target the improvement of the testing capability within an organization. They are assigned to three key participants in the testing process: managers, developers/testers, and users/clients.

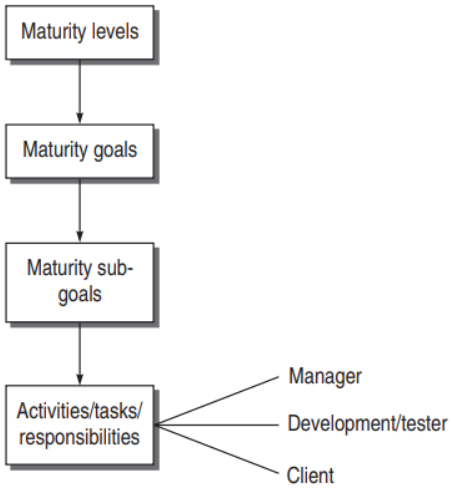


Figure 14.4 Framework of TMM

TMM provides a structured approach for organizations to assess their testing processes, set improvement goals, and advance through the maturity levels. It promotes the development of mature and effective testing practices, ultimately leading to higher-quality software products. TMM levels include

The Test Maturity Model (TMM) consists of five maturity levels that represent different stages in the evolution of a mature testing process:

1. Initial Level: This level is characterized by a random and ad-hoc testing process. Testing activities occur after coding, with the primary objective being to demonstrate the software's functionality.

2. Phase Definition: Testing and debugging are separate activities at this level. Testing is a planned activity that follows coding, and the main goal is to confirm that the software meets its specifications.

3. Integration: Testing is integrated into the entire software lifecycle and receives early consideration. Test objectives are established based on user and client requirements, and a dedicated test organization is formed with assigned responsibilities.

4. Management and Measurement: Testing is recognized as a measured and quantified process. Testing begins as soon as the Software Requirements Specification (SRS) is prepared, encompassing reviews in all development phases. Testware is maintained for reuse, and defects are adequately logged.

5. Optimization, Defect Prevention, and Quality Control: At this highest level, the testing process is well-defined and managed. It can be measured for costs and other parameters,

and its effectiveness can be monitored. The focus shifts to defect prevention instead of detection, and quality control practices are implemented. Mechanisms are established to fine-tune and continuously improve the testing process.

These maturity levels represent the evolutionary path towards achieving a mature and efficient testing process within an organization.

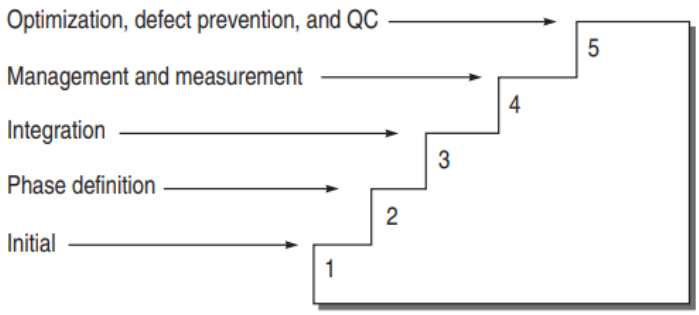


Figure 14.5 Five levels of TMM

The Assessment Model

The TMM assessment model (TMM-AM) has three main components:

1. Tools for Assessment: These tools are used for assessing the testing process. For instance, a questionnaire can be designed with questions related to maturity goals and process issues at each level.

2. Assessment Procedure: The assessment procedure provides guidelines to the assessment team on how to collect, organize, analyze, and interpret the data gathered from questionnaires and personal interviews.

3. Training Procedure: A training procedure is employed to educate personnel in test process assessment.