

Module 1

Testing Methodology

Evolution of Software Testing

- In the early days of software development, Software Testing was considered only as a debugging process for removing the errors after the development of software.
- By 1970, software engineering term was in common use. But software testing was just a beginning at that time.
- In 1978, G.J. Myers realized the need to discuss the techniques of software testing in a separate subject. He wrote the book “The Art of Software Testing” which is a classic work on software testing.
- Myers discussed the psychology of testing and emphasized that **testing should be done with the mind-set of finding the errors not to demonstrate that errors are not present.**
- By 1980, software professionals and organizations started talking about the quality in software. Organizations started Quality assurance teams for the project, which take care of all the testing activities for the project right from the beginning.

Evolution of Software Testing

- In the 1990s testing tools finally came into their own. There was flood of various tools, which are absolutely vital to adequate testing of the software systems. However, they do not solve all the problems and cannot replace a testing process.
- Gelperin and Hetzel [79] have characterized the growth of software testing with time. Based on this, we can divide the evolution of software testing in following phases:

Evolution of Software Testing

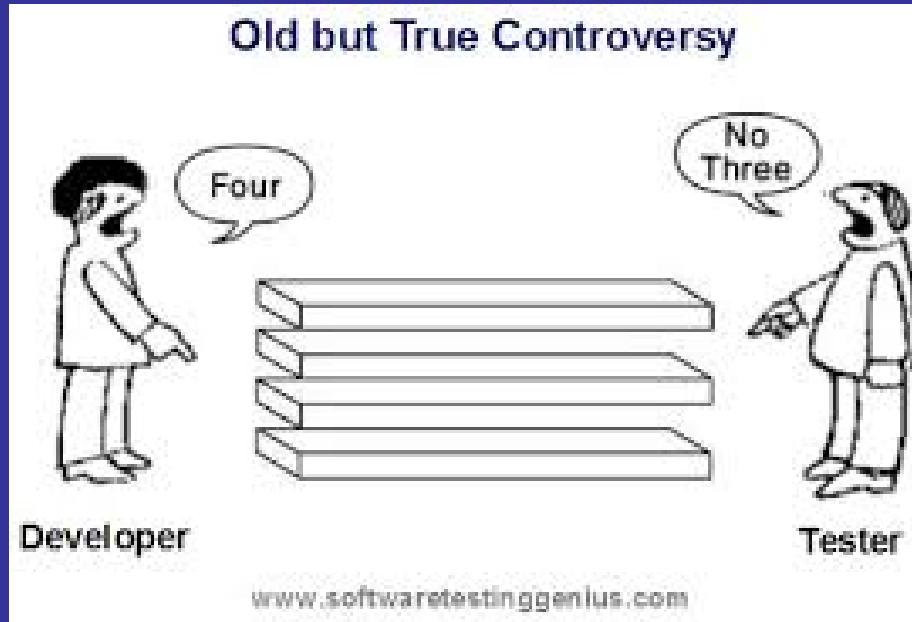
Debugging-oriented phase	Demonstration-oriented phase	Destruction-oriented phase	Evaluation-oriented phase	Prevention-oriented phase	Process-oriented phase
Checkout getting the system to run	Checkout of a program increased from program runs to program correctness	Separated debugging from testing To find more and more errors Effective testing	Quality of the software Verification and validation techniques	Bug-prevention rather than bug-detection	Process rather than a single phase
Debugging	Testing is to show the absence of errors				

Figure 1.1 Evolution phases of software testing

Psychology for Software Testing:

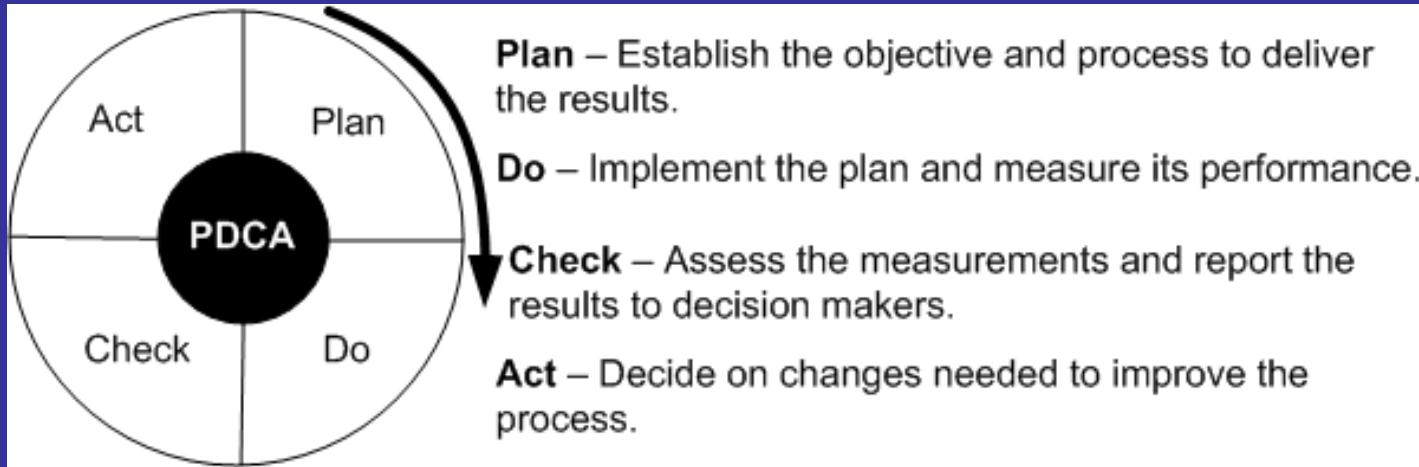
Testing is the process of demonstrating that there are no errors.

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



The Quality Revolution

The Shewhart cycle



- Deming introduced Shewhart's PDCA cycle to Japanese researchers
- It illustrates the activity sequence:
 - Setting goals
 - Assigning them to measurable milestones
 - Assessing the progress against the milestones
 - Take action to improve the process in the next cycle

Software Testing Goals

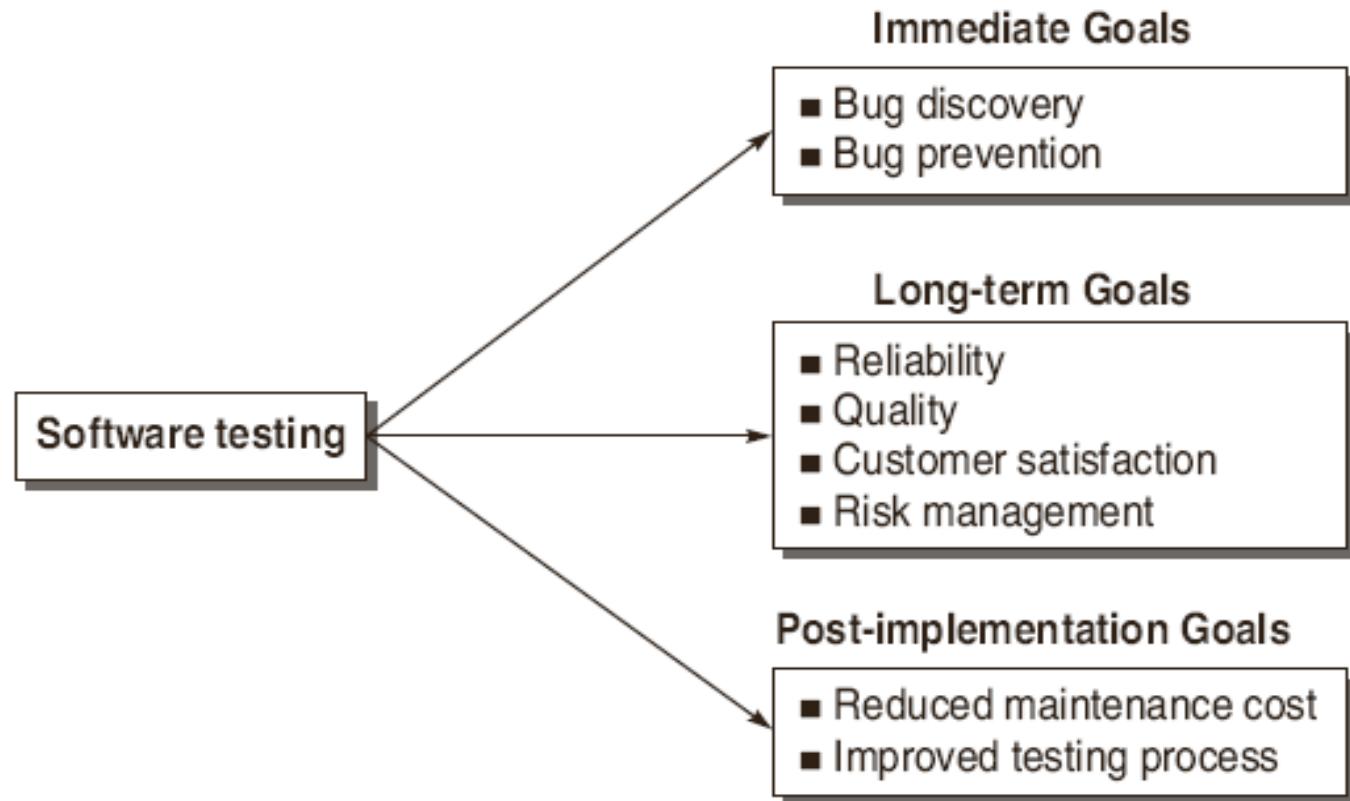
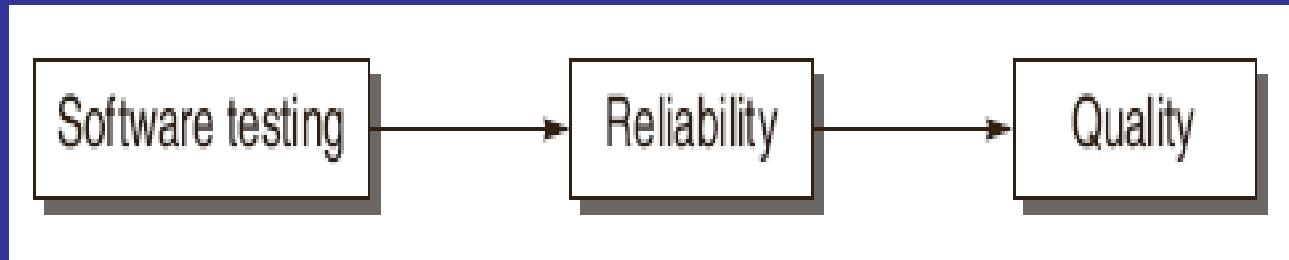
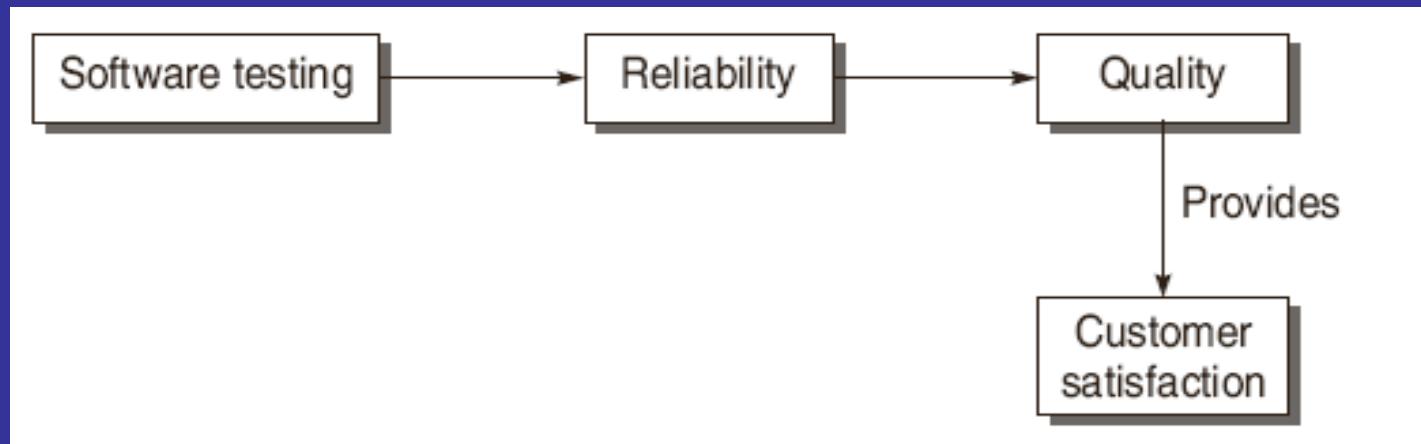


Figure 1.2 Software testing goals

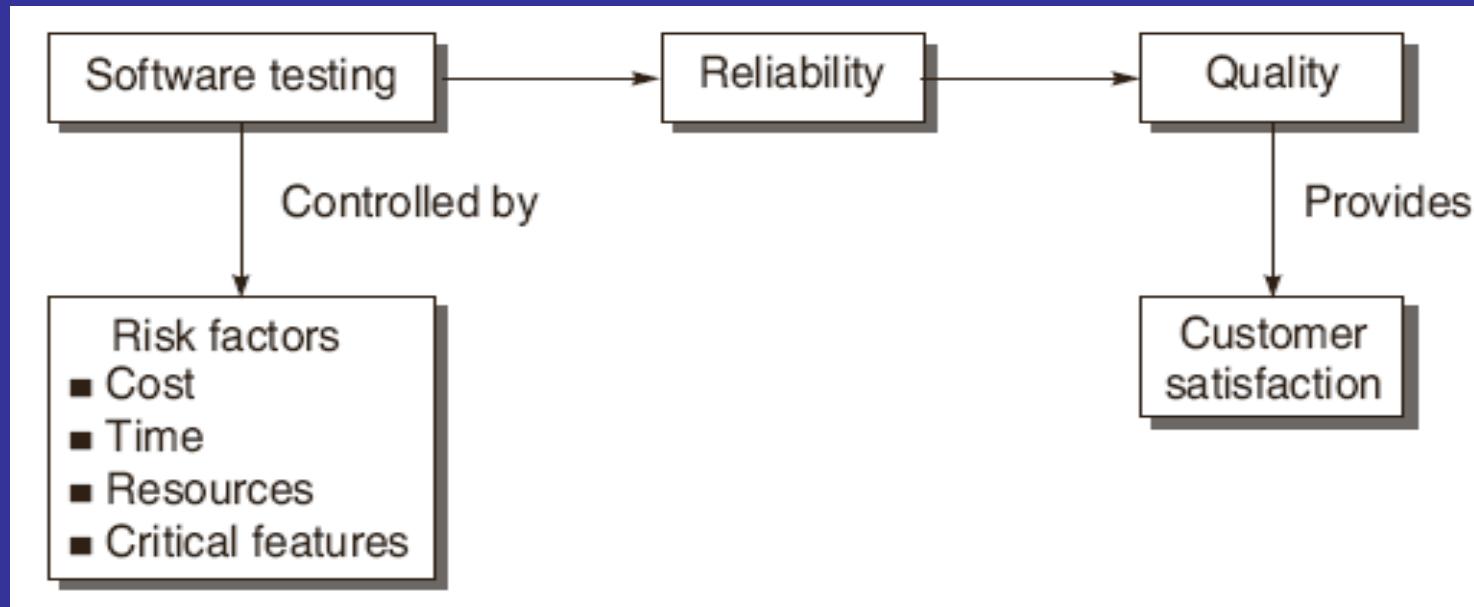
Testing produces Reliability and Quality



Quality leads to customer satisfaction



Testing control Risk factors



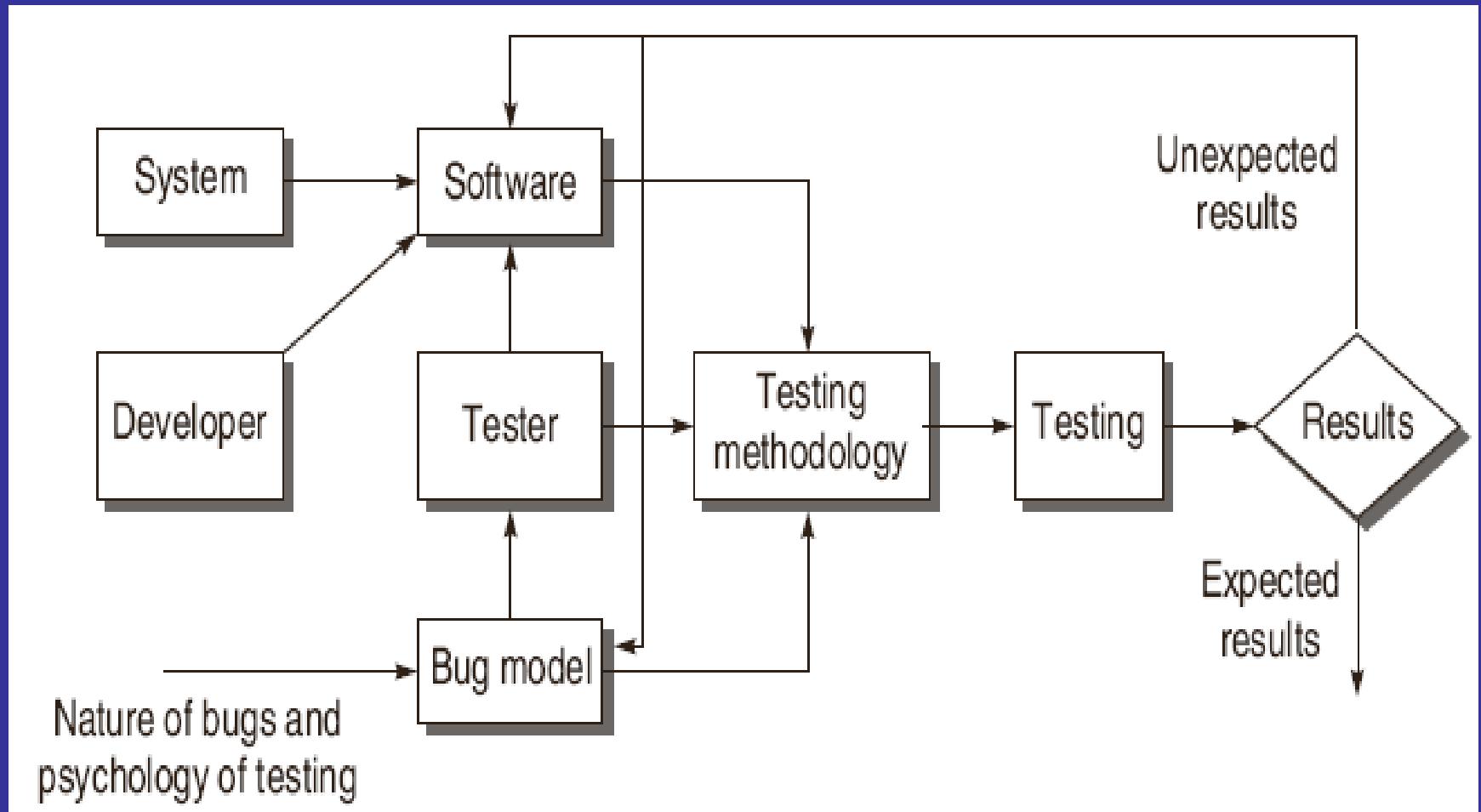
Software Testing Definitions

- “*Testing is the process of executing a program with the intent of finding errors.*”
- Myers [2]
- “*A successful test is one that uncovers an as-yet-undiscovered error.*”
- Myers [2]
- “*Testing can show the presence of bugs but never their absence.*”
- W. Dijkstra [125].

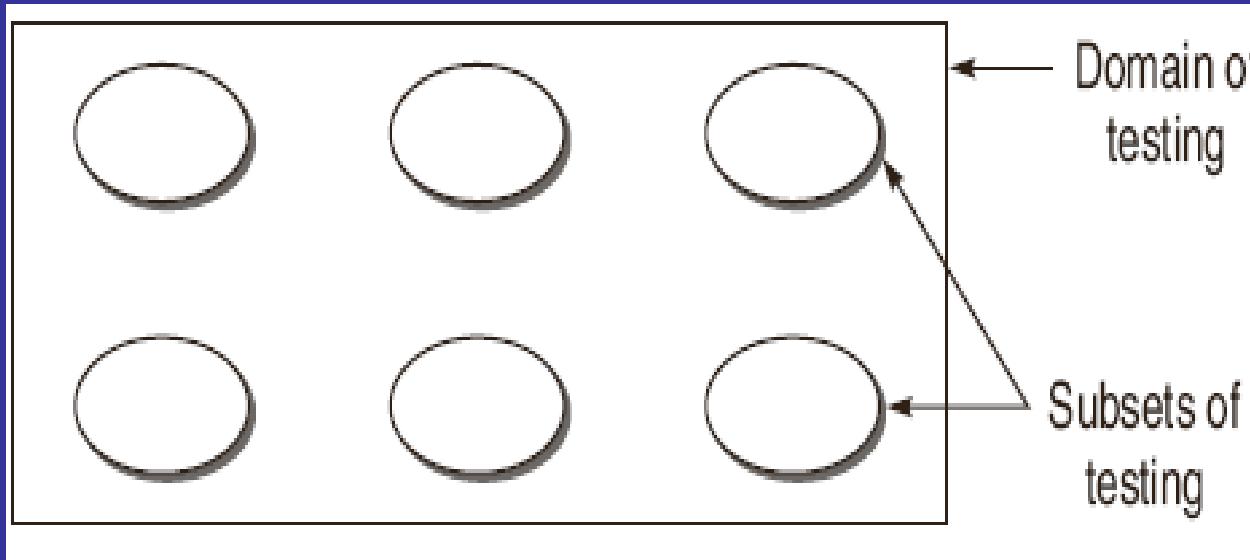
Software Testing Definitions

- “*Testing is a concurrent lifecycle process of engineering, using and maintaining testware (i.e. testing artifacts) in order to measure and improve the quality of the software being Tested.*”
- **Craig [117]**
- “*Software testing is a process that detects important bugs with the objective of having better quality software.*”

Model for Software Testing



Effective Software Testing vs Exhaustive Software Testing



Effective Software Testing vs Exhaustive Software Testing

The domain of possible inputs to the software is too large to test.

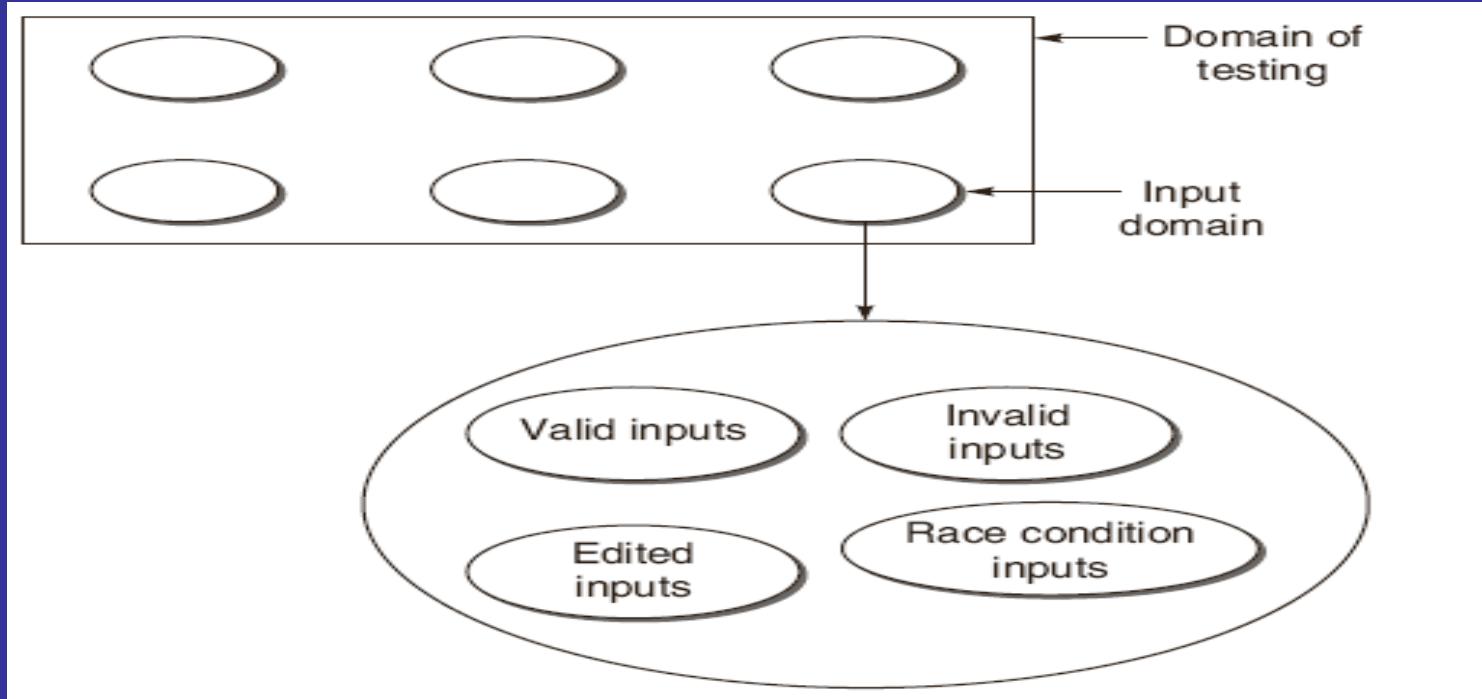
- **Valid Inputs**
- **Invalid Inputs**
- **Edited Inputs**
- **Race Conditions**

Effective Software Testing vs Exhaustive Software Testing

P2 - Exhaustive testing is impossible

- Testing everything is not feasible
- Instead of exhaustive testing, risk analysis and priorities should be used to focus testing efforts
- For example: In an application in one screen if there are 15 input fields, each having 5 possible values, then to test all the valid combinations you would need 30517578125 (5^{15}) tests. This is very unlikely that the project timescales would allow for this number of tests.

Effective Software Testing vs Exhaustive Software Testing

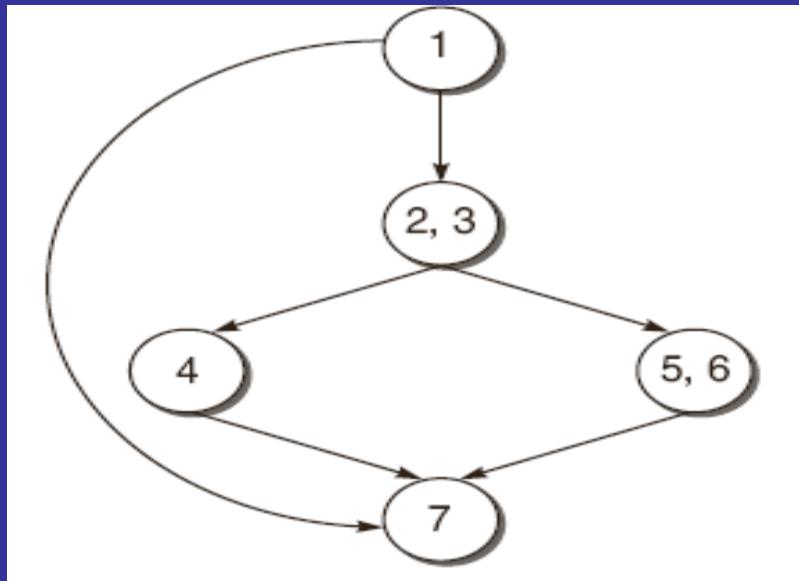


Effective Software Testing vs Exhaustive Software Testing

There are too many possible paths through the program to test.

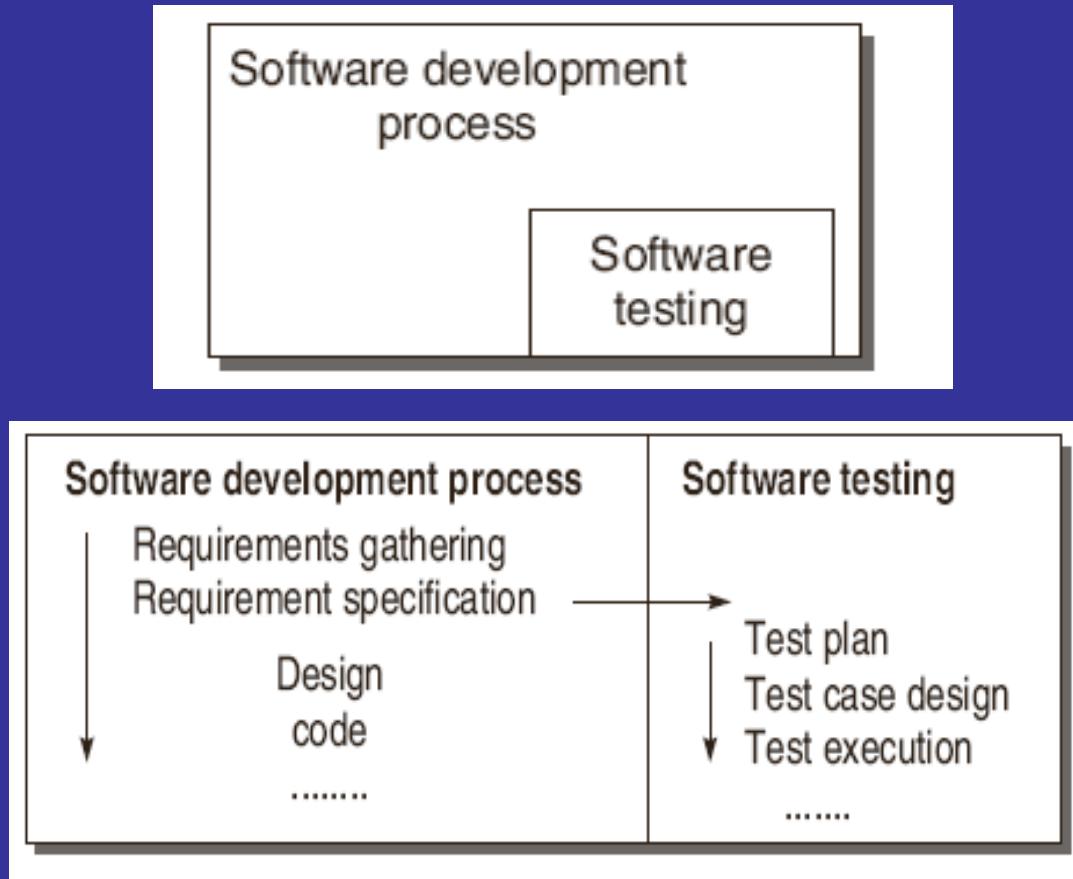
```
1for (int i = 0; i < n; ++i)
2{
3    if (m >=0)
4        x[i] = x[i] + 10;
5    else
6        x[i] = x[i] - 2;
7}...
```

Effective Software Testing vs Exhaustive Software Testing



- Total number of paths will be $2^n + 1$, where n is the number of times the loop will be carried out.
- if n is 20, then the number of paths will be $2^{20} + 1$, i.e. 1048577.
- **Every Design error cannot be found.**

Software Testing as a Process



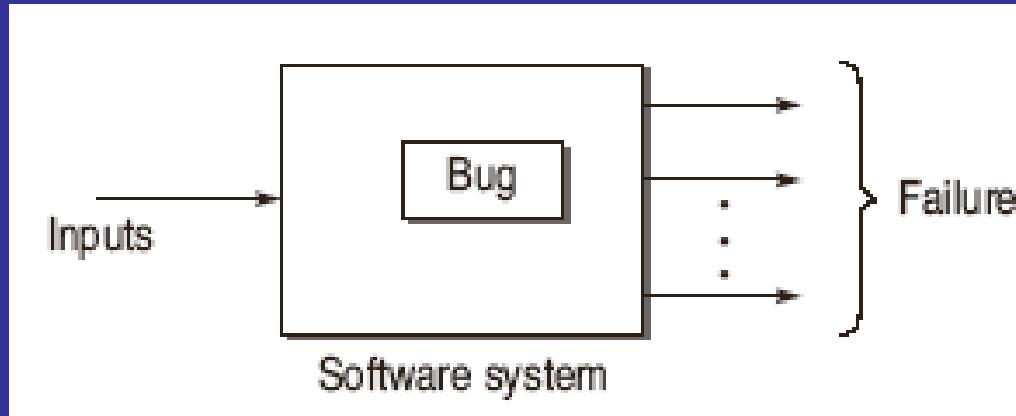
Software Testing as a Process

An organization for the better quality software must adopt a testing process and consider the following points:

- Testing process should be organized such that there is enough time for important and critical features of the software.
- Testing techniques should be adopted such that these techniques detect maximum bugs.
- Quality factors should be quantified so that there is clear understanding in running the testing process. In other words, process should be driven by the quantified quality goals. In this way, process can be monitored and measured.
- Testing procedures and steps must be defined and documented.
- There must be scope for continuous process improvement.

Software Testing Terminology

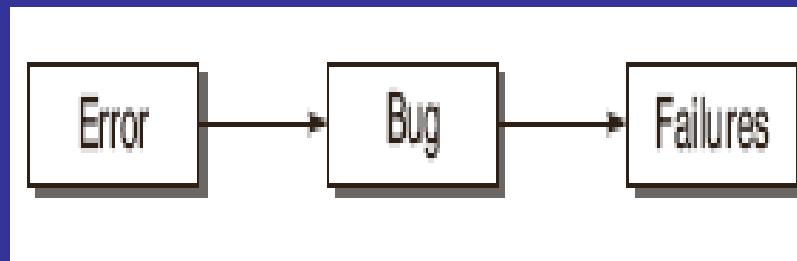
- **Failure**
The inability of a system or component to perform a required function according to its specification.
- **Fault / Defect / Bug**
Fault is a condition that in actual causes a system to produce failure. It can be said that failures are manifestation of bugs.



Software Testing Terminology

Error

Whenever a member of development team makes any mistake in any phase of SDLC, errors are produced. It might be a typographical error, a misleading of a specification, a misunderstanding of what a subroutine does and so on. Thus, error is a very general term used for human mistakes.



Software Testing Terminology

Module A()

{

While(a > n+1);

{

print("The value of x is",x);

}

}

Software Testing Terminology

Test Case is a well – documented procedure designed to test the functionality of a feature in the system.

Test Case ID
Purpose
Preconditions
Inputs
Expected Outputs

Software Testing Terminology

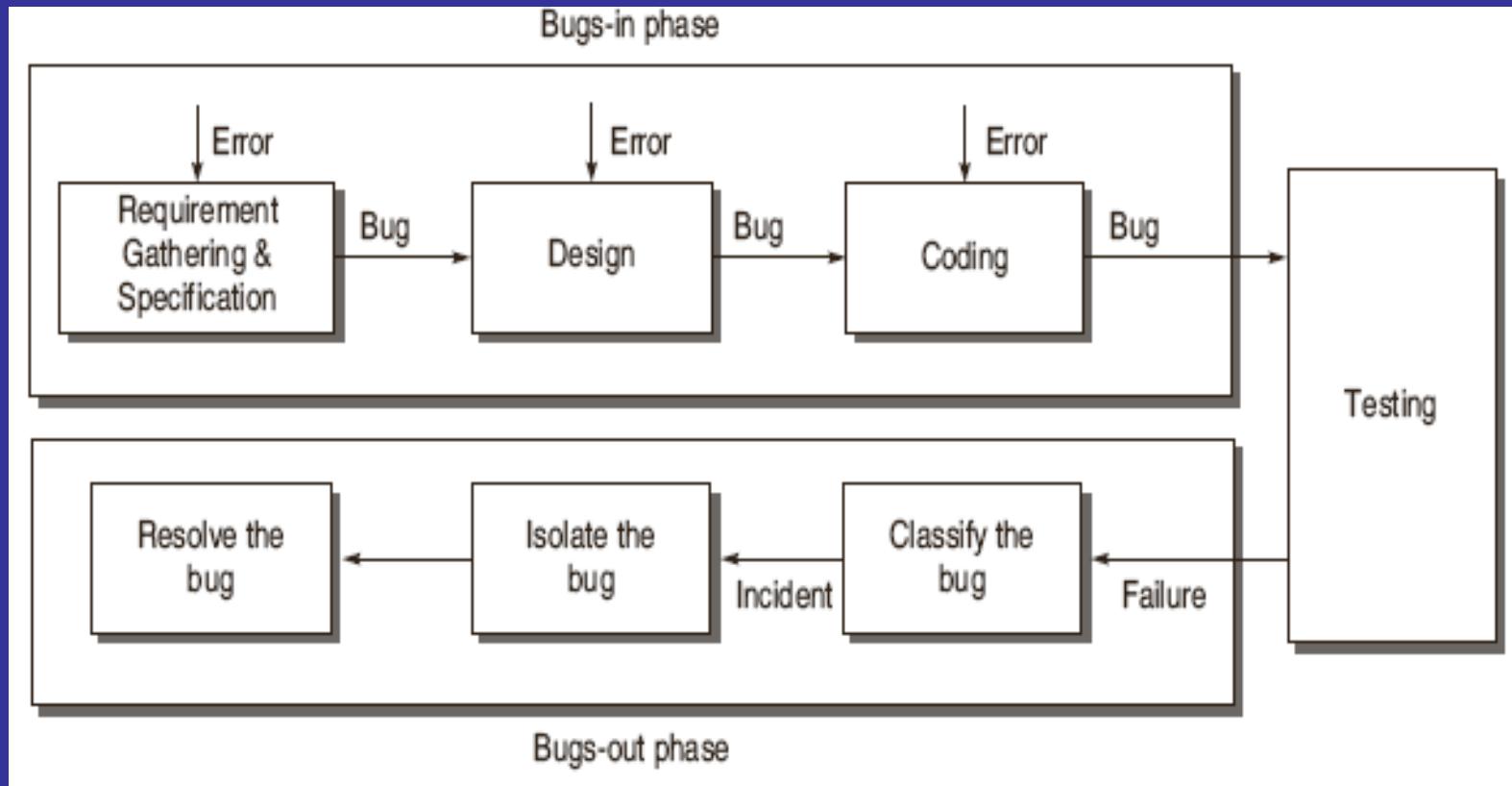
- **Testware**

The documents created during the testing activities are known as Testware. (Test Plan, test specifications, test case design , test reports etc.)
- **Incident**

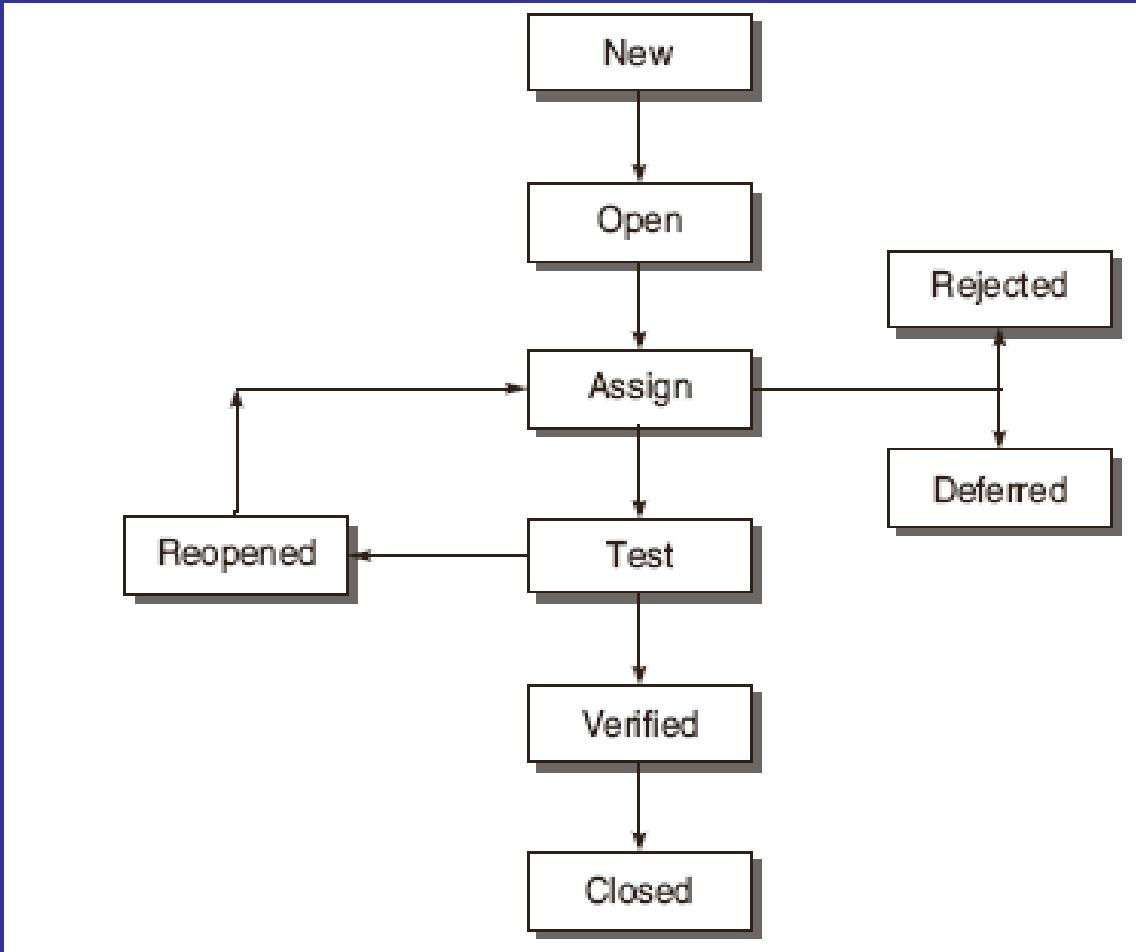
The symptom(s) associated with a failure that alerts the user to the occurrence of a failure.
- **Test Oracle**

To judge the success or failure of a test(correctness of the system for some test) *Comparing actual results with expected results by hand.*

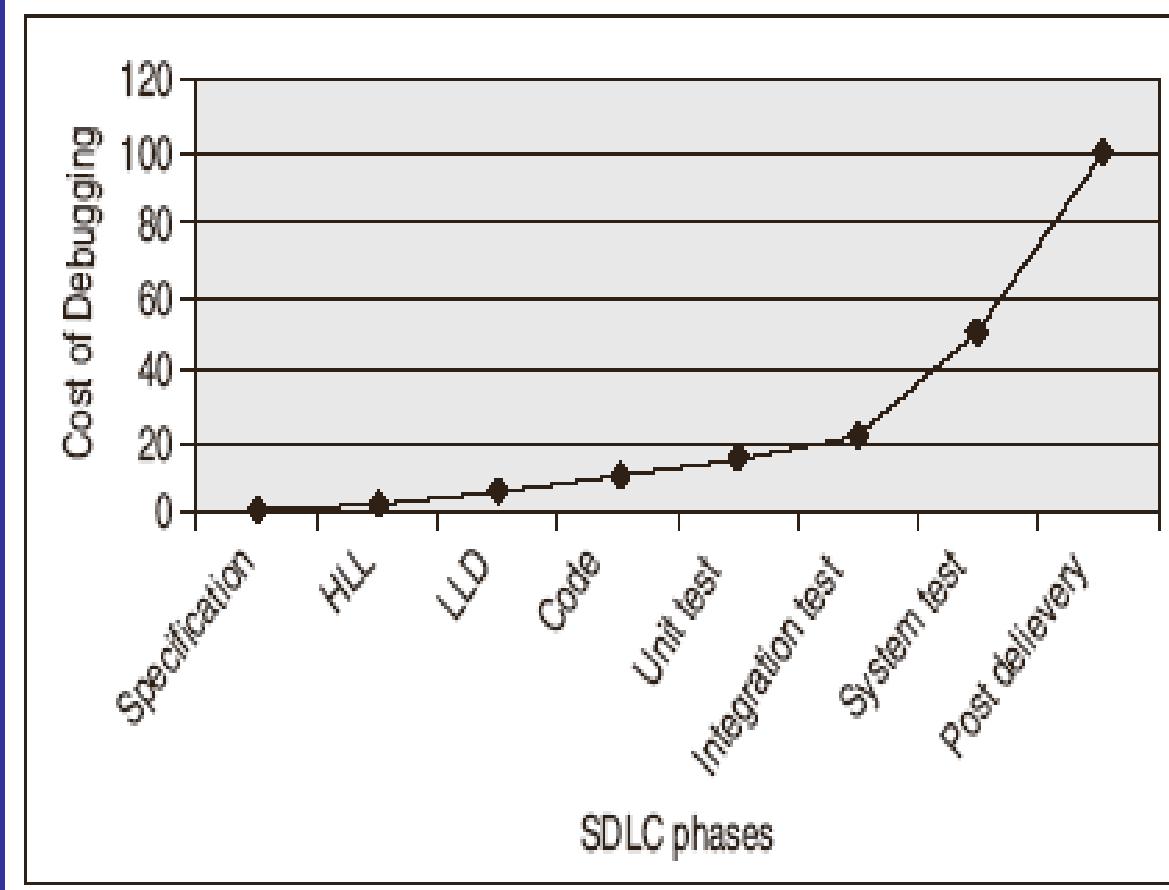
Life Cycle of a Bug



States of a Bug



Bug affects Economics of Software Testing



Bug Classification based on Criticality

- **Critical Bugs**
the worst effect on the functioning of software such that it stops or hangs the normal functioning of the software.
- **Major Bug**
This type of bug does not stop the functioning of the software but it causes a functionality to fail to meet its requirements as expected.
- **Medium Bugs**
Medium bugs are less critical in nature as compared to critical and major bugs.(not according to standards- Redundant /Truncated output)
- **Minor Bugs**
This type of bug does not affect the functioning of the software.(Typographical error or misaligned printout)

Bug Classification based on SDLC

Requirements and Specifications Bugs

Design Bugs

Control Flow Bugs

Logic Bugs: Improper layout of cases, missing cases

Processing Bugs: Arithmetic errors, Incorrect data conversion

Data Flow Bugs

Error Handling Bugs

Race Condition Bugs

Boundary Related Bugs

User Interface Bugs

Coding Bugs

Interface and Integration Bugs

System Bugs

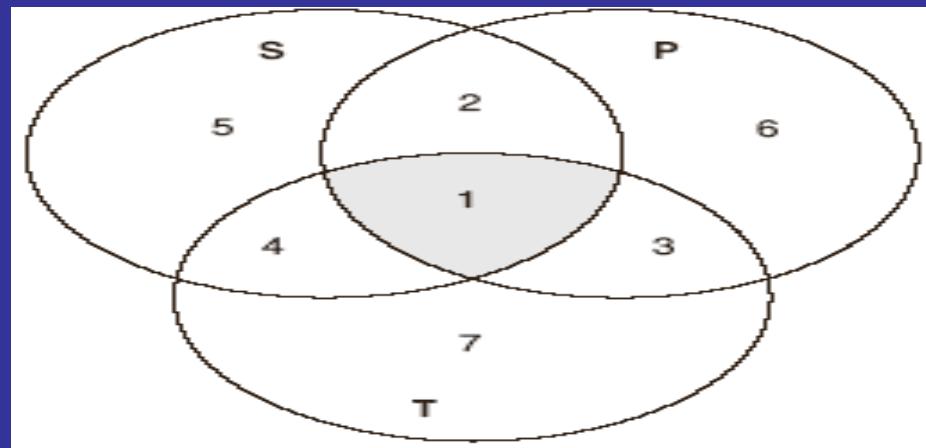
Testing Bugs

Testing Principles

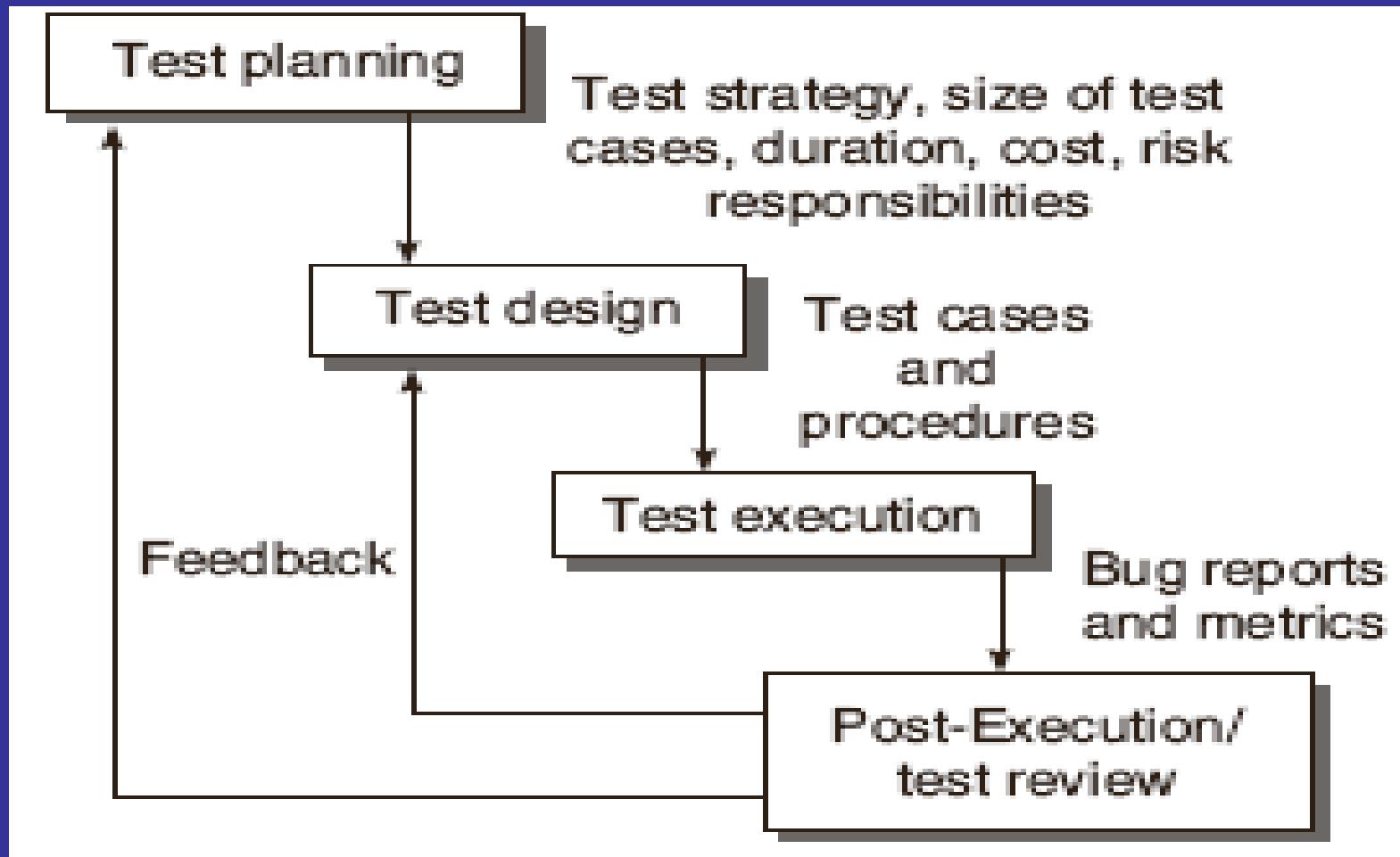
- **Effective Testing not Exhaustive Testing**
- **Testing is not a single phase performed in SDLC**
- **Destructive approach for constructive testing**
- **Early Testing is the best policy.**
- **The probability of the existence of an error in a section of a program is proportional to the number of errors already found in that section.**
- **Testing strategy should start at the smallest module level and expand toward the whole program.**

Testing Principles

- 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 Life Cycle (STLC): Well defined series of steps to ensure successful and effective testing.



Software Testing Life Cycle (STLC):Well defined series of steps to ensure successful and effective testing.

- 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.

Test Planning

- Defining the Test Strategy
- Estimate of 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, project metrics

Test Planning

The major output of test planning is the test plan document. Test plans are developed for each level of testing. After analysing the issues, the following activities are performed:

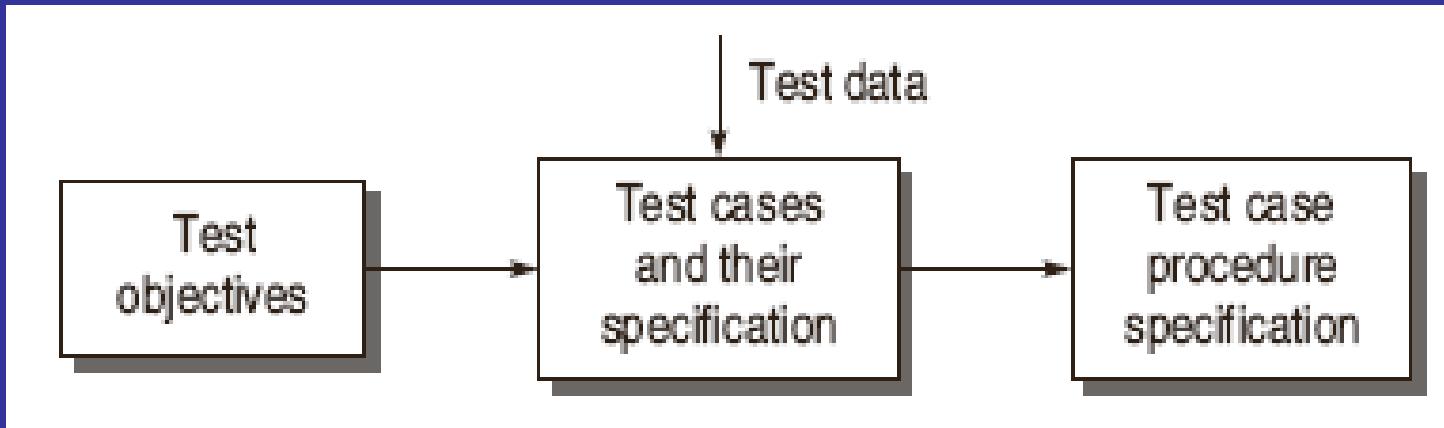
- Develop a test case format.
- Develop test case plans according to every phase of SDLC.
- Identify test cases to be automated.
- Prioritize the test cases according to their importance and criticality.
- Define areas of stress and performance testing.
- Plan the test cycles required for regression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

Test Plan: A document describing the scope, approach, resources and schedule of intended test activities as a roadmap.

Test Design

- Determining the test objectives and their Prioritization(broad categories of things to test)
- Preparing List of Items to be Tested under each objective
- Mapping items to test cases
- Selection of Test case design techniques(black box and white box)
- Creating Test Cases and Test Data
- Setting up the test environment and supporting tools
- Creating Test Procedure Specification(description of how the test case will be run). It is in the form of sequenced steps. This procedure is actually used by the tester at the time of execution of test cases.

Test Design

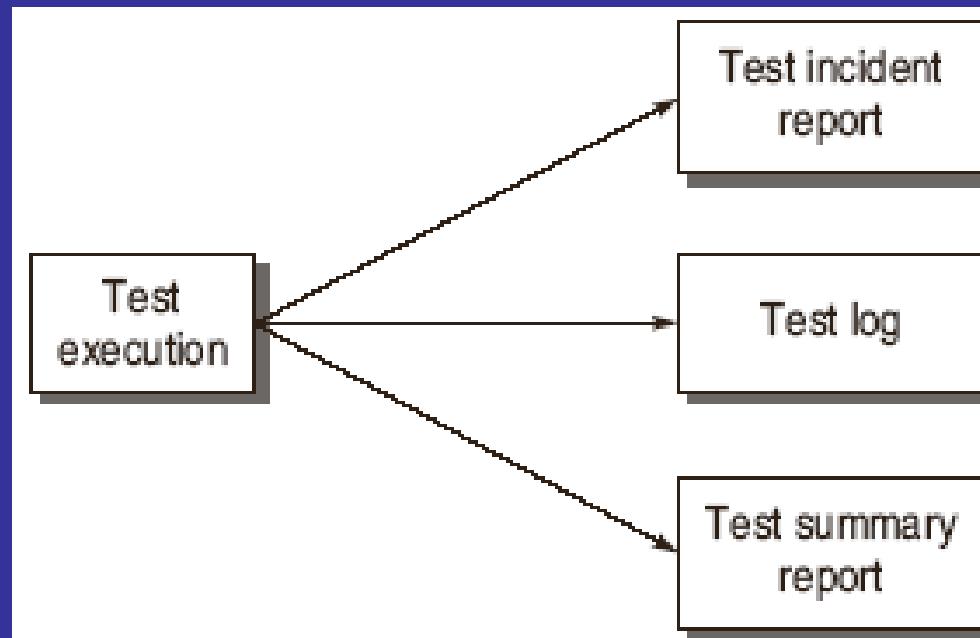


All the details specified in the test design phase are documented in the test design specification. This document provides the details of the input specifications, output specifications, environmental needs, and other procedural requirements for the test case.

Test Execution: Verification and Validation

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 etc.



Post-Execution / Test Review

After successful test execution, bugs will be reported to the concerned developers. This phase is to analyse bug-related issues and get feedback so that maximum number of bugs can be removed.

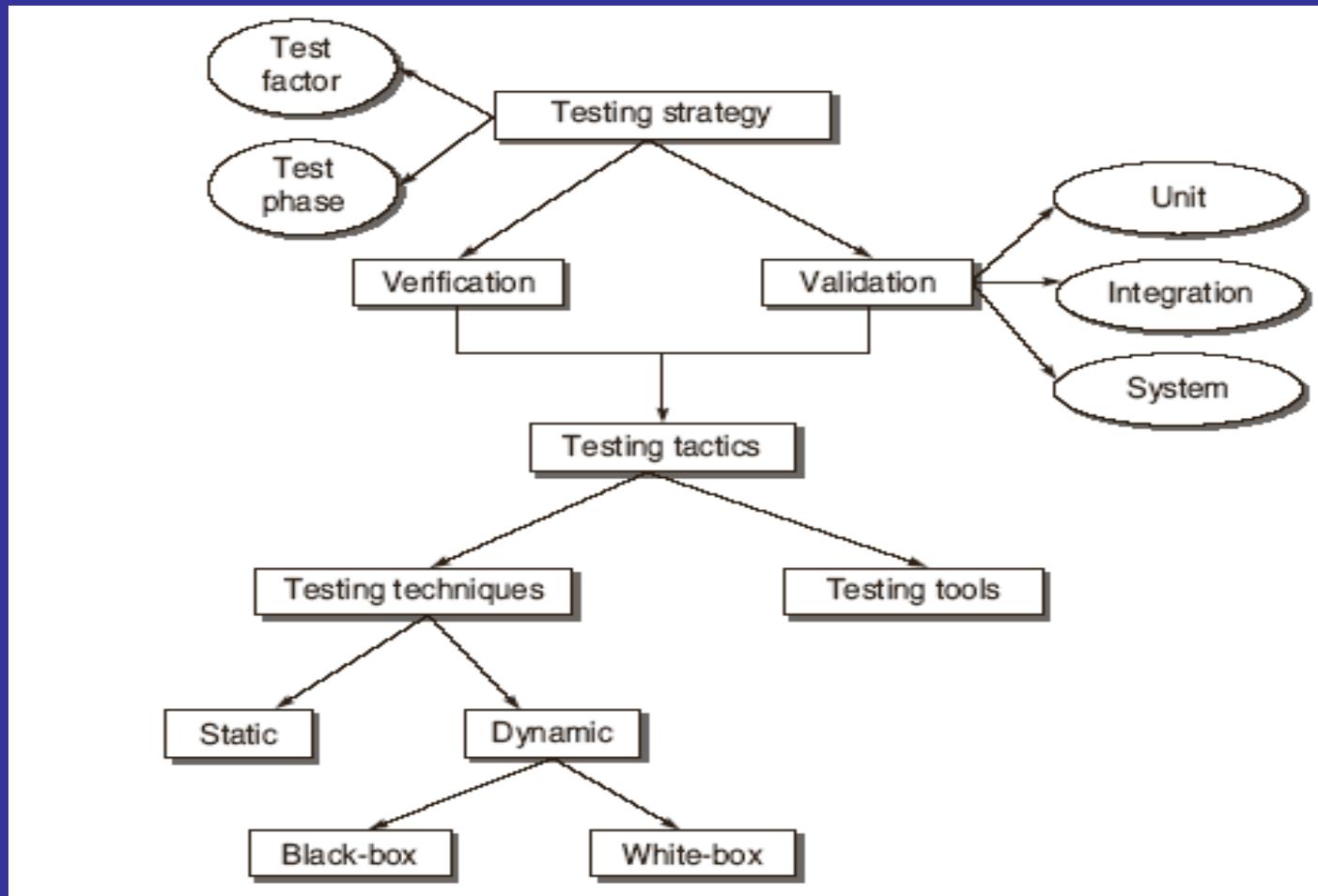
As soon as developer gets the bug report, he perform the following activities:

- Understanding the Bug
- Reproducing the bug
- Analyzing the nature and cause of the bug

Review Process:

- *Reliability analysis*
- *Coverage analysis*
- *Overall defect analysis* (Quality Improvement)

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



Test Strategy

Planning of the whole testing process into a well-planned series of steps. Test strategy provides a roadmap that includes very specific activities that must be performed by the test team in order to achieve a specific goal.

- Test Factors Test factors are risk factors or issues related to the system under development. Risk factors need to be selected and ranked according to a specific system under development.
- Test Phase This is another component on which the testing strategy is based. It refers to the phases of SDLC where testing will be performed. Testing strategy may be different for different models of SDLC, e.g. strategies will be different for waterfall and spiral models.

Test Strategy Matrix:

A test strategy matrix identifies the concerns that will become the focus of test planning and execution.

In this way, this matrix becomes an input to develop the testing strategy.

- **Select and Rank Test Factors**
- **Identify the System Development Phases**
- **Identify the Risks(concerns) associated with System under Development**
- **Plan the test strategy for every risk identified.**

Let's take a project as an example. Suppose a new operating system has to be designed, which needs a test strategy.

Test Factors	Test Phase					
	Requirements	Design	Code	Unit test	Integration test	System test
Portability	Is portability feature mentioned in specifications according to different hardware?					Is system testing performed on MIPS and INTEL platforms?
Service Level	Is time frame for booting mentioned?	Is time frame incorporated in design of the module?				

Risks and test objectives - examples

Risk	Test Objective
The web site fails to function correctly on the user's client operating system and browser configuration.	To demonstrate that the application functions correctly on selected combinations of operating systems and browser version combinations.
Bank statement details presented in the client browser do not match records in the back-end legacy banking systems.	To demonstrate that statement details presented in the client browser reconcile with back-end legacy systems.
Vulnerabilities that hackers could exploit exist in the web site networking infrastructure.	To demonstrate through audit, scanning and ethical hacking that there are no security vulnerabilities in the web site networking infrastructure.

Load Testing Objectives

- "To have a response time under XX seconds"
- "Error rate is under XX%"
- "The infrastructure can handle XXX users"

Development of Test Strategy

The testing strategy should start at the component level and finish at the integration of the entire system. Thus, a test strategy includes testing the components being built for the system, and slowly shifts towards testing the whole system. This gives rise to two basic terms—Verification and Validation—the basis for any type of testing. It can also be said that the testing process is a combination of verification and validation.

Software Verification : The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

Software Validation : The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements . (in conformance with customer expectation)

Verification: “Are we building the product right?”

Validation: “Are we building the right product?”

Verification would check the design doc and correcting the spelling mistake.
consider the following specification

A clickable button with name Submit

- Otherwise, the development team will create a button like



- So new specification is

A clickable button with name Submit

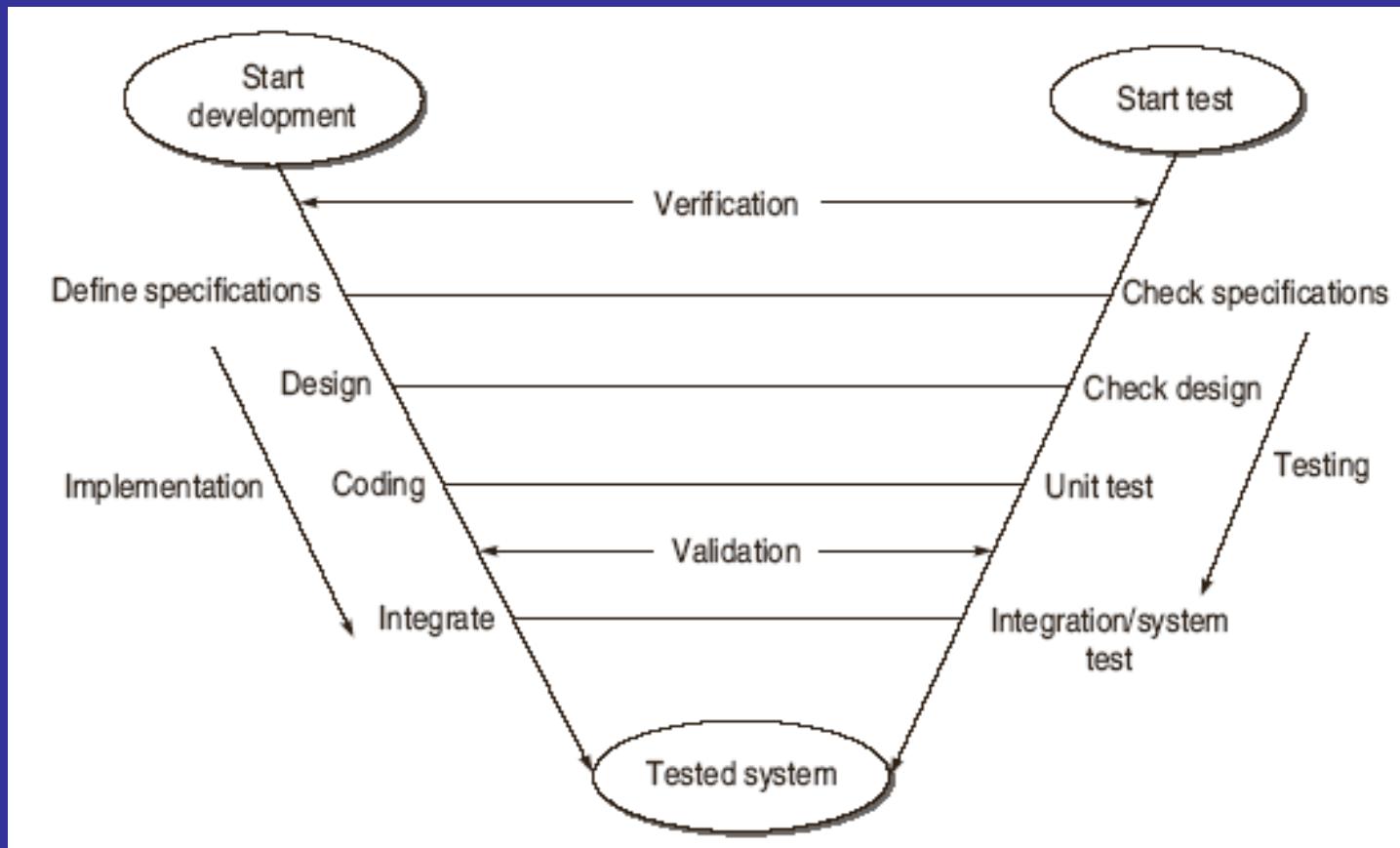
- Once the code is ready, Validation is done. A Validation test found –

Button **NOT Clickable**



Owing to Validation testing, the development team will make the submit button clickable

V Testing Life Cycle Model



Validation Activities

- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing

Testing Tactics

The ways to perform various types of testing under a specific test strategy.

#Manual Testing

#Automated Testing

Software Testing techniques: Methods for design test cases.

#Static Testing

#Dynamic Testing

- * White-Box Testing

- * Black-Box Testing

Testing Tools :

Resource for performing a test process

Considerations in Developing Testing Methodologies

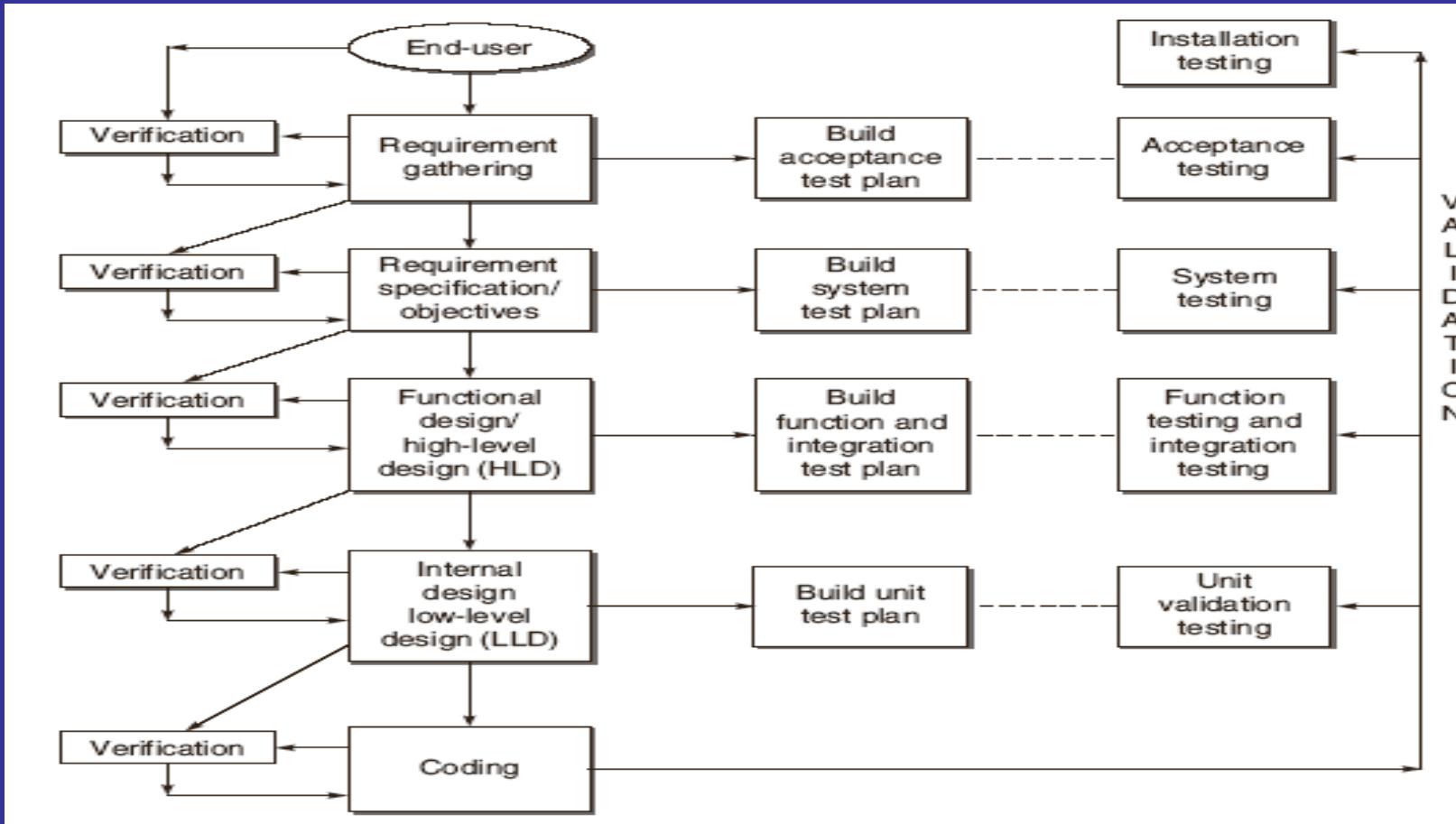
Determine project risks

Determine the type of development project

Identify test activities according to SDLC phase

Build test plan

Verification and Validation (V & V) Activities



VERIFICATION

Verification is a set of activities that ensures correct implementation of specific functions in a software.

Verification is to check whether the software conforms to specifications.

- If verification is not performed at early stages, there are 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.

VERIFICATION ACTIVITIES :All the verification activities are performed in connection with the different phases of SDLC. The following verification activities have been identified:

- Verification of Requirements and Objectives
- Verification of High-Level Design
- Verification of Low-Level Design
- Verification of Coding (Unit Verification)

Verification of Requirements

1. Verification of acceptance criteria

(An acceptance criterion defines the goals and requirements of the proposed system and acceptance limits for each of the goals and requirements.)

2. Acceptance Test plan

1. Verification of Objectives/Specifications(SRS): The purpose of this verification is to ensure that the user's needs are properly understood before proceeding with the project.

2. System Test plan

In verifying the requirements and objectives, the tester must consider both functional and non-functional requirements.

- Correctness
- Unambiguous(Every requirement has only one interpretation.)
- Consistent(No specification should contradict or conflict with another.)
- Completeness
- Updation
- Traceability
 - Backward Traceability
 - Forward Traceability.

Verification of High Level Design

1. The tester verifies the high-level design.
2. The tester also prepares a Function Test Plan which is based on the SRS. This plan will be referenced at the time of Function Testing .
3. The tester also prepares an Integration Test Plan which will be referred at the time of integration testing.
4. The tester verifies that all the components and their interfaces are in tune with requirements of the user. Every requirement in SRS should map the design.

Verification of High Level Design

Data Design: It creates a model of data and/or information that is represented at a high level of abstraction (the customer/user's view of data).

Verification of Data Design

- Check whether 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 requirements in SRS.

Verification of High Level Design

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

Verification of Architectural Design

- Check that every functional requirement in SRS has been taken care in this design.
- Check whether all exceptions handling conditions have been taken care.
- Verify the process of transform mapping and transaction mapping used for transition from the requirement model to architectural design.
- Check the functionality of each module according to the requirements specified.
- Check the inter-dependence and interface between the modules.
 - Coupling and Module Cohesion.

Verification of High Level Design

Interface Design: It creates an effective communication medium between the interfaces of different software modules, interfaces between the software system and any other external entity, and interfaces between a user and the software system.

Verification of Interface Design

- Check all the interfaces between modules according to 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 that the response time for all the interfaces are within required ranges.
- Help Facility
- Error messages and warnings
- Check mapping between every menu option and their corresponding commands.

Verify Low Level Design

- 1.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.
2. The tester also prepares the Unit Test Plan which will be referred at the time of Unit Testing

- Verify the SRS of each module.
- Verify the SDD of each module.
- In LLD, data structures, interfaces and algorithms are represented by design notations; so verify the consistency of every item with their design notations.
- Traceability matrix between SRS and SDD.

How to Verify Code

Coding is the process of converting LLD specifications into a specific language. This is the last phase when we get the operational software with the source code.

- Check that every design specification in HLD and LLD has been coded using traceability matrix.
- Examine the code against a language specification checklist.
- Verify every statement, control structure, loop, and logic
- Misunderstood or incorrect Arithmetic precedence
- Mixed mode operations
- Incorrect initialization
- Precision Inaccuracy
- Incorrect symbolic representation of an expression
- Different data types
- Improper or nonexistent loop termination
- Failure to exit

How to Verify Code

Two kinds of techniques are used to verify the coding:

(a) static testing, and (b) dynamic testing.

Static testing techniques :

This technique does not involve actual execution. It considers only static analysis of the code or some form of conceptual execution of the code.

Dynamic testing techniques:

It is complementary to the static testing technique. 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.

How to Verify Code

UNIT VERIFICATION :

Verification of coding cannot be done for the whole system. Moreover, the system is divided into modules. Therefore, verification of coding means the verification of code of modules by their developers. This is also known as unit verification testing.

Listed below are the points to be considered while performing unit verification :

- Interfaces are verified to ensure that information properly flows in and out of the program unit under test.
- The local data structure is verified to maintain data integrity.
- Boundary conditions are checked to verify that the module is working fine on boundaries also.
- All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.
- All error handling paths are tested.

Unit verification is largely white-box oriented

Validation

- Validation is the next step after verification.
- Validation is performed largely by black box testing techniques.
- Developing tests that will determine whether the product satisfies the users' requirements, as stated in the requirement specification.
- Developing tests that will determine whether the product's actual behavior matches the desired behavior, as described in the functional design specification.
- The bugs, which are still existing in the software after coding need to be uncovered.
- last chance to discover the bugs otherwise these bugs will move to the final product released to the customer.
- Validation enhances the quality of software.

Validation Activities

Validation Test Plan

- **Acceptance Test Plan**
- **System Test Plan**
- **Function Test Plan**
- **Integration Test Plan**
- **Unit Test Plan**
-

Validation Test Execution

- **Unit Validation Testing**
- **Integration Testing**
- **Function Testing**
- **System Testing**
- **Acceptance Testing**
- **Installation Testing**

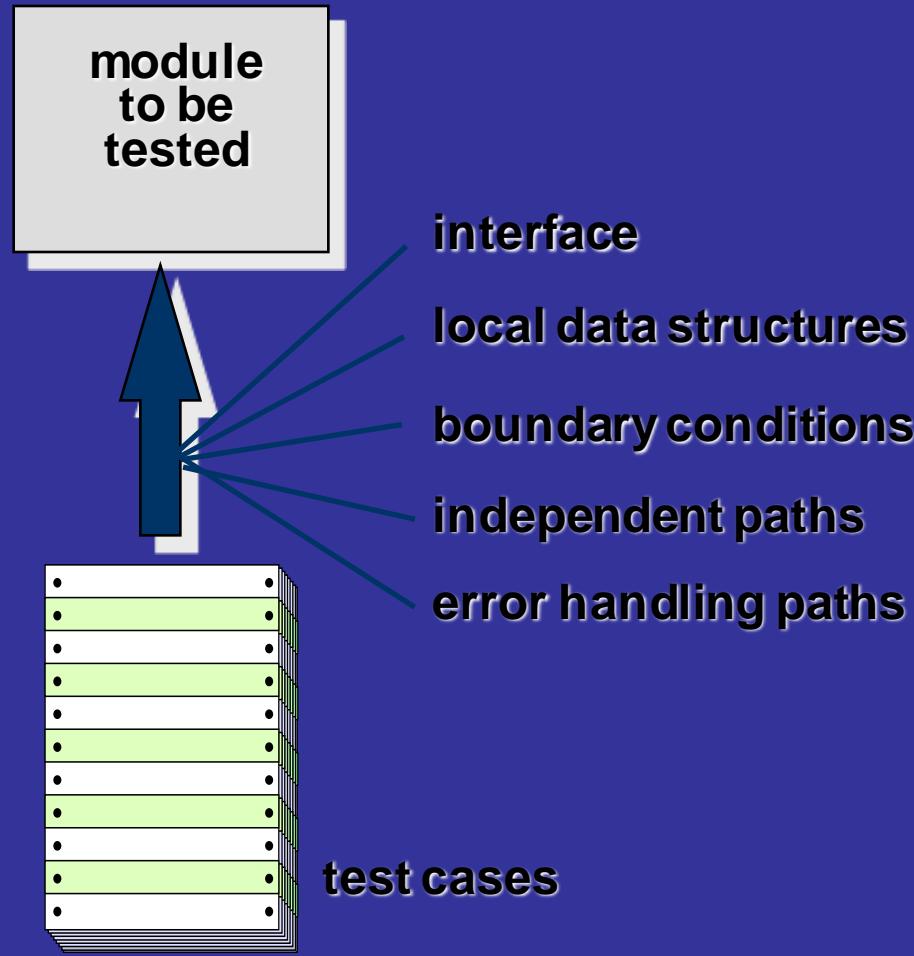
Concept of Unit Testing

- Unit is?
 - Function
 - Procedure
 - Method
 - Module
 - Component

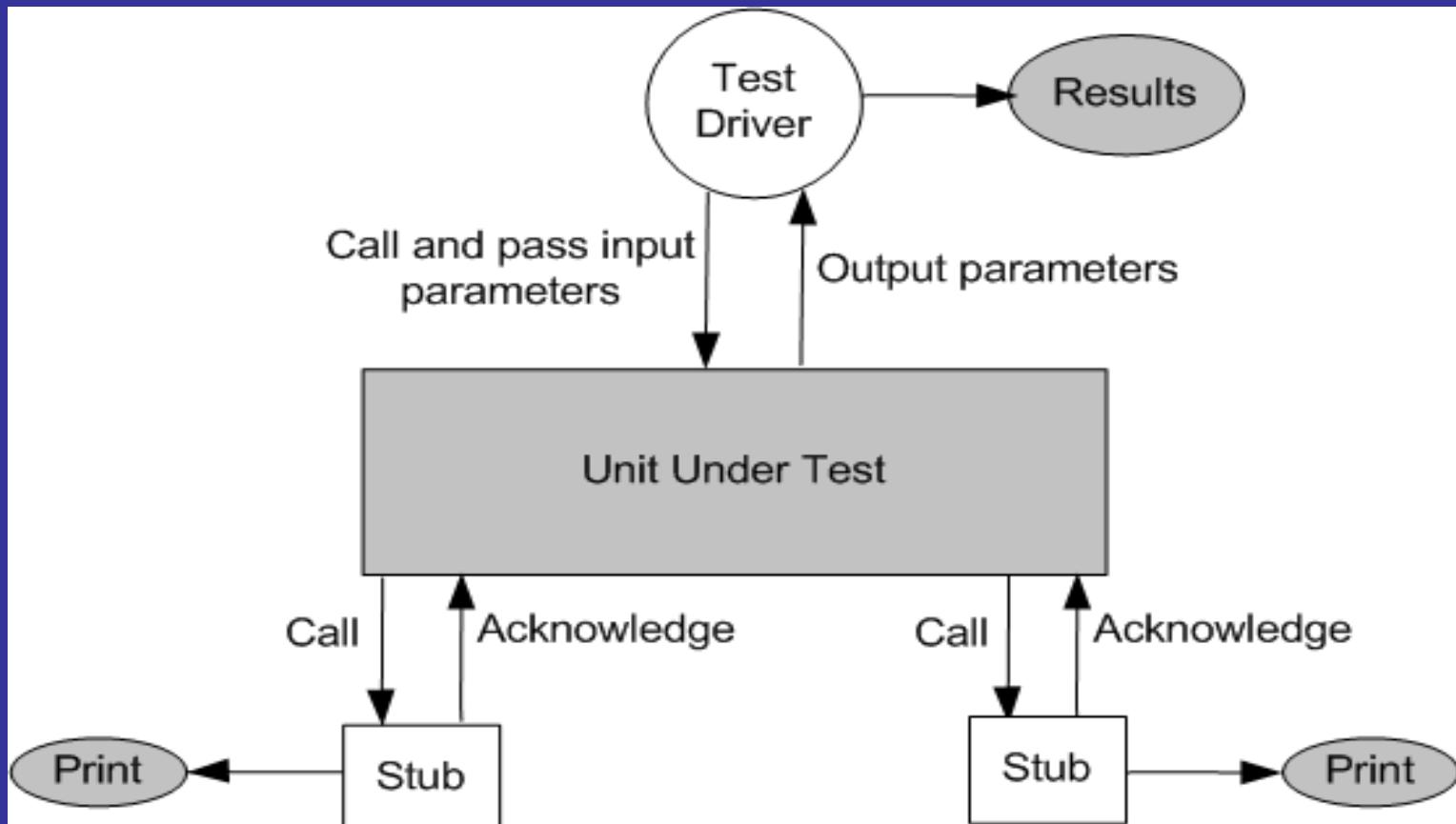
Unit Testing

- Testing program unit in isolation i.e. in a stand alone manner.
- Objective: Unit works as expected.

Unit Testing



Unit Testing



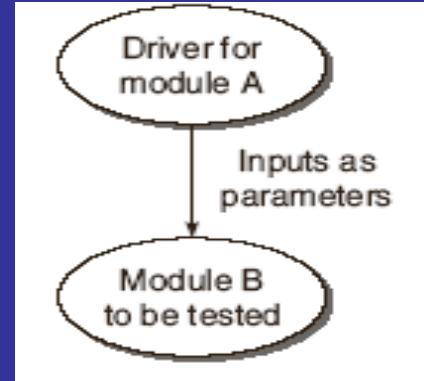
Dynamic unit test environment

Unit Testing

- The environment of a unit is emulated and tested in isolation
- The caller unit is known as *test driver*
 - A *test driver* is a program that invokes the unit under test (UUT)
 - It provides input data to unit under test and report the test result
- The emulation of the units called by the UUT are called *stubs*
 - It is a dummy program
- The *test driver* and the *stubs* are together called *scaffolding*
- The low-level design document provides guidance for selection of input test data

Unit Validation Testing

- **Drivers**

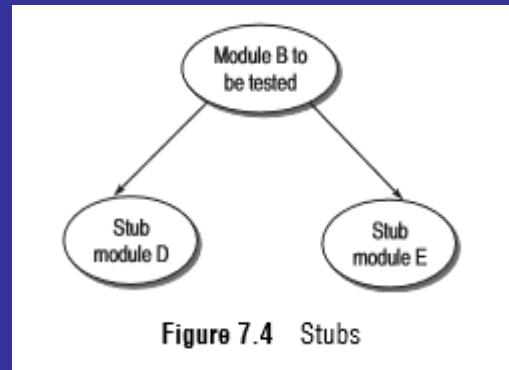


- **A test driver is supporting code and data used to provide an environment for testing part of a system in isolation.**
- **A test driver may take inputs in the following form and call the unit to be tested:**
 - It may hardcode the inputs as parameters of the calling unit.
 - It may take the inputs from the user.
 - It may read the inputs from a file.

Unit Validation Testing

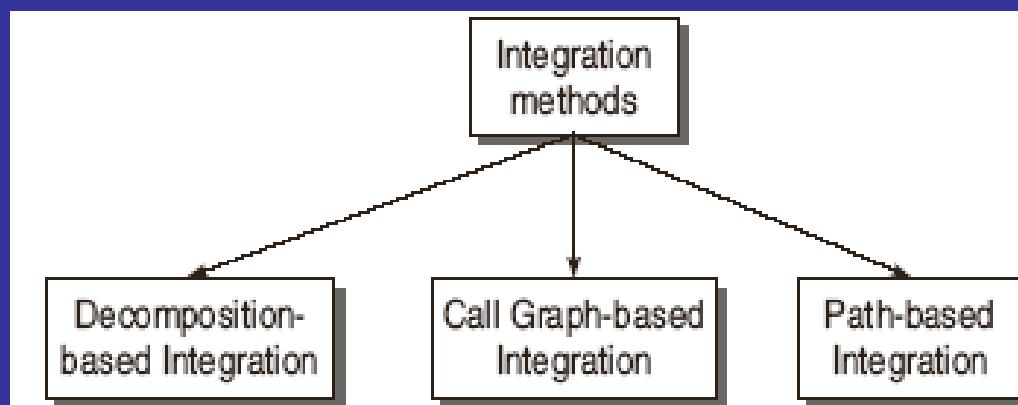
Stubs

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.

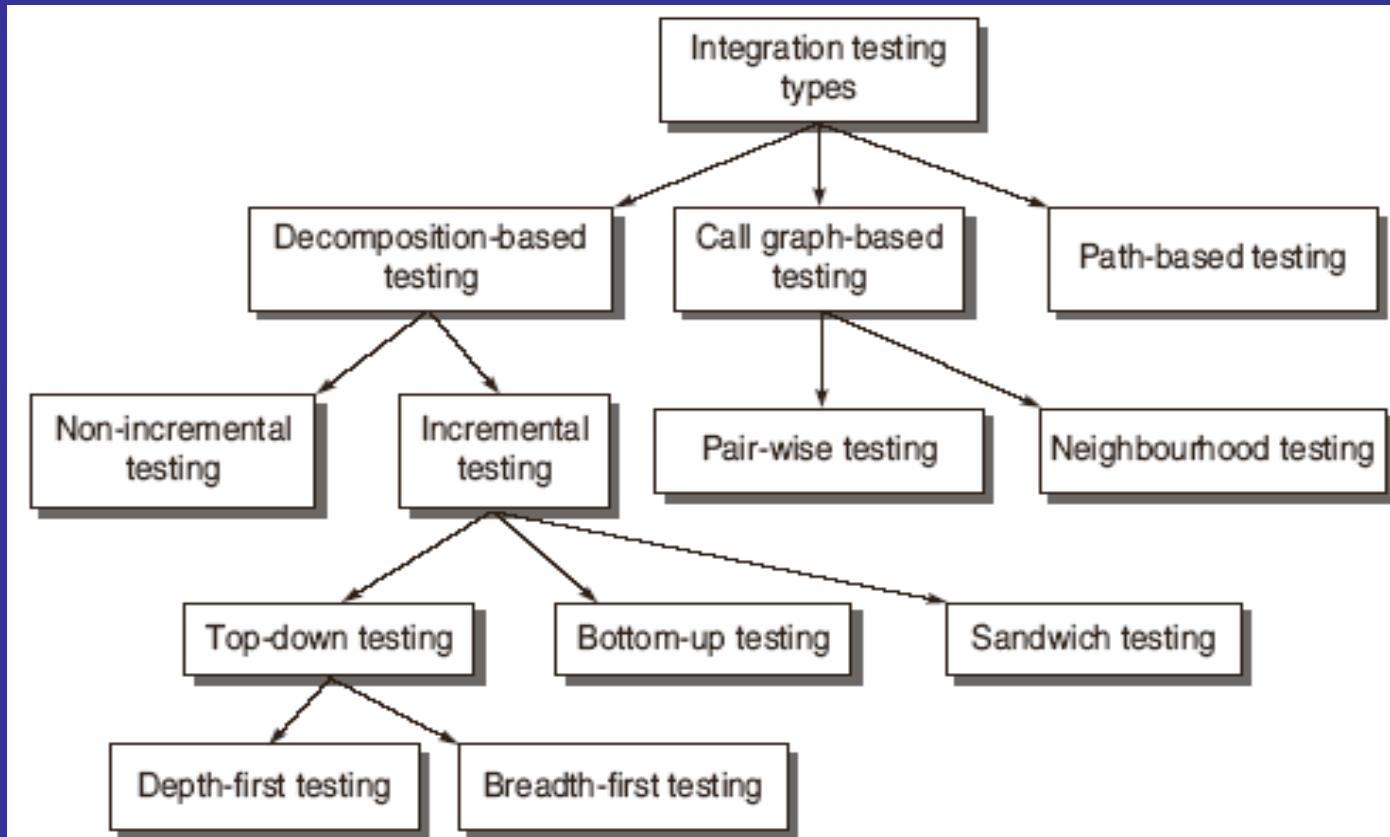


Integration Testing

- Integration testing exposes inconsistency between the modules such as improper call or return sequences.
- Data can be lost across an interface.
- One module when combined with another module may not give the desired result.
- Data types and their valid ranges may mismatch between the modules.



Integration Testing



Decomposition based integration testing

- Based on decomposition of design into functional components or modules.
- The integration testing effort is computed as number of test sessions. A test session is one set of test cases for a specific configuration.

The total test sessions in decomposition based integration is computed as:

$$\text{Number of test sessions} = \text{nodes} - \text{leaves} + \text{edges}$$

Decomposition based integration testing

1. Non-Incremental Integration Testing

- Big Bang Method

2. Incremental Integration Testing

- Top-down Integration Testing
- Bottom – up Integration Testing
- Practical approach for Integration Testing
 - Sandwich Integration Testing

Incremental Integration Testing

Issue	Top-Down Testing	Bottom-Up Testing
Architectural Design	It discovers errors in high-level design, thus detects errors at an early stage.	High-level design is validated at a later stage.
System Demonstration	Since we integrate the modules from top to bottom, the high-level design slowly expands as a working system. Therefore, feasibility of the system can be demonstrated to the top management.	It may not be possible to show the feasibility of the design. However, if some modules are already built as reusable components, then it may be possible to produce some kind of demonstration.
Test Implementation	$(nodes - 1)$ stubs are required for the subordinate modules.	$(nodes - leaves)$ test drivers are required for super-ordinate modules to test the lower-level modules.

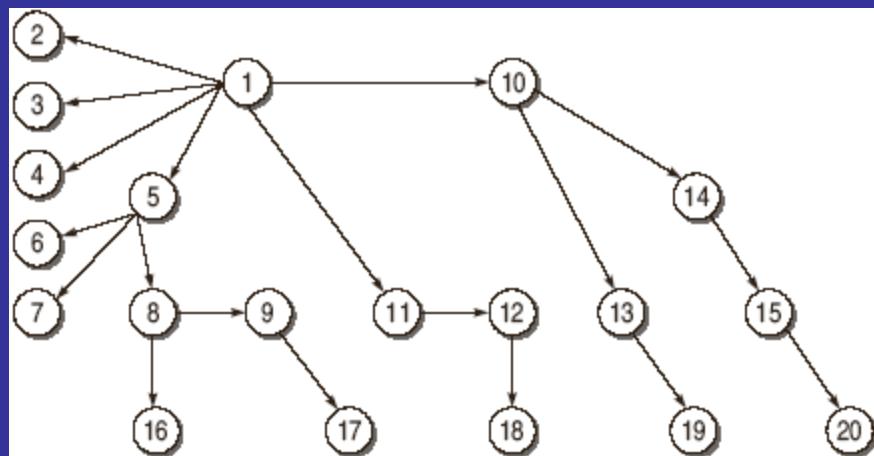
Practical Approach for Integration Testing

- ❖ There is no single strategy adopted for industry practice.
- ❖ For integrating the modules, one cannot rely on a single strategy. There are situations depending upon the project in hand which will force to integrate the modules by combining top-down and bottom-up techniques.
- ❖ This combined approach is sometimes known as Sandwich Integration testing.
- ❖ The practical approach for adopting sandwich testing is driven by the following factors:
 - Priority
 - Availability

Call Graph Based Integration

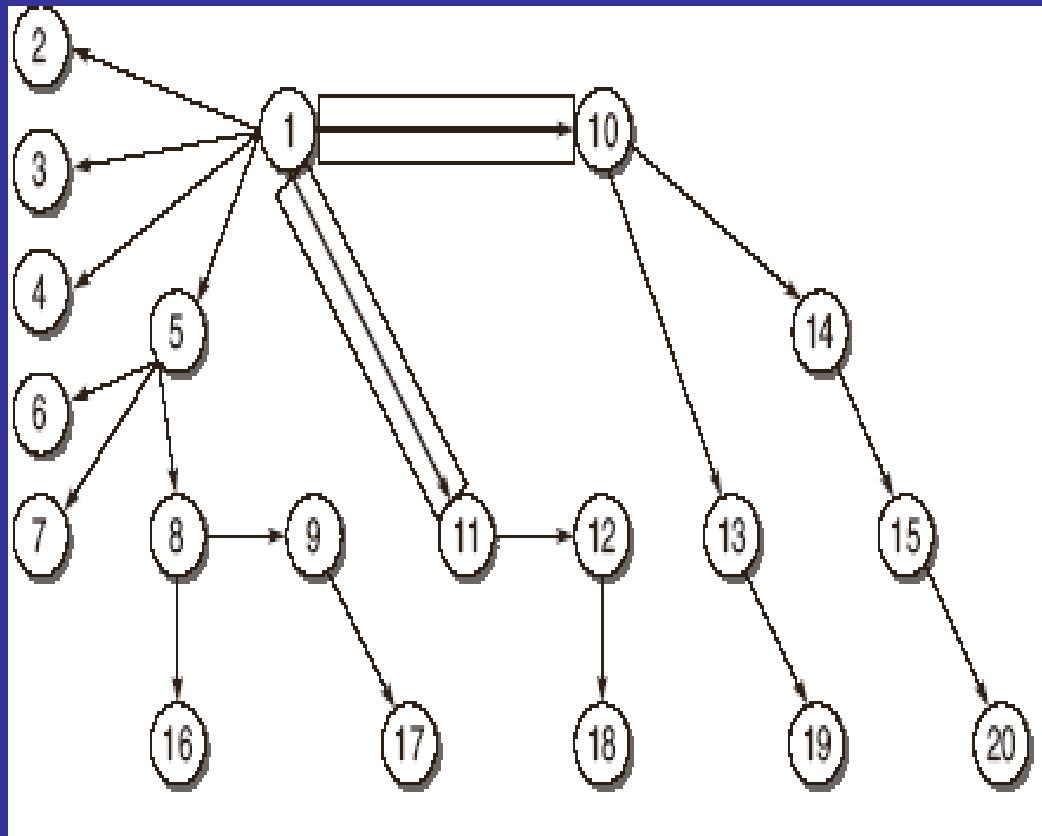
Refine the functional decomposition tree into a form of module calling graph, then we are moving towards behavioural testing at the integration level. This can be done with the help of a call graph

A call graph is a directed graph wherein nodes are modules or units and a directed edge from one node to another node means one module has called another module. The call graph can be captured in a matrix form which is known as adjacency matrix.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	x	x	x	x					x	x									
2																			
3																			
4																			
5							x	x	x										
6																			
7																			
8										x						x			
9																	x		
10											x	x							
11											x								
12												x							
13													x				x		
14													x					x	
15														x					x
16																			
17																			
18																			
19																			
20																			

Pair-wise Integration



Consider only one pair of calling and called modules and then we can make a set of pairs for all such modules.

Number of test sessions = no. of edges in call graph
= 19

Neighborhood Integration

Node	Neighbourhoods	
	Predecessors	Successors
1	---	2,3,4,5,10,11
5	1	6,7,8
8	5	9,16
9	8	17
10	1	13,14
11	1	12
12	11	18
13	10	19
14	10	15
15	14	20

The neighborhood for a node is the immediate predecessor as well as the immediate successor of the node.

The total test sessions in neighborhood integration can be calculated as:

$$\begin{aligned}\text{Neighborhoods} &= \text{nodes} - \text{sink nodes} \\ &= 20 - 10 \\ &= 10\end{aligned}$$

where Sink Node is an instruction in a module at which execution terminates.

Path Based Integration

This passing of control from one unit to another unit is necessary for integration testing. Also, there should be information within the module regarding instructions that call the module or return to the module. This must be tested at the time of integration. It can be done with the help of path-based integration defined by Paul C.

Source Node :It is an instruction in the module at which the execution starts or resumes. The nodes where the control is being transferred after calling the module are also source nodes.

Sink Node: It is an instruction in a module at which the execution terminates. The nodes from which the control is transferred are also sink nodes.

Path Based Integration

Module Execution Path (MEP) : It is a path consisting of a set of executable statements within a module like in a flow graph.

Message: When the control from one unit is transferred to another unit, then the programming language mechanism used to do this is known as a message.

MM-Path (Path consisting of MEPs and messages):

MM-path is a set of MEPs and transfer of control among different units in the form of messages.

MM-Path Graph:

It can be defined as an extended flow graph where nodes are MEPs and edges are messages. It returns from the last called unit to the first unit where the call was made.

In this graph, messages are highlighted with thick lines.

Path Based Integration

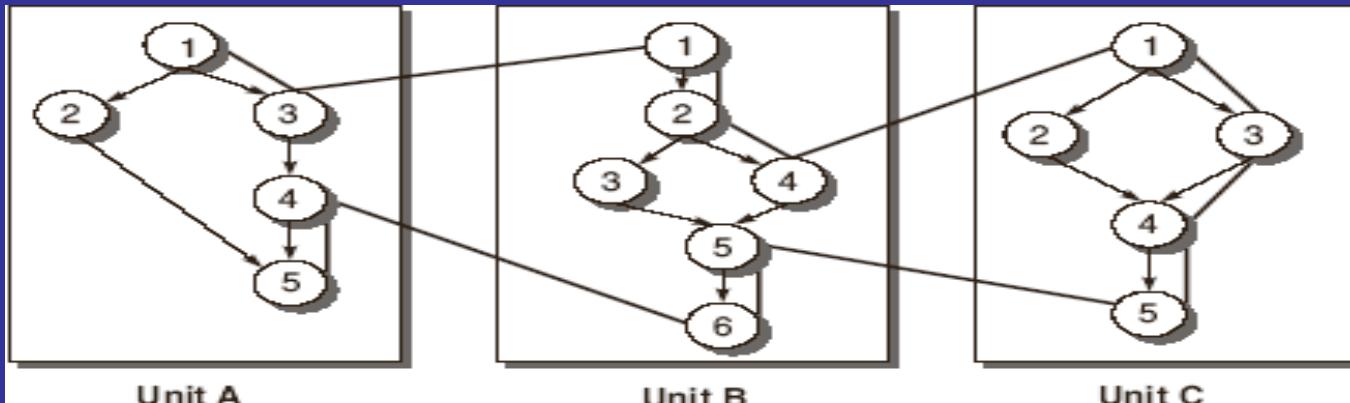


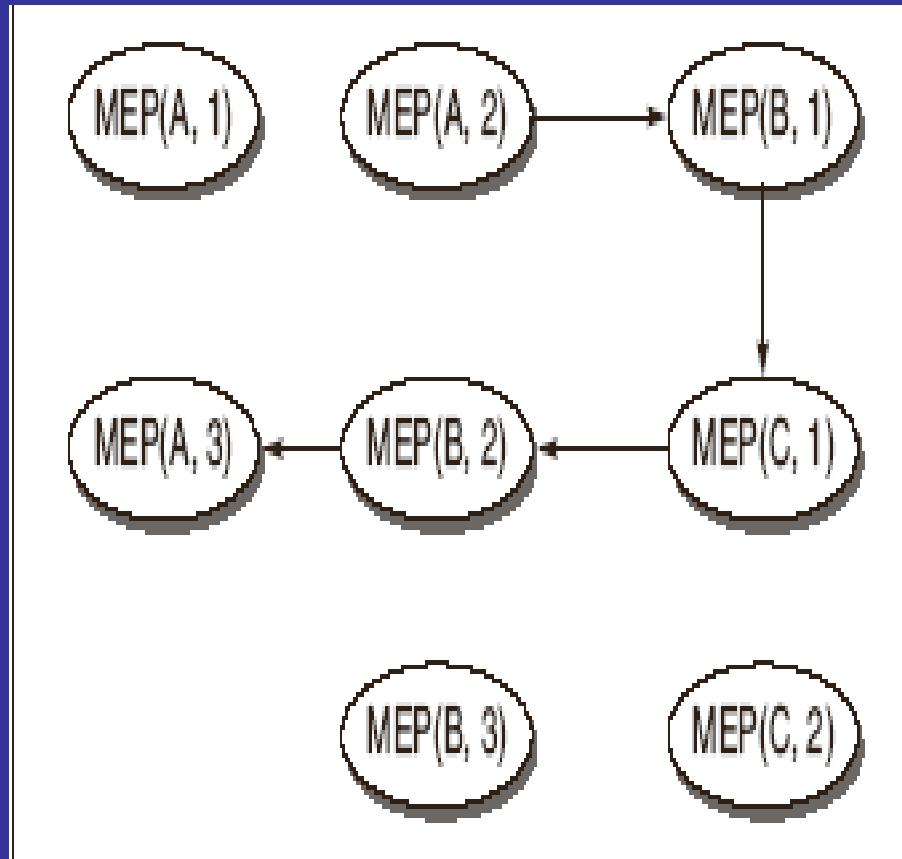
Figure 7.12 MM-path

Table 7.3 MM-path details

	Source Nodes	Sink Nodes	MEPs
Unit A	1,4	3,5	$\text{MEP}(A,1) = \langle 1,2,5 \rangle$ $\text{MEP}(A,2) = \langle 1,3 \rangle$ $\text{MEP}(A,3) = \langle 4,5 \rangle$
Unit B	1,5	4,6	$\text{MEP}(B,1) = \langle 1,2,4 \rangle$ $\text{MEP}(B,2) = \langle 5,6 \rangle$ $\text{MEP}(B,3) = \langle 1,2,3,4,5,6 \rangle$
Unit C	1	5	$\text{MEP}(C,1) = \langle 1,3,4,5 \rangle$ $\text{MEP}(C,2) = \langle 1,2,4,5 \rangle$

Path Based Integration

MEP Graph



Function Testing

Functional Testing is a testing technique that is used to test the features/functionality of the system or Software.

Every functionality of the system specified in the functions is tested according to its external specifications. An external specification is a precise description of the software behaviour from the viewpoint of the outside world

- The process of attempting to detect discrepancies between the functional specifications of a software and its actual behavior.
- The function test must determine if each component or business event:
 - performs in accordance to the specifications,
 - responds correctly to all conditions that may be presented by incoming events / data,
 - moves data correctly from one business event to the next (including data stores), and
 - business events are initiated in the order required to meet the business objectives of the system.

Function Testing

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

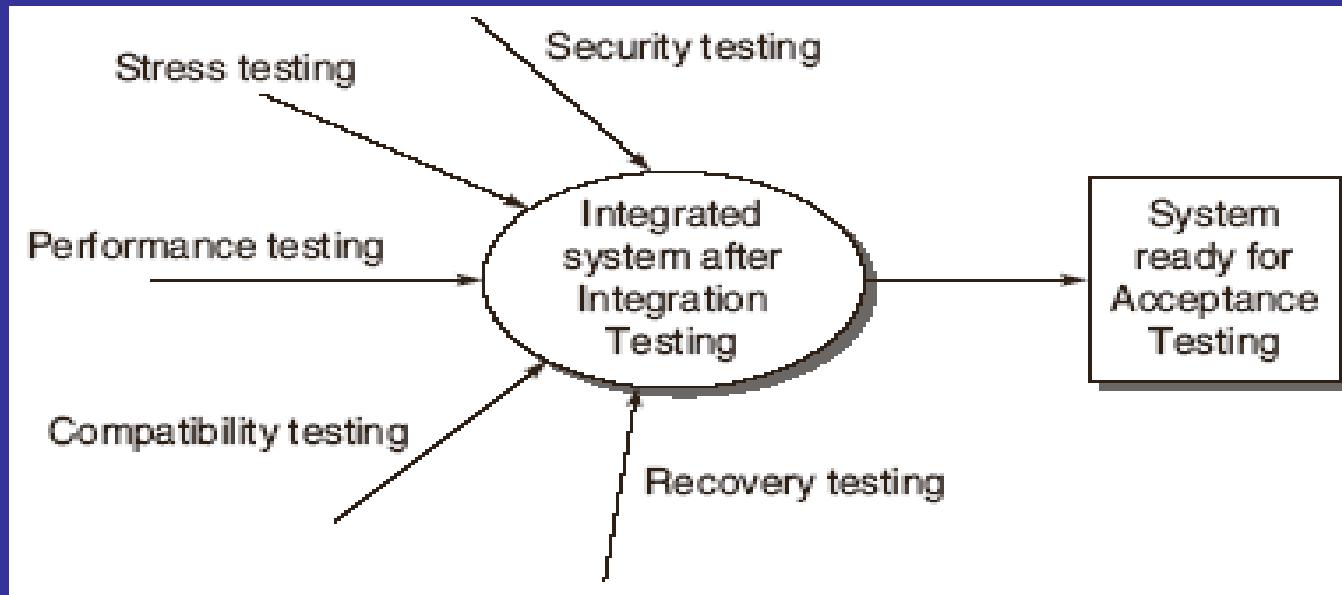
- Test Planning: During planning, the test leader with assistance from the test team defines the scope, schedule, and deliverables for the function test cycle.
- Functional Decomposition
- Requirement Definition: The testing organization needs specified requirements in the form of proper documents to proceed with the function test.
- Test Case Design: A tester designs and implements a test case to validate that the product performs in accordance with the requirements.
- Function Coverage Matrix

Functions	Priority	Test Cases
F1	3	T2,T4
F2	1	T1,T3

- Test Case Execution

System Testing

SYSTEM TESTING is a level of software testing where a complete and integrated software is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements.



Categories of System Tests

Recovery Testing

- Recovery is just like the exception handling feature in a programming language.
- Recovery is the ability of a system to restart operations after the integrity of the application has been lost.
- It reverts to a point where the system was functioning correctly and then reprocesses the transactions up until the point of failure .
- Recovery Testing is the activity of testing how well the software is able to recover from crashes , hardware failures and other similar problems.
- It is the forced failure of the software in various ways to verify that the recovery is properly performed.
 - Checkpoints
 - Swichover

Security Testing

Security tests are designed to verify that the system meets the security requirements. Security may include controlling access to data, encrypting data in communication, ensuring secrecy of stored data, auditing security events, etc

- **Confidentiality**-It is the requirement that data and the processes be protected from unauthorized disclosure
- **Integrity**-It is the requirement that data and process be protected from unauthorized modification
- **Availability**-It is the requirement that data and processes be protected from the denial of service to authorized users
- **Authentication**- A measure designed to establish the validity of a transmission, message, or originator. It allows the receiver to have confidence that the information it receives originates from a specific known source.
- **Authorization**- It is the process of determining that a requester is allowed to receive a service or perform an operation. Access control is an example of authorization.
- **Non-repudiation**- A measure intended to prevent the later denial that an action happened, or a communication took place, etc.

Security Testing

Security Testing is the process of attempting to devise test cases to evaluate the adequacy of protective procedures and countermeasures.

- Security test scenarios should include negative scenarios such as misuse and abuse of the software system.
- Security requirements should be associated with each functional requirement. For example, the log-on requirement in a client-server system must specify the number of retries allowed, the action to be taken if the log-on fails, and so on.
- A software project has security issues that are global in nature, and are therefore, related to the application's architecture and overall implementation. For example, a Web application may have a global requirement that all private customer data of any kind is stored in encrypted form in the database

Security Testing

- Useful types of security tests includes the following:
 - Verify that only authorized accesses to the system are permitted
 - Verify the correctness of both encryption and decryption algorithms for systems where data/messages are encoded.
 - Verify that illegal reading of files, to which the perpetrator is not authorized, is not allowed
 - Ensure that virus checkers prevent or curtail entry of viruses into the system
 - Try to identify any “backdoors” in the system usually left open by the software developers

Performance Testing

Performance testing is to test the run time performance of the system on the basis of various parameters.

- The performance testing requires that performance requirements must be clearly mentioned in SRS and system test plans. The main thing is that these requirements must be quantified.
- For example, a requirement that the system return a response to a query in a reasonable amount is not an acceptable requirement; the time must be specified in a quantitative way.

Performance Testing

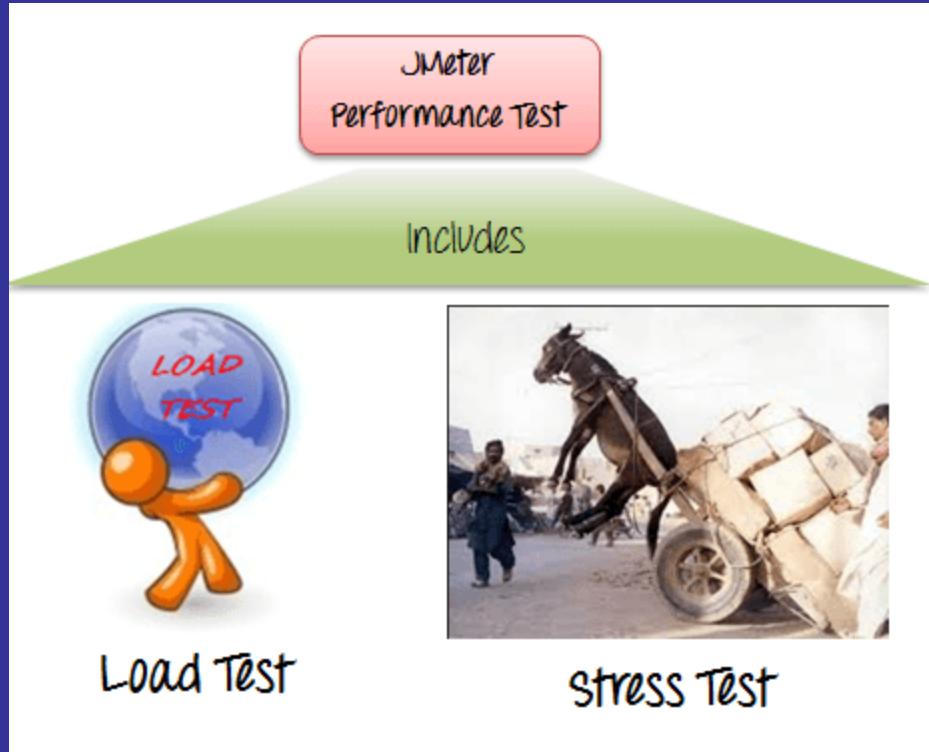
- Tests are designed to determine the performance of the actual system compared to the expected one
- Tests are designed to verify response time, execution time, throughput, resource utilization and traffic rate
- One needs to be clear about the specific data to be captured in order to evaluate performance metrics.
- For example, if the objective is to evaluate the response time, then one needs to capture
 - End-to-end response time (as seen by external user)
 - CPU time
 - Network connection time
 - Database access time
 - Network connection time
 - Waiting time

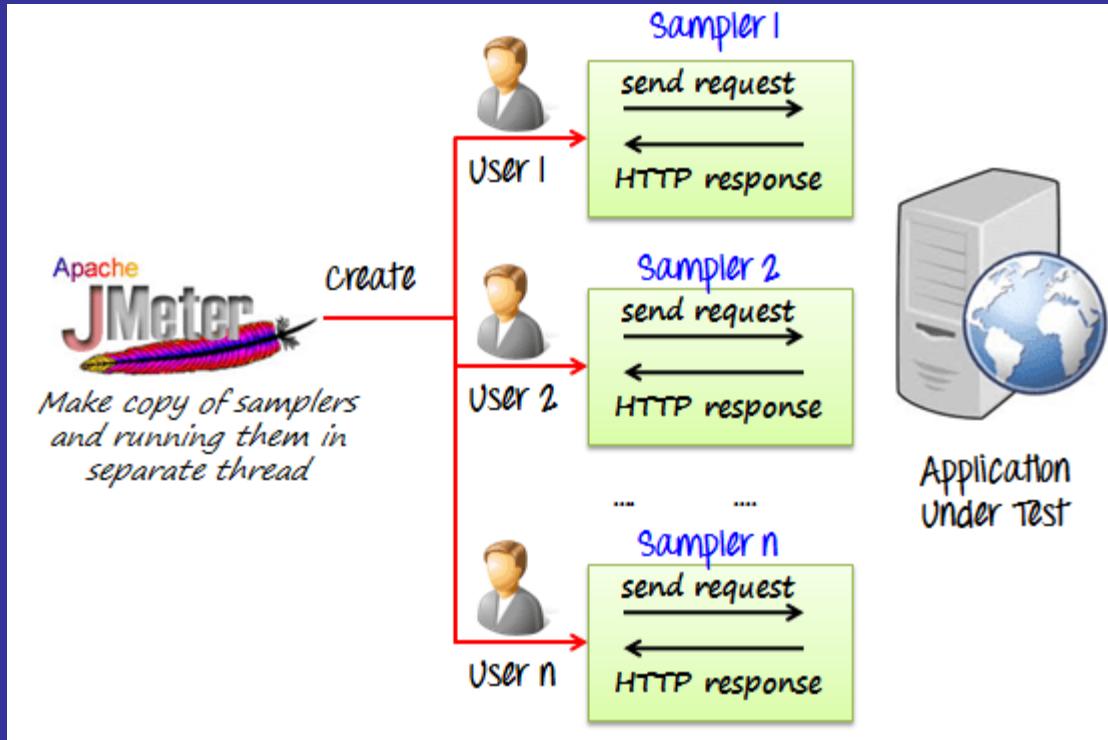
Stress Tests

- The goal of stress testing is to evaluate and determine the behavior of a software component while the offered load is in excess of its designed capacity
- The system is deliberately stressed by pushing it to and beyond its specified limits
- It ensures that the system can perform acceptably under worst-case conditions, under an expected peak load. If the limit is exceeded and the system does fail, then the recovery mechanism should be invoked
- Stress tests are targeted to bring out the problems associated with one or more of the following:
 - Memory leak: A failure in a program to release discarded memory
 - Buffer allocation: To control the allocation and freeing of buffers
 - Memory carving: A useful tool for analyzing physical and virtual memory dumps when the memory structures are unknown or have been overwritten.

Load and Stability Tests

- Tests are designed to ensure that the system remains stable for a long period of time under full load
- When a large number of users are introduced and applications that run for months without restarting, a number of problems are likely to occur:
 - the system slows down
 - the system encounters functionality problems
 - the system crashes altogether
- Load and stability testing typically involves exercising the system with virtual users and measuring the performance to verify whether the system can support the anticipated load
- This kind of testing help one to understand the ways the system will fare in real-life situations





Usability Testing

Testing to determine the extent to which the software product is understood, easy to learn, easy to operate and attractive to the users under specified conditions.

- Ease of Use
- Interface steps
- Response Time
- Help System
- Error Messages

Usability Testing

Graphical User Interface Tests

- Tests are designed to look-and-feel the interface to the users of an application system
- Tests are designed to verify different components such as icons, menu bars, dialog boxes, scroll bars, list boxes, and radio buttons
- The GUI can be utilized to test the functionality behind the interface, such as accurate response to database queries
- Tests the usefulness of the on-line help, error messages, tutorials, and user manuals
- The usability characteristics of the GUI is tested, which includes the following
 - ***Accessibility:*** Can users enter, navigate, and exit with relative ease?
 - ***Responsiveness:*** Can users do what they want and when they want in a way that is clear?
 - ***Efficiency:*** Can users do what they want to with minimum number of steps and time?
 - ***Comprehensibility:*** Do users understand the product structure with a minimum amount of effort?

Compatibility/Conversion/Configuration Testing

Compatibility Testing is a type of Software testing to check whether your software is capable of running on different hardware, operating systems, applications, network environments or Mobile devices.

- Operating systems: The specifications must state all the targeted end-user operating systems on which the system being developed will be run.
- Software/ Hardware: The product may need to operate with certain versions of web browsers, with hardware devices such as printers, or with other software, such as virus scanners or word processors.
- Conversion Testing: Compatibility may also extend to upgrades from previous versions of the software. Therefore, in this case, the system must be upgraded properly and all the data and information from the previous version should also be considered.
- Ranking of possible configurations(most to the least common, for the target system)
- Testers must identify appropriate test cases and data for compatibility testing.

Acceptance Testing

- Acceptance Testing is the formal testing conducted to determine whether a software system satisfies its acceptance criteria and to enable buyer to determine whether to accept the system or not.
 - Determine whether the software is fit for the user to use.
 - Making users confident about product
 - Determine whether a software system satisfies its acceptance criteria.
 - Enable the buyer to determine whether to accept the system.
-
- Alpha Testing Beta Testing

Acceptance Testing

Alpha Testing :

Alpha testing is a type of acceptance testing; performed to identify all possible issues/bugs before releasing the product to everyday users or public. The focus of this testing is to simulate real users by using blackbox and whitebox techniques. The aim is to carry out the tasks that a typical user might perform. Alpha testing is carried out in a lab environment and usually the testers are internal employees of the organization. To put it as simple as possible, this kind of testing is called alpha only because it is done early on, near the end of the development of the software, and before beta testing.

Acceptance Testing

Beta Testing:

Beta Testing of a product is performed by "real users" of the software application in a "real environment" and can be considered as a form of external user acceptance testing. Beta version of the software is released to a limited number of end-users of the product to obtain feedback on the product quality. Beta testing reduces product failure risks and provides increased quality of the product through customer validation. It is the final test before shipping a product to the customers. Direct feedback from customers is a major advantage of Beta Testing. This testing helps to tests the product in real time environment.

Acceptance Testing

Entry Criteria for Alpha testing:

- Software requirements document or Business requirements specification
- Test Cases for all the requirements
- Testing Team with good knowledge about the software application
- Test Lab environment setup
- QA Build ready for execution
- Test Management tool for uploading test cases and logging defects
- Traceability Matrix to ensure that each design requirement has atleast one test case that verifies it

Exit Criteria for Alpha testing

- All the test cases have been executed and passed.
- All severity issues need to be fixed and closed
- Delivery of Test summary report
- Make sure that no more additional features can be included
- Sign off on Alpha testing

Acceptance Testing

Entrance criteria for Beta Testing:

- Positive responses from alpha sites
- Sign off document on Alpha testing
- Beta version of the software should be ready
- Environment ready to release the software application to the public
- Beta sites are ready for installation

Exit Criteria for Beta Testing:

- Feedback report should be prepared from public(good feedback)
- Prepare Beta test summary report
- Notify bug fixing issues to developers

References:

1. Software Testing Principles and Practices, Naresh Chauhan, Second edition, Oxford Higher Education
2. <https://www.guru99.com/software-testing.html>

Module 2

Testing Techniques

Static Testing

Static Testing

Static testing is a complimentary technique to dynamic testing technique to acquire higher quality software. Static testing techniques do not execute the software.

Static testing can be applied for most of the verification activities.

The objectives of static testing can be summarized as follows:

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

Static Testing

- Static testing techniques do not demonstrate that the software is operational or one function of software is working;
- They check the software product at each SDLC stage for conformance with the required specifications or standards. Requirements, design specifications, test plans, source code, user's manuals, maintenance procedures are some of the items that can be statically tested.
- Static testing has proved to be a cost-effective technique of error detection.
- Another advantage in static testing is that a bug is found at its exact location whereas a bug found in dynamic testing provides no indication to the exact source code location.

Static Testing

Types of Static Testing

- **Software Inspections**
- **Walkthroughs**
- **Technical Reviews**

Inspections

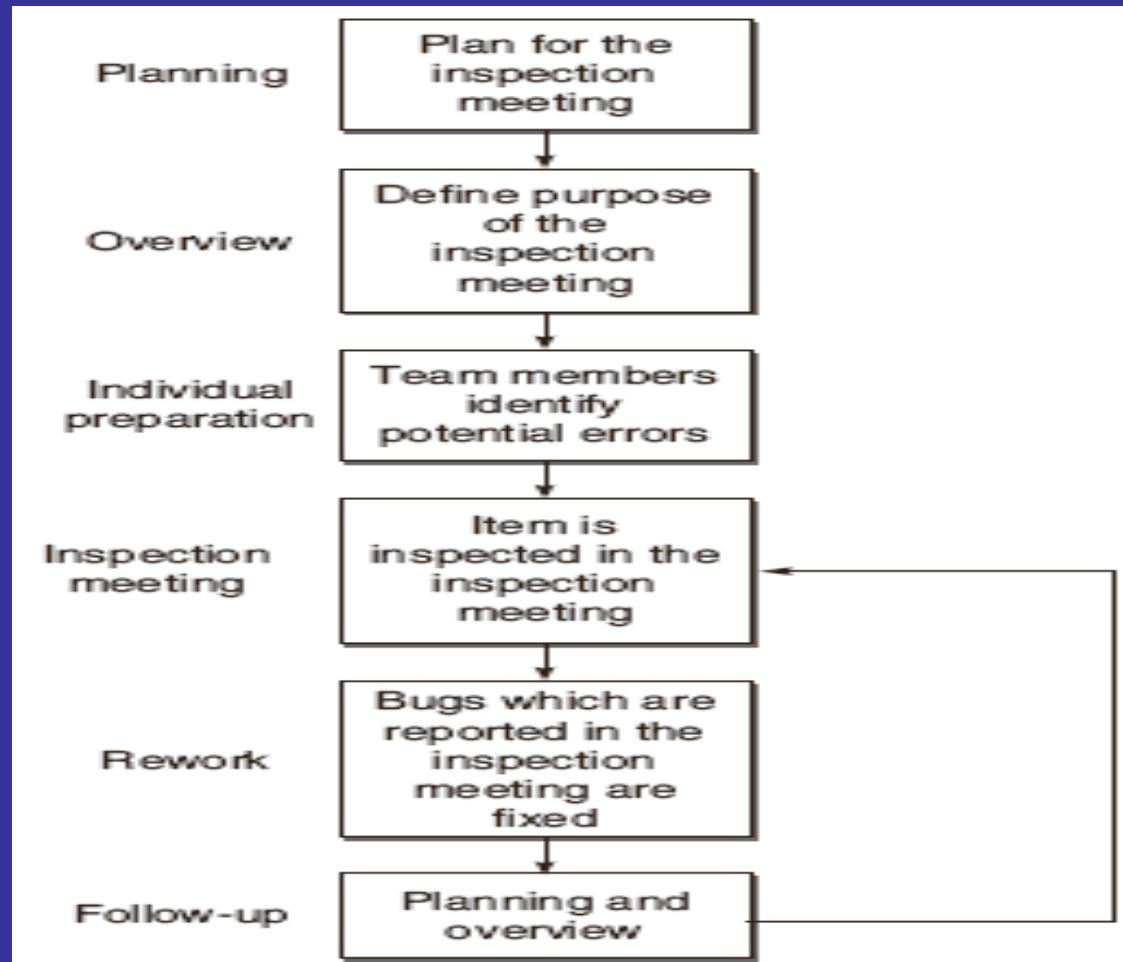
- Inspection process is an in-process manual examination of an item to detect bugs.
- Inspection process is carried out by a group of peers. The group of peers first inspects the product at individual level. After this, they discuss potential defects of the product observed in a formal meeting.
- It is a very formal process to verify a software product. The documents which can be inspected are SRS, SDD, code and test plan.

Inspections

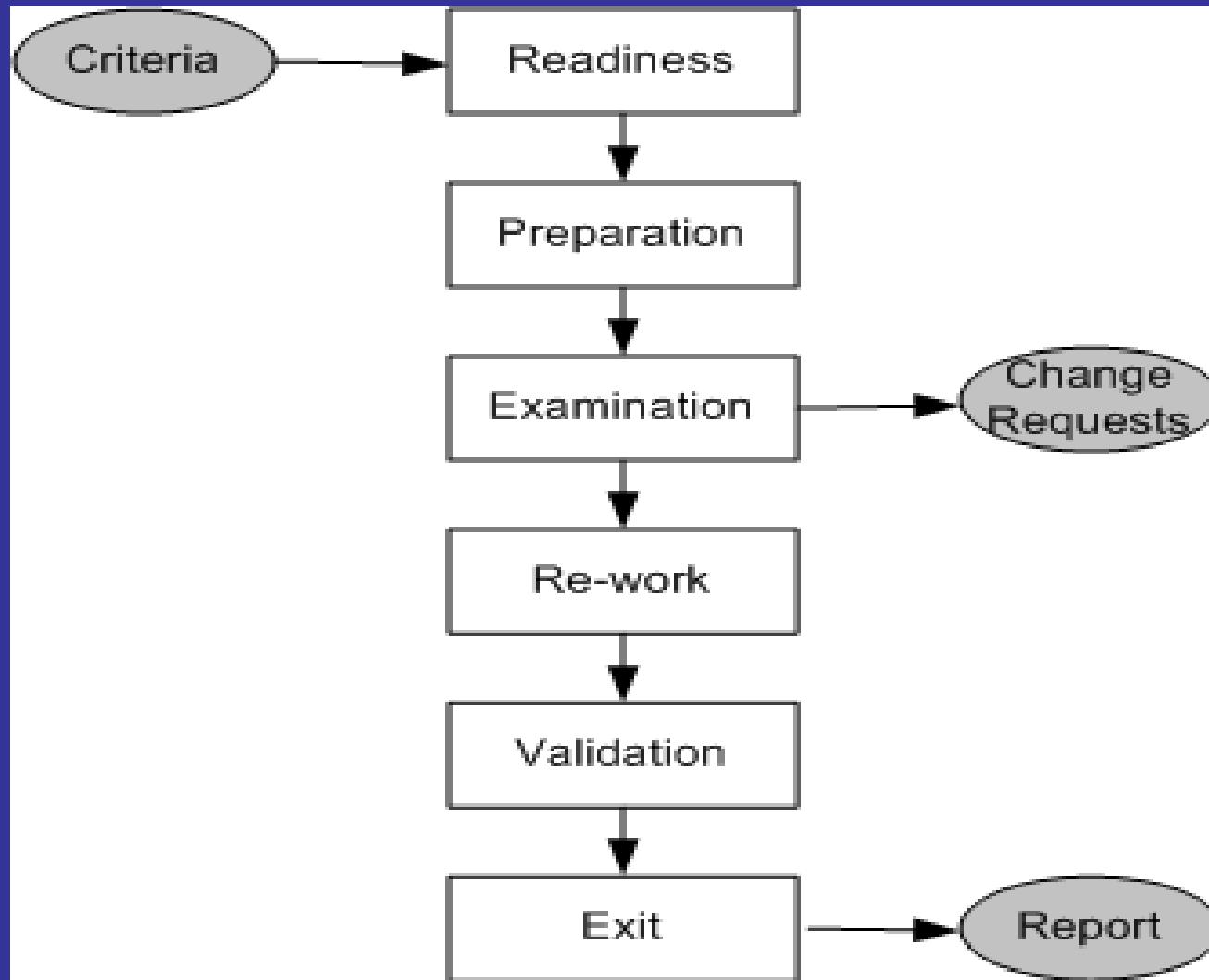
Inspection process involves the interaction of the following elements:

- Inspection steps
- Roles for participants
- Item being inspected

Inspection Process



Steps in the Inspection



Steps in the Inspection

1. Planning : During this phase, the following is executed:

- The product to be inspected is identified.
- A moderator is assigned.
- The objective of the inspection is stated. If the objective is defect detection, then the type of defect detection like design error, interface error, code error must be specified.

During planning, the 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.

Readiness Criteria

- Completeness ,Minimal functionality
- Readability, Complexity, Requirements and design documents

Steps in the Inspection

Inspection Team:

- Moderator
- Author
- Presenter
- Record keeper
- Reviewers
- Observer

2. Overview: In this stage, the inspection team is provided with the background information for inspection. The author presents the rationale for the product, its relationship to the rest of the products being developed, its function and intended use, and the approach used to develop it. This information is necessary for the inspection team to perform a successful inspection.

The opening meeting may also be called by the moderator. In this meeting, the objective of inspection is explained to the team members. The idea is that every member should be familiar with the overall purpose of the inspection.

Steps in the Inspection

3. Individual Preparation: After the overview, the reviewers individually prepare themselves for the inspection process by studying the documents provided to them in the overview session.

- List of questions
- Potential Change Request (CR)
- Suggested improvement opportunities

Completed preparation logs are submitted to the moderator prior to the inspection meeting.

Inspection Meeting/Examination:

- The author makes a presentation
- The presenter reads the code
- The record keeper documents the CR
- Moderator ensures the review is on track

Steps in the Inspection

At the end, the moderator concludes the meeting and produces a summary of the inspection meeting.

Change Request (CR) includes the following details:

- Give a brief description of the issue
- Assign a priority level (major or minor) to a **CR**
- Assign a person to follow it up
- Set a deadline for addressing a **CR**

Steps in the Inspection

4. Re-work: The summary list of the bugs that arise during the inspection meeting needs to be reworked by the author.

- Make the list of all the CRs
- Make a list of improvements
- Record the minutes meeting
- Author works on the CRs to fix the issue

5. Validation and Follow-up: It is the responsibility of the moderator to check that all the bugs found in the last meeting have been addressed and fixed.

- Moderator prepares a report and ascertains that all issues have been resolved. The document is then approved for release.
- If this is not the case, then the unresolved issues are mentioned in a report and another inspection meeting is called by the moderator.

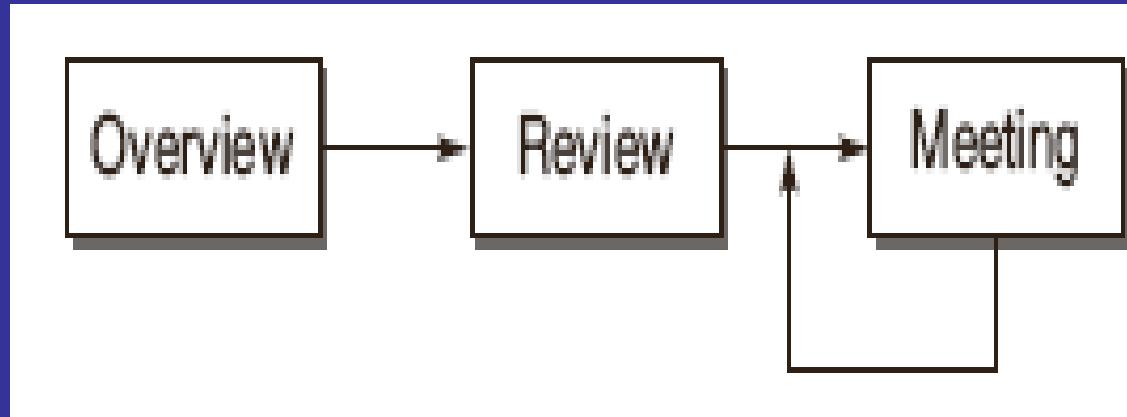
Benefits of Inspection Process

- **Bug Reduction**
- **Bug Prevention**
- **Productivity**
- **Real-time Feedback to Software Engineers**
- **Reduction in Development Resource**
- **Quality Improvement**
- **Project Management**
- **Checking Coupling and Cohesion**
- **Learning through Inspection**
- **Process Improvement**

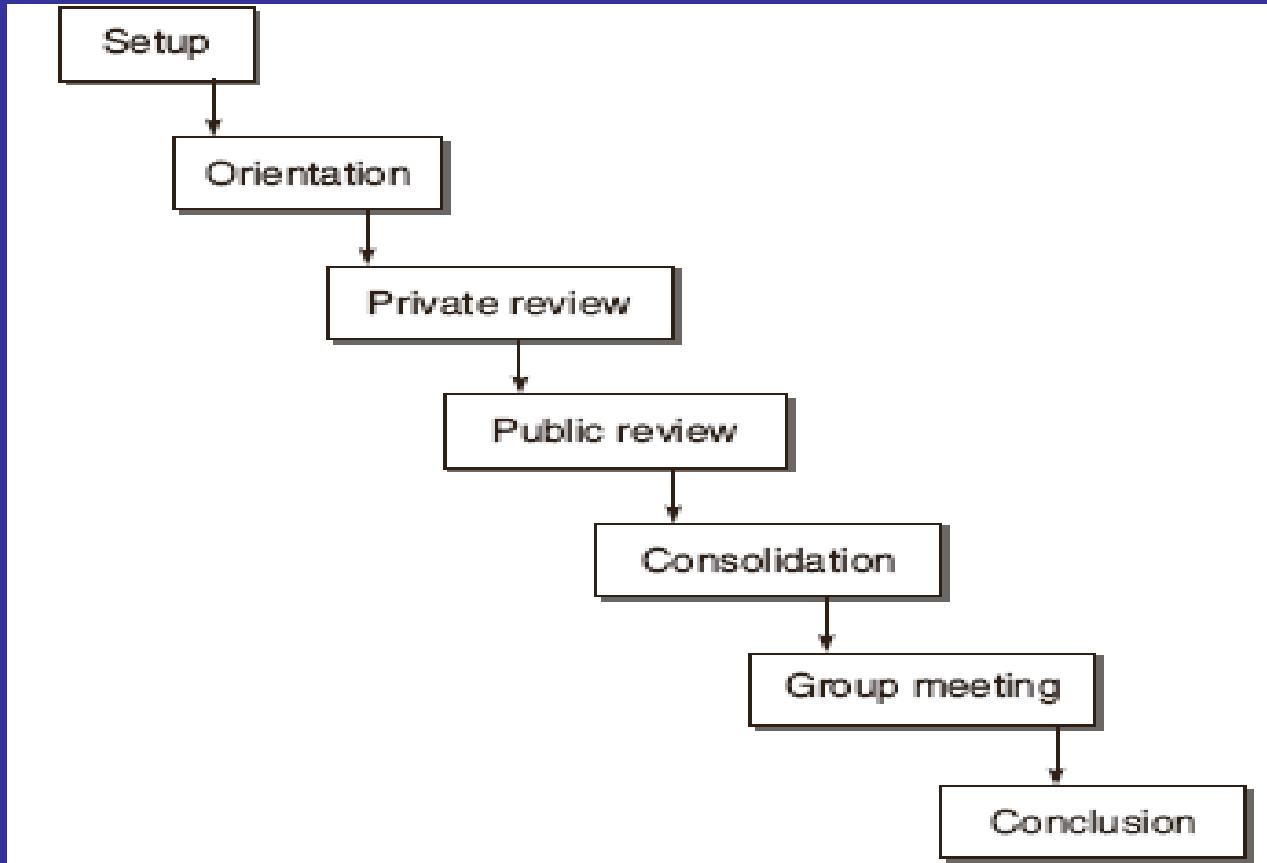
Variants of Inspection process

Active Design Reviews (ADRs)	Several reviews are conducted targeting a particular type of bugs and conducted by the reviewers who are experts in that area.
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.
Gilb Inspection	Defect detection is carried out by individual inspector at his level rather than in a group.
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 are analysed and combined into a single list.
N-Fold inspections	Inspection process's effectiveness can be increased by replicating it by having multiple inspection teams.
Phased Inspection	Phased inspections are designed to verify the product in a particular domain by experts in that domain only.
Structured Walkthrough	Described by Yourdon. Less formal and rigorous than formal inspections. Roles are coordinator, scribe, presenter, reviewers, maintenance oracle, standards bearer, user representative. Process steps are Organization, Preparation, Walkthrough, and Rework. Lacks data collection requirements of formal inspections.

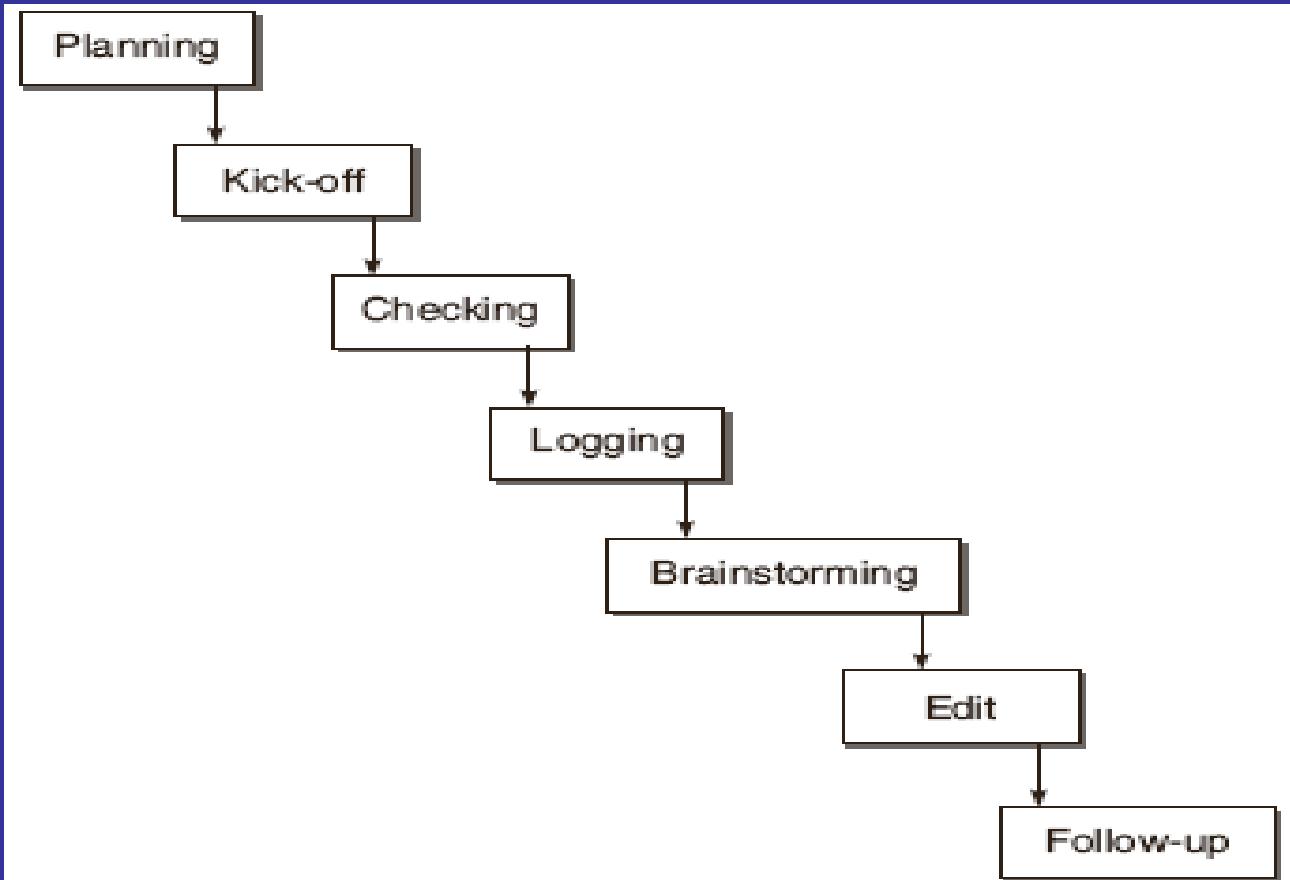
Active Design Reviews



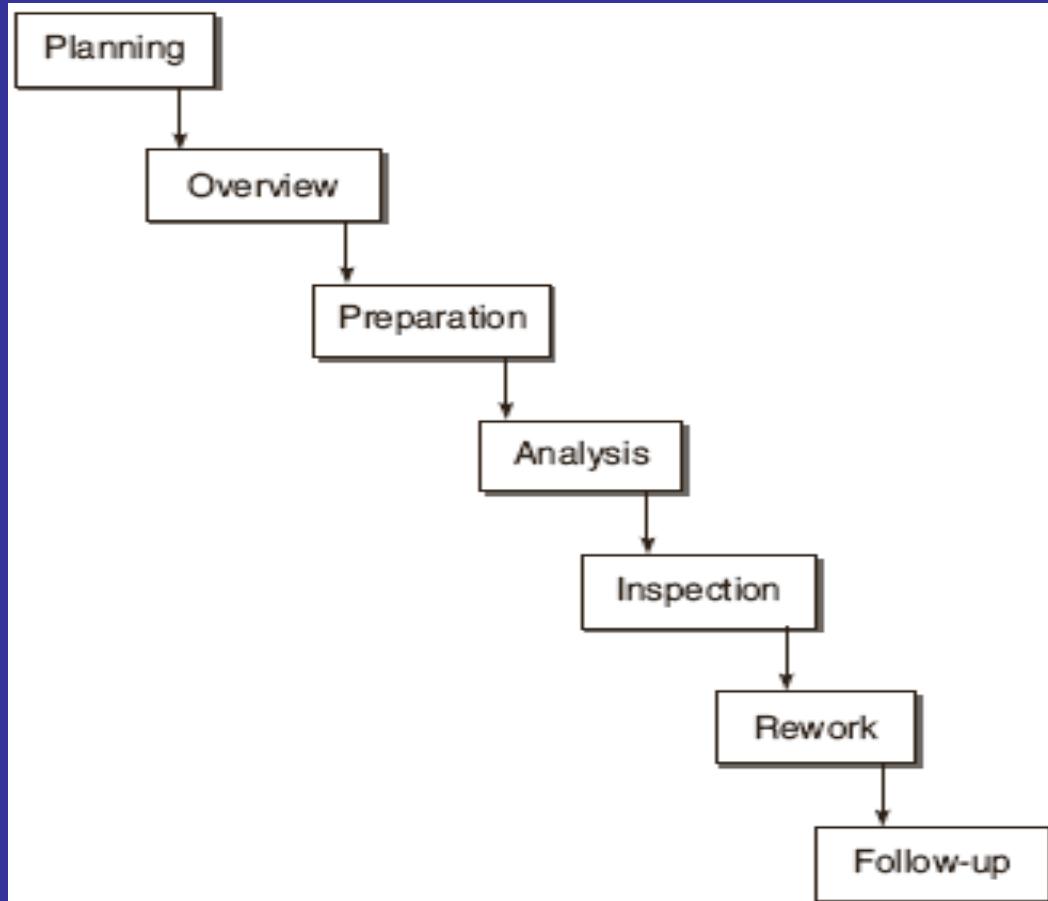
Formal Technical Asynchronous review method (FTArm)



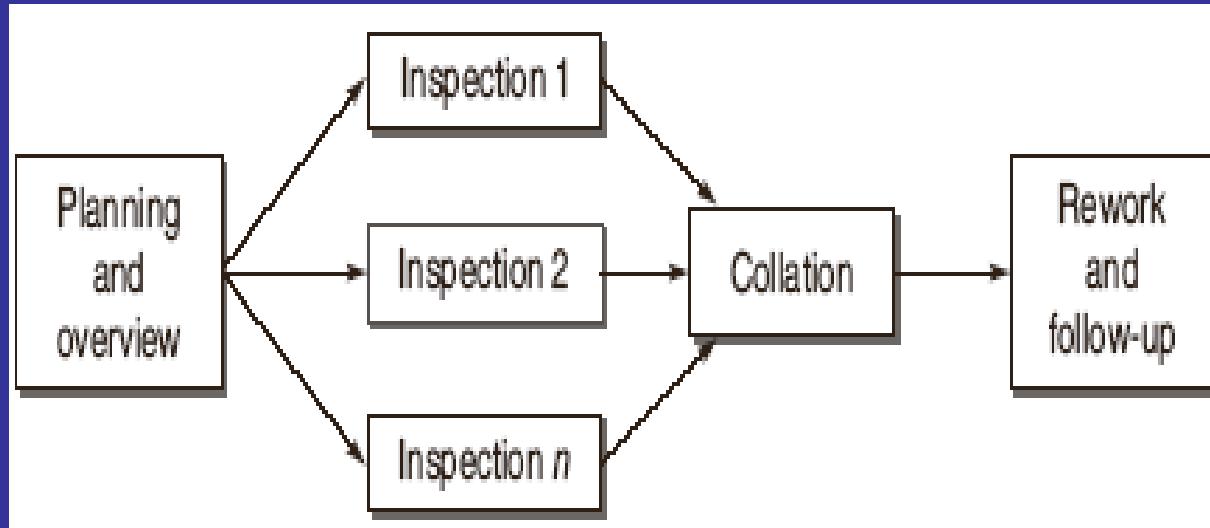
Gilb Inspection



Humphrey's Inspection Process



N-Fold Inspection



Checklist

Structure:

- Does the code completely and correctly implement the design?
- Does the code conform to any applicable coding standards?
- Is the code well-structured, consistent in style, and consistently formatted?
- Are there any uncalled or unneeded procedures or any unreachable code?
- Are there any leftover stubs or test routines in the code?
- Can any code be replaced by calls to external reusable components or library functions?
- Are there any blocks of repeated code that could be condensed into a single procedure?
- Is storage use efficient?
- Are any modules excessively complex and should be restructured or split into multiple routines?

Checklist

Arithmetic Operations:

- ❑ Does the code avoid comparing floating-point numbers for equality?
- ❑ Does the code systematically prevent rounding errors?
- ❑ Are divisors tested for zero or noise?

Checklist

Loops and Branches:

- Are all loops, branches, and logic constructs complete, correct, and properly nested?
- Are all cases covered in an IF- -ELSEIF or CASE block, including ELSE or DEFAULT clauses?
- Does every case statement have a default?
- Are loop termination conditions obvious and always achievable?
- Are indexes or subscripts properly initialized, just prior to the loop?
- Does the code in the loop avoid manipulating the index variable or using it upon exit from the loop?

Checklist

Documentation:

- Is the code clearly and adequately documented with an easy-to-maintain commenting style?
- Are all comments consistent with the code?

Variables:

- Are all variables properly defined with meaningful, consistent, and clear names?
- Do all assigned variables have proper type consistency or casting?
- Are there any redundant or unused variables?

Checklist

Input / Output errors:

- If the file or peripheral is not ready, is that error condition handled?
- Does the software handle the situation of the external device being disconnected?
- Have all error messages been checked for correctness, appropriateness, grammar, and spelling?
- Are all exceptions handled by some part of the code?

Scenario based Reading

Perspective based Reading

- software item should be inspected from the perspective of different stakeholders. Inspectors of an inspection team have to check software quality as well as the software quality factors of a software artifact from different perspectives.

Usage based Reading

- This method given is applied in design inspections. Design documentation is inspected based on use cases, which are documented in requirements specification.

Abstraction driven Reading

- This method is designed for code inspections. In this method, an inspector reads a sequence of statements in the code and abstracts the functions these statements compute.

Scenario based Reading

Task driven Reading

- This method is also for code inspections . In this method, the inspector has to create a data dictionary, a complete description of the logic and a cross-reference between the code and the specifications.

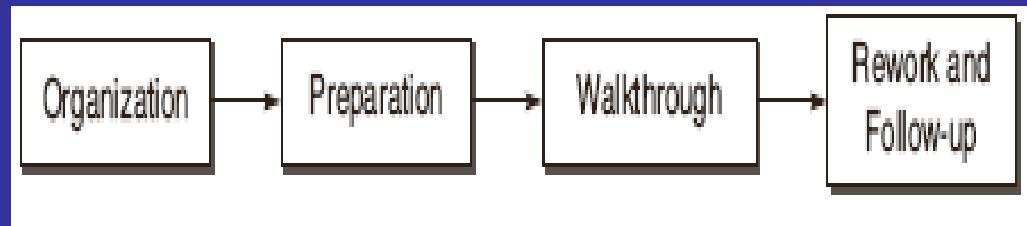
Function-point based Scenarios

- This is based on scenarios for defect detection in requirements documents [103]. The scenarios, designed around function-points are known as the Function Point Scenarios. A Function Point Scenario consists of questions and directs the focus of an inspector to a specific function-point item within the inspected requirements document.

Structured Walkthroughs

It is a less formal and less rigorous technique as compared to inspection.

- Author presents their developed artefact to an audience of peers.
- Peers question and comment on the artefact to identify as many defects as possible.
- It involves no prior preparation by the audience. Usually involves minimal documentation of either the process or any arising issues. Defect tracking in walkthroughs is inconsistent.
- A walk through is an evaluation process which is an informal meeting, which does not require preparation.
- The product is described by the author and queries for the comments of participants.
- The results are the information to the participants about the product instead of correcting it.



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 review is similar to an inspection or walkthrough, except that the review team also includes management. Therefore, it is considered a higher-level technique than inspection or walkthrough.

A technical review team is generally comprised of management-level representatives of the User 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. He should also prepare a set of indicators to measure the following points:

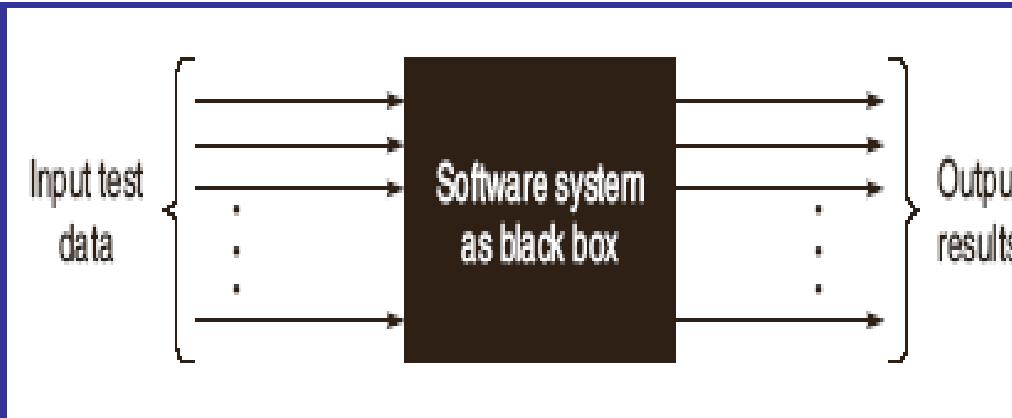
- Appropriateness of the problem definition and requirements
- Adequacy of all underlying assumptions
- Adherence to standards, Consistency , Completeness, Documentation

Dynamic Testing:

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.
- It is obvious that in black-box technique, test cases are designed based on functional specifications. Input test data is given to the system, which is a black box to the tester, and results are checked against expected outputs after executing the software,

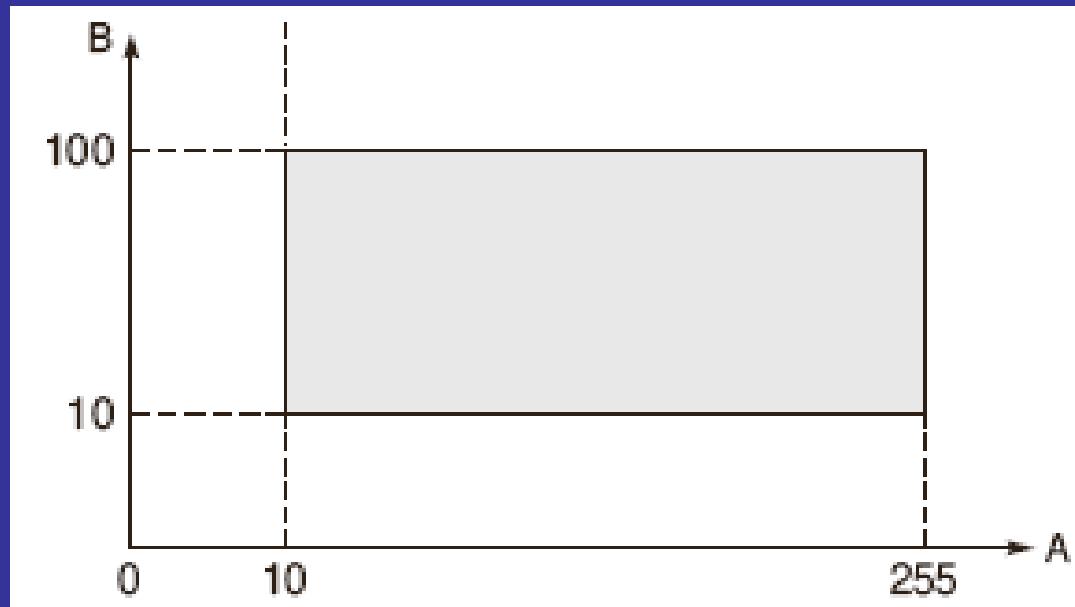


Black Box Testing

- To test the modules independently.
- To test the functional validity of the software
- Interface errors are detected.
- To test the system behavior and check its performance.
- To test the maximum load or stress on the system.
- Customer accepts the system within defined acceptable limits.

Boundary Value Analysis (BVA)

'Boundary value analysis' testing technique is used to identify errors at boundaries rather than finding those exist in centre of input domain.



Boundary Value Analysis (BVA)

- The BVA technique is an extension and refinement of the equivalence class partitioning technique
- In the BVA technique, the boundary conditions for each of the equivalence class are analyzed in order generate test cases

Guidelines for Boundary Value Analysis

- The equivalence class specifies a range
 - If an equivalence class specifies a range of values, then construct test cases by considering the boundary points of the range and points just beyond the boundaries of the range
- The equivalence class specifies a number of values
 - If an equivalence class specifies a number of values, then construct test cases for the minimum and the maximum value of the number
 - In addition, select a value smaller than the minimum and a value larger than the maximum value.
- The equivalence class specifies an ordered set
 - If the equivalence class specifies an ordered set, such as a linear list, table, or a sequential file, then focus attention on the first and last elements of the set.

BVA- “Single fault "assumption theory

“Single fault” assumption in reliability theory: failures are only rarely the result of the simultaneous occurrence of two (or more) faults.

The function f that computes the number of test cases for a given number of variables n can be shown as:

$$f = 4n + 1$$

As there are four extreme values this accounts for the $4n$. The addition of the constant one constitutes for the instance where all variables assume their nominal value.

BVA- “Single fault "assumption theory

The basic form of implementation is to maintain all but one of the variables at their nominal (normal or average) values and allowing the remaining variable to take on its extreme values. The values used to test the extremities are:

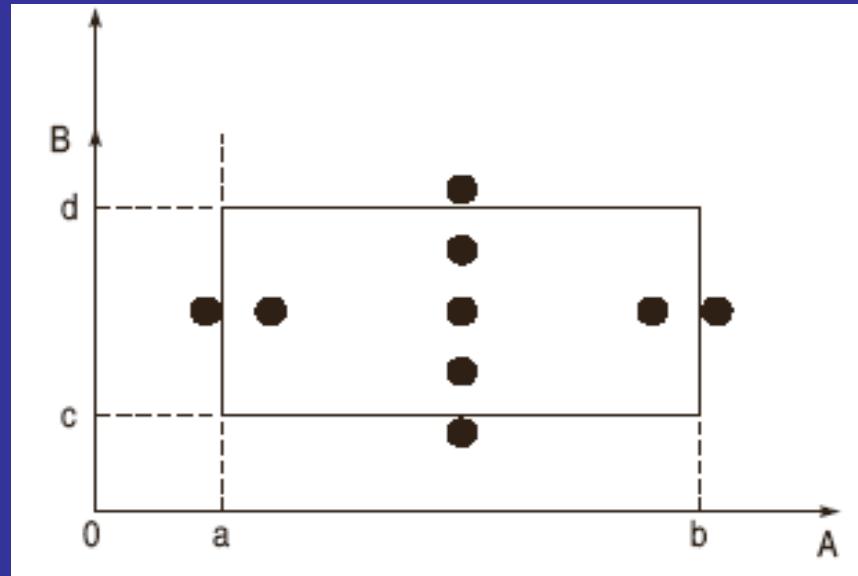
- Min ----- Minimal
- Min+ ----- Just above Minimal
- Nom ----- Average
- Max- ----- Just below Maximum
- Max ----- Maximum

Boundary Value Checking

- Test cases are designed by holding one variable at its extreme value and other variables at their nominal values in the input domain. The variable at its extreme value can be selected at:
 - Minimum value (Min)
 - Value just above the minimum value (Min+)
 - Maximum value (Max)
 - Value just below the maximum value (Max-)

Boundary Value Checking

- Anom, Bmin
- Anom, Bmin+
- Anom, Bmax
- Anom, Bmax-
- Amin, Bnom
- Amin+, Bnom
- Amax, Bnom
- Amax-, Bnom
- Anom, Bnom
- 4n+1 test cases can be designed with boundary value checking method.



Robustness Testing Method

A value just greater than the Maximum value (Max+)

A value just less than Minimum value (Min-)

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

- Amax+, Bnom
- Amin-, Bnom
- Anom, Bmax+
- Anom, Bmin-
-

It can be generalized that for n input variables in a module, 6n+1 test cases are designed with Robustness testing.

Worst Case Testing Method

- When more than one variable are in extreme values, i.e. when more than one variable are on the boundary. It is called Worst case testing method.
- It can be generalized that for n input variables in a module, 5^n test cases are designed with worst case testing.

10. A_{\min}, B_{\min}	11. $A_{\min+}, B_{\min}$
12. $A_{\min}, B_{\min+}$	13. $A_{\min+}, B_{\min+}$
14. A_{\max}, B_{\min}	15. $A_{\max-}, B_{\min}$
16. $A_{\max}, B_{\min+}$	17. $A_{\max-}, B_{\min+}$
18. A_{\min}, B_{\max}	19. $A_{\min+}, B_{\max}$
20. $A_{\min}, B_{\max-}$	21. $A_{\min+}, B_{\max-}$
22. A_{\max}, B_{\max}	23. $A_{\max-}, B_{\max}$
24. $A_{\max}, B_{\max-}$	25. $A_{\max-}, B_{\max-}$

Example

- A program reads an integer number within the range [1,100] and determines whether the number is a prime number or not. Design all test cases for this program using BVC, Robust testing and worst-case testing methods.
- 1) Test cases using BVC

Test Case ID	Integer Variable	Expected Output
1	1	Not a prime number
2	2	Prime number
3	100	Not a prime number
4	99	Not a prime number
5	53	Prime number

Min value = 1
Min ⁺ value = 2
Max value = 100
Max ⁻ value = 99
Nominal value = 50–55

Example

- **Test Cases Using Robust Testing**

Test Case ID	Integer Variable	Expected Output	
1	0	Invalid input	Min value = 1
2	1	Not a prime number	Min ⁻ value = 0
3	2	Prime number	Min ⁺ value = 2
4	100	Not a prime number	Max value = 100
5	99	Not a prime number	Max ⁻ value = 99
6	101	Invalid input	Max ⁺ value = 101
7	53	Prime number	Nominal value = 50–55

BVA- The triangle problem

The triangle problem accepts three integers (a, b and c) as its input, each of which are taken to be sides of a triangle . The values of these inputs are used to determine the type of the triangle (Equilateral, Isosceles, Scalene or not a triangle).

For the inputs to be declared as being a triangle they must satisfy the six conditions:

C1. $1 \leq a \leq 200$.

C2. $1 \leq b \leq 200$.

C3. $1 \leq c \leq 200$.

C4. $a < b + c$.

C5. $b < a + c$.

C6. $c < a + b$.

Otherwise this is declared not to be a triangle.

The type of the triangle, provided the conditions are met, is determined as follows:

1. If all three sides are equal, the output is Equilateral.
2. If exactly one pair of sides is equal, the output is Isosceles.
3. If no pair of sides is equal, the output is Scalene.

Test Cases for the Triangle Problem

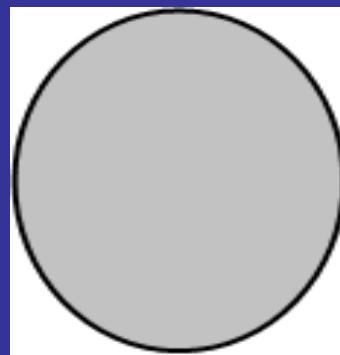
min = 1
min+ = 2
nom = 100
max- = 199
max = 200

Boundary Value Analysis Test Cases

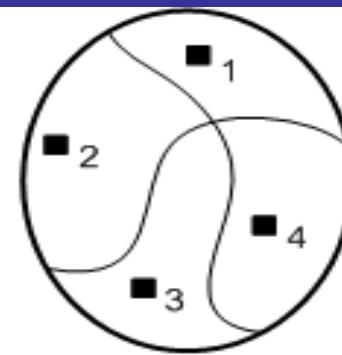
Case	a	b	c	Expected Output
1	100	100	1	Isosceles
2	100	100	2	Isosceles
3	100	100	100	Equilateral
4	100	100	199	Isosceles
5	100	100	200	Not a Triangle
6	100	1	100	Isosceles
7	100	2	100	Isosceles
8	100	199	100	Isosceles
9	100	200	100	Not a Triangle
10	1	100	100	Isosceles
11	2	100	100	Isosceles
12	199	100	100	Isosceles
13	200	100	100	Not a Triangle

Equivalence Class Testing

- An input domain may be too large for all its elements to be used as test input
- The input domain is partitioned into a finite number of subdomains
- Each subdomain is known as an equivalence class, and it serves as a source of at least one test input
- A valid input to a system is an element of the input domain that is expected to return a non error value
- An invalid input is an input that is expected to return an error value.



(a) Input Domain



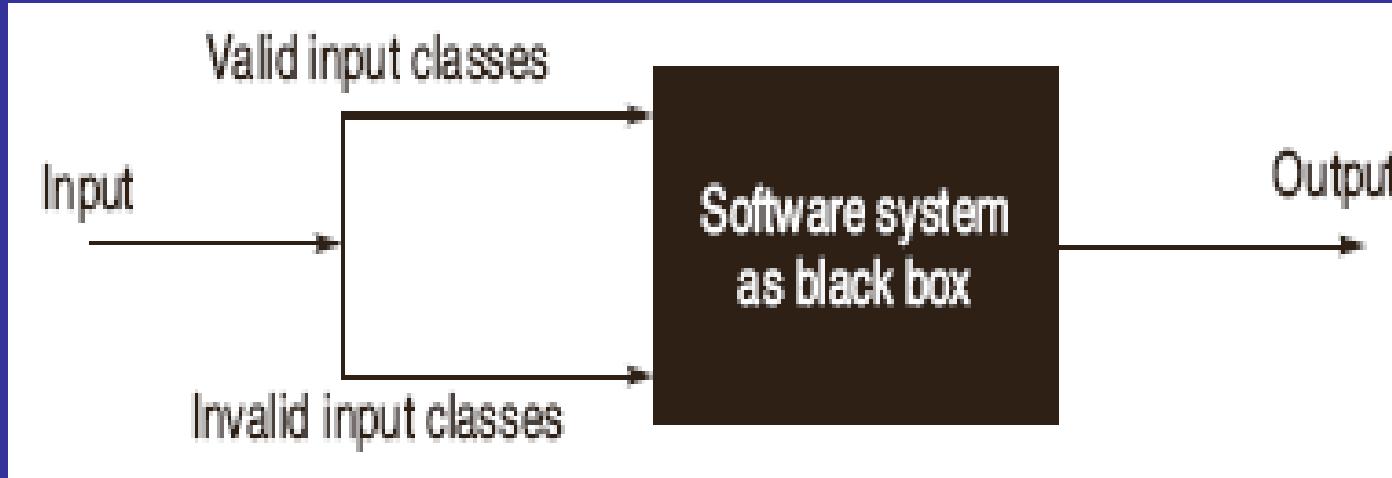
(b) Input Domain Partitioned into Four Sub-domains

Figure (a) Too many test input; (b) One input is selected from each of the subdomain

Reference: Software Testing Principles and Practices, Naresh Chauhan, Oxford University

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.



Guidelines for Equivalence Class Partitioning

- An input condition specifies a range $[a, b]$
 - one equivalence class for $a < X < b$, and
 - two other classes for $X < a$ and $X > b$ to test the system with invalid inputs
- An input condition specifies a set of values
 - one equivalence class for each element of the set $\{M_1\}, \{M_2\}, \dots, \{M_N\}$, and
 - one equivalence class for elements outside the set $\{M_1, M_2, \dots, M_N\}$
- Input condition specifies for each individual value
 - If the system handles each valid input differently then create one equivalence class for each valid input
- An input condition specifies the number of valid values (Say N)
 - Create one equivalence class for the correct number of inputs
 - two equivalence classes for invalid inputs – one for zero values and one for more than N values
- An input condition specifies a “must be” value
 - Create one equivalence class for a “must be” value, and
 - one equivalence class for something that is not a “must be” value

Identification of Test Cases

Test cases for each equivalence class can be identified by:

- Assign a unique number to each equivalence class
- For each equivalence class with valid input that has not been covered by test cases yet, write a new test case covering as many uncovered equivalence classes as possible
- For each equivalence class with invalid input that has not been covered by test cases, write a new test case that covers one and only one of the uncovered equivalence classes

Example

- A program reads three numbers A, B and C with range [1,50] and prints largest number. Design all test cases for this program using equivalence class testing technique.

I1 = { $\langle A, B, C \rangle : 1 \leq A \leq 50$ }

I2 = { $\langle A, B, C \rangle : 1 \leq B \leq 50$ }

I3 = { $\langle A, B, C \rangle : 1 \leq C \leq 50$ }

I4 = { $\langle A, B, C \rangle : A < 1$ }

I5 = { $\langle A, B, C \rangle : A > 50$ }

I6 = { $\langle A, B, C \rangle : B < 1$ }

I7 = { $\langle A, B, C \rangle : B > 50$ }

I8 = { $\langle A, B, C \rangle : C < 1$ }

I9 = { $\langle A, B, C \rangle : C > 50$ }

Test case ID	A	B	C	Expected result	Classes covered by the test case
1	13	25	36	C is greatest	I ₁ , I ₂ , I ₃
2	0	13	45	Invalid input	I ₄
3	51	34	17	Invalid input	I ₅
4	29	0	18	Invalid input	I ₆
5	36	53	32	Invalid input	I ₇
6	27	42	0	Invalid input	I ₈
7	33	21	51	Invalid input	I ₉

Example

- $I_1 = \{<A,B,C> : A > B, A > C\}$
- $I_2 = \{<A,B,C> : B > A, B > C\}$
- $I_3 = \{<A,B,C> : C > A, C > B\}$
- $I_4 = \{<A,B,C> : A = B, A \neq C\}$
- $I_5 = \{<A,B,C> : B = C, A \neq B\}$
- $I_6 = \{<A,B,C> : A = C, C \neq B\}$
- $I_7 = \{<A,B,C> : A = B = C\}$

Test case ID	A	B	C	Expected Result	Classes Covered by the test case
1	25	13	13	A is greatest	I_1, I_5
2	25	40	25	B is greatest	I_2, I_6
3	24	24	37	C is greatest	I_3, I_4
4	25	25	25	All three are equal	I_7

Advantages of Equivalence Class Partitioning

- A small number of test cases are needed to adequately cover a large input domain
- One gets a better idea about the input domain being covered with the selected test cases
- The probability of uncovering defects with the selected test cases based on equivalence class partitioning is higher than that with a randomly chosen test suite of the same size
- The equivalence class partitioning approach is not restricted to input conditions alone – the technique may also be used for output domains

Decision Table Based Testing

A major limitation of the EC-based testing is that it only considers each input separately. The technique does not consider combining conditions.

Different combinations of equivalent classes can be tried by using a new technique based on the decision table to handle multiple inputs.

Formation of Decision Table

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

- Condition Stub
- Action Stub
- Condition Entry
- Action Entry

Formation of Decision Table

- It comprises a set of conditions (or, causes) and a set of effects (or, results) arranged in the form of a column on the left of the table
- In the second column, next to each condition, we have its possible values: Yes (Y), No (N), and Don't Care (Immaterial) state.
- To the right of the “Values” column, we have a set of rules. For each combination of the three conditions {C1,C2,C3}, there exists a rule from the set {R1,R2,...}
- Each rule comprises a Yes (Y), No (N), or Don't Care (“-”) response, and contains an associated list of effects(actions) {E1,E2,E3}
- For each relevant effect, an effect sequence number specifies the order in which the effect should be carried out, if the associated set of conditions are satisfied
- Each rule of a decision table represents a test case

Test case design using decision table

The steps for developing test cases using decision table technique:

- **Step 1:** The test designer needs to identify the conditions and the actions/effects for each specification unit.
 - A condition is a distinct input condition or an equivalence class of input conditions
 - An action/effect is an output condition. Determine the logical relationship between the conditions and the effects
- **Step 2:** List all the conditions and actions in the form of a decision table. Write down the values the condition can take.
- **Step 3:** Fill the columns with all possible combinations – each column corresponds to one combination of values.
- Step 4: Define rules by indicating what action occurs for a set of conditions.

Test case design using decision table

- Interpret condition stubs as the inputs for the test case.
- Interpret action stubs as the expected output for the test case.
- Rule, which is the combination of input conditions becomes the test case itself.
- The columns in the decision table are transformed into test cases.
- If there are K rules over n binary conditions, there are at least K test cases and at the most 2^n test cases.

Decision Table Based Testing

Example

- A program calculates the total salary of an employee with the conditions that if the working hours are less than or equal to 48, then give normal salary. The hours over 48 on normal working days are calculated at the rate of 1.25 of the salary. However, on holidays or Sundays, the hours are calculated at the rate of 2.00 times of the salary. Design the test cases using decision table testing.

Decision Table Based Testing

The decision table for the program is shown below:

ENTRY		Rule 1	Rule 2	Rule3
Condition Stub	C1: Working hours > 48	I	F	T
	C2: Holidays or Sundays	T	F	F
Action Stub	A1: Normal salary		X	
	A2: 1.25 of salary			X
	A3: 2.00 of salary	X		

The test cases derived from the decision table are given below:

Test Case ID	Working Hour	Day	Expected Result
1	48	Monday	Normal Salary
2	50	Tuesday	1.25 of salary
3	52	Sunday	2.00 of salary

Dynamic Testing: White Box Testing Techniques

White-box testing (also known as clear box testing, glass box testing, transparent box testing, and structural testing) is a method of testing software that tests internal structures or workings of an application. White-box testing can be applied at the unit, integration and system levels of the software testing process.

- White box testing needs the full understanding of the logic/structure of the program.
- Test case designing using white box testing techniques
 - Control Flow testing method
 - Basis Path testing method
 - Loop testing
 - Data Flow testing method
 - Mutation testing method
- Control flow refers to flow of control from one instruction to another
- Data flow refers to propagation of values from one variable or constant to another variable

Basis Path Testing

Basis path testing is the technique of selecting the paths that provide a basis set of execution paths through the program.

- Path Testing is based on control structure of the program for which flow graph is prepared.
- requires complete knowledge of the program's structure.
- closer to the developer and used by him to test his module.
- The effectiveness of path testing is reduced with the increase in size of software under test.
- Choose enough paths in a program such that maximum logic coverage is achieved.

Control Flow Graph

The control flow graph is a 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 . Based on the concepts of directed graph, following notations are used for a flow graph:

Node: It represents one or more procedural statements. The nodes are denoted by a circle. These are numbered or labelled.

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.

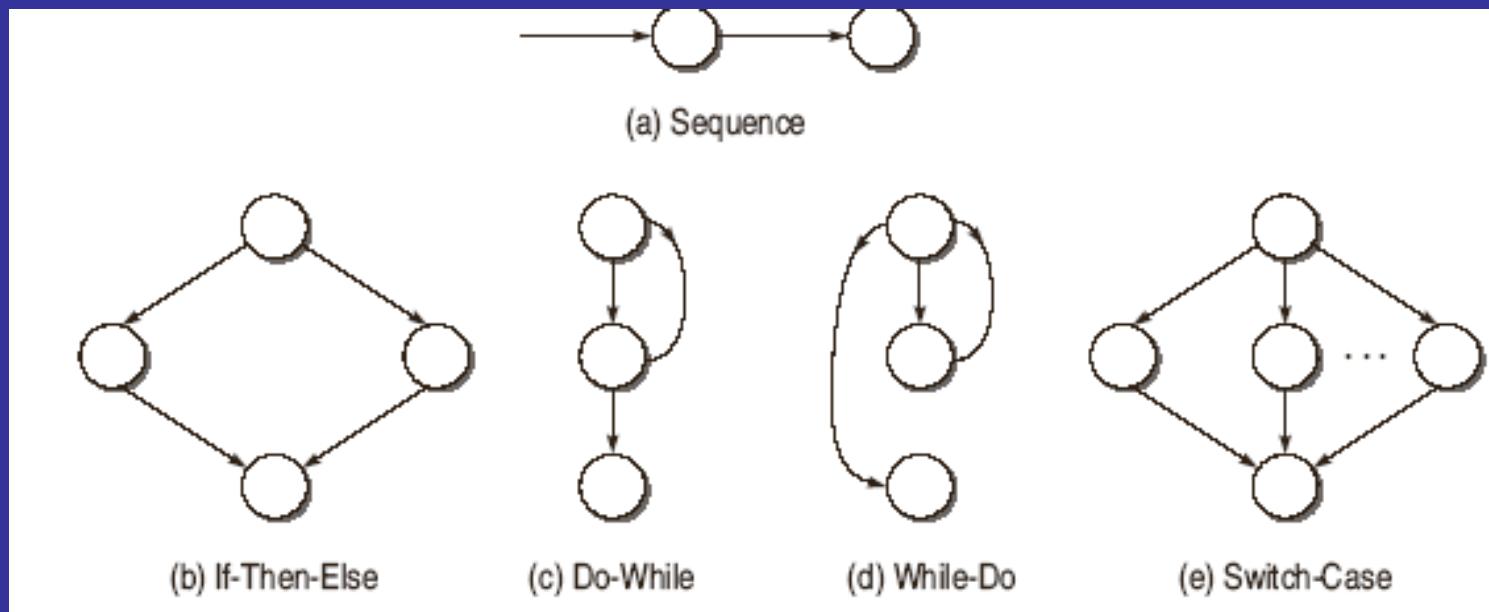
Decision node: A node with more than one arrow leaving it is called a decision node.

Junction node: A node with more than one arrow entering it is called a junction.

Regions: Areas bounded by edges and nodes are called regions. When counting the regions, the area outside the graph is also considered a region.

Control Flow Graph

Flow Graph Notations for Different Programming Constructs



Path Testing Terminology

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.

Segment: Paths consist of segments. The smallest segment is a link, that is, a single process that lies between two nodes (e.g., junction-process-junction, junction process-decision, decision-process-junction, decision-process-decision).

Path Segment: A path segment is a succession of consecutive links that belongs to some path.

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. An alternative way to measure the length of a path is by the number of nodes traversed.

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.

Path Testing Terminology

Cyclomatic complexity is a software metric used to indicate the complexity of a program. It is a quantitative measure of the number of linearly independent paths through a program's source code. It was developed by Thomas J. McCabe, Sr. in 1976.

- The testing strategy, called basis path testing by McCabe who first proposed it, is to test each linearly independent path through the program; in this case, the number of test cases will equal the cyclomatic complexity of the program.
- **Cyclomatic Complexity (logical complexity of program)**

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. $V(G) = \text{number of regions in the graph.}$

Path Testing Terminology

- **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

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

Guidelines for Basis Path Testing

We can use the cyclomatic complexity number in basis path testing. Cyclomatic number, which defines the number of independent paths, can be utilized as an upper bound for the number of tests that must be conducted to ensure that all the statements have been executed at least once. Thus, independent paths are prepared according to the upper limit of the cyclomatic number. The set of independent paths becomes the basis set for the flow graph of the program. Then test cases can be designed according to this basis set.

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

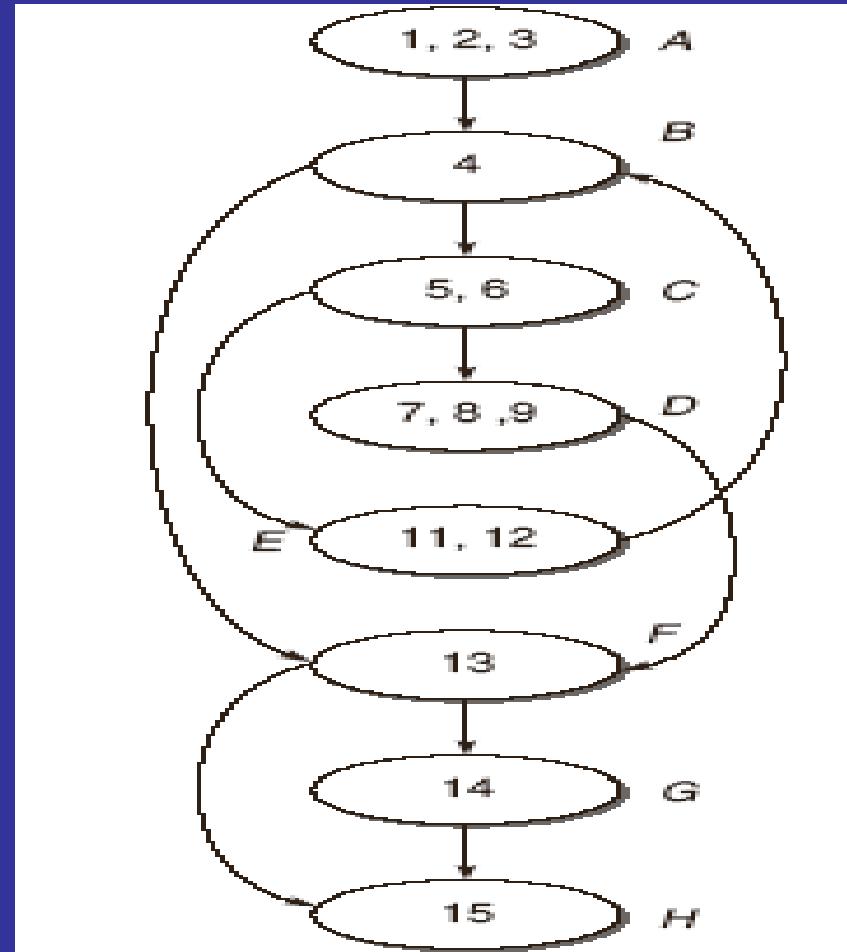
- Draw the flow graph using the code provided for which we have to write test cases.
- Determine the cyclomatic complexity of the flow graph.
- Cyclomatic complexity provides the number of independent paths.
- Determine a basis set of independent paths through the program control structure.
- The basis 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.

Example: Consider the following program segment:

```
main()
{
    int number, index;
1.   printf("Enter a number");
2.   scanf("%d, &number);
3.   index = 2;
4.   while(index <= number - 1)
5.   {
6.       if (number % index == 0)
7.       {
8.           printf("Not a prime number");
9.           break;
10.      }
11.      index++;
12.  }
13.  if(index == number)
14.      printf("Prime number");
15. } //end main
```

1. Draw the DD graph for the program.
2. Calculate the cyclomatic complexity of the program using all the methods.
3. List all independent paths.
4. Design test cases from independent paths.

Example



Example

Cyclomatic Complexity

$$\begin{aligned}V(G) &= e - n + 2 * P \\&= 10 - 8 + 2 \\&= 4\end{aligned}$$

$$\begin{aligned}V(G) &= \text{Number of predicate nodes} + 1 \\&= 3 \text{ (Nodes B,C and F)} + 1 \\&= 4\end{aligned}$$

$$\begin{aligned}V(G) &= \text{No. of Regions} \\&\bullet \quad = 4 \text{ (R1, R2, R3, R4)}\end{aligned}$$

Example

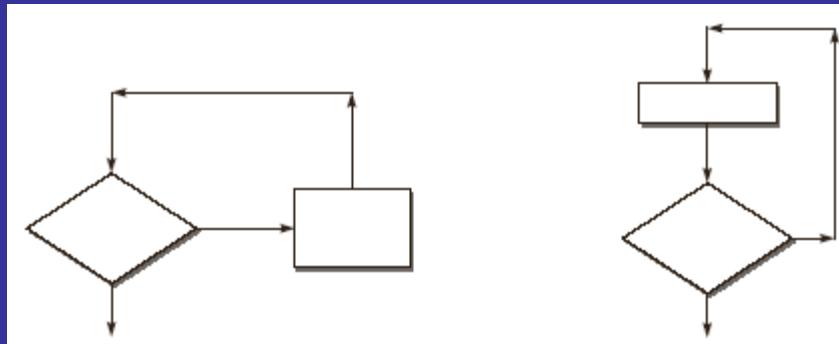
Independent Paths

- A-B-F-H
- A-B-F-G-H
- A-B-C-E-B-F-G-H
- A-B-C-D-F-H

Test case ID	Input num	Expected result	Independent paths covered by test case
1	1	No output is displayed	A-B-F-H
2	2	Prime number	A-B-F-G-H
3	4	Not a prime number	A-B-C-D-F-H
4	3	Prime number	A-B-C-E-B-F-G-H

Loop Testing

Simple Loops

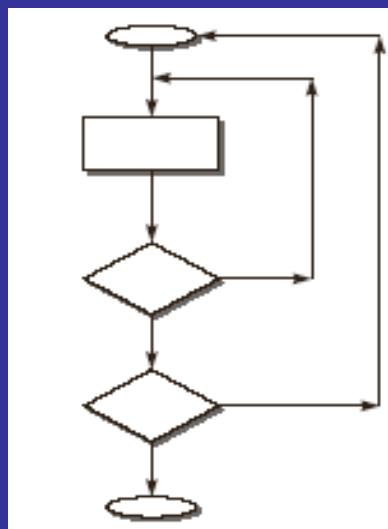


- Check whether you can bypass the loop or not. If the test case for bypassing the loop is executed and, still you enter inside the loop, it means there is a bug.
- Check whether the loop control variable is negative.
- Write one test case that executes the statements inside the loop.
- Write test cases for a typical number of iterations through the loop.
- Write test cases for checking the boundary values of maximum and minimum number of iterations defined (say min and max) in the loop. It means we should test for the min, min+1, min-1, max-1, max and max+1 number of iterations through the loop.

Loop Testing

Nested Loops: Nested loops When two or more loops are embedded, it is called a nested loop.

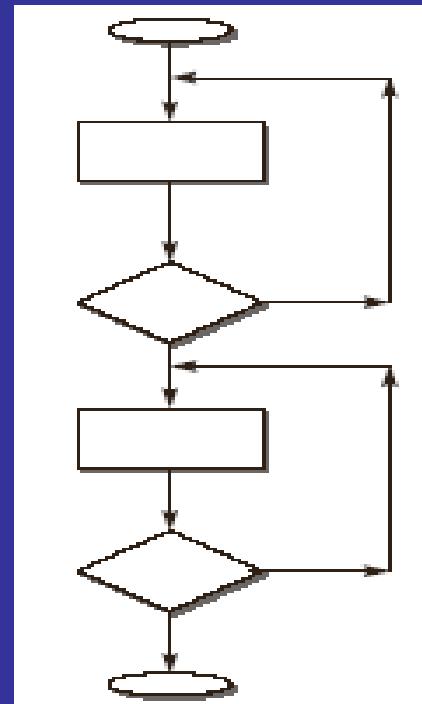
The strategy is to start with the innermost loops while holding outer loops to their minimum values. Continue this outward in this manner until all loops have been covered



Loop Testing

Concatenated Loops:

Loops are concatenated if it is possible to reach one after exiting the other while still on a path from entry to exit.



Data Flow Testing

A program unit accepts inputs, performs computations, assigns new values to variables, and returns results.

- One can visualize of “flow” of data values from one statement to another.
- A data value produced in one statement is expected to be used later.

Example:

Obtain a file pointer use it later.

If the later use is never verified, we do not know if the earlier assignment is write.

Two motivations of data flow testing:

1. The memory location for a variable is accessed in a “desirable” way.
2. Verify the correctness of data values “defined” (i.e. generated) – observe that all the “uses” of the value produce the desired results.

A programmer can perform a number of tests on data values. These tests are collectively known as data flow testing.

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. Errors may be unintentionally introduced in a program by programmers.

For instance, a programmer might use a variable without defining it. Moreover, he may define a variable, but not initialize it and then use that variable in a predicate.

For example,

```
int a;  
if(a == 67) { }
```

In this way, data flow testing gives a chance to look out for inappropriate data definition, its use in predicates, computations, and termination. It identifies potential bugs by examining the patterns in which that piece of data is used.

Data Flow testing helps us to pinpoint any of the following issues:

- A variable that is declared but never used within the program.
- A variable that is used but never declared.
- A variable that is defined multiple times before it is used.
- Deallocating a variable before it is used.

Data Flow Testing

Detect improper use of data values due to coding errors.

Closely examines the state of the data in the control flow graph resulting in a richer test suite than the one obtained from control flow graph based path testing strategies like branch coverage, all statement coverage, etc.

Data Flow Testing

States of data object

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

A=9; file opened etc.

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.

Reinitialized data, exiting from loop, closed file ,release of memory etc.

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.

Computational use (c-use) or predicate use (p-use).

Data Flow Testing

Data-Flow Anomalies: Data-flow anomalies represent the patterns of data usage which may lead to an incorrect execution of the code.

- **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.

Data Flow Testing

One -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.

It can be observed that not all data-flow anomalies are harmful, but most of them are suspicious and indicate that an error can occur.

Data Flow Testing

Terminology used in Data Flow Testing

- **Definition Node:** Defining a variable means assigning value to a variable for the very first time in a program.
Input statements, Assignment statements, Loop control statements, Procedure calls, etc.
- **Usage Node:** It means the variable has been used in some statement of the program.
Output statements, Assignment statements (Right), Conditional statements, Loop control statements, etc.
- **Loop Free Path Segment:** Path segment for which every node is visited once at most.
- **Simple Path Segment:** Segment: Path segment in which at most one node is visited twice.
- **Definition-Use Path (du-path)**
A du-path with respect to a variable v is a path between definition node and usage node of that variable. Usage node can be p-usage or c- usage node.
- **Definition-Clear path(dc-path)**
A dc-path with respect to a variable v is a path between definition node and usage node such that no other node in the path is a defining node of variable v.

Data Flow Testing

- The du-paths which are not dc-paths are important from testing viewpoint, as these are potential problematic spots for testing persons.
- Those du-paths which are definition-clear are easy to test in comparison to du-paths which are not dc-paths

Data Flow Testing

Static Data Flow Testing : With static analysis, the source code is analyzed without executing it.

Dynamic Data-Flow Testing : Dynamic data flow testing is performed with the intention to uncover possible bug in data usage during the execution of the code.

Strategies to create test cases:

- All-du Paths (ADUP)
- All-uses (AU)
- All-p-uses / Some-c-uses (APU + C)
- All-c-uses / Some-p-uses (ACU + P)
- All-Predicate-Uses(APU)
- All-Computational-Uses(ACU)
- All-Definition (AD)

Data Flow Testing

```
main()
{
    int work;
0. double payment =0;
1. scanf("%d", &work);
2. if (work > 0) {
3.     payment = 40;
4.     if (work > 20)
5.     {
6.         if(work <= 30)
7.             payment = payment + (work - 25) * 0.5;
8.         else
9.         {
10.             payment = payment + 50 + (work -30) * 0.1;
11.             if (payment >= 3000)
12.                 payment = payment * 0.9;
13.         }
14.     }
15. }
16. printf("Final payment", payment);
```

Data Flow Testing

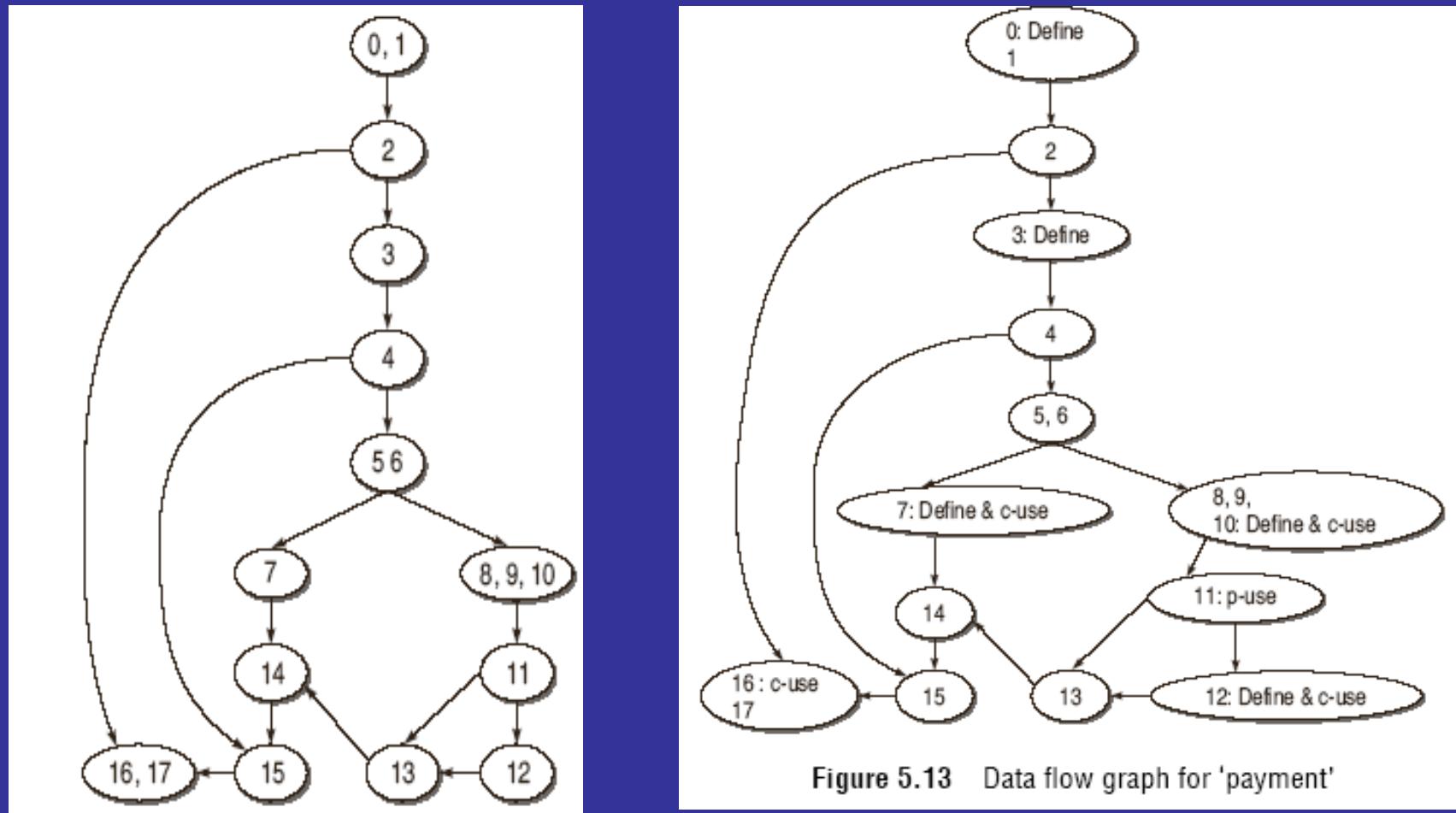


Figure 5.13 Data flow graph for 'payment'

Data Flow Testing

Find :

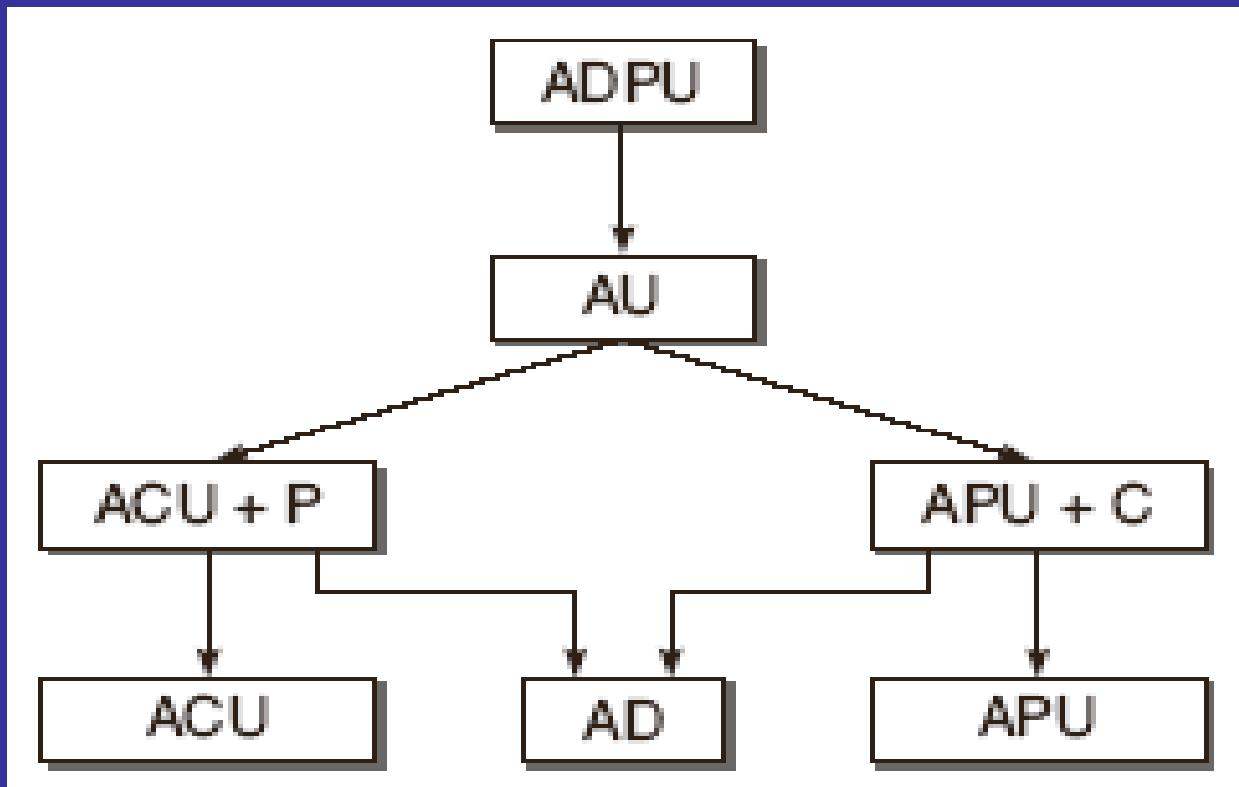
du Paths

dc Paths

for variable payment.

Variable	Defined at	Used at
Payment	0,3,7,10,12	7,10,11,12,16

Data Flow Testing



Mutation Testing

Mutation testing is a technique that focuses on measuring the adequacy of test data (or test cases).

The original intention behind mutation testing was to expose and locate weaknesses in test cases. Thus, mutation testing is a way to measure the quality of test cases.

Mutation Testing

- Mutation testing is the process of mutating some segment of code(putting some error in the code) and then testing this mutated code with some test data. If the test data is able to detect the mutations in the code, then the test data is quite good.
- Mutation testing helps a user create test data by interacting with the user to iteratively strengthen the quality of test data. During mutation testing, faults are introduced into a program by creating many versions of the program, each of which contains one fault. Test data are used to execute these faulty programs with the goal of causing each faulty program to fail.
- Faulty programs are called *mutants* of the original program and a mutant is said to be *killed* when a test case causes it to fail. When this happens, the mutant is considered *dead*

Mutation Testing

- Modify a program by introducing a single small change to the code
- A modified program is called *mutant*
- A mutant is said to be *killed* when the execution of test case cause it to fail. The mutant is considered to be *dead*
- A mutant is an *equivalent* to the given program if it always produce the same output as the original program
- A mutant is called *killable* or *stubborn*, if the existing set of test cases is insufficient to kill it.

Mutation Score

A mutation *score* for a set of test cases is the percentage of non-equivalent mutants *killed* by the test suite

$100*D/(N-E)$ where

D -> Dead

N-> Total No of Mutants

E-> No of equivalent mutants

The test suite is said to be *mutation-adequate* if its mutation score is 100%

Mutation Testing

Primary Mutants:

- Let us take one example of C program shown below

```
...
if (a>b)
    x = x + y;
else
    x = y;
printf("%d",x);
....
```

We can consider the following mutants for above example:

- M1: $x = x - y;$
- M2: $x = x / y;$
- M3: $x = x+1;$
- M4: $\text{printf}("%d",y);$

Mutation Testing

Test Data	x	y	Initial Program Result	Mutant Result
TD1	2	2	4	0 (M1)
TD2(x and y ≠ 0)	4	3	7	1.4 (M2)
TD3 (y ≠ 1)	3	2	5	4 (M3)
TD4(y = 0)	5	2	7	2 (M4)

Secondary Mutants:

Multiple levels of mutation are applied on the initial program.

Example Program:

If(a<b)

c=a;

Mutant for this code may be :

If(a==b)

c=a+1;

Mutation Testing Process

- Step 1: Begin with a program P and a set of test cases T known to be correct.
- Step 2: Run each test case in T against the program P .
 - If it fails (o/p incorrect) P must be modified and restarted. Else, go to step 3
- Step 3: Create a set of mutants $\{P_i\}$, each differing from P by a simple, syntactically correct modification of P .

Mutation Testing Process

- Step 4: Execute each test case in T against each mutant P_i .
- If the o/p is differ → the mutant P_i is considered incorrect and is said to be killed by the test case
- If P_i produces exactly the same results:
 - P and P_i are equivalent
 - P_i is killable (new test cases must be created to kill it)

Mutation Testing Process

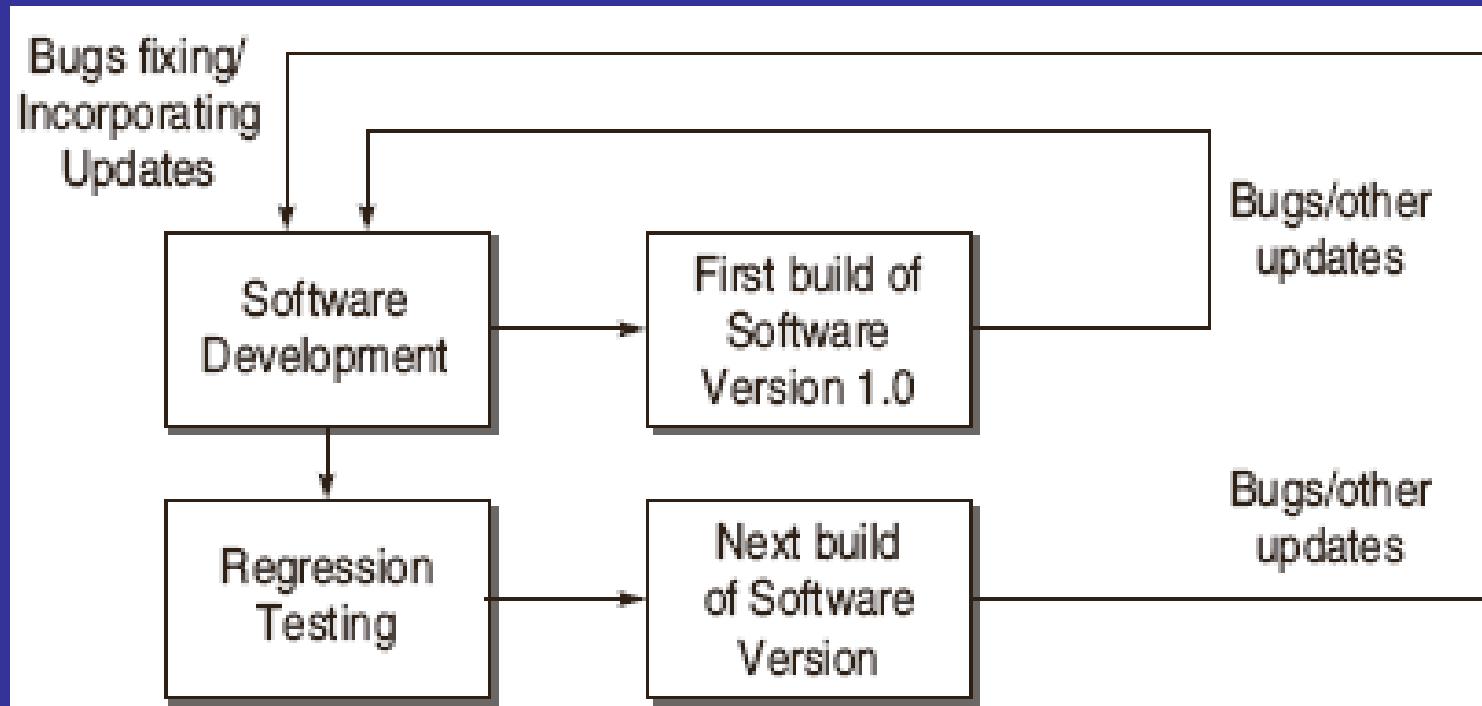
- Step 5: Calculate the mutation score for the set of test cases T .
- Mutation score = $100 \times D/(N - E)$,
- Step 6: If the estimated mutation adequacy of T *in step 5 is not sufficiently high*, then design a new test case that distinguishes P_i from P , add the new test case to T , and go to step 2.

Regression Testing

Progressive Vs Regressive Testing

- Regression testing is defined as a type of software testing to confirm that a recent program or code change has not adversely affected existing features.
- Regression testing is nothing but full or partial selection of already executed test cases which are re-executed to ensure existing functionalities work fine.
- This testing is done to make sure that new code changes should not have side effects on the existing functionalities. It ensures that old code still works once the new code changes are done.
- Regression Testing is necessary to maintain software whenever there is update in it.
- Regression testing is not another testing activity. Rather, it is the re-execution of some or all of the already developed test cases.
- Regression testing increases quality of software.

Regression Testing produces Quality Software



Objectives of Regression Testing

- **Regression Tests to check that the bug has been addressed**
- **Regression Tests to find other related bugs**
- **Regression tests to check the effect on other parts of the program**

Need / When to do regression testing?

- **Software maintenance**
Corrective maintenance
Adaptive maintenance
Perfective maintenance
Preventive maintenance

Adaptive – modifying the system to cope with changes in the software environment (DBMS, OS)

Perfective – implementing new or changed user requirements which concern functional enhancements to the software

Corrective – diagnosing and fixing errors, possibly ones found by users

Preventive – increasing software maintainability or reliability to prevent problems in the future

- **Rapid iterative development**
- **First step of integration**
- **Compatibility assessment and benchmarking**

Regression Testing Types

Bug-Fix Regression

Side-Effect Regression / Stability Regression

Regression Testing Types

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.

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.

Usability testing

- The testing that validates the ease of use, speed, and aesthetics of the product from the user's point of view is called usability testing.
- some of the characteristics of “usability testing” or “usability validation” are as follows:
 - Usability testing tests the product from the users' point of view. It encompasses a range of techniques for identifying how users actually interact with and use the product.
 - Usability testing is for checking the product to see if it is easy to use for the various categories of users.
 - Usability testing is a process to identify discrepancies between the user interface of the product and the human user requirements, in terms of the pleasantness and aesthetics aspects.

Usability testing

From the above definition it is easy to conclude that Something that is easy for one user may not be easy for another user due to different types of users a product can have . Something what is considered fast (interms of say, response time) by one user may be slow for another user as the machines used by them and the expectations of speed can be different. Something that is considered beautiful by someone may look ugly to another. A view expressed by one user of the product may not be the view of another.

Usability testing

Throughout the industry, usability testing is gaining momentum as sensitivity towards usability in products is increasing and it is very difficult to sell a product that does not meet the usability requirements of the users. There are several standards (for example, accessibility guidelines), organizations, tools (for example, Microsoft Magnifier), and processes that try to remove the subjectivity and improve the objectivity of usability testing.

Usability testing

Usability testing is not only for product binaries or executables. It also applies to documentation and other deliverables that are shipped along with a product. The release media should also be verified for usability. Let us take an example of a typical AUTORUN script that automatically brings up product setup when the release media is inserted in the machine. Sometimes this script is written for a particular operating system version and may not get auto executed on a different OS version. Even though the user can bring up the setup by clicking on the setup executable manually, this extra click (and the fact that the product is not automatically installed) may be considered as an irritant by the person performing the installation.

Who performs Usability testing

Generally, the people best suited to perform usability testing are typical representatives of the actual user segments who would be using the product, so that the typical user patterns can be captured, and People who are new to the product, so that they can start without any bias and be able to identify usability problems. A person who has used the product several times may not be able to see the usability problems in the product as he or she would have “got used” to the product's (potentially inappropriate) usability. Hence, a part of the team performing usability testing is selected from representatives outside the testing team. Inviting customer-facing teams (for example, customer support, product marketing) who know what the customers want and their expectations, will increase the effectiveness of usability testing.

Deliverables /Usability testing

A right approach for usability is to test every artifact that impacts users—such as product binaries, documentation, messages, media—covering usage patterns through both graphical and command user interfaces, as applicable.

Usability testing

Usability should not be confused with graphical user interface (GUI). Usability is also applicable to non-GUI interface such as command line interfaces (CLI). A large number of Unix/Linux users find CLIs more usable than GUIs. SQL command is another example of a CLI, and is found more usable by database users. Hence, usability should also consider CLI and other interfaces that are used by the users

WHEN TO DO USABILITY TESTING?

The most appropriate way of ensuring usability is by performing the usability testing in two phases. **First is design validation and the second is usability testing** done as a part of component and integration testing phases of a test cycle. When planning for testing, the usability requirements should be planned in parallel, upfront in the development cycle, similar to any other type of testing. Generally, however, usability is an ignored subject (or at least given less priority) and is not planned and executed from the beginning of the project. When there are two defects—one on functionality and other on usability—the functionality defect is usually given precedence. This approach is not correct as usability defects may demotivate users from using the software (even if it performs the desired function) and it may mean a huge financial loss to the product organization if users reject the product. Also, postponing usability testing in a testing cycle can prove to be very expensive as a large number of usability defects may end up as needing changes in design and needing fixes in more than one screen, affecting different code paths. All these situations can be avoided if usability testing is planned upfront.

WHEN TO DO USABILITY TESTING?

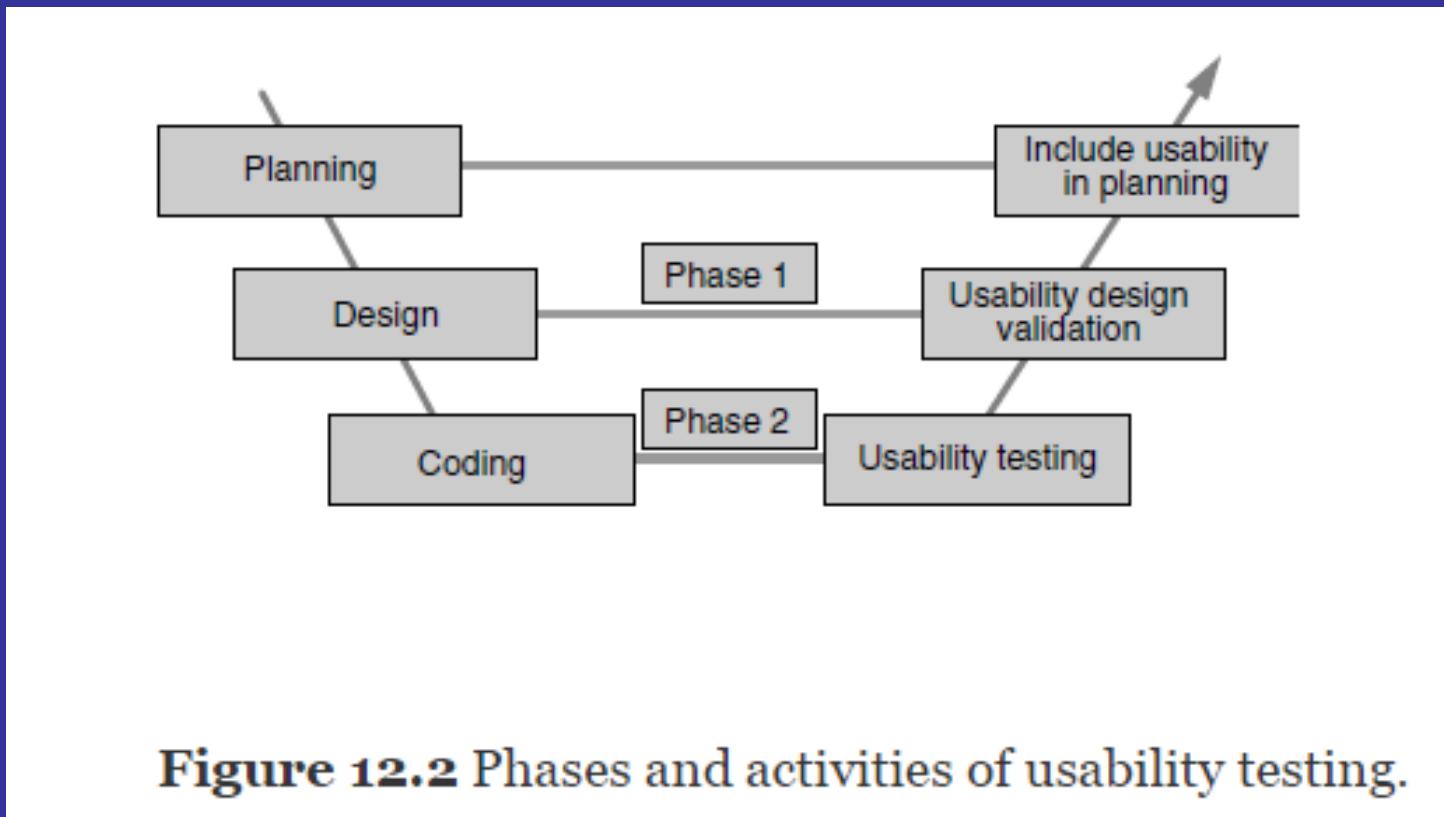


Figure 12.2 Phases and activities of usability testing.

WHEN TO DO USABILITY TESTING?

Usability design is verified through several means. Some of them are as follows:

- Style sheets : Style sheets are grouping of user interface design elements. Use of style sheets ensures consistency of design elements across several screens and testing the style sheet ensures that the basic usability design is tested. Style sheets also include frames, where each frame is considered as a separate screen by the user. Style sheets are reviewed to check whether they force font size, color scheme, and so on, which may affect usability.
- Screen prototypes : Screen prototype is another way to test usability design. The screens are designed as they will be shipped to the customers, but are not integrated with other modules of the product. Therefore, this user interface is tested independently without integrating with the functionality modules. This prototype will have other user interface functions simulated such as screen navigation, message display, and so on. The prototype gives an idea of how exactly the screens will look and function when the product is released. The test team and some real-life users test this prototype and their ideas for improvements are incorporated in the user interface. Once this prototype is completely tested, it is integrated with other modules of the product.

WHEN TO DO USABILITY TESTING?

- Paper designs : Paper design explores the earliest opportunity to validate the usability design, much before the actual design and coding is done for the product. The design of the screen, layout, and menus are drawn up on paper and sent to users for feedback. The users visualize and relate the paper design with the operations and their sequence to get a feel for usage and provide feedback. Usage of style sheets requires further coding, prototypes need binaries and resources to verify, but paper designs do not require any other resources. Paper designs can be sent through email or as a printout and feedback can be collected.
- Layout design : Style sheets ensure that a set of user interface elements are grouped and used repeatedly together. Layout helps in arranging different elements on the screen dynamically. It ensures arrangement of elements, spacing, size of fonts, pictures, justification, and so on, on the screen. This is another aspect that needs to be tested as part of usability design.

WHEN TO DO USABILITY TESTING?

- If an existing product is redesigned or enhanced, usability issues can be avoided by using the existing layout, as the user who is already familiar with the product will find it more usable. Making major usability changes to an existing product (for example, reordering the sequence of buttons on a screen) can end up confusing users and lead to user errors.
- **In the second phase**, tests are run to test the product for usability. Prior to performing the tests, some of the actual users are selected (who are new to the product and features) and they are asked to use the product. Feedback is obtained from them and the issues are resolved. Sometimes it could be difficult to get the real users of the product for usability testing. In such a case, the representatives of users can be selected from teams outside the product development and testing teams—for instance, from support, marketing, and sales teams. When to do usability also depends on the type of the product that is being developed.

QUALITY FACTORS FOR USABILITY

Some quality factors are very important when performing usability testing. As was explained earlier, usability is subjective and not all requirements for usability can be documented clearly. However focusing on some of the quality factors given below help in improving objectivity in usability testing are as follows.

Comprehensibility : **The product should have simple and logical structure of features and documentation.** They should be grouped on the basis of user scenarios and usage. The most frequent operations that are performed early in a scenario should be presented first, using the user interfaces. When features and components are grouped in a product, **they should be based on user terminologies, not technology or implementation.**

Consistency: A product needs to be consistent with any applicable standards, platform look-and-feel, base infrastructure, and earlier versions of the same product. Also, if there are multiple products from the same company, it would be worthwhile to have some consistency in the look-and-feel of these multiple products. Following same standards for usability helps in meeting the consistency aspect of the usability.

WHEN TO DO USABILITY TESTING?

Navigation : This helps in determining how easy it is to select the different operations of the product. An option that is buried very deep requires the user to travel to multiple screens or menu options to perform the operation. The number of mouse clicks, or menu navigations that is required to perform an operation should be minimized to improve usability. When users get stuck or get lost, there should be an easy option to abort or go back to the previous screen or to the main menu so that the user can try a different route.

Responsiveness : How fast the product responds to the user request is another important aspect of usability. This should not be confused with performance testing. Screen navigations and visual displays should be almost immediate after the user selects an option or else it could give an impression to the user that there is no progress and cause him or her to keep trying the operation again. Whenever the product is processing some information, the visual display should indicate the progress and also the amount of time left so that the users can wait patiently till the operation is completed. Adequate dialogs and popups to guide the users also improve usability.

USABILITY TESTING : AESTHETICS TESTING

AESTHETICS TESTING : Another important aspect in usability is making the product “beautiful.” Performing aesthetics testing helps in improving usability further. This testing is important as many of the aesthetics related problems in the product from many organizations are ignored on the ground that they are not functional defects. All the aesthetic problems in the product are generally mapped to a defect classification called “Cosmetic,” which is of low priority. Having a separate cycle of testing focusing on aesthetics helps in setting up expectations and also in focusing on improving the look and feel of the user interfaces. Aesthetics is not in the external look alone. It is in all the aspects such as messages, screens, colors, and images. A pleasant look for menus, pleasing colors, nice icons, and so on can improve aesthetics.

Accessibility testing

Accessibility Testing is a subset of usability testing, and it is performed to ensure that the application being tested is usable by people with disabilities like hearing, color blindness, old age and other disadvantaged groups.

- People with disabilities use assistive technology which helps them in operating a software product.

Accessibility testing involves testing these alternative methods of using the product and testing the product along with accessibility tools. Accessibility is a subset of usability and should be included as part of usability test planning.

Verifying the product usability for physically challenged users is called accessibility testing.

Accessibility testing

Accessibility testing may be challenging for testers because they are unfamiliar with disabilities. It is better to work with disabled people who have specific needs to understand their challenges.

Accessibility to the product can be provided by two means :

Making use of accessibility features provided by the underlying infrastructure (for example, operating system), called basic accessibility, and

Providing accessibility in the product through standards and guidelines, called product accessibility.

Accessibility testing

Basic Accessibility: Basic accessibility is provided by the hardware and operating system. All the input and output devices of the computer and their accessibility options are categorized under basic accessibility.

Examples:

Keyboard accessibility

Screen accessibility

- **Speech Recognition Software** - It will convert the spoken word to text , which serves as input to the computer.
- **Screen reader software** - Used to read out the text that is displayed on the screen
- **Screen Magnification Software**- Used to enlarge the monitor and make reading easy for vision-impaired users.
- **Special keyboard** made for the users for easy typing who have motion control difficulties.

Accessibility testing

Product Accessibility :A good understanding of the basic accessibility features is needed while providing accessibility to the product. A product should do everything possible to ensure that the basic accessibility features are utilized by it. For example, providing detailed text equivalent for multimedia files ensures the captions feature is utilized by the product.

Accessibility testing

Sample requirement #1: Text equivalents have to be provided for audio, video, and picture images.

Sample requirement #2: Documents and fields should be organized so that they can be read without requiring a particular resolution of the screen, and templates (known as style sheets).

Sample requirement #3: User interfaces should be designed so that all information conveyed with color is also available without color.

Sample requirement #4: Reduce flicker rate, speed of moving text; avoid flashes and blinking text.

Sample requirement #5: Reduce physical movement requirements for the users when designing the interface and allow adequate time for user responses.

Table 12.2 Sample list of usability and accessibility tools.

Name of the tool	Purpose
JAWS	For testing accessibility of the product with some assistive technologies.
HTML validator	To validate the HTML source file for usability and accessibility standards.
Style sheet validator	To validate the style sheets (templates) for usability standards set by W3C.
Magnifier	Accessibility tool for vision challenged (to enable them to enlarge the items displayed on screen)
Narrator	Narrator is a tool that reads the information displayed on the screen and creates audio descriptions for vision-challenged users.
Soft keyboard	Soft keyboard enables the use of pointing devices to use the keyboard by displaying the keyboard template on the screen.

Accessibility testing

Following are the point's needs to be checked for application to be used by all users. This **checklist** is used for signing off accessibility testing.

- Whether an application provides keyboard equivalents for all mouse operations and windows?
- Whether instructions are provided as a part of user documentation or manual? Is it easy to understand and operate the application using the documentation?
- Whether tabs are ordered logically to ensure smooth navigation?
- Whether shortcut keys are provided for menus?
- Whether application supports all operating systems?
- Whether color of the application is flexible for all users?
- Whether images or icons are used appropriately, so it's easily understood by the end users?

Accessibility testing

- Whether an application has audio alerts?
- Whether user can adjust or disable flashing, rotating or moving displays?
- Check to ensure that color-coding is never used as the only means of conveying information or indicating an action
- Whether highlighting is viewable with inverted colors? Testing of color in the application by changing the contrast ratio
- Whether audio and video related content are properly heard by the disability people ? Test all multimedia pages with no speakers in websites
- Whether training is provided for users with disabilities that will enable them to become familiar with the software or application?

References:

1. Software Testing Principles and Practices, Naresh Chauhan, Second edition, Oxford Higher Education
2. Software Testing: Principles and Practice by Srinivasan Desikan, Gopalaswamy Ramesh

Module 3 :

Testing Metrics for Monitoring and Controlling the Testing Process

Software Metrics

Metrics can be defined as “STANDARDS OF MEASUREMENT”.

Software Metrics are used to measure the quality of the project. Simply, Metric is a unit used for describing an attribute. Metric is a scale for measurement.

Suppose, in general, “Kilogram” is a metric for measuring the attribute “Weight”. Similarly, in software, “How many issues are found in thousand lines of code?”

Test metrics example:

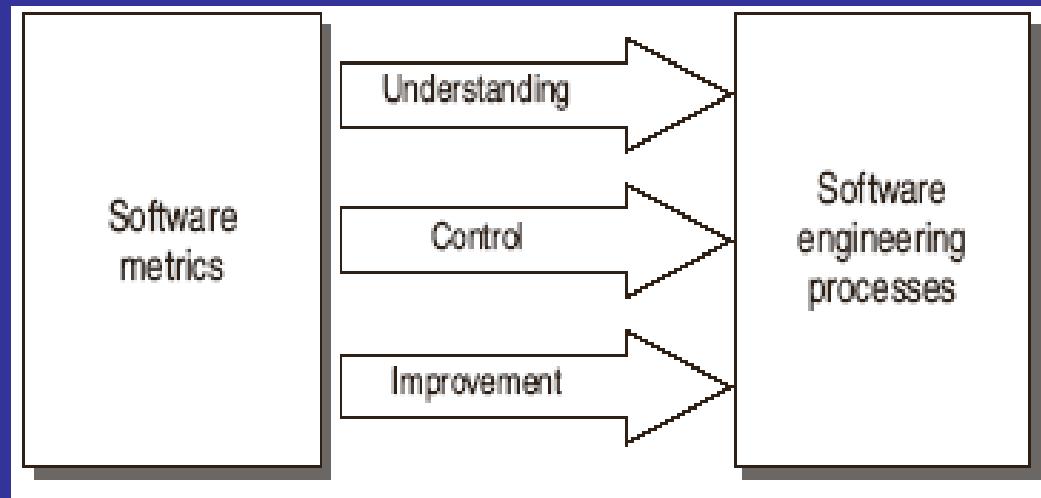
How many defects exist within the module?

How many test cases are executed per person?

What is the Test coverage %?

Need of Software Measurement

- **Understanding**
- **Control**
- **Improvement**



Software Metrics

Product Metrics

**Measures of the software product at any stage of its development
From requirements to installed system.**

- Complexity of S/W design and code
- Size of the final program
- Number of pages of documentation produced

Process Metrics

Measures of the S/W development process

- Overall development time
- Type of methodology used
- Average level of experience of programming staff

Measurement Objectives for Testing

The objectives for assessing a test process should be well defined.

GQM(Goal Question Metric) Framework:

- List the major goals of the test process.
- Drives 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 in Software Testing

Category	Attributes to be Measured
Progress	<ul style="list-style-type: none">• Scope of testing• Test progress• Defect backlog• Staff productivity• Suspension criteria• Exit criteria
Cost	<ul style="list-style-type: none">• Testing cost estimation• Duration of testing• Resource requirements• Training needs of testing group and tool requirement• Cost-effectiveness of automated tool
Quality	<ul style="list-style-type: none">• Effectiveness of test cases• Effectiveness of smoke tests• Quality of test plan• Test completeness
Size	<ul style="list-style-type: none">• Estimation of test cases• Number of regression tests• Tests to automate

Attributes : Progress

- Scope of testing: Overall amount of work involved
- Test Progress: Schedule, budget, resources
 - Major milestones
 - NPT
 - Test case Escapes (TCE)
 - Planned versus Actual Execution (PAE) Rate
 - Execution Status of Test (EST) Cases(*Failed, Passed, Blocked, Invalid and Untested*)
- *Defect Backlog:* Number of defects that are unresolved/outstanding
- *Staff productivity:* Time spent in test planning, designing, Number of test cases developed. (useful to estimate the cost and duration for testing activities)

Attributes : Progress

Suspension criteria: It describe the circumstances under which testing would stop temporarily.

- Incomplete tasks on critical path, Large volume of bugs, critical bugs, incomplete test environment

Exit criteria: It indicates the conditions that move the testing activities forward from one level to the next.

- Rate of fault discovery in regression tests, frequency of failing fault fixes , fault detection rate.

Attributes: Cost

Cost	<ul style="list-style-type: none">• Testing cost estimation• Duration of testing• Resource requirements• Training needs of testing group and tool requirement• Cost-effectiveness of automated tool
------	---

Attributes: Cost

Effectiveness of Test Cases

- Number of faults found in testing.
- Number of failures observed by the customer which can be used as a reflection of the effectiveness of the test cases.
- Defect age

Defect age is used in another metric called defect spoilage to measure the effectiveness of defect removal activities.

- Spoilage = Sum of (Number of Defects x defect age) / Total number of defects

Effectiveness of Test Cases

- Defect Removal Efficiency (DRE) metric defined as follows:

$$DRE = \frac{\text{Number of Defects Found in Testing}}{\text{Number of Defects Found in Testing} + \text{Number of Defects Not Found}}$$

Spoilage Metric

- Defects are injected and removed at different phases of a software development cycle
- The cost of each defect injected in phase X and removed in phase Y increases with the increase in the distance between X and Y
- An effective testing method would find defects earlier than a less effective testing method .
- A useful measure of test effectiveness is defect age, called PhAge

Phase Injected	Phase Discovered							
	Requirements	High-Level Design	Detailed Design	Coding	Unit Testing	Integration Testing	System Testing	Acceptance Testing
Requirements	0	1	2	3	4	5	6	7
High-Level Design		0	1	2	3	4	5	6
Detailed Design			0	1	2	3	4	5
Coding				0	1	2	3	4

Table: Scale for defect age

Pooja Malhotra

Spoilage Metric

Phase Injected	Phase Discovered								Total Defects
	Requirements	High-Level Design	Detailed Design	Coding	Unit Testing	Integration Testing	System Testing	Acceptance Testing	
Requirements	0	7	3	1	0	0	2	4	17
High-Level Design		0	8	4	1	2	6	1	22
Detailed Design			0	13	3	4	5	0	25
Coding				0	63	24	37	12	136
Summary	0	7	11	18	67	30	50	17	200

Table: Defect injection and versus discovery on project *Boomerang*

A new metric called spoilage is defined as

$$\text{Spoilage} = \frac{\sum (\text{Number of Defects} \times \text{Discovered Phase})}{\text{Total Number of Defects}}$$

Spoilage Metric

Phase Injected	Phase Discovered							Weight	Total Defects	$\text{Spoilage} = \frac{\text{Weight}}{\text{Total Defects}}$
	Requirements	High-Level Design	Detailed Design	Coding	Unit Testing	Integration Testing	System Testing			
Requirements	0	7	6	3	0	0	12	28	56	17 3.294117647
High-Level		0	8	8	3	8	30	6	63	22 2.863636364
Detailed Design			0	13	6	12	20	0	51	25 2.04
Coding				0	63	48	111	48	270	136 1.985294118
Summary	0	7	14	24	72	68	173	82	440	200 2.2

Table: Number of defects weighted by defect age on project *Boomerang*

Spoilage Metric

- The spoilage value for the Boomerang test project is 2.2
- A spoilage value close to 1 is an indication of a more effective defect discovery process
- This metric is useful in measuring the long-term trend of test effectiveness in an organization

Measuring Test completeness

Refer to how much of code and requirements are covered by the test set.

The relationship between code coverage and the number of test cases:

$$-(p/N)*x$$

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

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

- Requirement traceability matrix

Quality

Effectiveness of smoke tests (establish confidence over stability of a system)

SMOKE TESTING, also known as “Build Verification Testing”, is a type of software testing that comprises of a non-exhaustive set of tests that aim at ensuring that the most important functions work. The result of this testing is used to decide if a build is stable enough to proceed with further testing.

Quality of Test Plan : The quality of test plan is measured in concern with the possible number of errors.

- Multidimensional qualitative method using rubrics

Attributes : Size

- Number of test cases reused
- Number of test cases added to test database
- Number of test cases rerun when changes are made to the S/W
- Number of planned regression tests executed
- Number of planned regression tests executed and passed

Size	<ul style="list-style-type: none">• Estimation of test cases• Number of regression tests• Tests to automate
------	---

Size Metrics

- **Line of Code (LOC)**
- **Token Count (Halstead Product Metrics) : counting the number of operators and operands.**

Program Vocabulary

$$n = n_1 + n_2$$

where n = program vocabulary

n_1 = number of unique operators

n_2 = number of unique operands

Program Length

$$N = N_1 + N_2$$

Where N = program length

N_1 = all operators appearing in the implementation

N_2 = all operands appearing in the implementation

Token Count

Program Volume : Refers to the size of the program.

$$V = N \log_2 n$$

where V = Program volume

N = Program length

n = Program vocabulary

Function Point Analysis

Estimation models for estimating testing efforts

1) Halstead metrics for estimating testing effort

- $PL = 1 / [(n_1 / 2) * (N_2 / n_2)]$
- $e(\text{effort}) = V/PL$
- V= describe number of volume of information in bits required to specify a program.
- PL= measure of software complexity
- Percentage of testing effort (k) for module k $= e(k) / \sum e(i)$

$e(k)$: effort required for module k

$\sum e(i)$: sum of halstead effort across all modules of the system

2) Development Ratio Method(No. of testing personal required is estimated on the basis of developer – tester ratio)

- Type and complexity of the software being developed
- Testing level
- Scope of testing
- Test effectiveness
- Error tolerance level for testing
- Available budget

Estimation models for estimating testing efforts

3) Project staff ratio method

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

	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

Estimation models for estimating testing efforts

5) Task planning method

Number of Test Procedures (NTP)	Person-hours consumed for testing (PH)	Hours per test procedure = PH/NTP
840	6000	7.14
1000 (New project)	7140	7.14

Testing activity	Historical value	% time of the project consumed on the test activity	Preliminary estimate of person hours	Adjusted estimate of person-hours
Test planning	210	3.5	249	
Test design	150	2.5	178	
Test execution	180	3	214	
Project total	6000	100%	7140	6900

	NTP	PH (Adjusted estimate)	Number of hours per test procedure = PH/NTP	TP	Number of testers = PH/TP
New project estimate	1000	6900	6.9	1856 hrs	3.7

Architectural Design Metric used for Testing

Structural Complexity

2

$$S(m) = f \text{out}(m)$$

where S is the structural complexity of a module m
and $f_{\text{out}}(m)$ is the fan-out of module m.

This metric gives us the number of stubs required for unit testing of the module m.(Unit Testing)

Data Complexity

$$D(m) = v(m) / [f_{\text{out}}(m) + 1]$$

where $v(m)$ is the number of input and output variables that are passed to and from module m.

This metric measures the complexity in the internal interface for a module m and indicates the probability of errors in module m.

System Complexity

$$SC(m) = S(m) + D(m)$$

It is defined as the sum of structural and data complexity.

Overall architectural complexity of system is the sum total of system complexities of all the modules.

Efforts required for integration testing increases with the architectural complexity of the system.

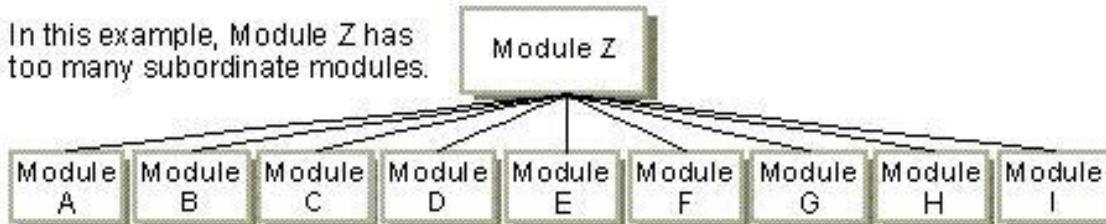
Information Flow Metrics used for Testing

- **Local Direct Flow**
- **Local InDirect Flow**
- **Global Flow**
- **Fan-in of a module**
- **Fan-out of a module**

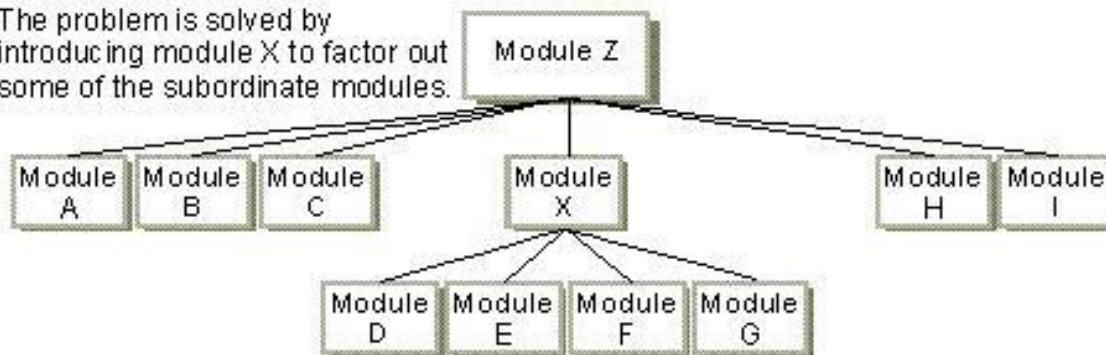
Information Flow Metrics used for Testing

Example of a Solution to Excessively High Fan-Out

In this example, Module Z has too many subordinate modules.



The problem is solved by introducing module X to factor out some of the subordinate modules.



Information Flow Metrics used for Testing :

Henry & Kafura Design Metric

Measure the total level of information flow between individual modules and rest of the system

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

The higher the IF complexity for m, greater is the effort in integration and integration testing, thereby increasing the probability of errors in the module.

Cyclomatic Complexity Measures for Testing

- Since 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.
- McCabe has suggested that ideally, cyclomatic number should be less than equal to 10. This number provides a quantitative measure of testing difficulty. If cyclomatic number is more than 10, then testing effort increases due to :
 - Number of errors increases.
 - Time required to find and correct the errors increases

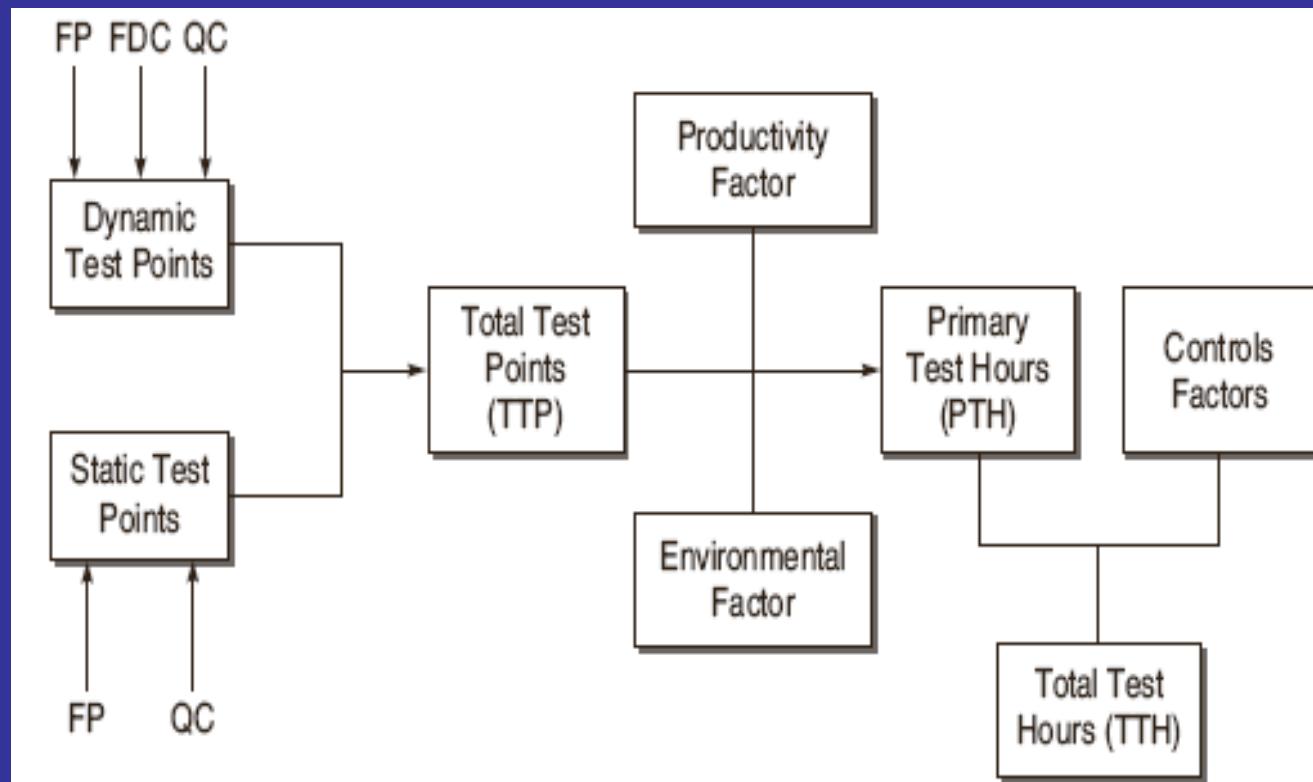
Function Point Metrics for Testing

Used for measuring size of a software, Testing effort

- Number of Hours required for testing per Function Point (FP)
- Number of FPs tested per person-months
- Total cost of testing per FP
- Defect density = *Number of defects (by phase or in total) / Total number of FPs*
- Test case Coverage = *Number of Test cases / Total number of FPs*
- Number of Test Cases = $(\text{Function Points})^{1.2}$
- Number of Acceptance Test Cases = $(\text{Function Points}) \times 1.2$

Test Point Analysis: Black Box test effort estimation

TPA calculates the test effort estimation in test points for important functions.



Test Point Analysis

- **Calculating Dynamic Test Points**

$$DTP = FP \times FDCw \times QCdw$$

Where DTP = number of dynamic test points

FP = Function point assigned to function

FDCw = Function dependent factor wherein weights are assigned to function dependent factors

QCdw = Quality characteristic factor wherein weights are assigned to dynamic quality characteristics

Test Point Analysis

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

Where FI_w = Function importance rated by user

UIN_w = Weights given to Usage intensity of the function, i.e. how frequently function is being used

I = Weights given to the function for interfacing with other functions, i.e. if there is change in function, how many functions in the system will be affected

C = Weights given to complexity of function, i.e. how many conditions are in the algorithm of function

U = Uniformity factor

Test Point Analysis

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

Table 11.8 Ratings for FDC_w factors

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	

Table 11.9 Ratings and weights for QC_{dw}

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

Test Point Analysis

- **Calculating Static Test Points**

- $STP = FP \times \sum QCsw / 500$

Where STP = Static Test Point

FP = Total function point assigned to system

- QCsw = Quality characteristic factor wherein weights are assigned to static quality characteristics
- Static quality characteristic is what can be tested with a checklist.
- QCsw is assigned the value 16 for each quality characteristic which can be tested statically using the check list.
- **Total Test Points (TTP) = DTP + STP**

Test Point Analysis

Calculating Primary Test Hours

- **Primary Test Hours (PTH) = TTP x Productivity factor x Environmental factor**
- **Productivity Factor** ranges between 0.7 to 2.0.
 - Skill set,knowledge,experience of tester
- **Environmental factor = weights of (Test Tools + Development Testing + Test Basis + Development Environment + Testing Environment + Testware) / 21**

Test Point Analysis

Table 11.10 Test tool ratings

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

Table 11.11 Development testing ratings

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.

Table 11.12 Test basis rating

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

Table 11.13 Development environment rating

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

Table 11.14 Test environment rating

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.

Table 11.15 Testware rating

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

Test Point Analysis

Calculating Total Test Hours

Total Test Hour = PTH + Planning and control allowance

- **Planning & Control Allowance (%)= weights of (Team size + Planning and Control Tools)**
- **Planning & Control Allowance (hours)= Planning & Control Allowance (%) x PTH**

Testing Progress Metrics

- Test Procedure Execution Status:

Test proc Exec. Status=Number of executed test cases/Total number of test cases

- Defect Aging :Turnaround time for a bug to be corrected.

Defect aging = closing date of bug - start date when bug was opened

- Defect Fix Time to Retest:

Defect Fix Time to Retest = Closing date of bug and releasing in new build – Date of retesting the bug

- Defect Trend Analysis: defined as the trend in the number of defects found as testing progresses.

- Number of defects of each type detected in unit test per hour
- Number of defects of each type detected in integration test per hour
- Severity level for all defects

Testing Progress Metrics

- Recurrence Ratio: Indicates the quality of bug –fixes.
Number of bugs remaining per fix.
- Defect Density:
 1. Defect Density = Total number of defects found for a requirement /Total number of test cases executed for that requirement
 2. Pre- ship defect density/Post-ship defect density
- Coverage Measures: Helps in identifying the work to be done.

White-box testing

Degree of statement , branch , data flow and basis path coverage

Actual degree of coverage/Planned degree of coverage

Black-box Testing

Number of features or Ecs actually covered/ Total number of features or Ecs.

Testing Progress Metrics

Tester 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

Testing Progress Metrics

- Budget and Resource Monitoring Measures:

Earned value tracking

For the planned earned values, we need the following measurement data :

1. Total estimated time or cost for overall testing effort
2. Estimated time or cost for each testing activity
3. Actual time or cost for each testing activity

Estimated time or cost for testing activity / Actual time or cost of testing activity

- Test case effectiveness Metric:

TCE= $100 * (\text{Number of defects found by the test cases} / \text{total number of defects})$

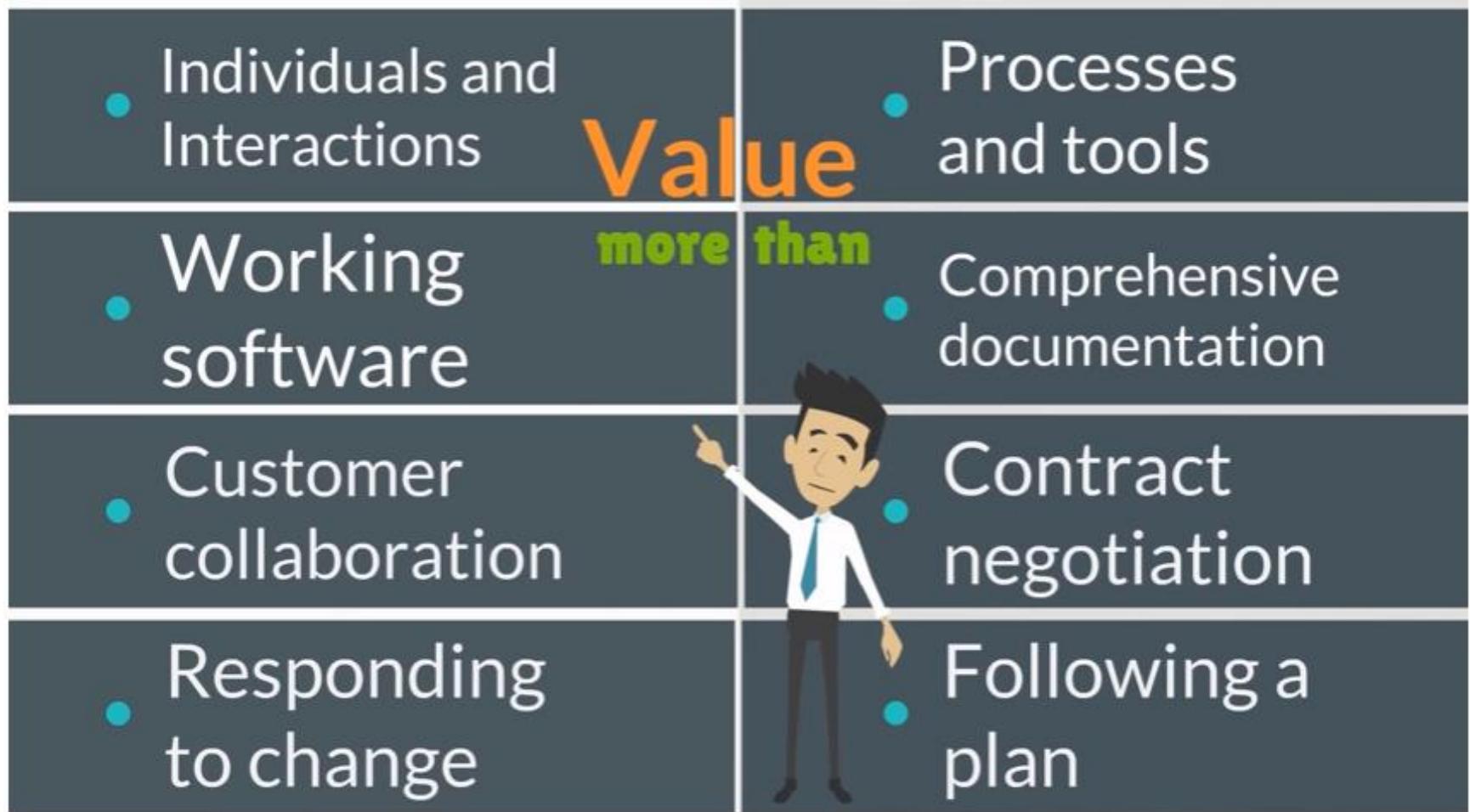
Agile software Testing

Agile Methodology

AGILE methodology is a practice that promotes **continuous iteration** of development and testing throughout the software development lifecycle of the project. Both development and testing activities are concurrent unlike the Waterfall model.

- The agile software development emphasizes on four core values.
 - Individual and team interactions over processes and tools
 - Working software over comprehensive documentation
 - Customer collaboration over contract negotiation
 - Responding to change over following a plan

Agile Methodology : Values more for items on left side than right one



What is Agile Testing?

A software testing practice that follows the principles of agile software development is called Agile Testing. Agile is an iterative development methodology, where requirements evolve through collaboration between the customer and self-organizing teams and agile aligns development with customer needs.

Advantages of Agile Testing

- Agile Testing Saves Time and Money
- Less Documentation
- Regular feedback from the end user
- Daily meetings can help to determine the issues well in advance

Principles of Agile Testing

- **Testing is NOT a Phase:** Agile team tests continuously and continuous testing is the only way to ensure continuous progress.
- **Testing Moves the project Forward:** When following conventional methods, testing is considered as quality gate but agile testing provide feedback on an ongoing basis and the product meets the business demands.
- **Everyone Tests:** In conventional SDLC, only test team tests while in agile including developers and BA's test the application.
- **Shortening Feedback Response Time:** In conventional SDLC, only during the acceptance testing, the Business team will get to know the product development, while in agile for each and every iteration, they are involved and continuous feedback shortens the feedback response time and cost involved in fixing is also less.
- **Clean Code:** Raised defects are fixed within the same iteration and thereby keeping the code clean.
- **Reduce Test Documentation:** Instead of very lengthy documentation, agile testers use reusable checklist, focus on the essence of the test rather than the incidental details.
- **Test Driven:** In conventional methods, testing is performed after implementation while in agile testing, testing is done while implementation.
- **More is less : More Interactions and More Focus leads to Less doubts and Less Defects.**

Scrum

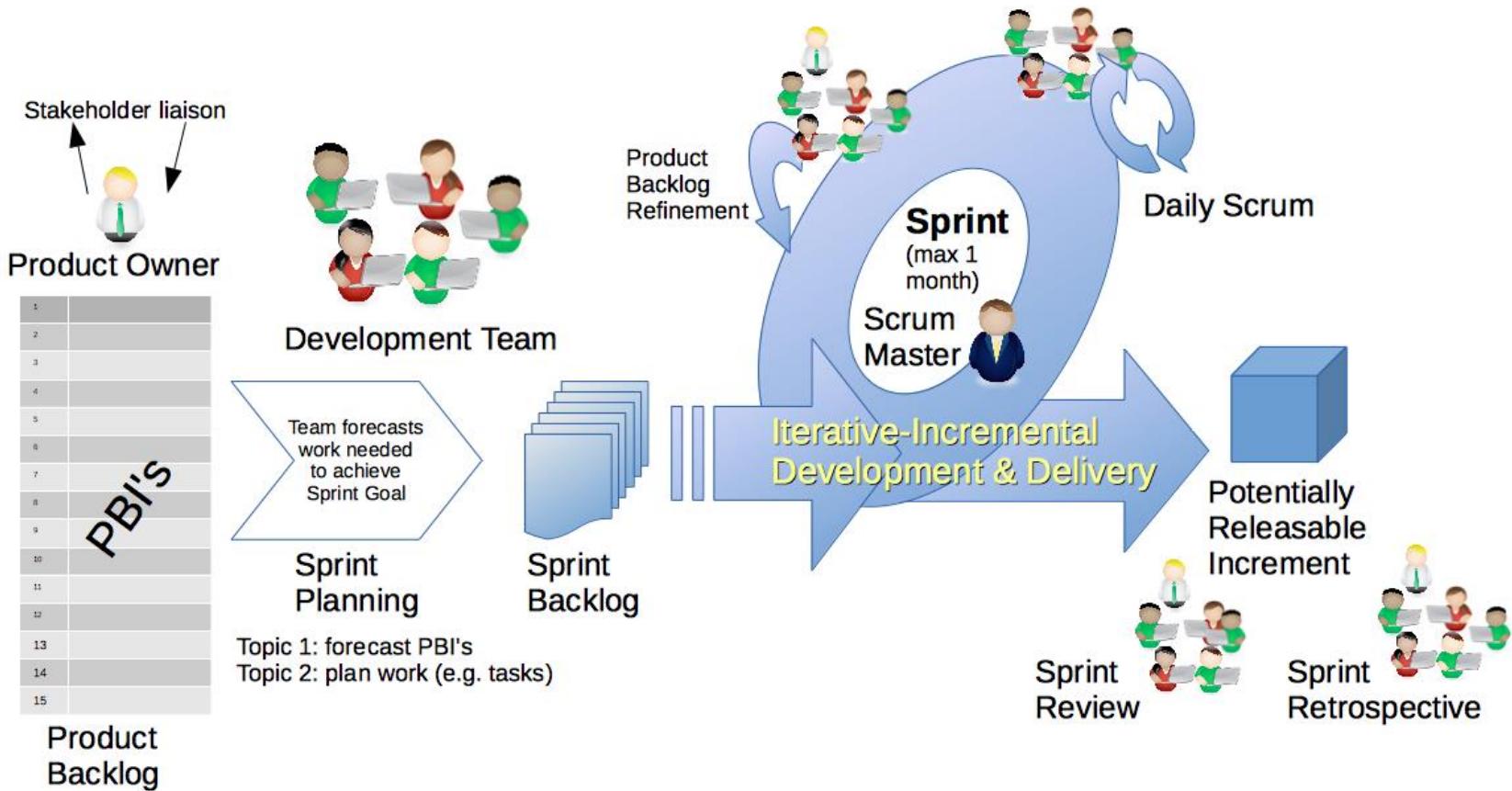
Scrum is a framework for managing software development. It is designed for teams of three to nine developers who: break their work into actions that can be completed within fixed duration cycles (called "sprints"), track progress and re-plan in daily 15-minute stand-up meetings, and collaborate to deliver workable software every sprint.

- SCRUM is an agile development method which concentrates specifically on how to manage tasks within a team-based development environment. Basically, Scrum is derived from activity that occurs during a rugby match. Scrum believes in empowering the development team and advocates working in small teams (say- 7 to 9 members).

Scrum

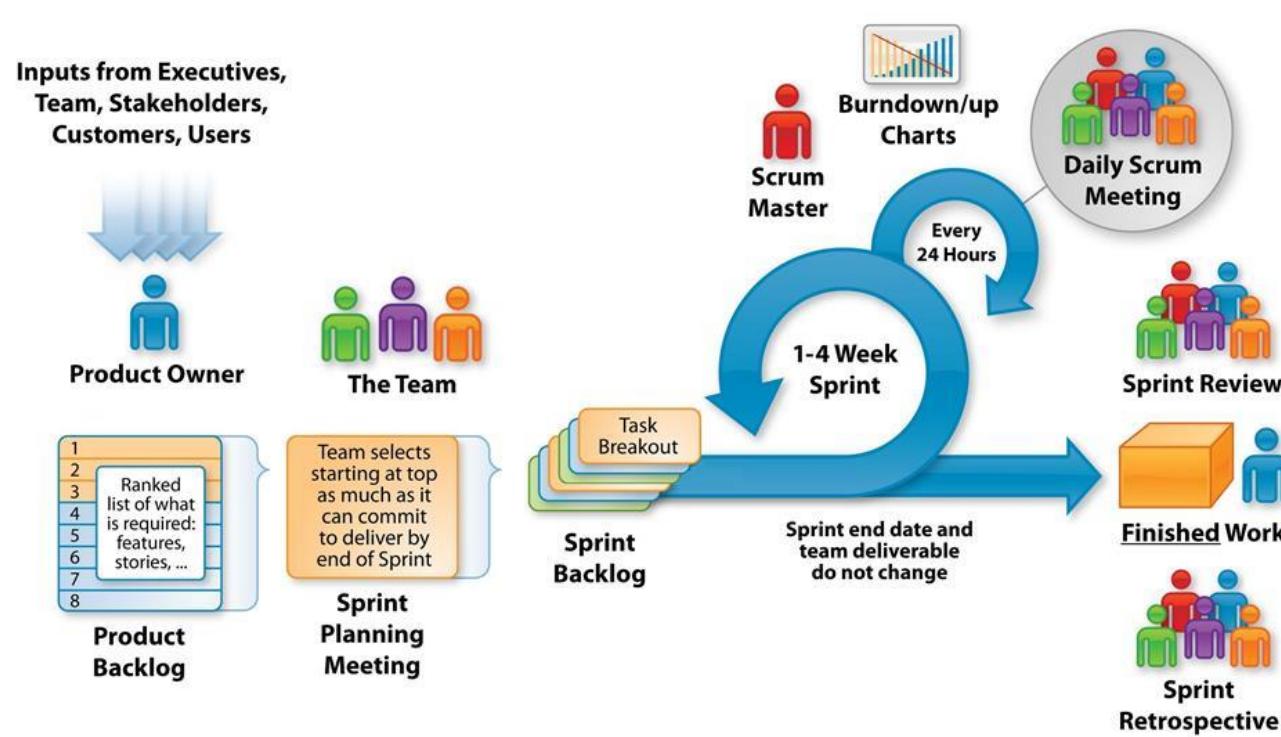


Scrum



Scrum

The Agile - Scrum Framework

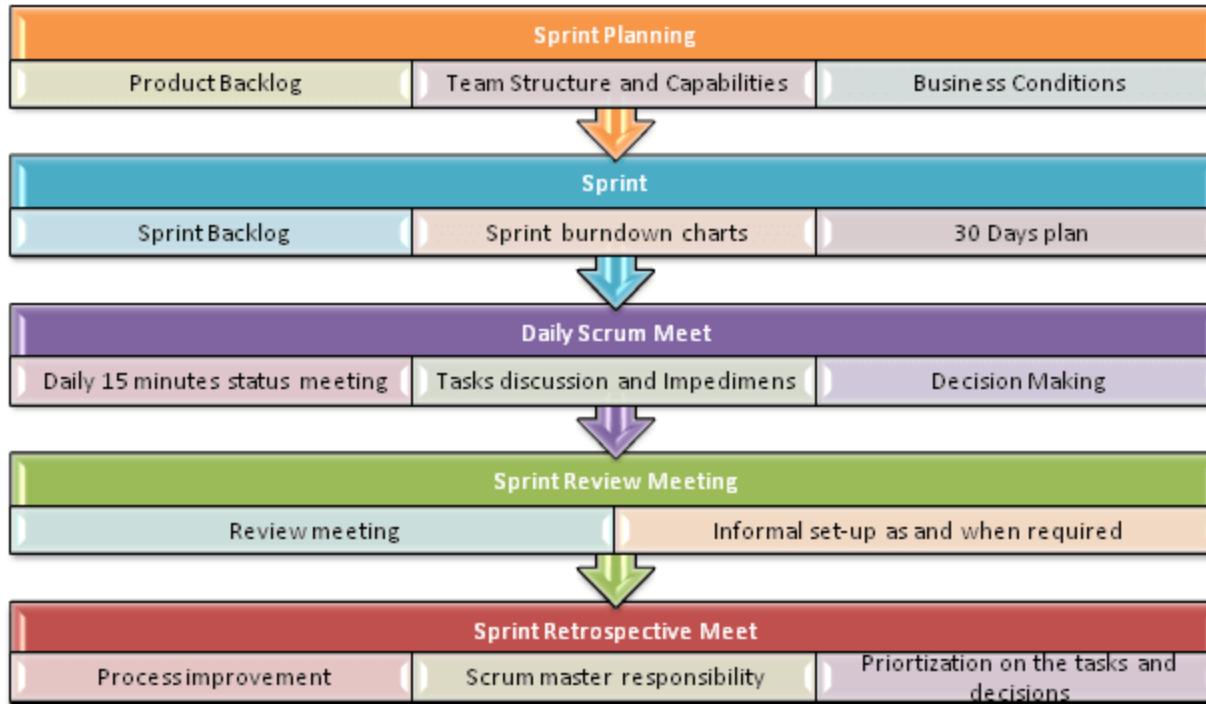


It consists of three roles, and their responsibilities are explained as follows:

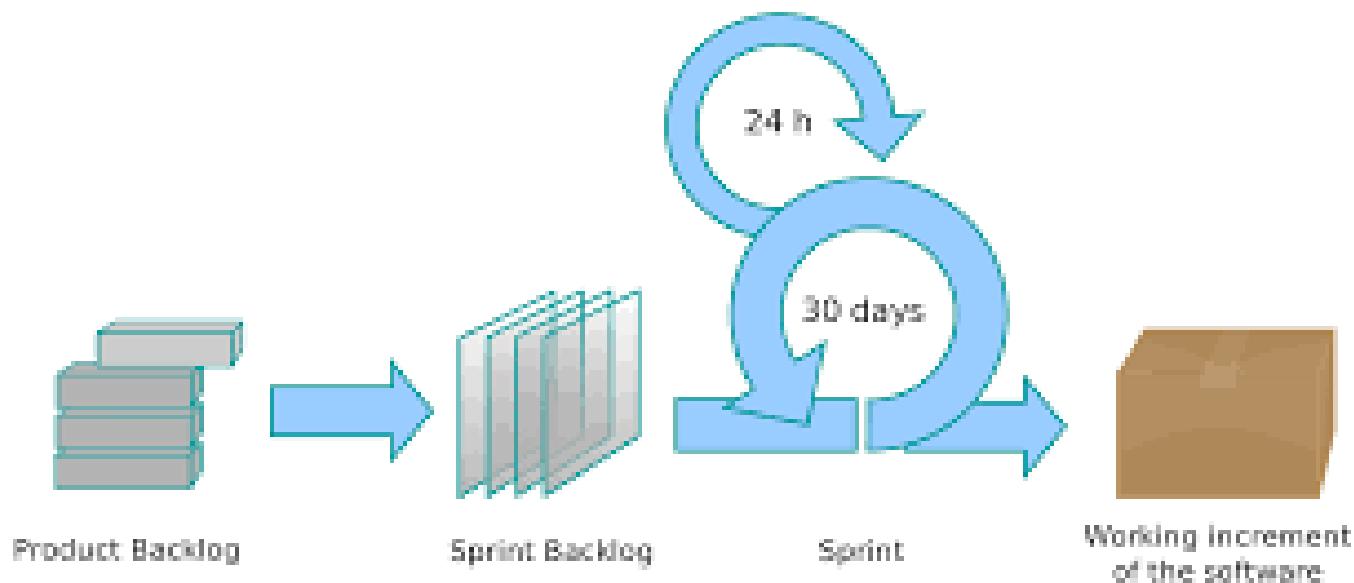
- Scrum Master
 - Master is responsible for setting up the team, sprint meeting and removes obstacles to progress
 - Helping the product owner maintain the product backlog in a way that ensures the needed work is well understood so the team can continually make forward progress
 - Helping the team to determine the definition of done for the product, with input from key stakeholders
 - Coaching the team, within the Scrum principles, in order to deliver high-quality features for its product
 - Promoting self-organization within the team
 - Helping the scrum team to avoid or remove impediments to its progress, whether internal or external to the team
 - Facilitating team events to ensure regular progress
 - Educating key stakeholders in the product on Scrum principles
- Product owner
 - The product owner represents the product's stakeholders and the voice of the customer; and is accountable for ensuring that the team delivers value to the business. The product owner defines the product in customer-centric terms (typically user stories), adds them to the product backlog, and prioritizes them based on importance and dependencies.¹Scrum teams should have one product owner
 - is responsible for the delivery of the functionality at each iteration
- Scrum Team
 - The development team is responsible for delivering potentially shippable product increments every sprint (the sprint goal).
 - The team has from three to nine members who carry out all tasks required to build the product increments (analysis, design, development, testing, technical writing, etc.)

Product Backlog

- This is a repository where requirements are tracked with details on the no of requirements to be completed for each release. It should be maintained and prioritized by Product Owner, and it should be distributed to the scrum team. Team can also request for a new requirement addition or modification or deletion



- The Sprint Review is equivalent to a user acceptance test.
- Sprint Retrospective is equivalent to a project post-mortem.
- The Sprint Review is focused on the "product" and maximizing the business value of the results of the work of the previous sprint and the Sprint Retrospective is focused on the process and continuous process improvement.



Scrum

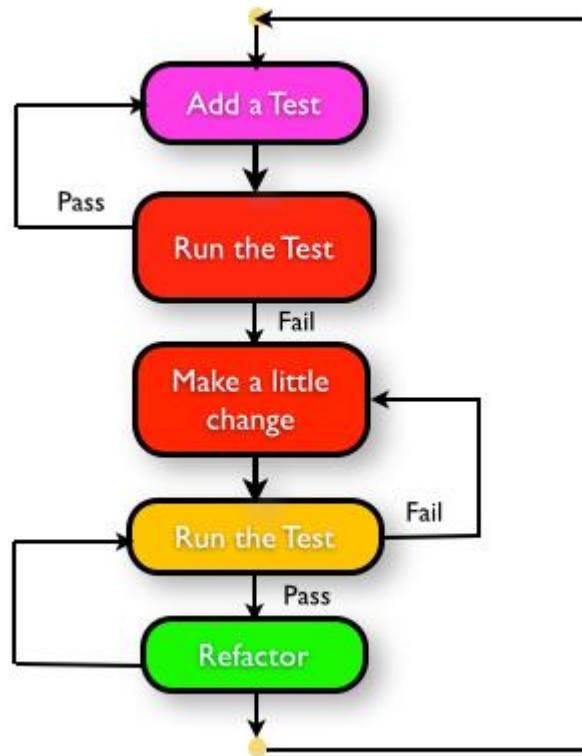
Process flow of Scrum Methodologies:

- Each iteration of a scrum is known as Sprint
- Product backlog is a list where all details are entered to get end product
- During each Sprint, top items of Product backlog are selected and turned into Sprint backlog
- Team works on the defined sprint backlog
- Team checks for the daily work
- At the end of the sprint, team delivers product functionality

Agile Testing: Test-Driven Development (TDD)

- Test-driven development starts with developing test for each one of the features. The test might fail as the tests are developed even before the development. Development team then develops and refactors the code to pass the test.
- Test-driven development is related to the test-first programming evolved as part of extreme programming concepts.
- **Test-Driven Development Process:**
 - Add a Test
 - Run all tests and see if the new one fails
 - Write some code
 - Run tests and Refactor code
 - Repeat

Test-Driven Development

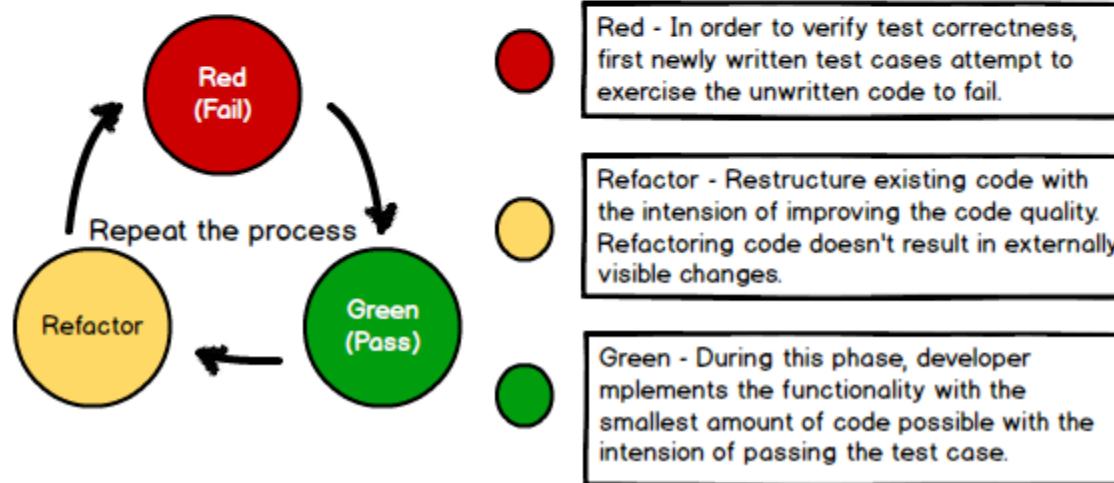


Test-Driven Development

Test Driven Development (TDD)

An Agile Development Approach for

TDD is a quality-focussed, extreme programming practice in which test cases are built prior to feature



Agile Testing Life Cycle

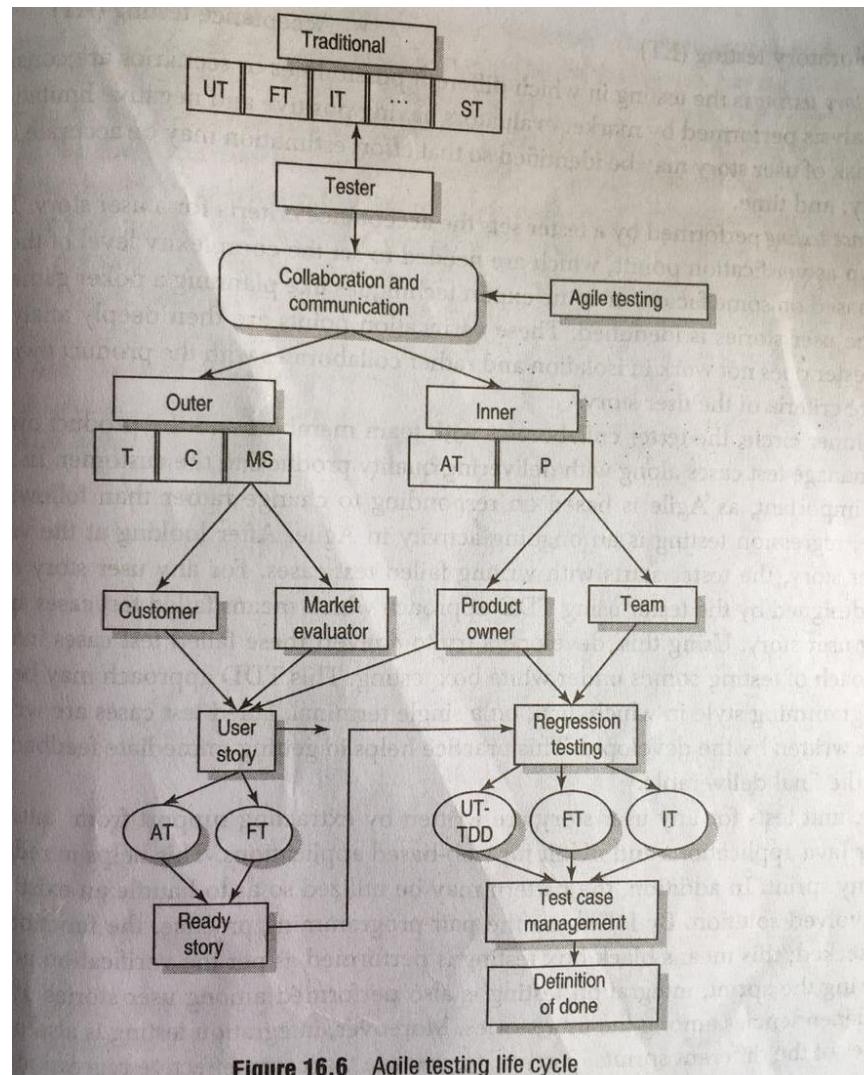
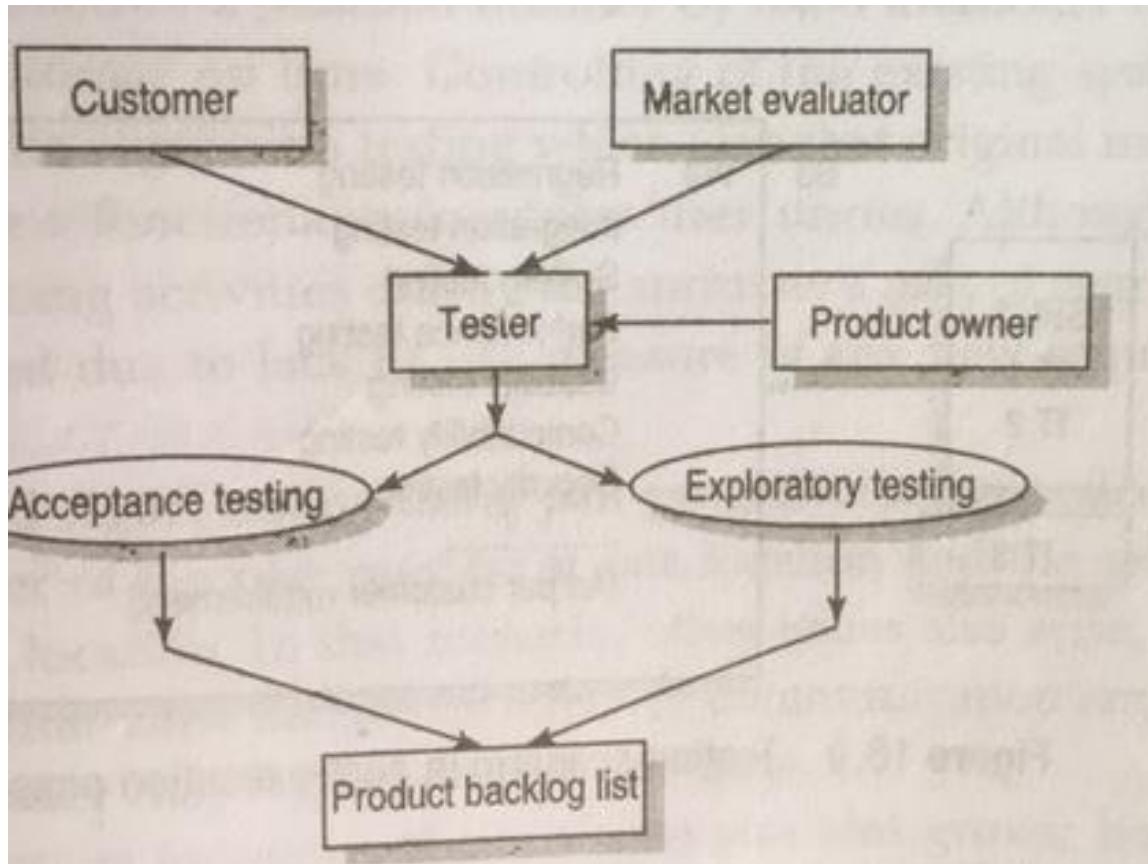


Figure 16.6 Agile testing life cycle

Testing in Scrum Phases



Pre- Execution Phase

Testing in Scrum Phases

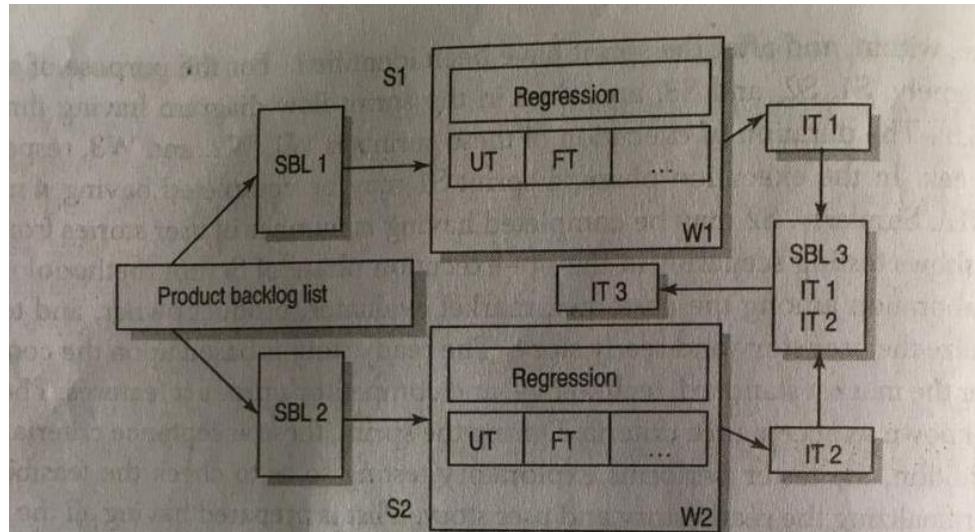


Figure 16.8 Testing scenario in execution phase

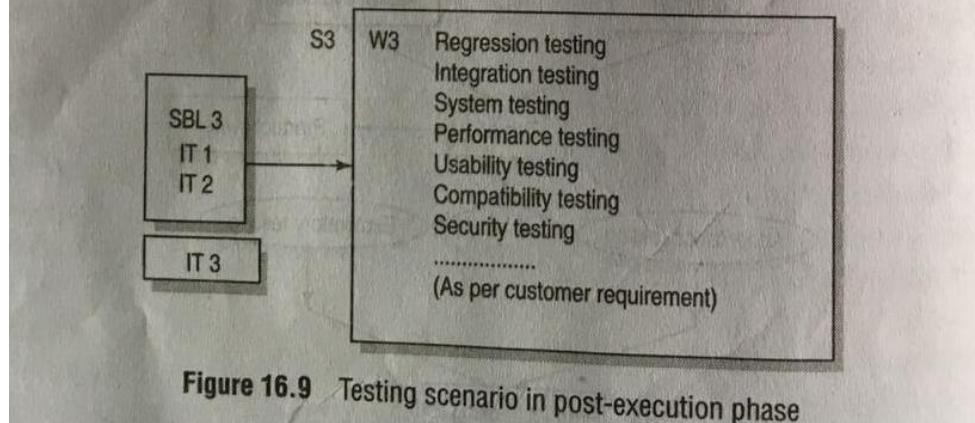


Figure 16.9 Testing scenario in post-execution phase

Execution and post - Execution Phases

Test Automation

Module 4

Automation and Testing Tools

- Need of automating the testing activities
- Static and Dynamic testing tools
- Selection of a testing tool
- Costs incurred in adopting a testing tool
- Guidelines of testing automation

Automation and Testing Tools

- Test automation is the use of special software (separate from the software being tested) to control the execution of tests and the comparison of actual outcomes with predicted outcomes.
- Test automation can automate some repetitive but necessary tasks in a formalized testing process already in place,
- or perform additional testing that would be difficult to do manually.
- The automation software can also enter test data into the System under Test , compare expected and actual results and generate detailed test reports.
- Goal of Automation is to reduce number of test cases to be run manually and not eliminate manual testing all together.
- Successive development cycles will require execution of same test suite repeatedly . Using a test automation tool it's possible to record this test suite and re-play it as required.

Need of Automation

- **Reduction of testing Effort**
- **Reduces the testers' involvement in executing tests**
- **Facilitates Regression Testing**
- **Avoids Human mistakes**
- **Reduces overall cost of the software**
- **Simulated testing**
- **Internal Testing**
- **Test Enablers**
- **Test case Design**

Categorization of Testing Tools

Static Testing Tools

Dynamic Testing Tools

Static Testing Tools

Static Program Analyzers which scan the source code and detect possible faults and anomalies.

- **Control Flow Analysis**
- **Data use Analysis**
- **Interface Analysis**
- **Path Analysis**

Dynamic Testing Tools

Dynamic Testing Tools supports the dynamic testing activities

Program Monitors:

- List the number of times a component is called or line of code is executed. This information is used by testers about the statement or path coverage of their test cases.
- Report on whether a decision point has branched in all directions, thereby providing information about branch coverage.
- Report summary statistics providing 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

Tools for Review and Inspections

- Complexity Analysis Tools
- Code Comprehension

Tools for Test Planning

- Templates for test plan documentation
- Test schedule and staffing estimates
- Complexity analyzer

Tools for Test Design and Development

- Test data generator
- Test case generator

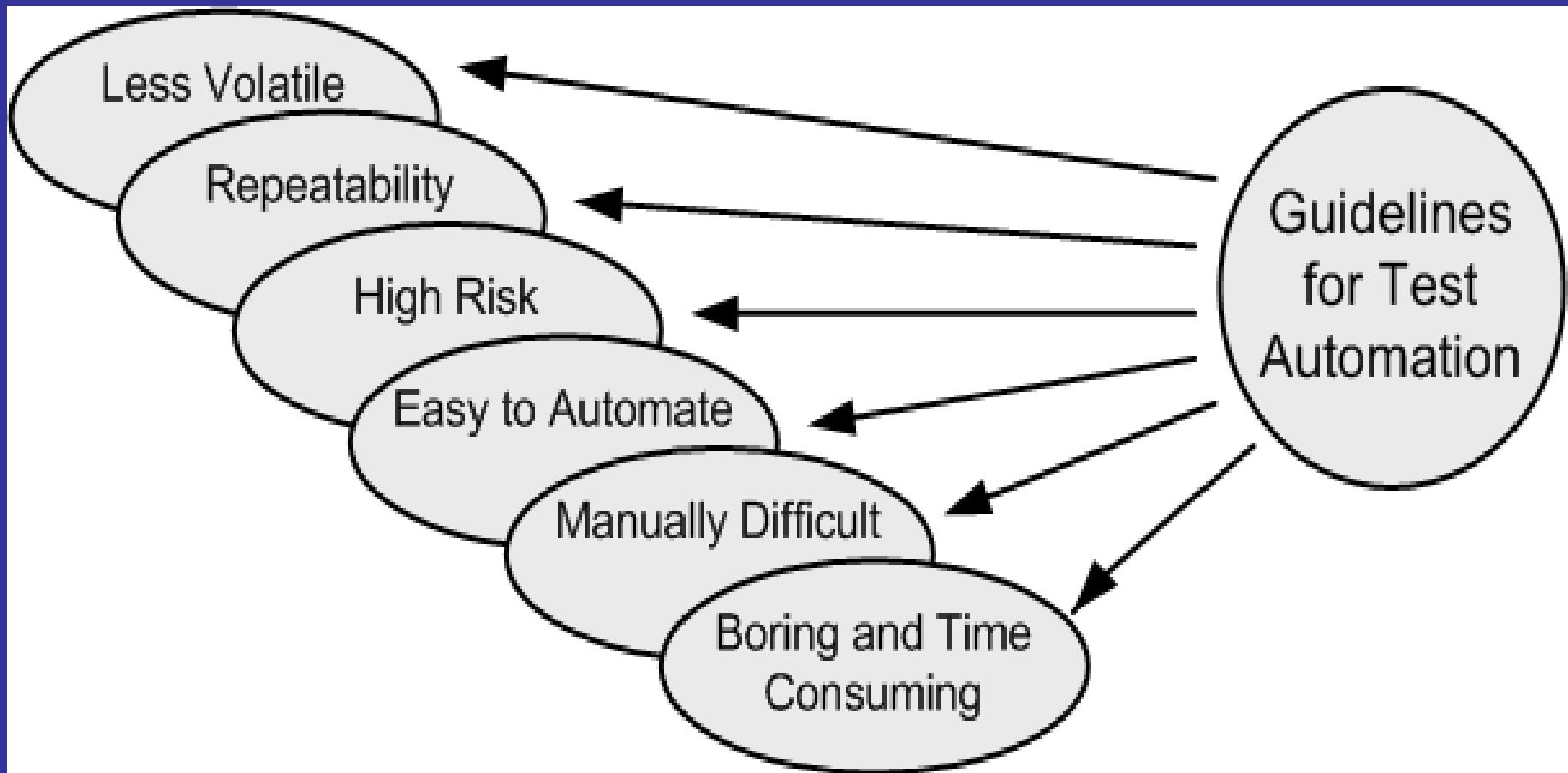
Test Execution and Evaluation Tools

- Capture/Playback Tools
- Coverage Analysis Tools
- Memory Testing Tools
- Test management Tools
- Network-testing Tools
- Performance testing Tools

Selection of Testing Tools

- **Match the tool to its appropriate use**
- **Select the tool to its appropriate SDLC phase**
- **Select the tool to the skill of the tester**
- **Select a tool which is affordable**
- **Determine how many tools are required for testing the system**
- **Select the tool after having the schedule of testing**

Test Selection Guidelines for Automation



Test selection guideline for test automation

Costs incurred in Testing Tools

- **Automated Script Development**
- **Training is required**
- **Configuration Management**
- **Learning Curve for the Tools**
- **Testing Tools can be Intrusive**
- **Multiple Tools are required**

Guidelines for Automated Testing

- Consider building a tool instead of buying one if possible.
- Test the tool on an application prototype
- Not all the tests should be automated
- Focus on the needs of the Organization and know the resources (budget, schedule) before choosing the automation tool.
- Use proven test-script development techniques
- Automate the regression tests whenever feasible.

Test Automation Framework

Framework is a constructive blend of various guidelines, coding standards, concepts, processes, practices, project hierarchies, modularity, reporting mechanism, test data injections etc. to pillar automation testing. Thus, user can follow these guidelines while automating application to take advantages of various productive results.

Advantage of Test Automation framework:

- Reusability of code
- Maximum coverage
- Recovery scenario
- Low cost maintenance
- Minimal manual intervention
- Easy Reporting

Test Automation Framework

Types of Test Automation Framework

- Module Based Testing Framework
- Data Driven Testing Framework
- Keyword Driven Testing Framework
- Hybrid Testing Framework

Example of keywords

Keywords	Description
Login	Login to demo site
Emails	Send Email
logouts	Log out from demo site
Notifications	Find unread notifications

Some Commercial Testing Tools

- **Mercury Interactive's WinRunner**
- **Segue Software's SilkTest**
- **IBM rational SQA Robot**
- **Mercury Interactive's LoadRunner**
- **Apache's JMeter**
- **Mercury Interactive's TestDirector**

Reference:

Software Testing Principles and Practices,
Naresh Chauhan, Second edition, Oxford
Higher Education

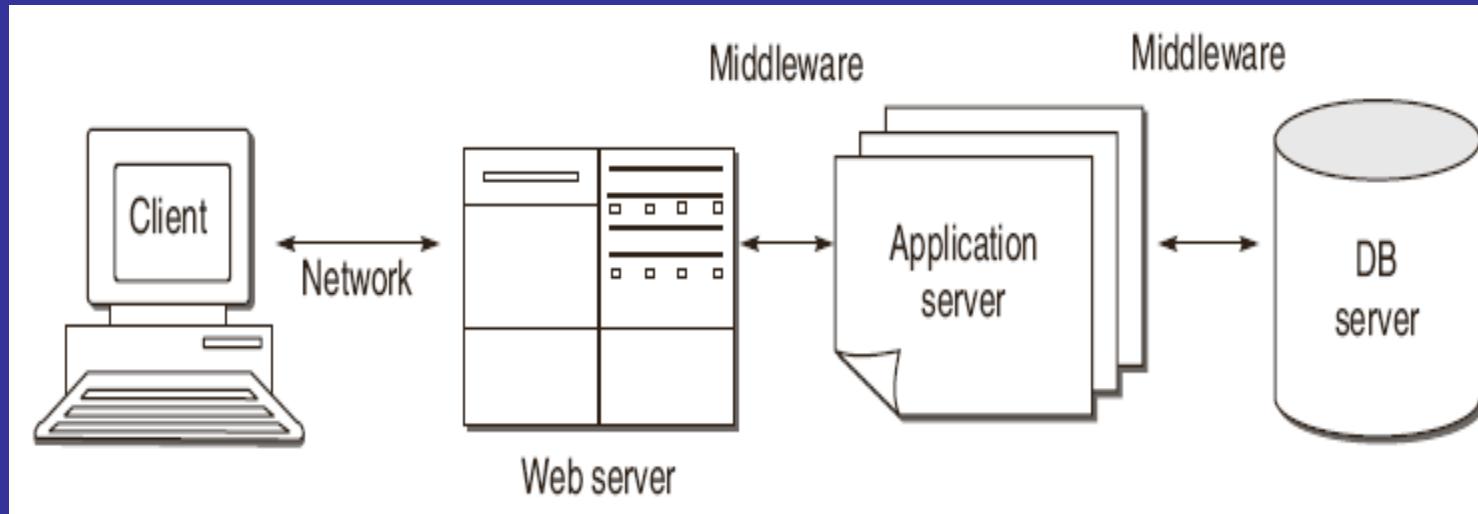
Testing Web based Systems

Module 4

Testing Web based Systems

Web application testing, a software testing technique exclusively adopted to test the applications that are hosted on web in which the application interfaces and other functionalities are tested.

Web Technology Evolution



Middleware are packages provided by software vendors to handle communications , data translation and process distribution.

Challenges in Testing for Web-based Software

- **Diversity and Complexity**
- **Dynamic Environment**
client side programs and contents may be generated dynamically
- **Very short development time**
- **Continuous evolution**
- **Compatibility & Interoperability**

Quality Aspects

- **Reliability**
- **Performance**
- **Security :**
 - Security of Infrastructure hosting the web application
 - Vulnerability is a cyber-security term that refers to a flaw in a system that can leave it open to attack. A vulnerability may also refer to any type of weakness in a computer system itself, in a set of procedures, or in anything that leaves information security exposed to a threat.
- **Usability**
- **Scalability**
- **Availability**
- **Maintainability**

Navigation Testing

Navigation testing is performed on various possible paths in the web application. Design the test cases such that the following navigations are correctly executing.

- 1.Internal Links**
- 2.External Links**
- 3.Redirected Links**
- 4.Navigation for searching inside the web application**

The errors must be checked during navigation testing for the following:

- The links should not be broken due to any reasons.**
- The redirected links should be with proper messages displayed to the user.**
- Check that all possible navigation paths are active.**
- Check that all possible navigation paths are relevant.**
- Check the navigations for the Back and Forward buttons, whether these are properly working if allowed.**

Security Testing

Security Test Plan

Securing testing can be planned into two categories:

Testing the security of the infrastructure hosting the Web application and testing for vulnerabilities of the web application.

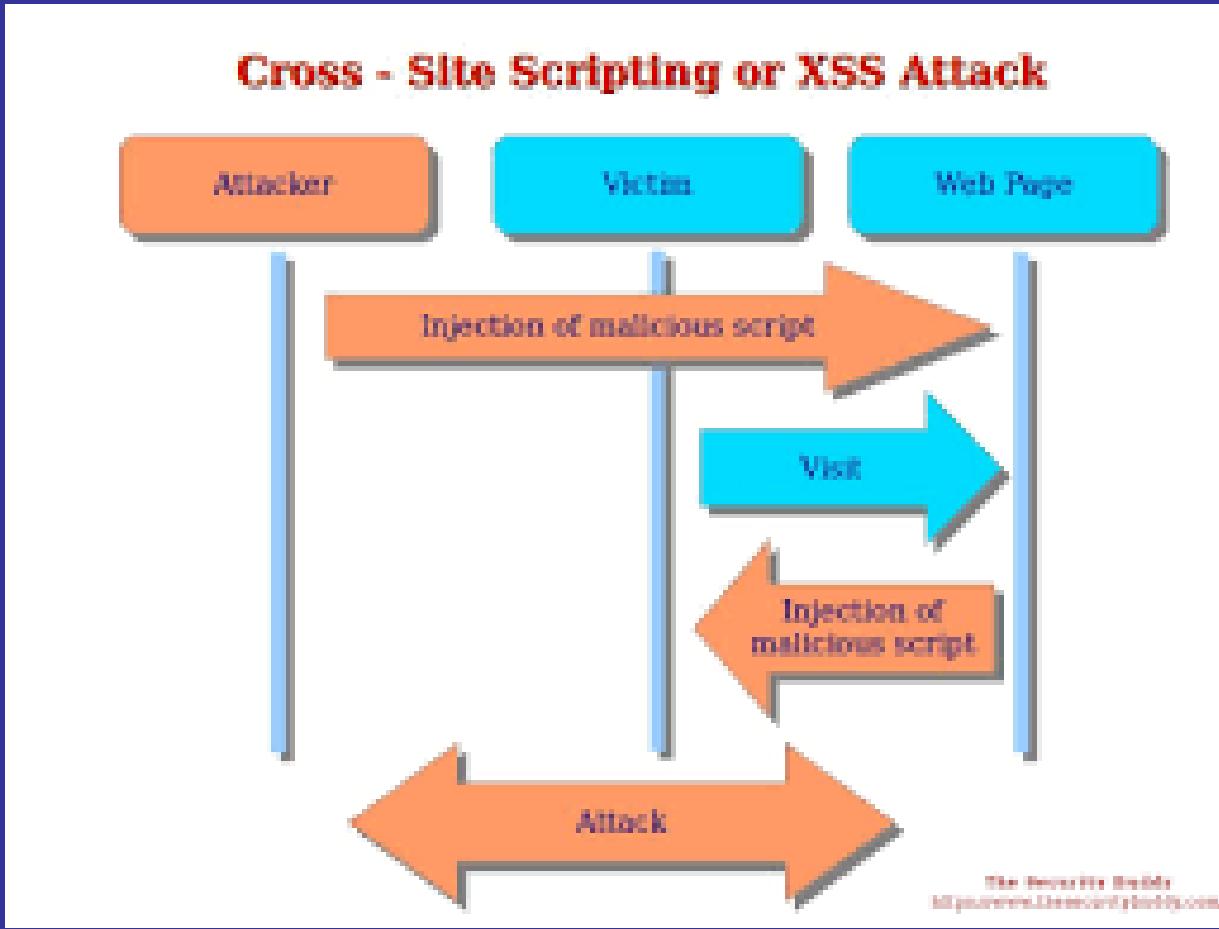
- Firewalls and port scans can be the solution for security of infrastructure.
- For vulnerabilities, user authentication, restricted and encrypted use of cookies, data communicated must be planned. Moreover, users should not be able to browse through the directories in the server.

Security Testing

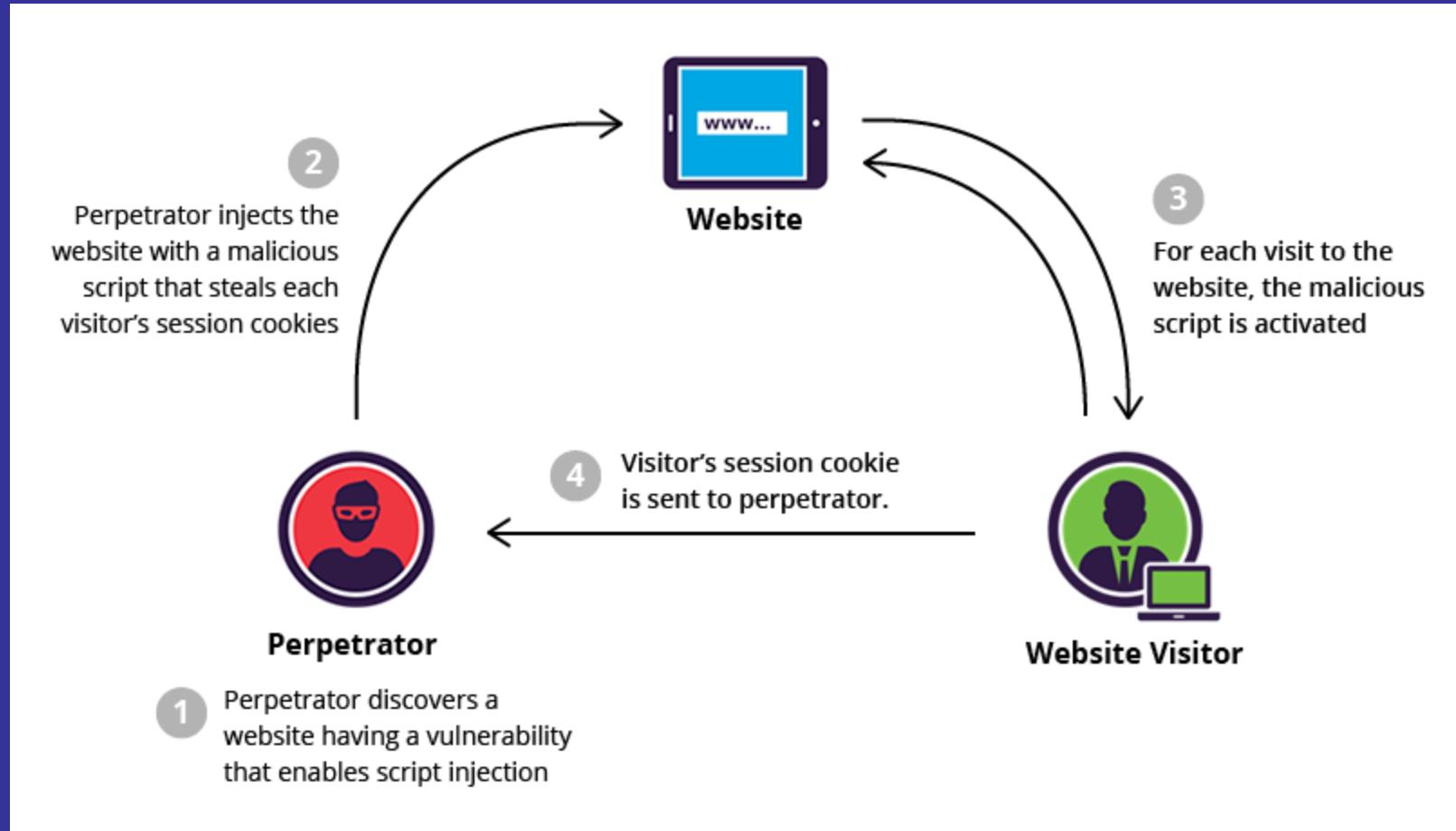
Various Threat Types and their corresponding Test cases

- **Unauthorized User / Fake Identity / Password Cracking**
- **Cross-site scripting (XSS):**
- **Buffer overflows**
- **URL Manipulation:** The attacker may change some information in query string passed from GET request so that he may get some information or corrupt the data.
- **SQL Injection :** Hackers can put SQL statements through the web application user interface into some queries meant for querying the database . In this way he/she can get vital information from the server database.
- **Denial of Service**

Cross-site scripting



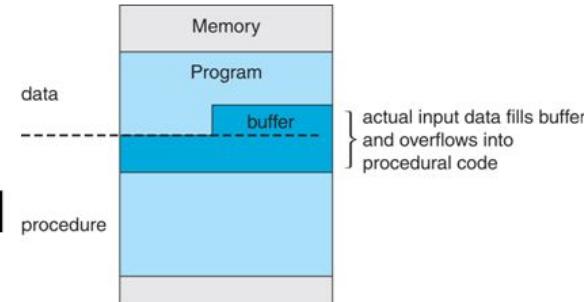
Cross-site scripting



Buffer overflows

Buffer Overflow

- Buffer: a memory allocation intended to hold input or output
- Buffer overflow: input is too large; excess input overflows into subsequent memory
- If the overflow area contains instructions, the attacker can effectively take control of the application by rewriting critical parts



© 2010 Jones and Bartlett Publishers, LLC (www.jbpub.com)

Buffer overflows

**Instruction Buffer
with out attack**

Instruction-1
Instruction-2
Instruction-3
Instruction-4
Instruction-5

glibc - getaddrinfo Stack-Based Buffer Overflow

*Call to Local
application*

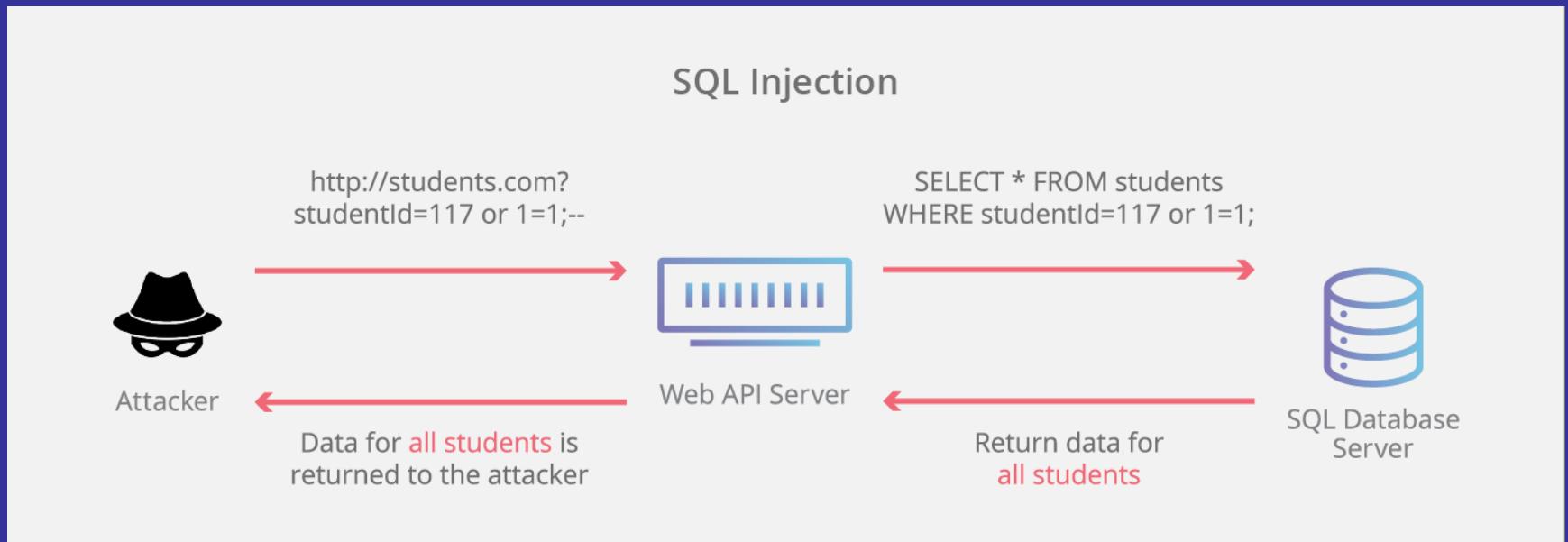
**Instruction Buffer
after attack**

Instruction-1
Instruction-2
Instruction-3
New instruction
Instruction-5

*Call to Attacker's
application*

www.hackthesec.co.in

SQL Injection



Performance Testing

Performance Parameters

- **Resource utilization**
- **Throughput**
- **Response time**
- **Database load**
- **Scalability**
- **Round trip time** (connection time and processing time)

Performance Testing

- Load testing

Many users, large I/P data from users, simultaneous connection to DB ,heavy load on some specific pages etc

- Capacity testing
- Scalability Testing

- Stress testing : Stretching the system beyond its specification limits.

Stressful conditions :

Limited memory
Insufficient disk space
Server failure

Reference:

Software Testing Principles and Practices,
Naresh Chauhan, Second edition, Oxford
Higher Education

Mobile Application Testing

Module 4

Mobile application testing

Mobile application testing is a process by which application software developed for handheld mobile devices is tested for its functionality, usability and consistency. Mobile application testing can be an automated or manual type of testing. Mobile applications either come pre-installed or can be installed from mobile software distribution platforms.

Types of Mobile Testing

There are broadly 2 kinds of testing that take place on mobile devices:

#1. Hardware testing:

- The device including the internal processors, internal hardware, screen sizes, resolution, space or memory, camera, radio, Bluetooth, WIFI etc. This is sometimes referred to as, simple "Mobile Testing".

#2. Software or Application testing:

- The applications that work on mobile devices and their functionality are tested. It is called the "Mobile Application Testing" to differentiate it from the earlier method. Even in the mobile applications, there are few basic differences that are important to understand:

a) Native apps: A native application is created for use on a platform like mobile and tablets.

b) Mobile web apps are server-side apps to access website/s on mobile using different browsers like chrome, Firefox by connecting to a mobile network or wireless network like WIFI.

c) Hybrid apps are combinations of native app and web app. They run on devices or offline and are written using web technologies like HTML5 and CSS.

Basic differences

- Native apps have single platform affinity while mobile web apps have cross platform affinity.
- Native apps are written in platforms like SDKs while Mobile web apps are written with web technologies like html, css, asp.net, java, php.
- For a native app, installation is required but for mobile web apps, no installation is required.
- Native app can be updated from play store or app store while mobile web apps are centralized updates.
- Many native app don't require Internet connection but for mobile web apps it's a must.
- Native app works faster when compared to mobile web apps.
- Native apps are installed from app stores like [Google play store](#) or [app store](#) where mobile web are websites and are only accessible through Internet.

Challenges

Testing applications on mobile devices is more challenging than testing web apps on desktop due to :

- **Wide varieties of mobile devices** like HTC, Samsung, Apple and Nokia.
- **Different range of mobile devices** with different screen sizes and hardware configurations like hard keypad, virtual keypad (touch screen) and trackball etc.
- **Different mobile operating systems** like Android, Symbian, Windows, Blackberry and IOS.
- **Different versions of operation system** like iOS 5.x, iOS 6.x, BB5.x, BB6.x etc.
- **Different mobile network operators** like GSM(Global System for Mobile Communication) and CDMA(Code Division Multiple Access).
- **Frequent updates** (like android- 4.2, 4.3, 4.4, iOS-5.x, 6.x) – with each update a new testing cycle is recommended to make sure no application functionality is impacted.

Basic Difference between Mobile and Desktop Application Testing

- On desktop, the application is tested on a central processing unit. On a mobile device, the application is tested on handsets like Samsung, Nokia, Apple and HTC.
- Mobile device screen size is smaller than desktop.
- Mobile devices have less memory than desktop.
- Mobiles use network connections like 2G, 3G, 4G or WIFI where desktop use broadband or dial up connections.
- The automation tool used for desktop application testing might not work on mobile applications.

Types of Mobile App Testing

- **Compatibility testing**— Testing of the application in different mobile devices, browsers, screen sizes and OS versions according to the requirements.
- **Interface testing**— Testing of menu options, buttons, bookmarks, history, settings, and navigation flow of the application.
- **Services testing**— Testing the services of the application online and offline.
- **Low level resource testing:** Testing of memory usage, auto deletion of temporary files, local database growing issues known as low level resource testing.

Types of Mobile App Testing

- **Performance testing**— Testing the performance of the application by changing the connection from 2G, 3G to WIFI, sharing the documents, battery consumption, etc.
- Testing the performance and behaviour of the application under certain conditions such as low battery, bad network coverage, low available memory, simultaneous access to application's server by several users and other conditions. Performance of an application can be affected from two sides: application's server side and client's side. Performance testing is carried out to check both.
- **Operational testing**— Testing of backups and recovery plan if battery goes down, or data loss while upgrading the application from store.
- **Security Testing**— Testing an application to validate if the information system protects data or not.

Types of Mobile App Testing

The fundamental objective of **security testing** is to ensure that the application's data and networking security requirements are met as per guidelines.

The following are the most crucial areas for checking the security of Mobile applications.

- To validate that the application is able to withstand any brute force attack which is an automated process of trial and error used to guess a person's username, password or credit-card number.
- To validate whether an application is not permitting an attacker to access sensitive content or functionality without proper authentication.
- To validate that the application has a strong password protection system and it does not permit an attacker to obtain, change or recover another user's password.
- To validate that the application does not suffer from insufficient session expiration.
- To identify the dynamic dependencies and take measures to prevent any attacker for accessing these vulnerabilities.
- To prevent from SQL injection related attacks.
- To identify and recover from any unmanaged code scenarios.
- To ensure whether the certificates are validated, does the application implement Certificate Pinning or not.
- To protect the application and the network from the denial of service attacks.
- To enable the session management for preventing unauthorized users to access unsolicited information.
- To check if any cryptography code is broken and ensure that it is repaired.
- To validate whether the business logic implementation is secured and not vulnerable to any attack from outside.
- To analyze file system interactions, determine any vulnerability and correct these problems.
- To validate the protocol handlers for example trying to reconfigure the default landing page for the application using a malicious iframe.
- To protect against malicious client side injections. And malicious runtime injections.
- To investigate file caching and prevent any malicious possibilities from the same.
- To prevent from insecure data storage in the keyboard cache of the applications.
- To investigate cookies and preventing any malicious deeds from the cookies.
- Investigate custom created files and preventing any malicious deeds from the custom created files.
- To prevent from buffer overflows and memory corruption cases.

Types of Mobile App Testing

- **Functional Testing:** Functional testing ensures that the application is working as per the requirements. Most of the test conducted for this is driven by the user interface and call flow
- **Laboratory Testing:** Laboratory testing, usually carried out by network carriers, is done by simulating the complete wireless network. This test is performed to find out any glitches when a mobile application uses voice and/or data connection to perform some functions.
- **Installation testing:** Validation of the application by installing /uninstalling it on the devices.
 - Certain mobile applications come pre-installed on the device whereas others have to be installed from the store. Installation testing verifies that the installation process goes smoothly without the user having to face any difficulty. This testing process covers installation, updating and uninstalling of an application

Types of Mobile App Testing

- **Memory Leakage Testing:** Memory leakage happens when a computer program or application is unable to manage the memory it is allocated resulting in poor performance of the application and the overall slowdown of the system. As mobile devices have significant constraints of available memory, memory leakage testing is crucial for the proper functioning of an application
- **Interrupt Testing:** An application while functioning may face several interruptions like incoming calls or network coverage outage and recovery. The different types of interruptions are:
 - Incoming and Outgoing [SMS](#) and [MMS](#)
 - Incoming and Outgoing calls
 - Incoming Notifications
 - Battery Removal
 - Cable Insertion and Removal for data transfer
 - Network outage and recovery
 - Media Player on/off
 - Device Power cycle
 - An application should be able to handle these interruptions by going into a suspended state and resuming afterwards.

Types of Mobile App Testing

Usability testing: To make sure that the mobile app is easy to use and provides a satisfactory user experience to the customers.

Usability testing is carried out to verify if the application is achieving its goals and getting a favourable response from users. This is important as the usability of an application is its key to commercial success (it is nothing but user friendliness). Another important part of usability testing is to make sure that the user experience is uniform across all devices. This section of testing hopes to address the key challenges of the variety of mobile devices and the diversity in mobile platforms/OS, which is also called device fragmentation. One key portion of this type of usability testing is to be sure that there are no major errors in the functionality, placement, or sizing of the user interface on different devices.

Types of Mobile App Testing

- **Certification Testing:** To get a certificate of compliance, each mobile device needs to be tested against the guidelines set by different mobile platforms.
- The Certified Mobile Application Tester (CMAT) certification exam is offered by the Global Association for Quality Management (GAQM) via Pearson Vue Testing Center worldwide to benefit the Mobile Application Testing Community.
- **Location Testing:** Connectivity changes with network and location, but you can't mimic those fluctuating conditions in a lab. Only in Country non automated testers can perform comprehensive usability and functionality testing.
- **Outdated Software Testing:** Not everyone regularly updates their operating system. Some Android users might not even have access to the newest version. Professional Testers can test outdated software.
- **Load Testing:** When many users all attempt to download, load, and use your app or game simultaneously, slow load times or crashes can occur causing many customers to abandon your app, game, or website. In-country human testing done manually is the most effective way to test load.

Types of Mobile App Testing

- **Black box Testing:** This type of testing doesn't include the internally coding logic of the application. Tester tests the application with functionality without peering with internally structure of the application. This method of test can be applied virtually to every level of software testing: unit, integration, system and acceptance.
- **CrowdSourced Testing:** In recent years, crowdsourced testing has become popular as companies can test mobile applications faster and cheaper using a global community of testers. Due to growing diversity of devices and operating systems as well as localization needs, it is difficult to comprehensively test mobile applications with small in-house testing teams. A global community of testers provides easy access to different devices and platforms. A globally distributed team can also test it in multiple locations and under different network conditions. Finally, localization issues can be tested by hiring testers in required geographies. Since real users using real devices test the application, it is more likely to find issues faced by users under real world conditions.

Mobile Application Testing Strategy

The Test strategy should make sure that all the quality and performance guidelines are met. A few pointers in this area:

1) Selection of the devices – Analyze the market and choose the devices that are widely used. (This decision mostly relies on the clients. The client or the app builders consider the popularity factor of a certain devices as well as the marketing needs for the application to decide what handsets to use for testing.)

2) Emulators – The use of these is extremely useful in the initial stages of development, as they allow quick and efficient checking of the app. Emulator is a system that runs software from one environment to another environment without changing the software itself. It duplicates the features and work on real system.

Types of Mobile Emulators

- Device Emulator- provided by device manufacturers
- Browser Emulator- simulates mobile browser environments.
- Operating systems Emulator- Apple provides emulators for iPhones, Microsoft for Windows phones and Google Android phones

List of few free and easy to use mobile device emulators

- Mobile Phone Emulator – Used to test handsets like iPhone, blackberry, HTC, Samsung etc.
- MobiReady – With this, not only can we test the web app, we can also check the code.
- Responsivepx – It checks the responses of the web pages, appearances and functionality of the websites.
- Screenfly – It is a customizable tool and used to test websites under different categories.

Mobile Application Testing Strategy

- 3) After a satisfactory level of development is complete for the mobile app, you could move to test on the physical devices for a more real life scenarios based testing.
- 4) **Consider cloud computing based testing:** Cloud computing is basically running devices on multiple systems or networks via Internet where applications can be tested, updated and managed. For testing purposes, it creates the web based mobile environment on a simulator to access the mobile app.

Mobile Application Testing Strategy

5) Automation vs. Manual testing

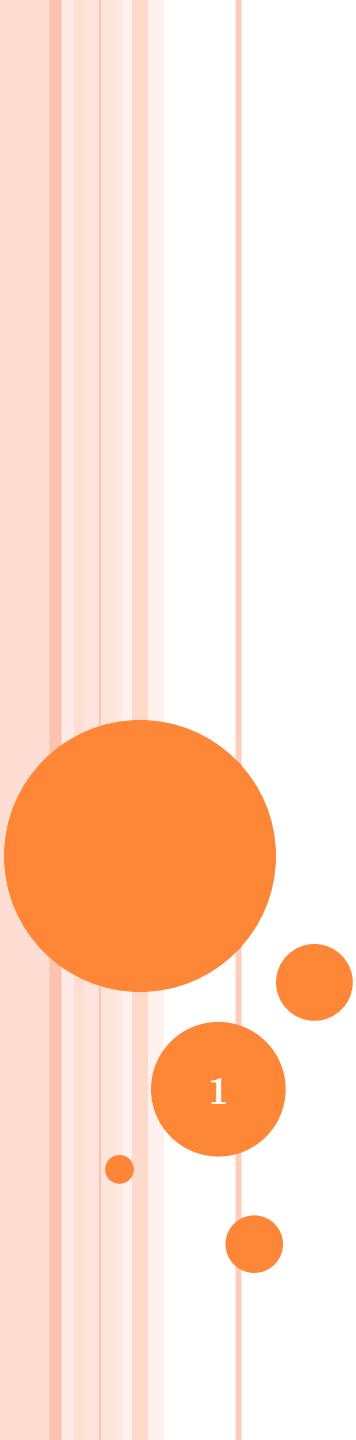
- If the application contains new functionality, test it manually.
- If the application requires testing once or twice, do it manually.
- Automate the scripts for regression test cases. If regression tests are repeated, automated testing is perfect for that.
- Automate the scripts for complex scenarios which are time consuming if executed manually.

6) **Network configuration** is also necessary part of mobile testing. It's important to validate the application on different networks like 2G, 3G, 4G or WIFI.

Sample Test Cases for Testing a Mobile App

In addition to functionality based test cases, Mobile application testing requires **special test cases** which should cover following scenarios.

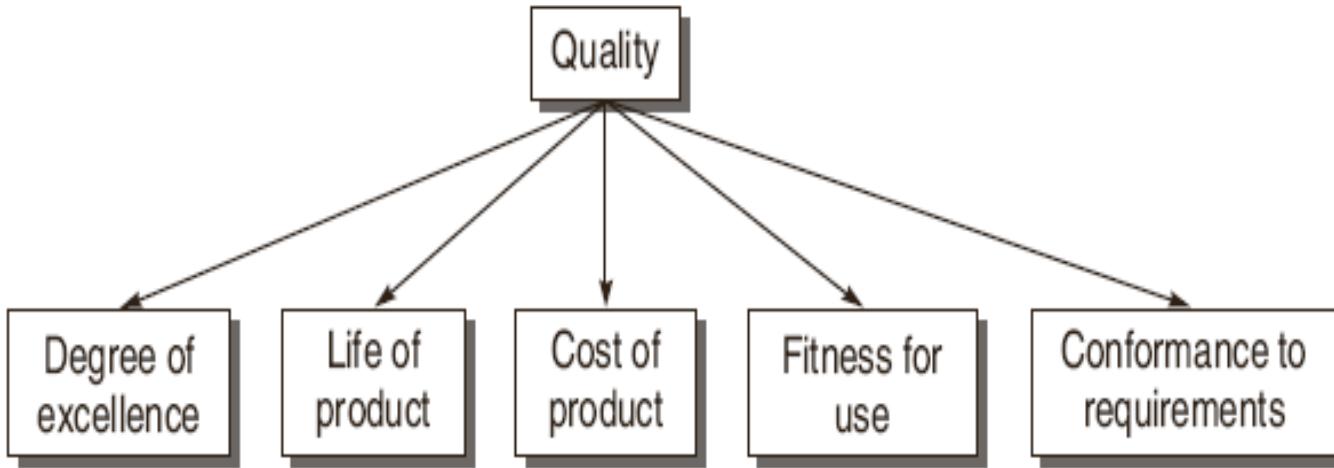
- **Battery usage**— It's important to keep a track of battery consumption while running application on the mobile devices.
- **Speed of the application**- the response time on different devices, with different memory parameters, with different network types etc.
- **Data requirements** – For installation as well as to verify if the user with limited data plan will able to download it.
- **Memory requirement**— again, to download, install and run
- **Functionality of the application**— make sure application is not crashing due to network failure or anything else.



SOFTWARE QUALITY MANAGEMENT

Module 5

Quality as Multidimensional Concept



The degree to which a product or service possesses a desired combination of attributes

WHAT IS QUALITY?

- “an inherent or distinguishing characteristic; a property; having a high degree of excellence”
- Features & functionality
 - “fitness for use”
 - “conformance to requirements”

Broadening the Concept of Quality

- Product Quality
- Process Quality

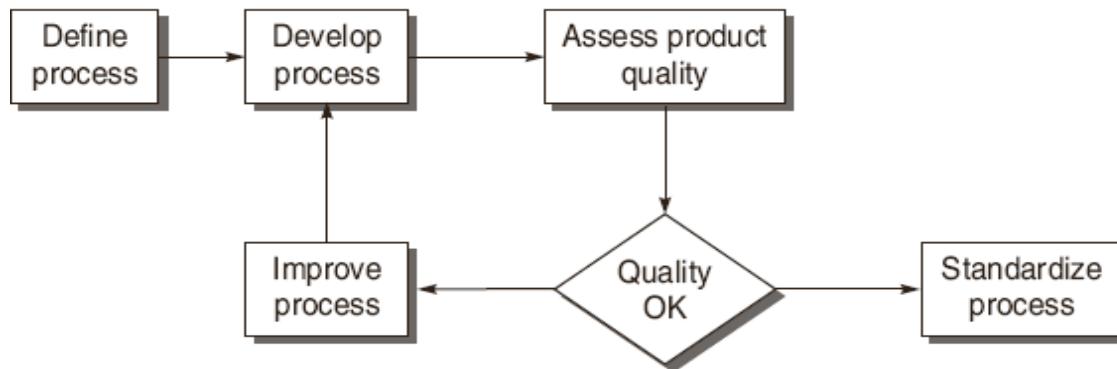


Figure 13.2 Process affects quality

FIVE VIEWS OF SOFTWARE QUALITY

- Transcendental view
- User view
- Manufacturing view
- Product view
- Value-based view

FIVE VIEWS OF SOFTWARE QUALITY

○ Transcendental view

- Quality is something that can be recognized through experience, but not defined in some tractable form.
- A good quality object stands out, and it is easily recognized.
- It is "something toward which **we strive as an ideal**, but may never implement completely"
- It's mainly feelings about something.

○ User view

- Quality concerns the extent to which a product meets user needs and expectations.
- Is a product fit for use?
- This view is highly personalized.
 - A product is of good quality if it satisfies a large number of users.
 - It is useful to identify the product attributes which the users consider to be important.
- This view may encompass many subject elements, such as *usability*, *reliability*, and *efficiency*.

FIVE VIEWS OF SOFTWARE QUALITY

○ Manufacturing view

- This view has its genesis in the manufacturing industry – auto and electronics.
- Key idea: Does a product satisfy the requirements?
 - Any deviation from the requirements is seen as reducing the quality of the product.
- The concept of process plays a key role.
- Products are manufactured “right the first time” so that the cost is reduced
 - Development cost
 - Maintenance cost
- Conformance to requirements leads to uniformity in products.
- Product quality can be incrementally improved by improving the process.
 - The CMM and ISO 9001 models are based on the manufacturing view.

FIVE VIEWS OF SOFTWARE QUALITY

- Product view

- Hypothesis: If a product is manufactured with good internal properties, then it will have good external properties.
- One can explore the causal relationship between *internal properties* and *external qualities*.
- Example: *Modularity* enables *testability*.

- Value-based view

- This represents the merger of two concepts: *excellence* and *worth*.
- Quality is a measure of excellence, and value is a measure of worth.
- Central idea
 - How much a customer is willing to pay for a certain level of quality.
 - Quality is meaningless if a product does not make economic sense.
 - The value-based view makes a trade-off between cost and quality.

MCCALL'S QUALITY FACTORS AND CRITERIA

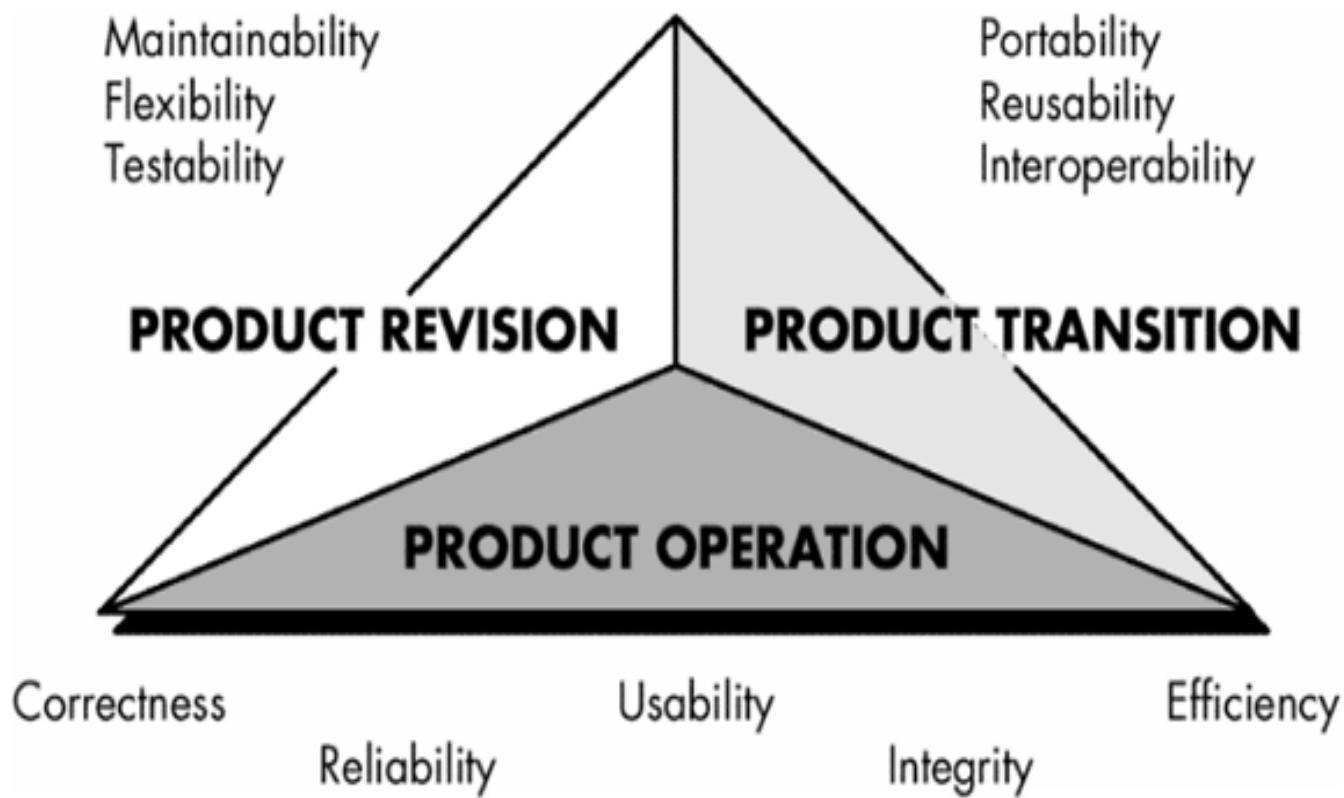
○ Quality Factors

- McCall, Richards, and Walters studied the concept of software quality in terms of two key concepts as follows:
 - *quality factors*, and
 - *quality criteria*.
- A quality factor represents the behavioral characteristic of a system.
 - Examples: correctness, reliability, efficiency, testability, portability,
...

Quality Factors	Definitions
Correctness	The extent to which a program satisfies its specifications and fulfills the user's mission objectives.
Reliability	The extent to which a program can be expected to perform its intended function with required precision.
Efficiency	The amount of computing resources and code required by a program to perform a function.
Integrity	The extent to which access to software or data by unauthorized persons can be controlled.
Usability	The effort required to learn, operate, prepare input, and interpret output of a program.
Maintainability	The effort required to locate and fix a defect in an operational program.
Testability	The effort required to test a program to ensure that it performs its intended functions.
Flexibility	The effort required to modify an operational program.
Portability	The effort required to transfer a program from one hardware and/ or software environment to another.
Reusability	The extent to which parts of a software system can be reused in other applications.
Interoperability	The effort required to couple one system with another.

Table : McCall's quality factors

MCCALL'S QUALITY FACTORS



MCCALL'S QUALITY FACTORS AND CRITERIA

Quality Categories	Quality Factors	Broad Objectives
Product Operation	Correctness Reliability Efficiency Integrity Usability	Does it do what the customer wants? Does it do it accurately all of the time? Does it quickly solve the intended problem? Is it secure? Can I run it?
Product Revision	Maintainability Testability Flexibility	Can it be fixed? Can it be tested? Can it be changed?
Product Transition	Portability Reusability Interoperability	Can it be used on another machine? Can parts of it be reused? Can it interface with another system?

MCCALL'S QUALITY FACTORS AND CRITERIA

○ Quality Criteria

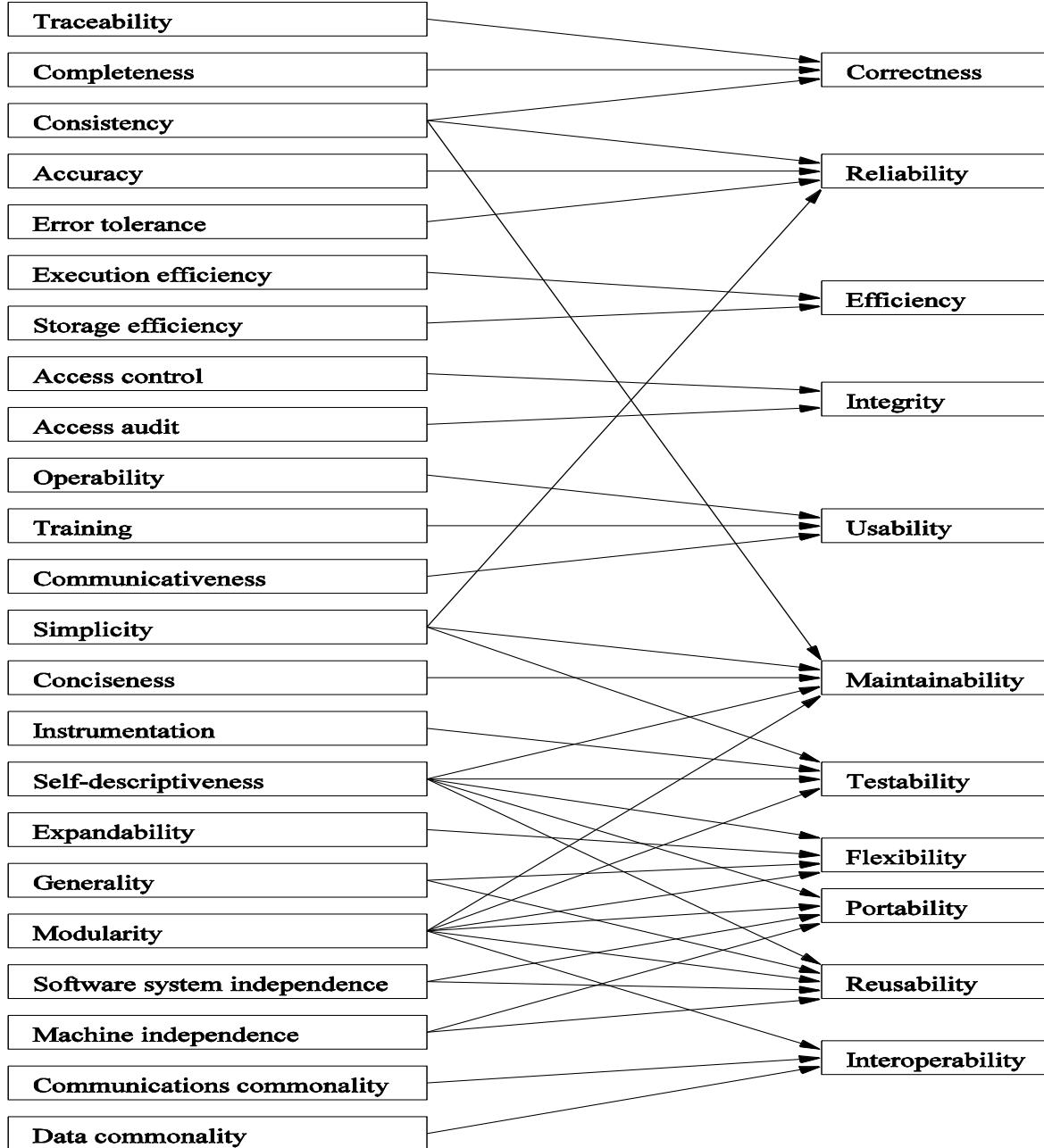
- A quality criterion is an attribute of a quality factor that is related to software development.
- Example:
 - Modularity is an attribute of the architecture of a software system.
 - A highly modular software allows designers to put cohesive components in one module, thereby increasing the maintainability of the system.
- In Table:- quality criteria have been listed.

Quality Criteria	Definitions of Quality Criteria
Access audit	The ease with which software and data can be checked for compliance with standards or other requirements.
Access control	The provisions for control and protection of the software and data.
Accuracy	The precisions of computations and output.
Communication commonality	The degree to which standard protocols and interfaces are used.
Completeness	The degree to which a full implementation of the required functionalities has been achieved.
Communicativeness	The ease with which inputs and outputs can be assimilated.
Conciseness	The compactness of the source code, in terms of lines of code.
Consistency	The use of uniform design and implementation techniques and notations throughout a project.
Data commonality	The use of standard data representations.
Error tolerance	The degree to which continuity of operation is ensured under adverse conditions.
Execution efficiency	The run-time efficiency of the software.
Expandability	The degree to which storage requirements or software functions can be expanded.
Generality	The breadth of the potential application of software components.
Hardware independence	The degree to which the software is dependent on the underlying hardware.
Instrumentation	The degree to which the software provides for measurements of its use or identification of errors.
Modularity	The provision of highly independent modules.
Operability	The ease of operation of the software.
Self-documentation	The provision of in-line documentation that explains the implementation of components.
Simplicity	The ease with which the software can be understood.
Software system independence	The degree to which the software is independent of its software environment – non-standard language constructs, operating system, libraries, database management system, etc.
Software efficiency	The run-time storage requirements of the software.
Traceability	The ability to link software components to requirements.
Training	The ease with which new users can use the system.

MCCALL'S QUALITY FACTORS AND CRITERIA

- Relationship Between Quality Factors and Quality Criteria
 - Each quality factor is positively influenced by a set of quality criteria, and the same quality criterion impacts a number of quality factors.
 - Example: Simplicity impacts reliability, usability, and testability.
 - If an effort is made to improve one quality factor, another quality factor may be degraded.
 - Portable code may be less efficient.
 - Some quality factors positively impact others.
 - An effort to improve the correctness of a system will increase its reliability.

Quality criteria



McCALL'S QUALITY FACTORS AND CRITERIA

Pooja Malhotra

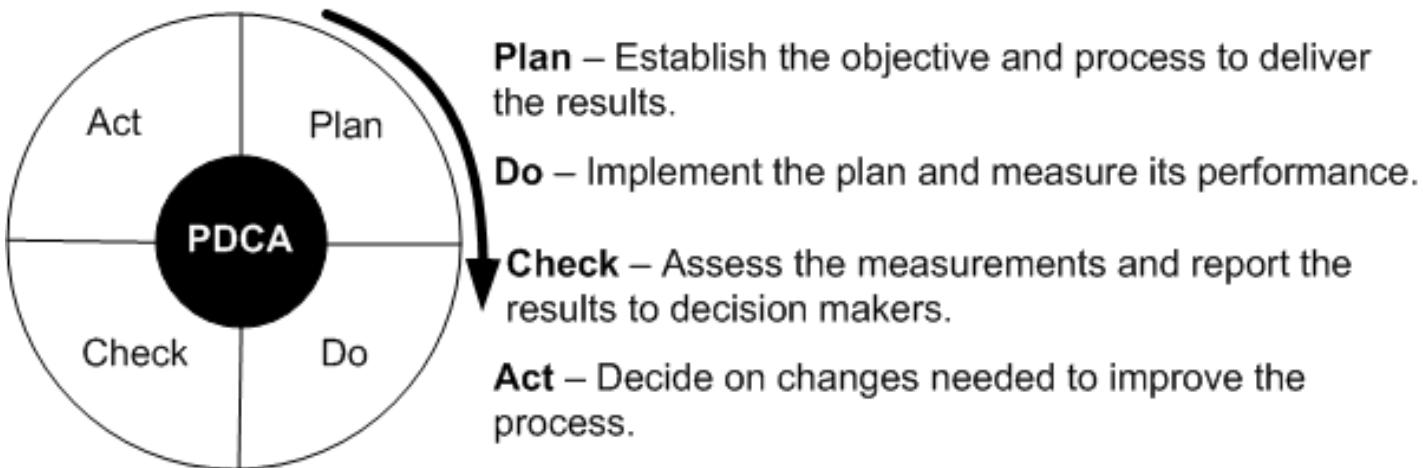
QUALITY MANAGEMENT

- Quality management is to
- plan suitable Quality control and Quality assurance activities,
 - define procedures and standards which should be used during software development and verifying that these are being followed by everyone
 - Properly execute and control activities.
 - Verifying and improving the software process.
-
- **Quality Management and Project Management**



THE QUALITY REVOLUTION - PDCA CYCLE

The Shewhart cycle

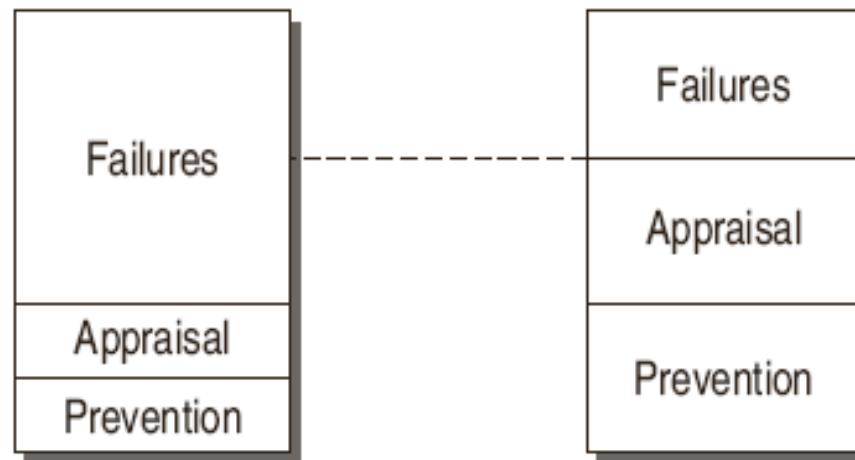


- Deming introduced Shewhart's PDCA cycle to Japanese researchers
- It illustrate the activity sequence:
 - Setting goals
 - Assigning them to measurable milestones
 - Assessing the progress against the milestones
 - Take action to improve the process in the next cycle

QUALITY COST & BENEFITS OF INVESTMENT ON QUALITY

Cost of quality is decomposed into three major categories:

1. Prevention Costs(Quality Planning, FTRs, Testing , training etc.)
2. Appraisal Costs(SMP, testing, inspection, equipment calibrations and maintenance)
3. Failure Costs(analyse and remove failures)
 - o Internal Failure Costs(developer's site)
 - o External Failure Costs(customer's site)



QUALITY CONTROL AND QUALITY ASSURANCE

Quality Control is basically related to software product such that there is minimum variation, according to the desired specifications. This variation is checked at each step of development. Quality control may include the following activities: Reviews, Testing using manual techniques or with automated tools (V & V).

Quality Assurance is largely related to the process. In addition, quality assurance activities are in management zone. Therefore auditing and reporting of quality based on quantitative measurements are also performed.



PROCEDURAL APPROACH TO QM

- SQA activities (Product evaluation and process monitoring)
 - Products- Standards and Process- Procedures
- SQA relationships with other assurance activities
 - Configuration Management monitoring
 - Verification and validation monitoring
 - Formal test monitoring
 - SQA during SDLC

SOFTWARE QUALITY ASSURANCE BEST PRACTICE

- **Continuous improvement:** All the standard process in SQA must be improved **frequently** and made **official** so that the other can follow. This process should be **certified** by popular organization such as ISO, CMMI... etc.
- **Documentation:** All the QA policies and methods, which are defined by QA team, should be documented for training and reuse for future projects.
- **Experience:** Choosing the members who are seasoned SQA auditors is a good way to ensure the quality of management review
- **Tool Usage:** Utilizing tool such as the tracking tool, management tool for SQA process reduces SQA effort and project cost.
- **Metrics:** Developing and creating metrics to track the software quality in its current state, as well as to compare the improvement with previous versions, will help increase the value and maturity of the Testing process
- **Responsibility:** The SQA process is not the SQA member's task, but **everyone's** task. Everybody in the team is responsible for quality of product, not just the test lead or manager.

QUANTITATIVE APPROACH TO QM

Major Issues

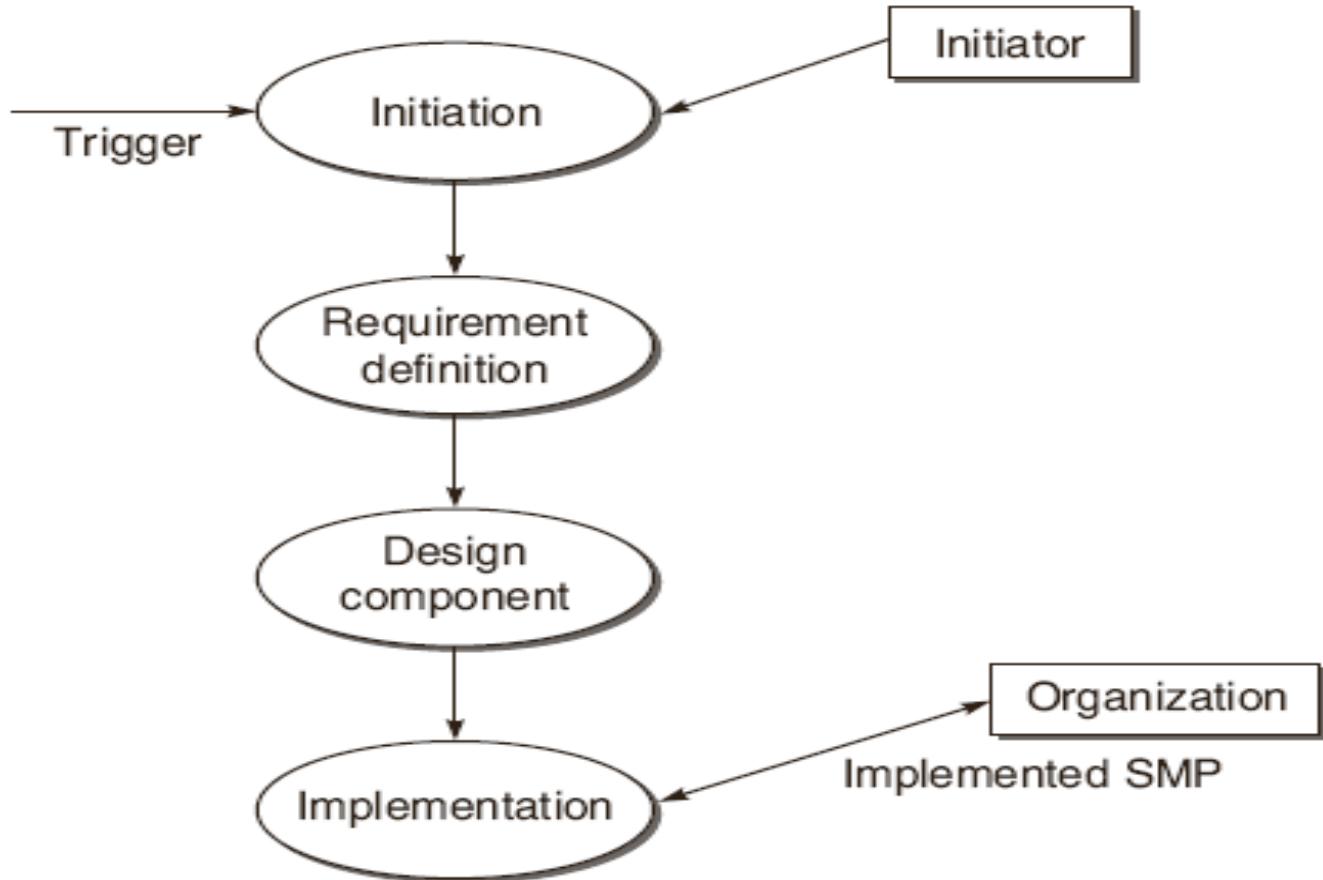
- Setting Quality Goal

Estimate for defects(P)current Project=

Defects(SP)X effort estimate(P)/Actual effort(SP)

- Managing software development process quantitatively- **Intermediate goals**

PAUL GOODMAN MODEL FOR SOFTWARE METRICS PROGRAM



SOFTWARE QUALITY METRICS

Product Quality Metrics

- **Mean time to failure (MTTF)**

MTTF metric is an estimate of the average or mean time until a product's first failure occurs.

Defect Density Metrics

- Defect Density = Number of Defects / Size of Product

Customer problem metrics

- problems per user month (PUM) = Total problems reported by the customer for a time period / Total number of licensed months of the software during the period.
- **Customer Satisfaction Metrics**



SOFTWARE QUALITY METRICS

In-Process Quality Metrics

- **Defect Density during testing**
- **Defect Arrival pattern during Testing**
- **Defect Removal Efficiency**
$$\text{DRE} = (\text{Defects removed during the month} / \text{No. of problems arrivals during the month}) \times 100$$



Software Quality Metrics

Metrics for Software maintenance

Fix backlog and backlog management index

BMI = (Number of problems closed during the month / Number of problem arrivals during the month) x 100 %

Fix response time and fix responsiveness

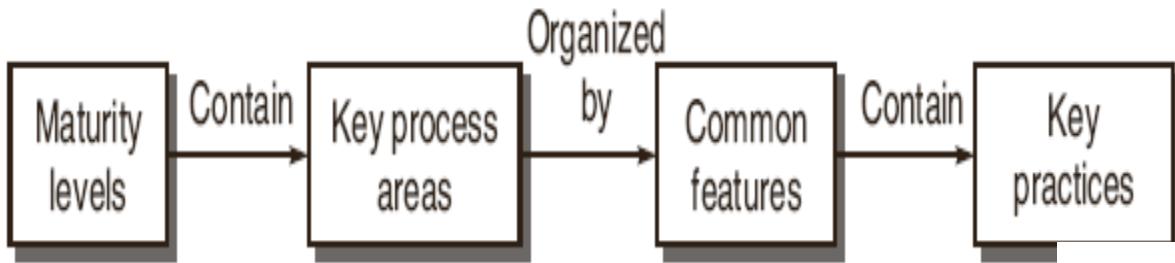
Percent Delinquent fixes

Percent Delinquent fixes = (Number of fixes that exceeded the response time criteria by severity level / Number of fixes delivered in a specified time) X 100%

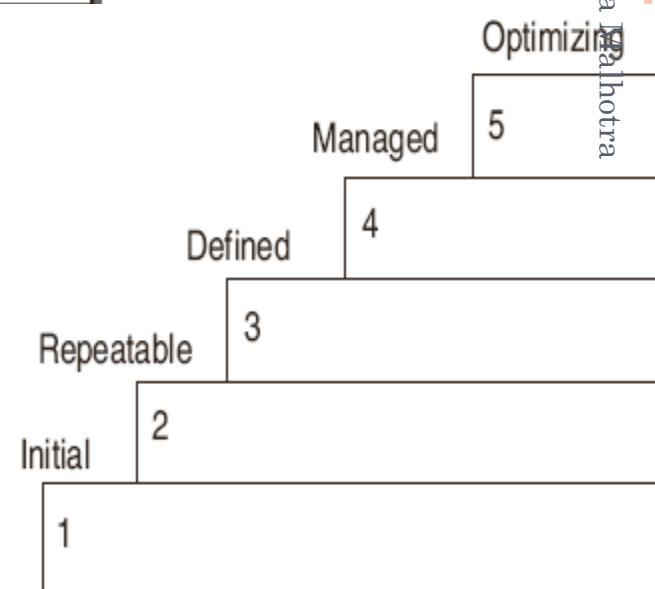
Fix Quality: Defective fix percentage



Capability Maturity Model (CMM)



Maturity levels	Indicate	Process capability
Key process areas	Achieve	Goals
Common features	Address	Implementation/Institutionalization
Key practices	Describe	Infrastructure/Activities



CAPABILITY MATURITY MODEL (CMM)

Table 13.2 KPIs for level 2

<i>KPIs, at this level, focus on the project's concerns related to establishing basic project management controls.</i>	
KPA	Goals
Requirement Management	Document the requirements properly. Manage the requirement changes properly.
Software Project Planning	Ensure proper project plan including estimation and listing of activities to be done.
Software Project Tracking and Oversight	Evaluate the actual performance of the project against the plans during project execution. Action, if there is deviation from plan.
Software Sub-contract Management	Selects qualified software sub-contractors. Maintain ongoing communications between prime contractor and the software sub-contractor. Track the software sub-contractor's actual results and performance against its commitments.
Software Quality Assurance	Plan the SQA activities. Ensure that there are proper processes by conducting review and audits. Take proper actions, if projects fail.
Software Configuration Management	Identify work products and documents to be controlled in the project. Control the changes in the software.

CAPABILITY MATURITY MODEL (CMM)

Table 13.3 KPIAs for level 3

<i>KPIAs, at this level, address both project and organizational issues, as the organization establishes an infrastructure that institutionalizes effective software engineering and management processes across all projects.</i>	
KPA	Goals
Organization Process Focus	Coordinate process development and improvement activities across the organization. Compare software processes with a process standard. Plan organization-level process development and improvement activities.
Organization Process Definition	Standard software processes are defined and documented.
Training Program	Identify training needs of various team members and implementation of training programs.
Integrated Software Management	Tailor the project from the standard defined process. Manage the project according to defined process.
Software Product Engineering	Define, integrate, and perform software engineering tasks. Keep the work products consistent especially if there are changes.
Inter-group Coordination	Ensure coordination between different groups.
Peer Reviews	Plan peer review activities. Identify and remove defects in the software work products.

Pooja Malhotra

CAPABILITY MATURITY MODEL (CMM)

Table 13.4 KPAs for level 4

KPAs, at this level, focus on establishing a quantitative understanding of both the software process and the software work products being built.

Pooja

Malhotra

KPA	Goals
Quantitative Process Management	Plan the quantitative process management activities. Control the performance of the project's defined software process quantitatively. The process capability of the organization's standard software process is known in quantitative terms.
Software Quality Management	Set and plan quantitative quality goals for the project. Measure the actual performance of the project with quality goals and compare them with plans.

CAPABILITY MATURITY MODEL (CMM)

Table 13.5 KPAs for level 5

KPAs, at this level, cover the issues that both the organization and the projects must address to implement continuous and measurable software process improvement.

KPA	Goals
Defect Prevention	Plan the defect prevention activities. Identify, prioritize, and eliminate the common causes of bugs.
Technology Change Management	Plan the technology changes to be incorporated in the project, if any. Evaluate the effect of new technologies on quality and productivity.
Process Change Management	Plan the process improvement activities such that an organization-wide participation is there. Measure the change of improvement in the process.

6-SIGMA

- Originated by Motorola in Schaumburg, IL
- Based on competitive pressures in 1980s – “Our quality stinks”

Sigma	Defects Per Million
1 δ	690,000
2 δ	308,537
3 δ	66,807
4 δ	6,210
5 δ	233
6 δ	3.4

6-SIGMA

3 δ	6 δ
Five short or long landings at any major airport	One short or long landing in 10 years at all airports in the US
Approximately 1,350 poorly performed surgical operations in one week	One incorrect surgical operation in 20 years
Over 40,500 newborn babies dropped by doctors or nurses each year	Three newborn babies dropped by doctors or nurses in 100 years
Drinking water unsafe to drink for about 2 hours each month	Water unsafe to drink for one second every six years

6-SIGMA D-M-A-I-C CYCLE

○ Define

- The first step is to define customer satisfaction goals and subgoals; for example, reduce cycle time, costs, or defects. These goals then provide a baseline or benchmark for the process improvement.

○ Measure

- The Six Sigma team is responsible for identifying a set of *relevant* metrics.

○ Analyze

- With data in hand, the team can analyze the data for trends, patterns, or relationships. Statistical analysis allows for testing hypotheses, modeling, or conducting experiments.

○ Improve

- Based on solid evidence, improvements can be proposed and implemented. The Measure-Analyze-Improve steps are generally iterative to achieve target levels of performance.

○ Control

- Once target levels of performance are achieved, control methods and tools are put into place in order to maintain performance.

6-SIGMA ROLES & RESPONSIBILITIES

- Master black belts
 - People within the organization who have the highest level of technical and organizational experience and expertise. Master black belts train black.
- Black belts
 - Should be technically competent and held in high esteem by their peers. They are actively involved in the Six Sigma change process.
- Green belts
 - Are Six Sigma team leaders or project managers. Black belts generally help green belts choose their projects, attend training with them, and then assist them with their projects once the project begins.
- Champions
 - Leaders who are committed to the success of the Six Sigma project and can ensure that barriers to the Six Sigma project are removed. Usually a high-level manager who can remove obstacles that may involve funding, support, bureaucracy, or other issues that black belts are unable to solve on their own.

SOFTWARE TOTAL QUALITY MANAGEMENT (STQM)

TQM is defined as a quality-centered, customer-focused, fact-based, team-driven, senior-management-led process to achieve an organization's strategic imperative through continuous process improvement.

Poonia Malhotra

T = Total = everyone in the organization

Q = Quality = customer satisfaction

M = Management = people and processes

SOFTWARE TOTAL QUALITY MANAGEMENT (STQM)

The elements of TQM / STQM

- Customer-focus/Customer-focus in software development
- Process / Process, technology, and development quality
- Human-side of quality/Human-side of software quality
- Measurement and analysis

ISO 9000 STANDARD

The ISO 9000 family of quality management systems standards is designed to help organizations ensure that they meet the needs of customers and other stakeholders while meeting statutory and regulatory requirements related to a product or program. ISO 9000 deals with the fundamentals of quality management systems, including the seven quality management principles upon which the family of standards is based. ISO 9001 deals with the requirements that organizations wishing to meet the standard must fulfill.

ISO 9000:2015 SOFTWARE QUALITY STANDARD AND FUNDAMENTALS

This International Standard provides the fundamental concepts, principles and vocabulary for quality management systems (QMS) and provides the foundation for other QMS standards. This International Standard is intended to help the user to understand the fundamental concepts, principles and vocabulary of quality management, in order to be able to effectively and efficiently implement a QMS and realize value from other QMS standards.

This International Standard proposes a well-defined QMS, based on a framework that integrates established fundamental concepts, principles, processes and resources related to quality, in order to help organizations realize their objectives. It is applicable to all organizations, regardless of size, complexity or business model. Its aim is to increase an organization's awareness of its duties and commitment in fulfilling the needs and expectations of its customers and interested parties, and in achieving satisfaction with its products and services.

ISO 9000:2015 SOFTWARE QUALITY STANDARD AND FUNDAMENTALS

This International Standard describes the fundamental concepts and principles of quality management which are universally applicable to the following:

- organizations seeking sustained success through the implementation of a quality management system;
- customers seeking confidence in an organization's ability to consistently provide products and services conforming to their requirements;
- organizations seeking confidence in their supply chain that product and service requirements will be met;
- organizations and interested parties seeking to improve communication through a common understanding of the vocabulary used in quality management;
- organizations performing conformity assessments against the requirements of ISO 9001;
- providers of training, assessment or advice in quality management;
- developers of related standards.

ISO 9000:2015 SOFTWARE QUALITY STANDARD AND FUNDAMENTALS

Essentially the layout of the standard is similar to the previous ISO standard in that it follows the Plan, Do, Check, Act cycle in a process based approach, but is now further encouraging this to have risk based thinking. The purpose of the quality objectives is to determine the conformity of the requirements (customers and organizations), facilitate effective deployment and improve the quality management system

ISO 9000:2015 SOFTWARE QUALITY STANDARD AND FUNDAMENTALS

- Risk-based thinking is essential for achieving an effective quality management system. The concept of risk-based thinking has been implicit in previous editions of this International Standard including, for example, carrying out preventive action to eliminate potential nonconformities, analysing any nonconformities that do occur, and taking action to prevent recurrence that is appropriate for the effects of the nonconformity.
- To conform to the requirements of this International Standard, an organization needs to plan and implement actions to address risks and opportunities. Addressing both risks and opportunities establishes a basis for increasing the effectiveness of the quality management system, achieving improved results and preventing negative effects.
- Opportunities can arise as a result of a situation favourable to achieving an intended result, for example, a set of circumstances that allow the organization to attract customers, develop new products and services, reduce waste or improve productivity. Actions to address opportunities can also include consideration of associated risks. Risk is the effect of uncertainty and any such uncertainty can have positive or negative effects. A positive deviation arising from a risk can provide an opportunity, but not all positive effects of risk result in opportunities.

ISO 9000:2015 SOFTWARE QUALITY STANDARD AND FUNDAMENTALS

The ISO 9000 series are based on seven quality management principles (QMP)

QMP 1 – Customer focus

QMP 2 – Leadership

QMP 3 – Engagement of people

QMP 4 – Process approach

QMP 5 – Improvement

QMP 6 – Evidence-based decision making

QMP 7 – Relationship management

ISO 9001:2015 REQUIREMENTS

ISO 9001:2015 Quality management systems — Requirements is a document of approximately 30 pages which is available from the national standards organization in each country.

ISO 9001:2015 is the latest revision of the ISO 9001 standard. In it, there are 10 sections (clauses) with supporting subsections (sub clauses). The requirements to be applied to your quality management system (QMS) are covered in sections 4-10. To successfully implement ISO 9001:2015 within your organization, you must satisfy the requirements within clauses 4-10.

ISO 9001:2015 REQUIREMENTS

Some of the key changes include:

High Level Structure of 10 clauses is implemented. Now all new standard released by ISO will have this High level structure.

- Greater emphasis on building a management system suited to each organization's particular needs
- A requirement that those at the top of an organization be involved and accountable, aligning quality with wider business strategy
- Risk-based thinking throughout the standard makes the whole management system a preventive tool and encourages continuous improvement
- Less prescriptive requirements for documentation: the organization can now decide what documented information it needs and what format it should be in
- Alignment with other key management system standards through the use of a common structure and core text
- Inclusion of Knowledge Management principles
- Quality Manual & Management representative is now not mandatory requirements.

ISO 9001:2015 REQUIREMENTS

Contents of ISO 9001:2015 are as follows:

Section 1: Scope

Section 2: Normative references

Section 3: Terms and definitions

Section 4: Context of the organization

Section 5: Leadership

Section 6: Planning

Section 7: Support

Section 8: Operation

Section 9: Performance evaluation

Section 10: Improvement

SOFTWARE QUALITY TOOLS

- Ishikawa Diagram
- Check List
- Control Chart
- Flow Chart
- Pareto Chart
- Histogram

CAUSE & EFFECT DIAGRAMS

Kaoru Ishikawa (1915 - 1989)

- Studied under Deming
- Believed quality is a continuous process that relies on all levels of the organization

Cause and effect diagrams (Ishikawa Diagram) are used for understanding organizational or business problem causes.

Organizations face problems everyday and it is required to understand the causes of these problems in order to solve them effectively. Cause and effect diagrams exercise is usually a teamwork.

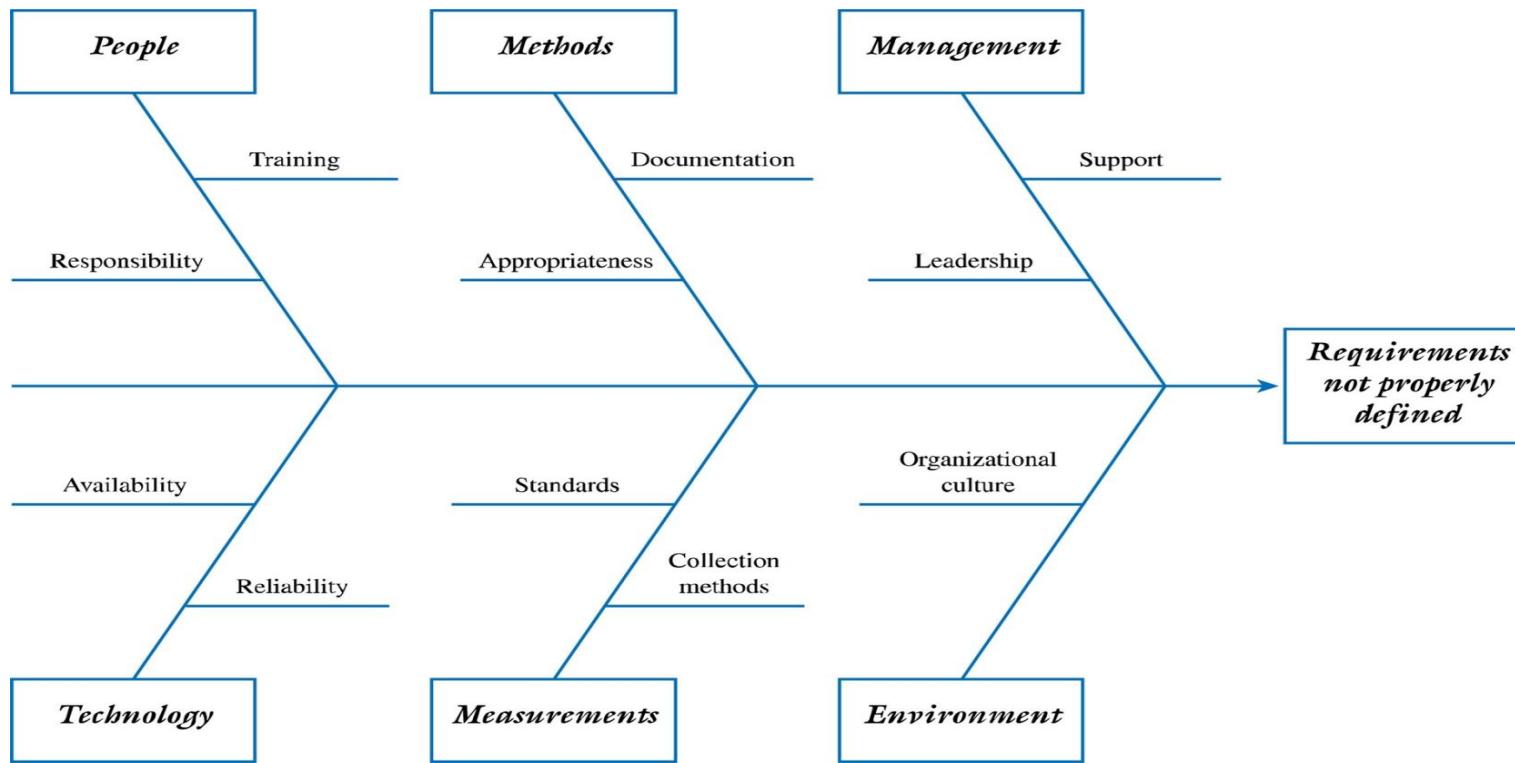
A brainstorming session is required in order to come up with an effective cause and effect diagram.

All the main components of a problem area are listed and possible causes from each area is listed.

Then, most likely causes of the problems are identified to carry out further analysis.

ISHIKAWA, OR FISHBONE DIAGRAM

BEST DEVELOPED BY BRAINSTORMING OR BY USING A LEARNING CYCLE APPROACH



CONTROL CHARTS

- Walter A. Shewhart (1891 – 1967)
 - Worked for Western Electric Company (Bell Telephones)
 - Introduced the concept of the control chart as a tool better to understand variation and to allow management to shift its focus away from inspection and more towards the prevention of problems and the improvement of processes.

CONTROL CHARTS

A **quality control chart** is a graphic that depicts whether sampled products or processes are meeting their intended specifications and, if not, the degree by which they vary from those specifications.

The control chart is a graph used to study how a process changes over time. Data are plotted in time order. A control chart always has a central line for the average, an upper line for the upper control limit, and a lower line for the lower control limit. These lines are determined from historical data. By comparing current data to these lines, you can draw conclusions about whether the process variation is consistent (in control) or is unpredictable (out of control, affected by special causes of variation).

CONTROL CHARTS

Common and special causes are the two distinct origins of variation in a process, as defined in the statistical thinking and methods of Walter A. Shewhart and W. Edwards Deming. Briefly, "common causes", also called Natural patterns, are the usual, historical, quantifiable variation in a system, while "special causes" are unusual, not previously observed, non-quantifiable variation.

CONTROL CHARTS

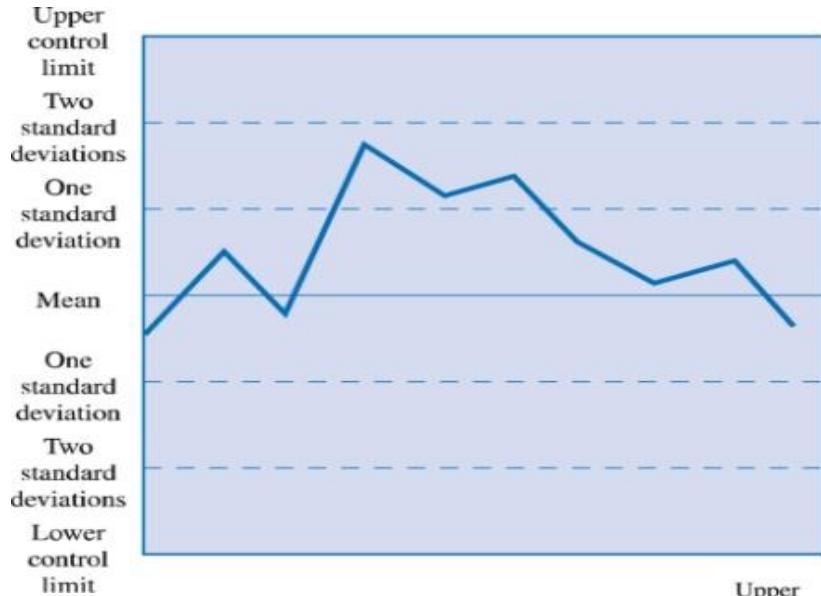
Common causes

- Inappropriate procedures
- Poor design
- Poor maintenance of machines
- Measurement error
- Quality control error

Special causes

- Faulty controllers
- Machine malfunction
- Computer crash
- Poor batch of raw material

CONTROL CHARTS



CONTROL CHARTS

Purpose:

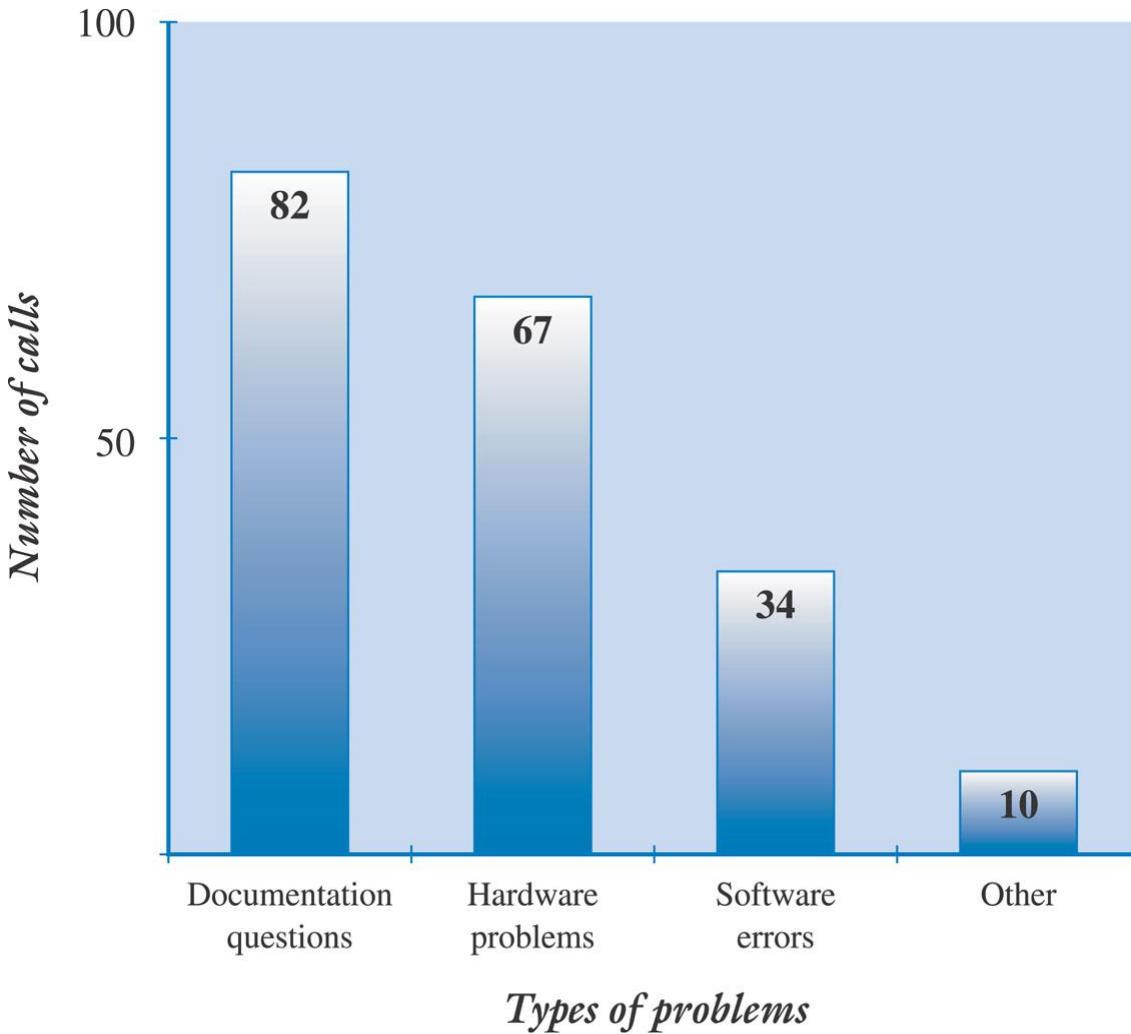
The primary purpose of a control chart is to predict expected product outcome.

Benefits:

- Predict process out of control and out of specification limits
- Distinguish between specific, identifiable causes of variation
- Can be used for statistical process control



PARETO CHART



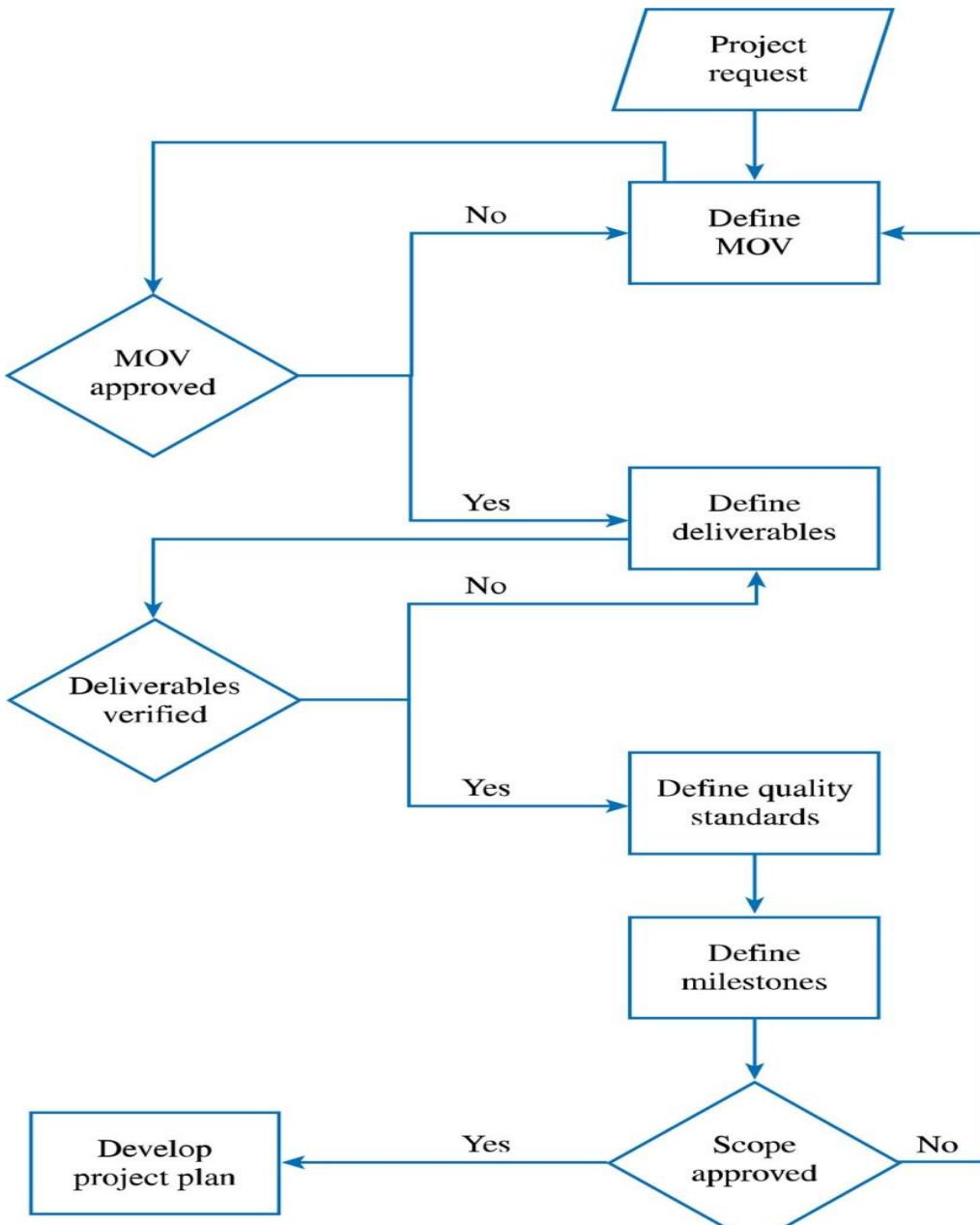
PARETO CHART

Pareto charts are used for identifying a set of priorities. You can chart any number of issues/variables related to a specific concern and record the number of occurrences.

This way you can figure out the parameters that have the highest impact on the specific concern.

This helps you to work on the propriety issues in order to get the condition under control.

FLOW CHART FOR PROJECT SCOPE VERIFICATION



CHECK LIST

Project Name:		Check Sheet							
Project Manager:		Dates:							
Defect/Issue		Sun	Mon	Tues	Wed	Thurs	Fri	Sat	Total
									0
									0
									0
									0
									0
									0
									0
Total		0	0	0	0	0	0	0	0

CHECK LIST/SHEET

A check sheet can be introduced as the most basic tool for quality. A check sheet is basically used for gathering and organizing data.

When this is done with the help of software packages such as Microsoft Excel, you can derive further analysis graphs and automate through macros available.

Therefore, it is always a good idea to use a software check sheet for information gathering and organizing needs.

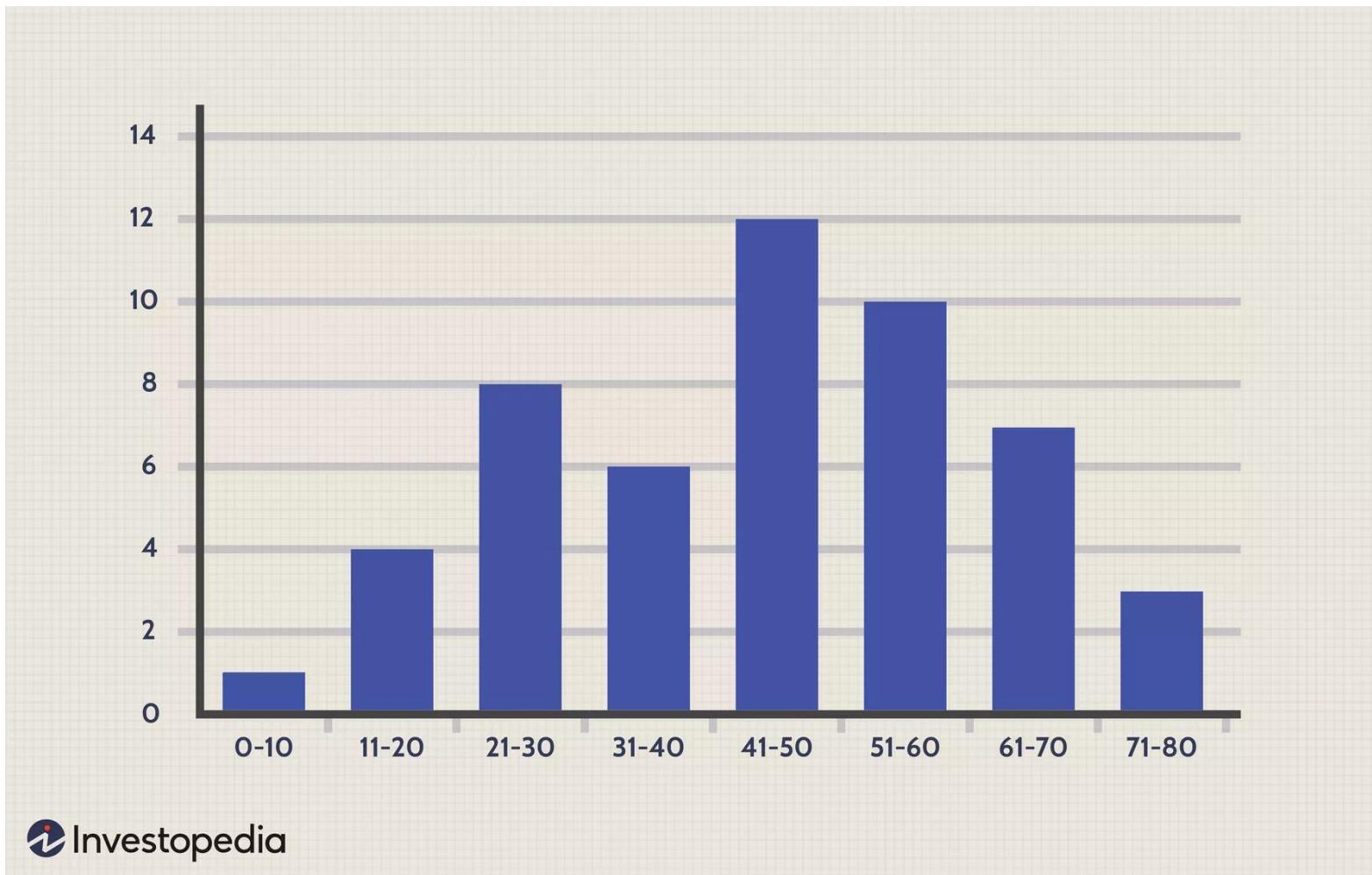
One can always use a paper-based check sheet when the information gathered is only used for backup or storing purposes other than further processing.

HISTOGRAM

A histogram is a graphical representation that organizes a group of data points into user-specified ranges. It is similar in appearance to a bar graph. The histogram condenses a data series into an easily interpreted visual by taking many data points and grouping them into logical ranges or bins.

- A histogram is a bar graph-like representation of data that buckets a range of outcomes into columns along the x-axis.
- The y-axis represents the number count or percentage of occurrences in the data for each column and can be used to visualize data distributions.

HISTOGRAM



HISTOGRAM

