SW TESTING

- ✓ Software Testing is a process where the software is evaluated to determine that it meets the **user needs (validation)** and that the **process to build the software was followed correctly(verification).**
- ✓ in other way,

  Validation: Are we building the right product?

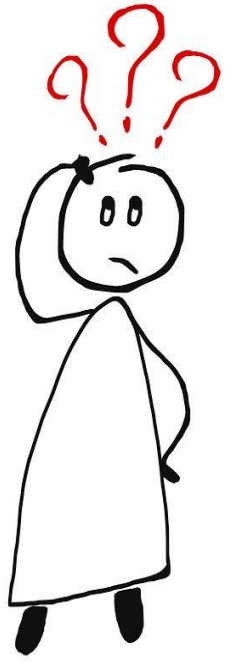
  Verification: Are we building the product right?

TESTING : WHY DO WE PERFORM

Testing fulfills two primary purposes:

- ✓ to demonstrate quality or proper behavior;
- ✓ to detect and fix problems.

finding and eventually fixing defects is the Number one reason for software testing. But the truth is, no Software is free from Defect.



If all software is released to customers with defects, why should we spend so much time, effort, and money on testing?

Testing: Why is it Important

- ✓ Most of us have had an experience with systems that don't work as expected. This can have a huge impact on a company, including:

  Financial loss – losing business is bad enough but in extreme cases there are financial penalties for non-compliance to regulations

  Increased costs – these tend to be the result of having to implement a manual workaround but can include staff not being able to work due to a fault or failure and the cost of fixing the issues after go-live

  Reputational loss – if the company is unable to provide service to their customers due to software problems then the customers will probably take their business elsewhere

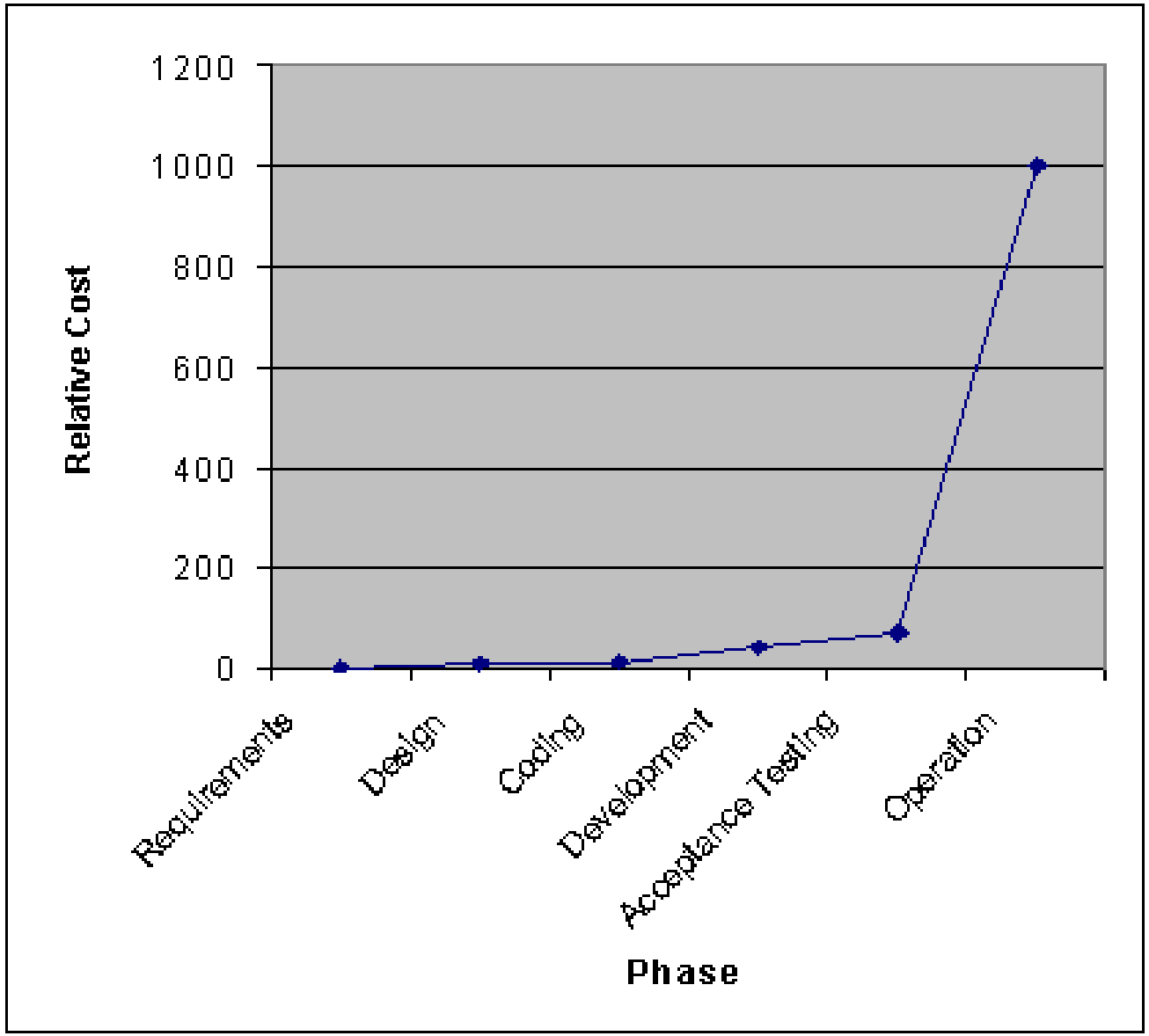
It is important to test the system to ensure it is error free

Cost of Late Testing

- ✓ IBM mainframe z/OS system defect that was found by a customer in the field was estimated to cost at minimum 10K per defect.
- ✓ It's not about a complicated defect, any defect even a one line change in the code would cost that much to fix.
- ✓ Researchers found that if finding an error in requirements phase costs 1 unit then it increases so significantly that the same defect if undetected until operations phase can cost to fix anywhere from 29 units to more than 1,500 units.
- ✓ So if a unit costs $1000 then at requirements phase a defect would cost a $1,000 to fix while in the operations it can cost from $29,000 to $150,000 to fix.

So what is the bottom line message?

Early testing can save significant money for the company!


1200
1000
800
600
400
200
0
Relative Cost
Requirements
Design
Coding
Development
Acceptance Testing
Operation
Phase

SOFTWARE TESTING — MYTHS AND FACTS

Myth	Testing is a single phase in **SDLC** .
Truth	in reality, testing starts as soon as we get the requirement specifications for the software. And the testing work continues throughout the **SDLC**, even post-implementation of the software.
Myth	Testing is easy..
Truth	is not easy, as they have to plan and develop the test cases manually and it requires a thorough understanding of the project being developed with its overall design. Overall, testers have to shoulder a lot of responsibility which sometimes make their job even harder than that of a developer..

SOFTWARE TESTING — MYTHS AND FACTS

Myth	Software development is worth more than testing.
Truth	Testing is a complete process like development, so the testing team enjoys equal status and importance as the development team.
Myth	Complete testing is possible.
Truth	there are many things which cannot be tested completely, as it may take years to do so.
Myth	Testing starts after program development.
Truth	the work of a tester begins as soon as we get the specifications. The tester performs testing at the end of every phase of SDLC in the form of verification and plans for the validation testing.

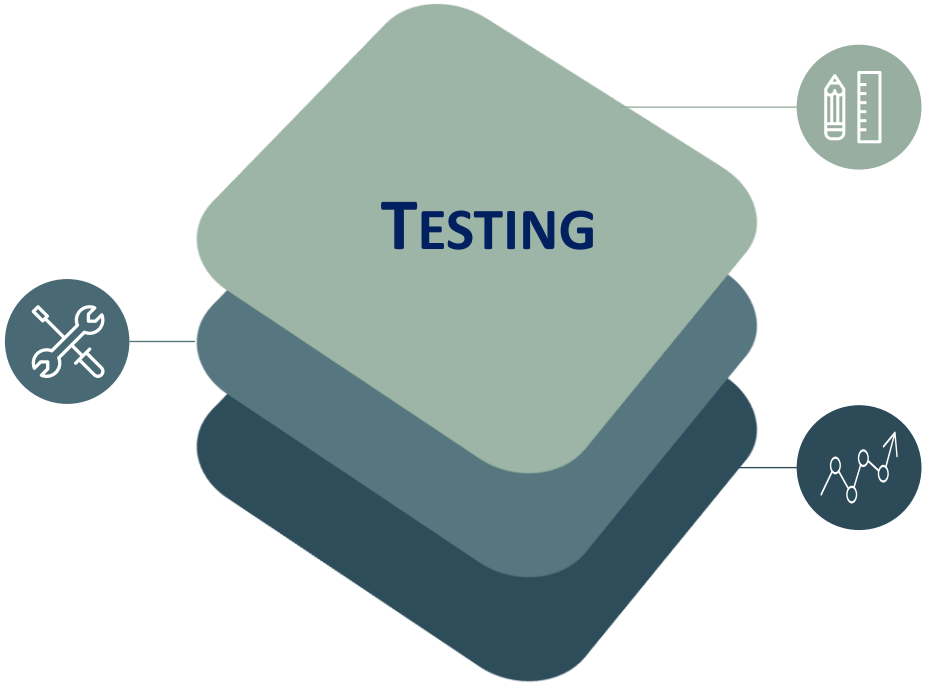
SOFTWARE TESTING — MYTHS AND FACTS

Myth	The purpose of testing is to check the functionality of the software.
Truth	quality does not always imply checking only the functionalities of all the modules. There are various things related to quality of the software, for which test cases must be executed..
Myth	Anyone can be a tester.
Truth	??????????????????

GOALS OF SOFTWARE TESTING

Post-implementation Goals

- Reduced maintenance cost
- Improved testing process



Immediate Goals

- Bug discovery
- Bug prevention

Long-term Goals

- Reliability
- Risk management
- Quality
- Customer satisfaction

TESTING TERMINOLOGY

Failure

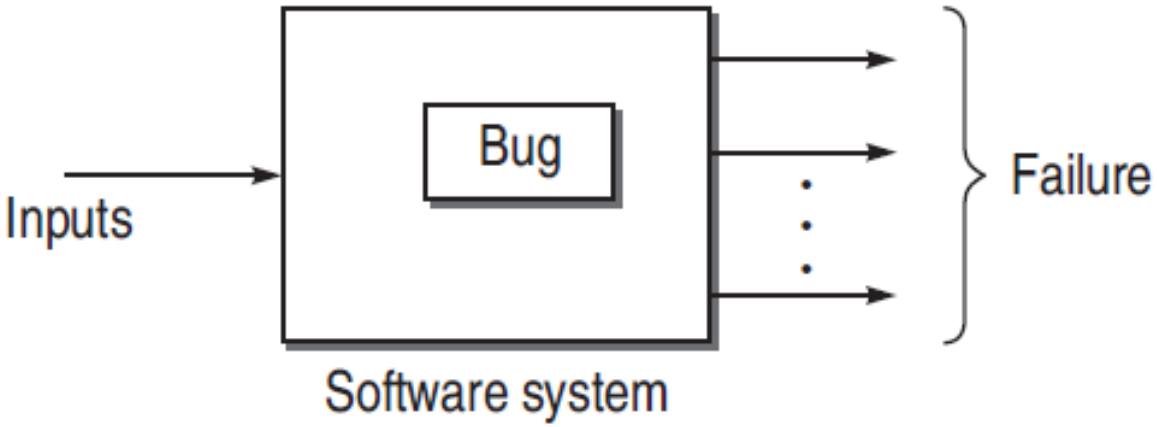
Is the inability of a system or component to perform a required function according to its specification.

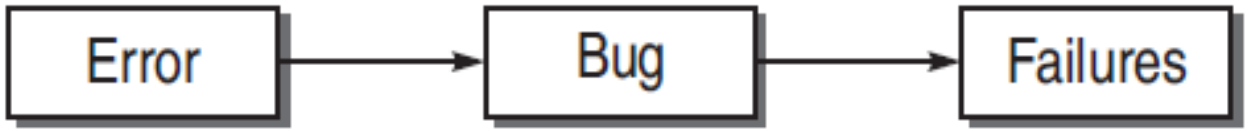
Fault/Defect/Bug

is a fault due to an **error** in code whether due to incorrect design, logic, or implementation where the software does not behave as expected.

Error

Whenever a development team member makes a 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.





Why Do Bugs Occur?

Faulty definition of requirements

- Erroneous requirement definitions
- Absence of important requirements
- Incomplete requirements
- Unnecessary requirements included

Client-developer communication failures

- Misunderstanding of client requirements presented in writing, orally, etc.
- Misunderstanding of client responses to design problems

Coding errors

- Errors in interpreting the design document, errors related to incorrect use of programming language constructs, etc.

Why Do Bugs Occur?

Deliberate deviations from software requirements

- Reuse of existing software components from previous projects without complete analysis
- Functionality omitted due to budget or time constraints
- “Improvements” to software that are not in requirements

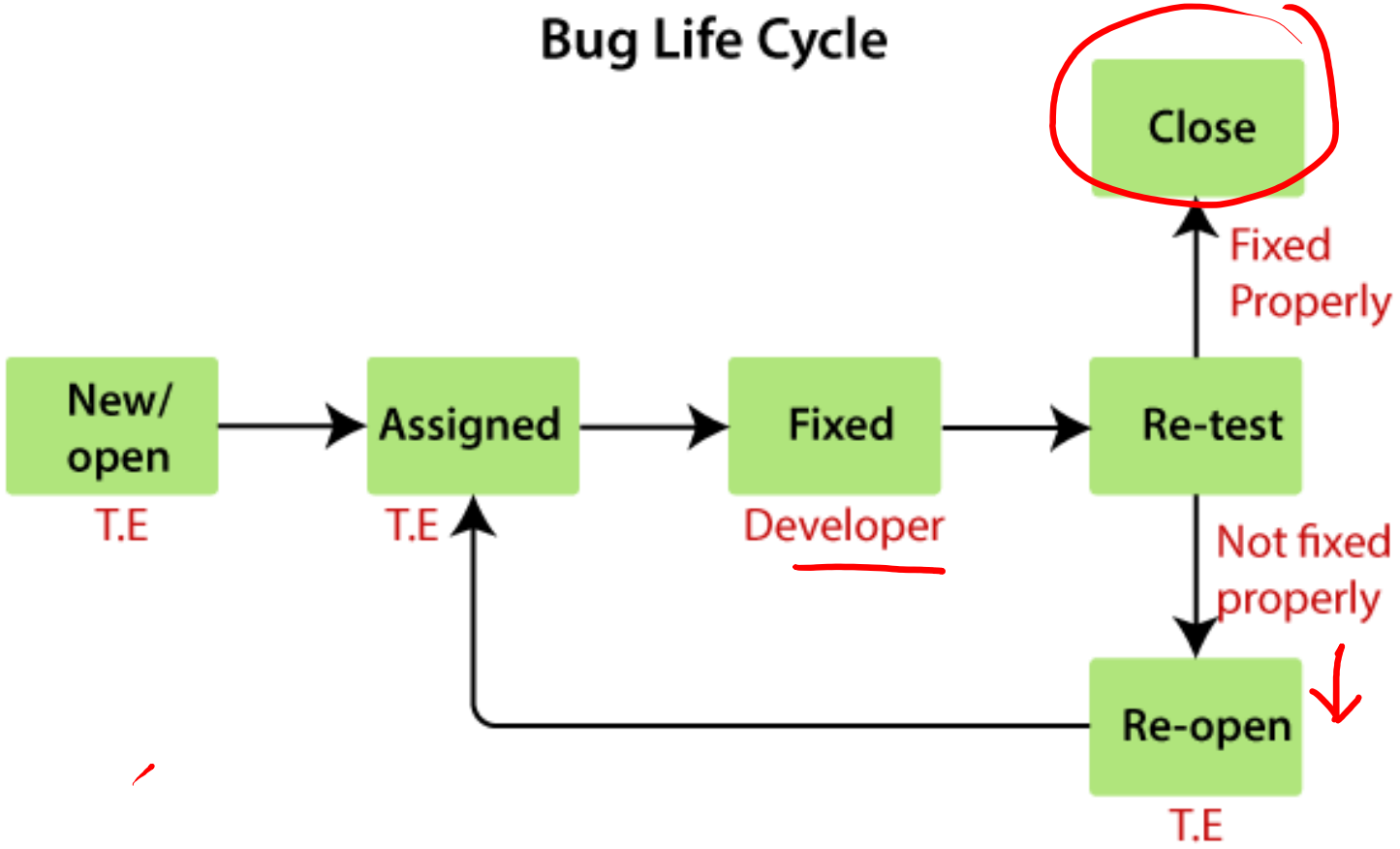
Logical design errors

- Errors in interpreting the requirements into a design (e.g. errors in definitions of boundary conditions, algorithms, reactions to illegal operations,...)

Shortcomings of the testing process

- Incomplete test plan
- Failure to report all errors/faults resulting from testing
- Incorrect reporting of errors/faults
- Incomplete correction of detected errors

DEFECT/BUG LIFE CYCLE

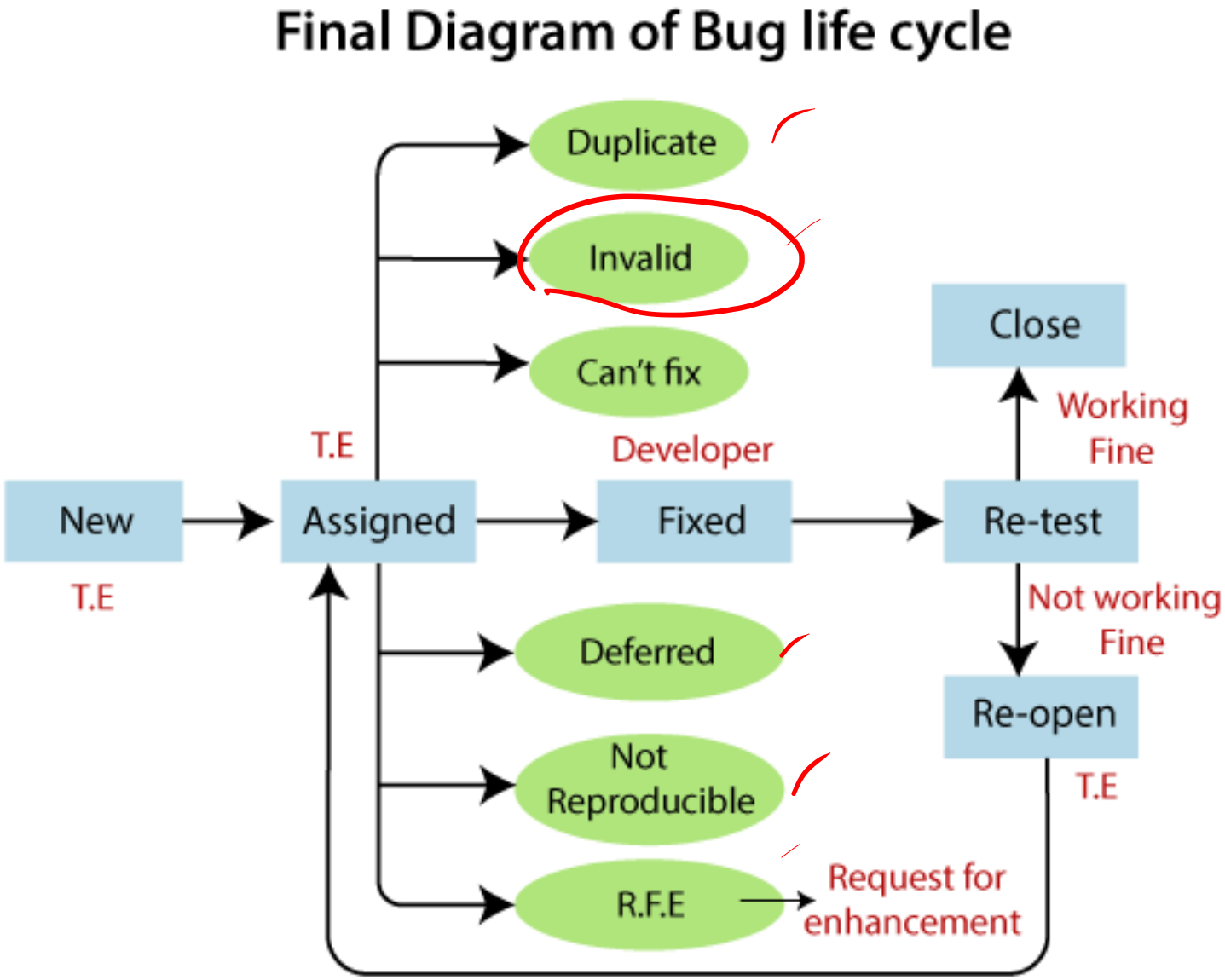


Others status of the bug

Following are the different status of the bug:

- ✓ Invalid/rejected
- ✓ Duplicate
- ✓ Postpone/deferred
- ✓ Can't fix
- ✓ Not reproducible
- ✓ RFE (Request for Enhancement)

Defect/Bug Life Cycle

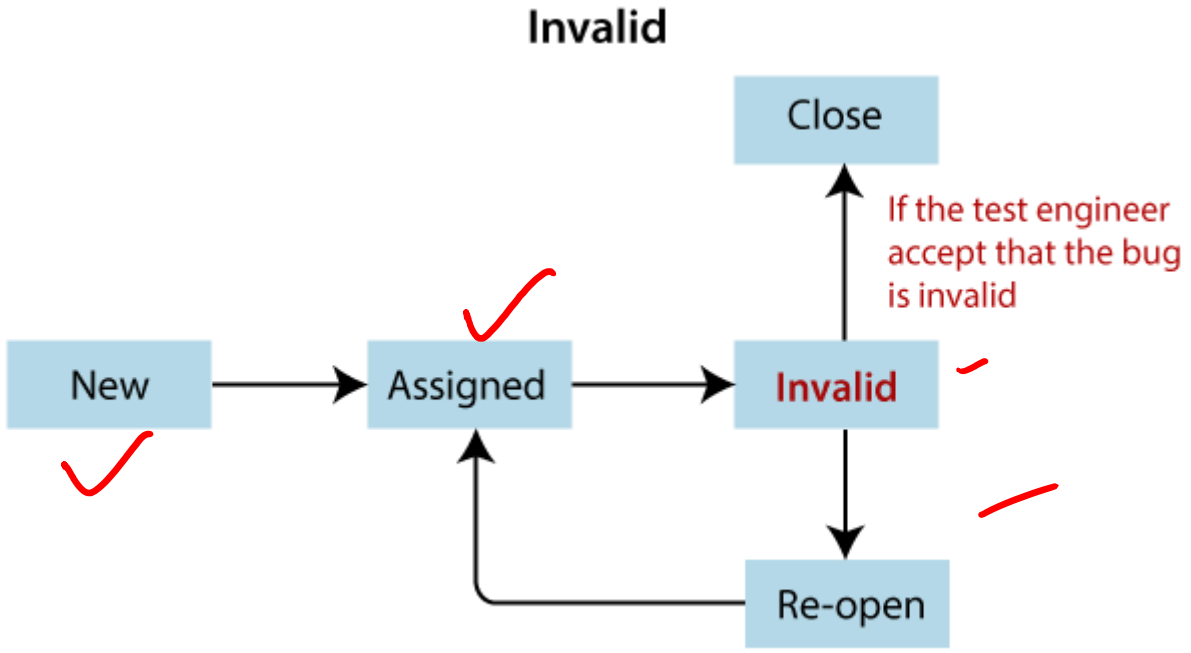


INVALID BUG

Any bug which is not accepted by the developer is known as an invalid bug.

Reasons for an invalid status of the bug :

1. Test Engineer misunderstood the requirements
2. Developer misunderstood the requirements

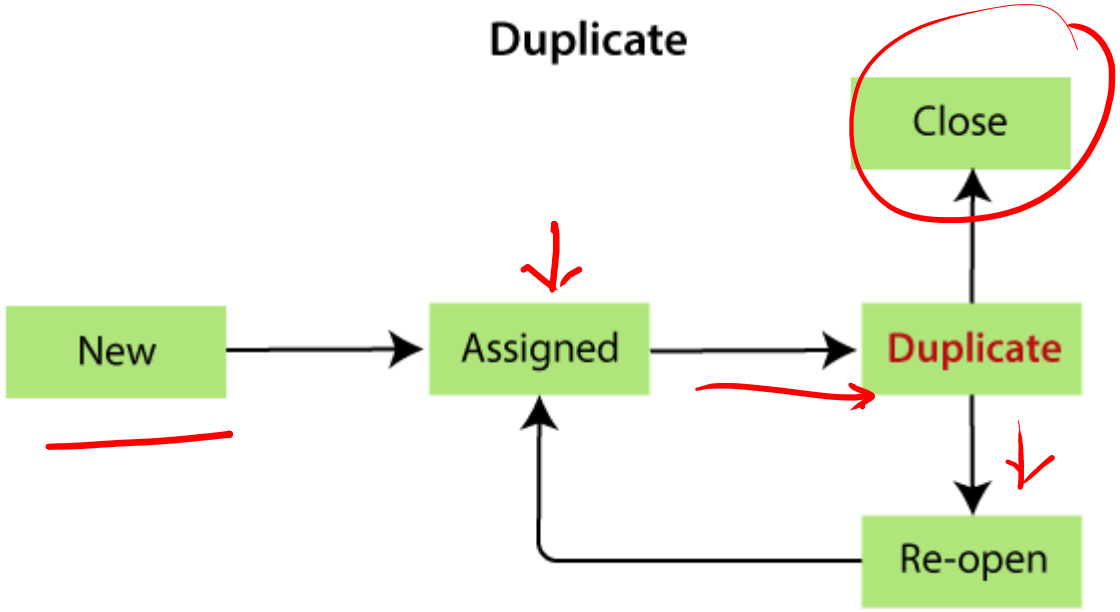


DUPLICATE BUG

When the same bug has been reported multiple times by the different test engineers are known as a duplicate bug.

Reasons for an duplicate status of the bug :

1. Common features
2. Dependent Modules



DUPLICATE BUG

To avoid the duplicate bug:

If the Developer got the duplicate bug, then he/she will go to the bug repository and search for the bug and also check whether the bug exist or not.

If the same bug exist, then no need to log the same bug in the report again.

Or

If the bug does not exist, then log a bug and store in the bug repository, and send to Developers and Test Engineers adding them in [CC].

NOT REPRODUCIBLE

These are the bug where the developer is not able to find it, after going through the navigation step given by the test engineer in the bug report.

Reasons for the not reproducible status of the bug

1. **Incomplete bug report**
   The Test engineer did not mention the complete navigation steps in the report.
2. **Environment mismatch**
   Environment mismatch can be described in two ways:
   - Server mismatch
   - Platform mismatch

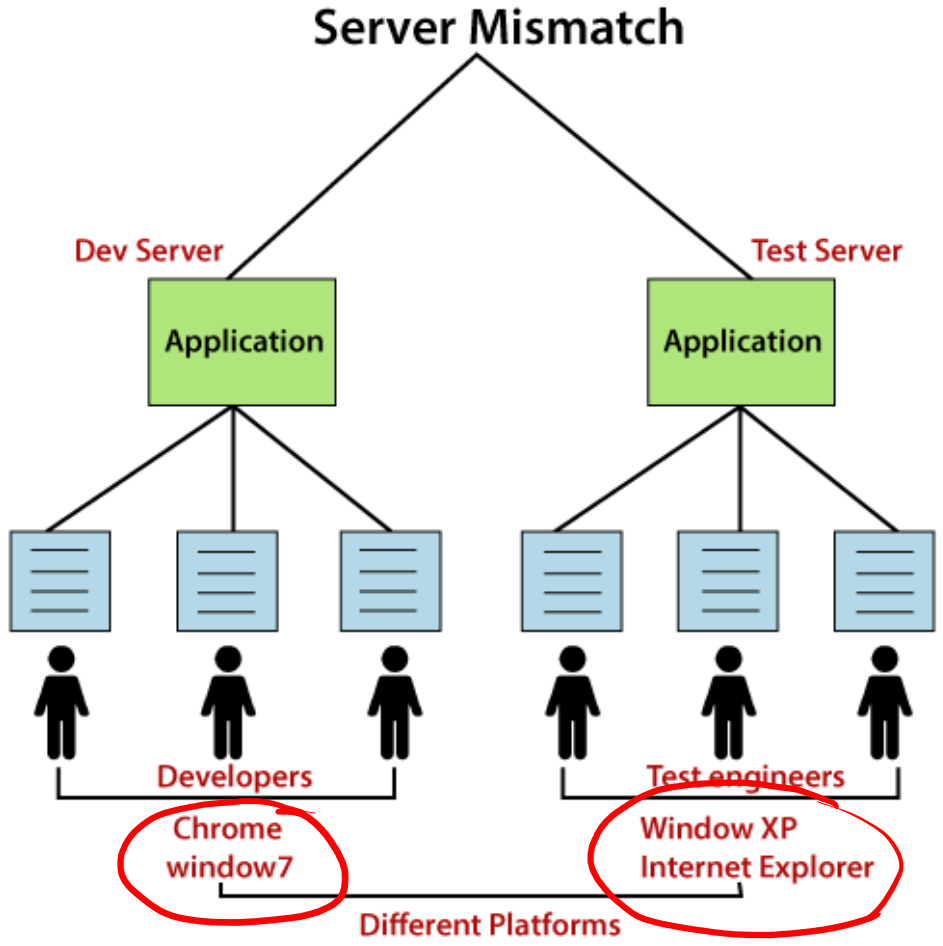
NOT REPRODUCIBLE

Server mismatch

Test Engineer is using a different server (**Test Server**), and the Developer is using the different server (**Development Server**) for reproducing the bug

Platform mismatch:

Test engineer using the different Platform (**window 7 and Google Chrome browser**), and the Developer using the different Platform (**window XP and internet explorer**) as well.



NOT REPRODUCIBLE

Data mismatch

Different Values used by test engineer while testing & Developer uses different values.

For example:

The requirement is given for admin and user.

Test engineer(user) using the below requirements:	**the Developer (admin)** using the below requirements:
User name → abc Password → 123	User name → aaa Password → 111

NOT REPRODUCIBLE

Build mismatch

The test engineer will find the bug in one Build, and the Developer is reproducing the same bug in another build. The bug may be automatically fixed while fixing another bug.

Inconsistent bug

The Bug is found at some time, and sometime it won't happen.

Solution for inconsistent bug:

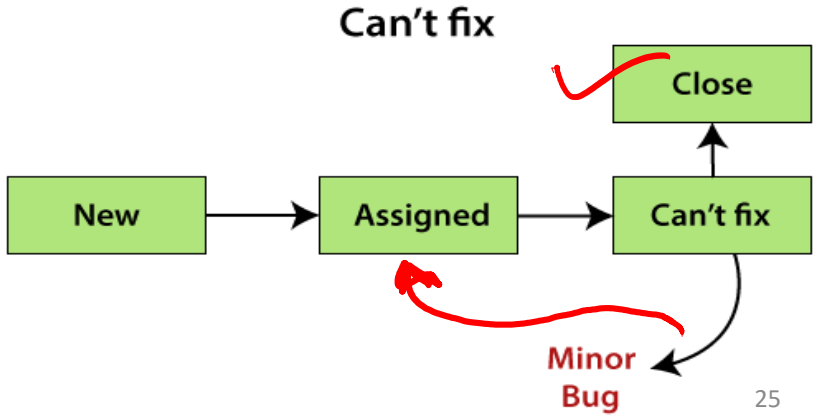
As soon as we find the bug, first, **take the screenshot**, then developer will **re-confirm** the bug and fix it if exists.

CAN'T FIX

When Developer accepting the bug and also able to reproduce, but can't do the necessary code changes due to some constraints.

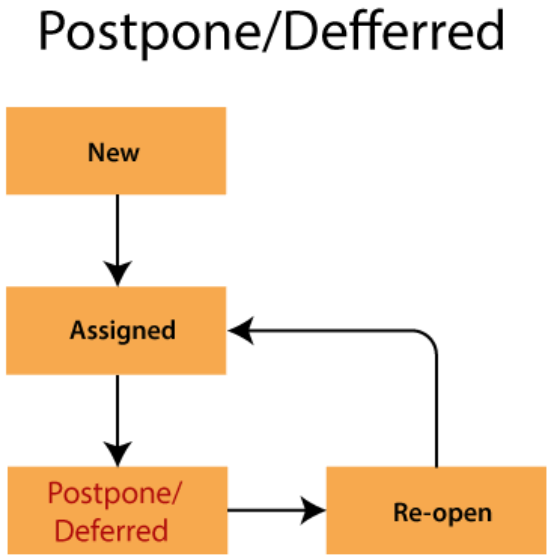
Reasons for the can't fix status of the bug

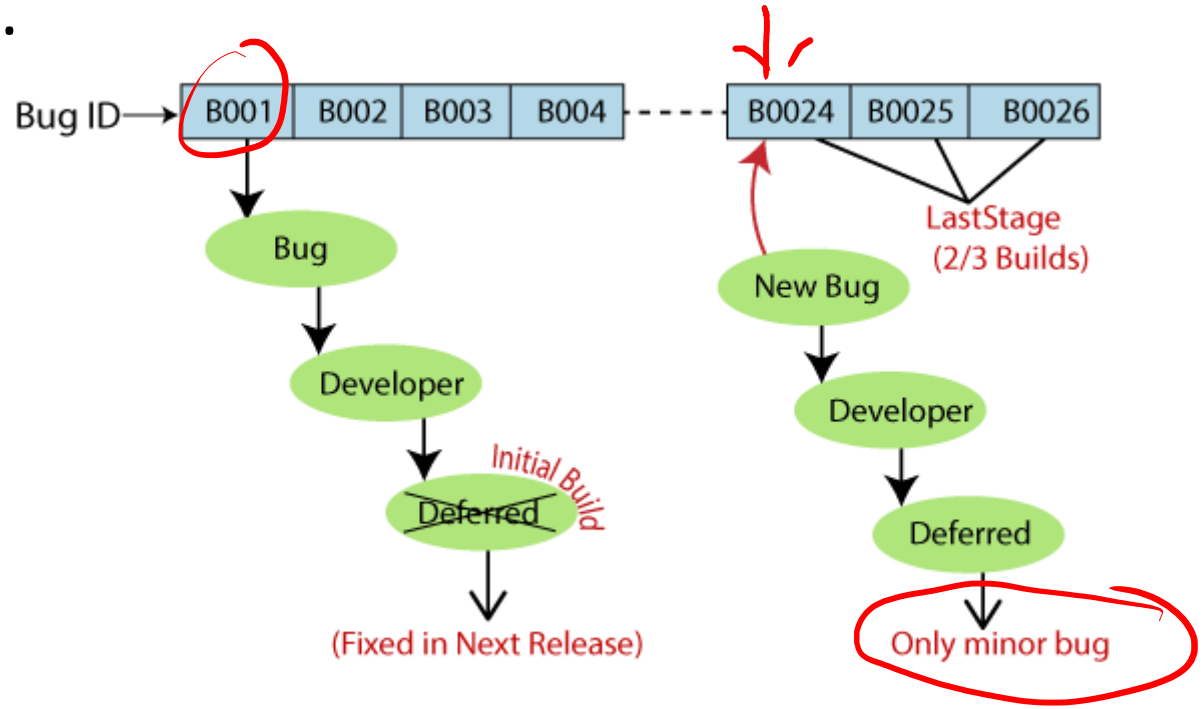
1. **No technology support:** The programming language we used itself not having the capability to solve the problem.
2. **The Bug is in the core of code (framework):** If the bug is **minor** (not important and does not affect the application), the development lead says it can be fixed in the next release.
   Or , if the bug is **critical** (regularly used and important for the business) and development lead cannot reject the bug.
3. **The cost of fixing a bug is more than keeping it.**



DEFERRED/POSTPONED

The deferred/postpone is a status in which the bugs are postponed to the future release due to time constraints.



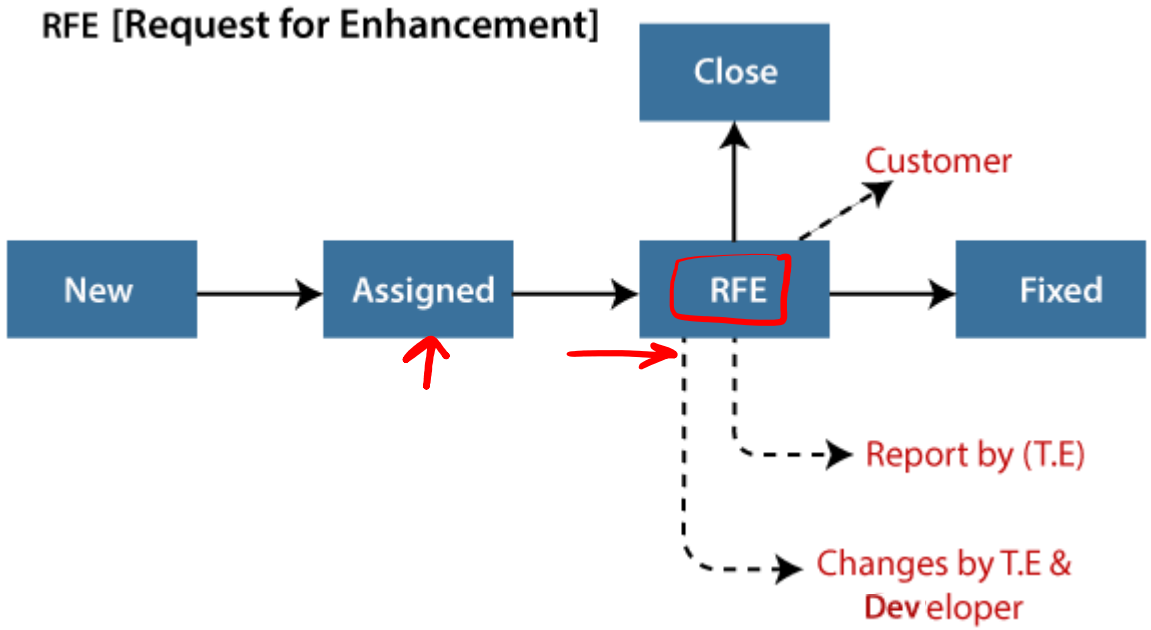


The **Bug ID-B001** bug is found at the initial build, but it will not be fixed in the same build, it will postpone, and **fixed in the next release**.

And **Bug ID- B0024, B0025, and B0026** are those bugs, which are found in the last stage of the build, **and they will not be fixed because these bugs are the minor bugs**.

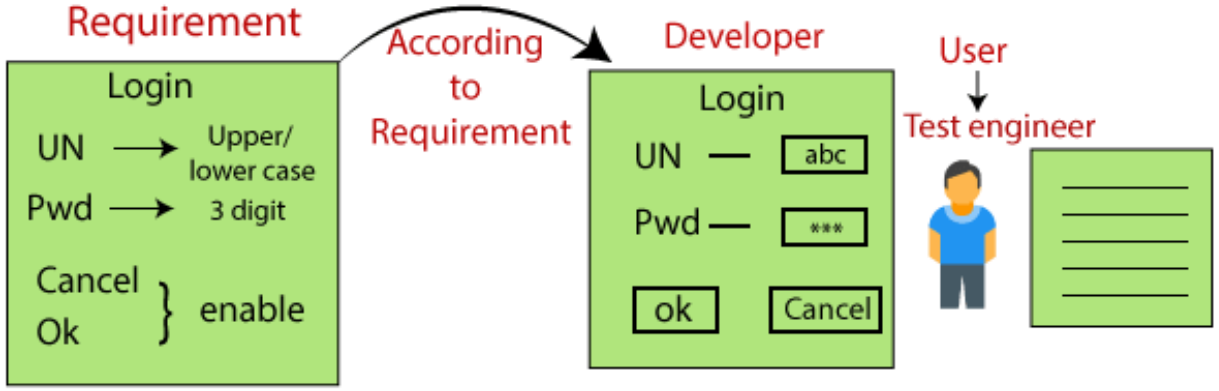
RFE (REQUEST FOR ENHANCEMENT)

These are the suggestions given by the test engineer towards the enhancement of the application in the form of a bug report. The RFE stands for Request for Enhancement.



RFE (Request for Enhancement)

As we can see in the below image that the test engineer thinks that the look and feel of the application or software are not good because the test engineer is testing the application as an end-user, and he/she will change the status as RFE.



And if the customer says Yes, then the status should be Fix.

Or

If the customer says no, then the status should be Close.

BUG SEVERITY

The impact of the bug on the application is known as severity.

It can be a blocker, critical, major, and minor for the bug.

Blocker: if the severity of a bug is a blocker, which means we cannot proceed to the next module, and unnecessarily test engineer sits ideal.

Critical: if it is critical, that means the main functionality is not working, and the test engineer cannot continue testing.

Major: if it is major, which means that the supporting components and modules are not working fine, but test engineer can continue the testing.

Minor: if the severity of a bug is major, which means that all the U.I problems are not working fine, but testing can be processed without interruption.

BUG PRIORITY

- ✓ Priority is important for fixing the bug or which bug to be fixed first or how soon the bug should be fixed.
- ✓ It can be urgent, high, medium, and low.

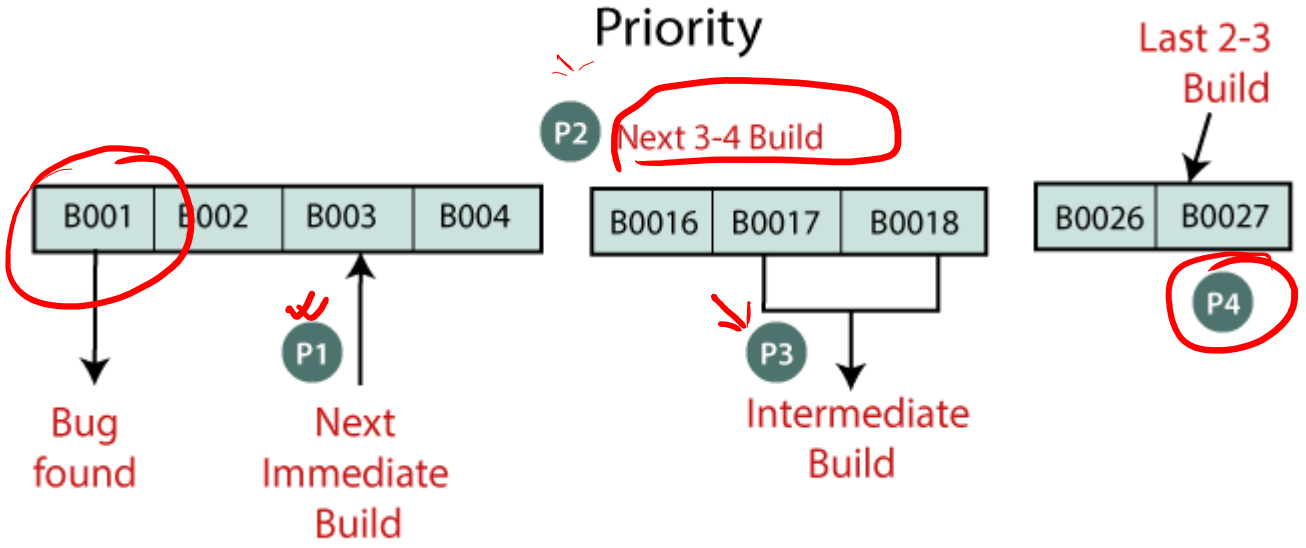
High: it is a major impact on the customer application, and it has to be fixed first.

Medium: In this, the problem should be fixed before the release of the current version in development.

Low: The flow should be fixed if there is time, but it can be deferred with the next release.

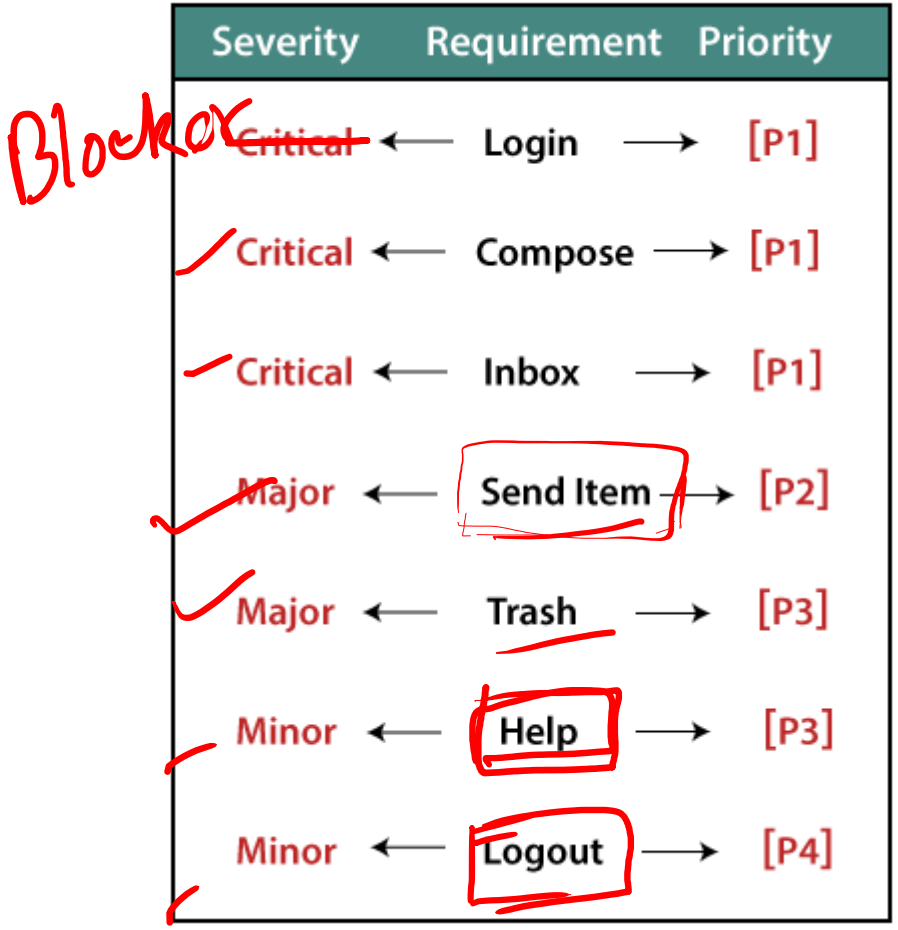
EXAMPLE OF SEVERITY AND PRIORITY

- ✓ When the bug is just found, it will be fixed in the next immediate build, and give the Priority as P1 or urgent.
- ✓ If the Priority of the bug is P2 or high, it will be fixed in the next 3-4 builds.
- ✓ When the Priority of the bug is P3/medium, then it will be fixed in the intermediate build of the application.
- ✓ And at last, if the Priority is P4/low, it will be fixed in the last 2-3 build of the software as we can see in the below image:



Example of severity and Priority

If we take the case of a login module, then the severity and Priority could depend on the application like as we can see in the below image:



WRITING DEFECT/BUG

What makes a good defect report?

Often defect reports do not have enough information or they are inaccurate and so they are either rejected or when reviewed later, it is impossible to say what the actual problem was.

Parameters of a Defect/Bug

The Following details should be part of a Bug:

- ✓ Date of issue, author, approvals and status.
- ✓ Severity and priority of the incident.
- ✓ The associated test case that revealed the problem
- ✓ Expected and actual results.
- ✓ Description of the incident with steps to Reproduce
- ✓ Status of the incident
- ✓ Conclusions, recommendations and approvals.

CHARACTERISTICS OF A GOOD BUG REPORT

- ✓ **Objective** – criticizing someone else's work can be difficult. Care should be taken that defects are objective, non-judgmental and unemotional. e.g. don't say "your program crashed" say "the program crashed" and don't use words like "stupid" or "broken".
- ✓ **Specific** – one report should be logged per defect and only one defect per report.
- ✓ **Concise** – each defect report should be simple and to-the-point. Defects should be reviewed and edited after being written to reduce unnecessary complexity.
- ✓ **Persuasive** –the pinnacle of good defect reporting is the ability to present them in a way which makes developers want to fix them

CHARACTERISTICS OF A GOOD BUG REPORT

- ✓ **Reproducible** – the single biggest reason for developers rejecting defects is because they can't reproduce them. As a minimum, a defect report must contain enough information to allow anyone to easily reproduce the issue if it can be reproduced at will. Timing issues may be hard to reproduce but in that case there should be other information that can help in determining where in the code the error may be
- ✓ **Explicit** – defect reports should state information clearly or they should refer to a specific source where the information can be found. e.g. "click the button to continue" implies the reader knows which button to click, whereas "click the 'Next' button" explicitly states what they should do.

Sample Bug Report Template

Bug Report Template	
Bug ID	
Module	
Requirements	
Test Case Name	
Release	
Version	
Status	
Reporter	
Date	
Assign To	
CC	
Severity	
priority	
Server	
Platform	
Build No.	
Test Data	
Attachment	
Brief Description	
Navigation Steps	
Observation	
Expected Result	
Actual Result	
Additional Comments	

DEFECT/BUG MANAGEMENT

- ✓ Defects need to be handled in a methodical and systematic fashion
- ✓ There's no point in finding a defect if it's not going to be fixed
- ✓ There's no point getting it fixed if you don't know it has been fixed and there's no point in releasing software if you don't know which defects have been fixed and which remains.

How will you know?

The answer is to have a defect tracking system

The simplest can be a database or a spreadsheet. A better alternative is a dedicated system, which enforce the rules and process of defect handling and makes reporting easier. Some of these systems are costly but there are many freely available variants.

TEST CASE

The tests are usually defined in a document called Test Plan which spells out all the details of the testing to be done, schedules and milestones, responsibilities, testing systems and installation and setup, and the list of test cases to be executed.

- ✓Test cases can be manual where a tester follows conditions and steps by hand or automated where a test is written as a program to run against the system.
- ✓Tests are usually first defined in a document and often have to be approved by developers and/or testers.
- ✓Automated tests may have all the documentation in the code implementing the test case and in the comments.

Fields of a Test Case

There is no specific Rule for it.

But the common information that all test cases should have:

- ✓ **Unique Test Case name and/or ID:** Each test case needs an identifier. Some tests will have both an id and a meaningful name which is derived from the requirement or the functionality being tested.
- ✓ **Test Scenario and test summary or description:** This description should specify what behavior/functionality will be tested by this test case.
- ✓ **Pre-requisites or setup:** What needs to be setup first, such as previous input or commands to the system to execute this test case.
- ✓ **Test data or inputs:** Data to be used for this test case.
- ✓ **Execution Steps:** The steps that have to be done against the system to test the behavior for this test case.
- ✓ **Expected behavior/Result:** what is expected to happen after execution
- ✓ **Assumptions:** Any assumptions that are made about the system or the test case. For example, data dependency or other test case dependency.
- ✓ **Actual results:** what actually happened when the test case was run.
- ✓ **Status:** What is the current status of the test case such as passed, failed, or not tested.

CHARACTERISTICS OF A GOOD TEST CASE

So how can you develop good test cases?

There are a number of guidelines or best practices in writing efficient test cases.

1. Only test one thing in a test case

Your test case should not be complicated and it should only test a single condition or value. It should be as “atomic” as possible and it should not overlap other test cases either

2. Test case should have an exact and accurate purpose

There should be no confusion what the test is supposed to be checking and what the expected behavior and result should be. The test case should be accurate on what it tests and it should test what it is intended to test.

3. Test case should be written in a clear and easy to understand language

This applies to both the test definition in the documentation and in the actual code. Just like the test definition should be clear, the code that implements it should be straight forward and clear. Any developer or tester should be able to understand exactly what the test case is trying to do by reading it once.

CHARACTERISTICS OF A GOOD TEST CASE

4. Test case should be relatively small

If your test case is following the rule for only testing a single thing, it should not be large. If you find yourself writing a long test case, then chances are you are trying to test too much. In that case you need to break it down into multiple test cases.

5. Test case should be independent

You should be able to execute the test cases in any order and independently of other test cases. This makes the test case simpler as it is self-sufficient and there is no reason to track or worry about any other test cases.

6. Test case should not have unnecessary steps or words

The test case should be precise and economical and only have what it needs to have to describe what it is testing and how it should be tested. It should not be cluttered with any unnecessary and confusing verbiage.

CHARACTERISTICS OF A GOOD TEST CASE

7. Test case should be traceable to requirements or design

The test case needs to test some behavior or characteristic that the code is supposed to have. So that means that there is an explicit requirement from the user or product owner, explicit organization requirement (e.g. serviceability for every product), or implicit requirement derived from some explicit requirement that the test case can be traced to.

8. Test case should be repeatable

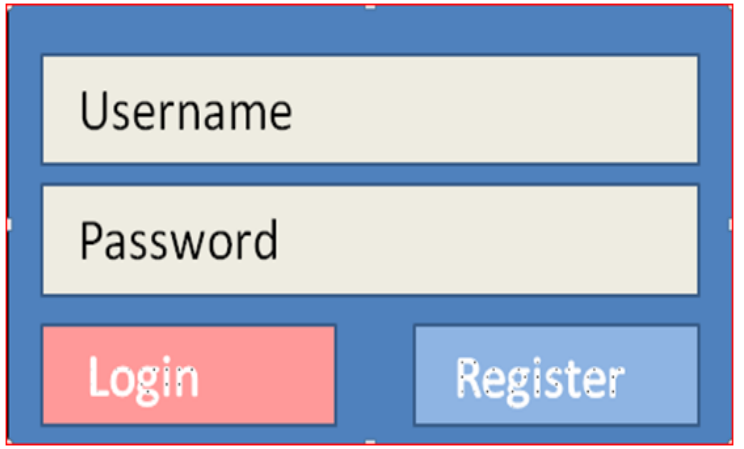
The test cases should be able to be re-run anytime and in any order. This allows for regression testing, re-runs for fixes, and continuous integration where tests are run every time changed code is integrated into the product.

9. Test case should use consistent terminology and identification of functionality

When naming or identifying a feature, functionality, or widget, there should be the same and consistent terminology within the test case and across test cases. So for example, if test cases are describing the tests for login page for a college class, they should not have “user”, “person”, and “student” to refer to the same identity. One of these should be picked and used consistently across.

Test Case Exercise

Let's say we are to test a login functionality that was designed to look like below:



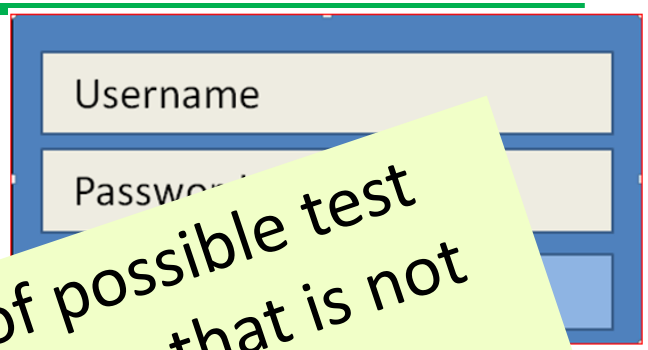
So we have a username and a password fields.
The username is a valid email address and no longer than max valid email address and the password has to be at least 8 characters and no more than 12 characters with only letters and numbers allowed.
Once both registered username and password are entered and Login button clicked, the welcome page will be displayed.

The design indicates that if either username or password is incorrect, we should get the following message above the login area above and the username and password fields should be cleared:

The Username and/or password you entered does not match our records. Try again.

Test Case Exercise

So we define our Test Scenario and Test Cases as follow:



Test Scenario: Verify the login functionality

Test Case 1: Click on Login without typing in us

Test Case 2: Type in correct usernan

Test Case 3: Type in correct password

Test Case 4: Type in incorrect username

Test Case 5: Type in correct username ar word

Test Case 6: Type in incorrect username a correct password

Test Case 7: Type in correct username and correct password

This is not a complete set of possible test cases. For example a username that is not correctly formatted email address; username greater than max size for email address; password less than 8 characters; etc.

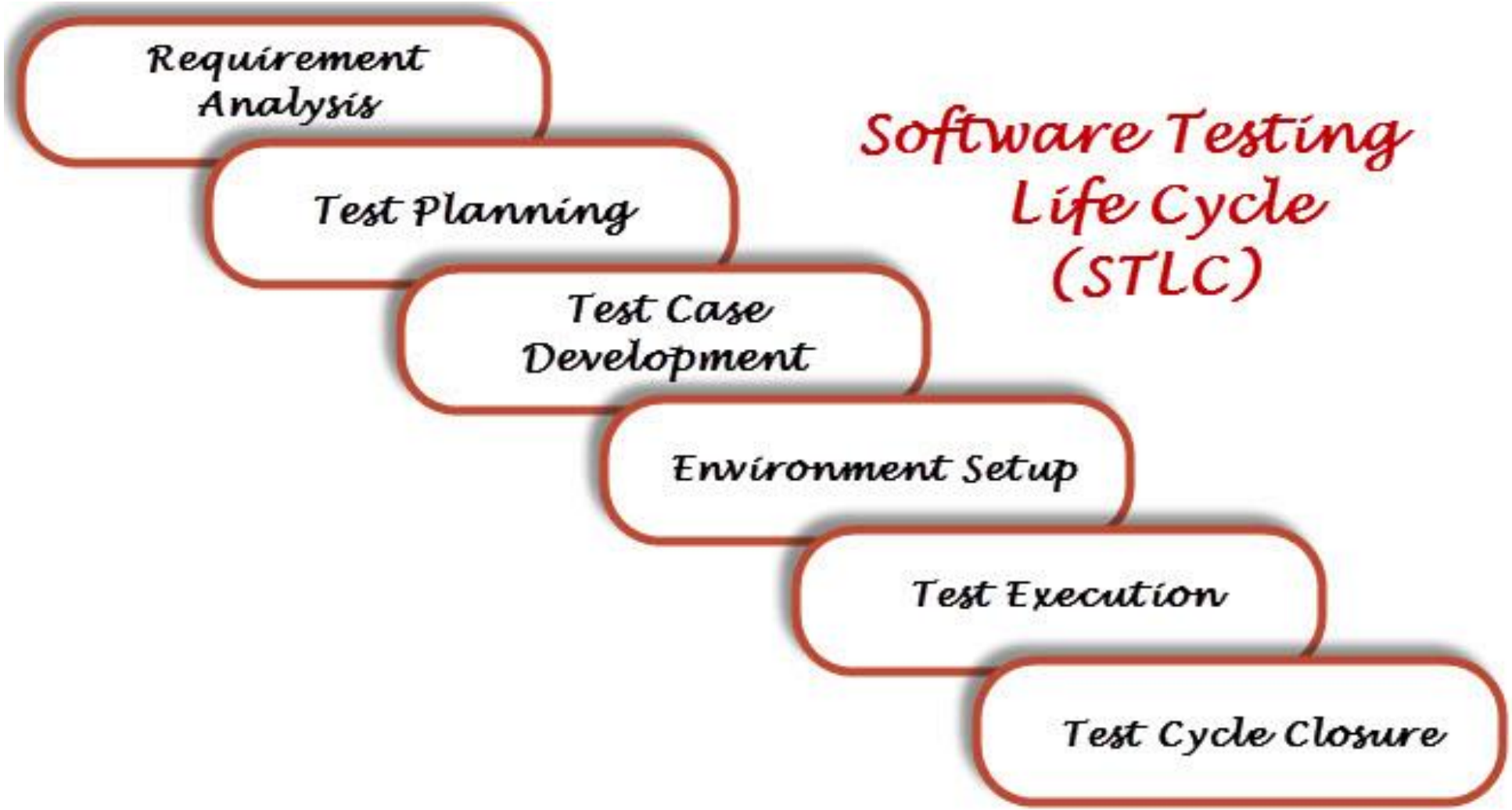
Test Case Exercise

Your **Test Case 1** would then be defined something like:

Test Case Name/ID	T1-ClickLogin
Test Scenario	Login:
Test Case description	Missing both username and password
Pre-requisites	None
Execution Steps	1. Click Login button 2. Check the error message 3. Check that there is no text in the username and password fields
Expected behavior/Result	Message will be displayed above login box "The Username and/or password you entered does not match our records. Try again." Both the username and password fields will be cleared
Assumptions	The GUI design will be tested in a separate test scenario
Actual results	
Status	Not Tested

So Now You Try!!!

Software Testing Life Cycle (STLC)



STLC : REQUIREMENT ANALYSIS

- ✓ test team studies the requirements from a testing point of view to identify testable requirements and the QA team may interact with various stakeholders to understand requirements in detail.

Activities

- Identify types of tests to be performed.
- Gather details about testing priorities and focus.
- Prepare Requirement Traceability Matrix (RTM).
- Identify test environment details where testing is supposed to be carried out.
- Automation feasibility analysis (if required).

Entry Criteria

- Requirements Specification.
- Application architectural view

Deliverables

- List of questions with all answers to be resolved from testable requirements
- Automation feasibility report

STLC : TEST PLANNING

- ✓ a Senior QA manager determines the test plan strategy along with efforts and cost estimates for the project. Moreover, the resources, test environment, test limitations and the testing schedule are also determined.

Activities

- Preparation of test plan/strategy document for various types of testing
  - Test tool selection
  - Test effort estimation
  - Resource planning and determining roles and responsibilities.
  - Training requirement.

Entry Criteria

- Updated requirements document.
- Test feasibility reports “
- Automation feasibility report.

Deliverables

- Test plan /strategy document.
- Effort estimation document.

STLC : Test Case Development

- ✓ the Test data is identified then created and reviewed and then reworked based on the preconditions. Then the QA team starts the development process of test cases for individual units.

Activities

- Create test cases, automation scripts (if applicable)
- Review and baseline test cases and scripts
- Create test data (If Test Environment is available)

Entry Criteria

- Reviewed and signed test Cases/scripts
- Reviewed and signed test data

Deliverables

- Test cases/scripts
- Test data

STLC : Environment Setup

- ✓ setup of software and hardware for the testing teams to execute test cases. It supports test execution with hardware, software and network configured.

Activities

- Understand the required architecture, environment set-up and prepare hardware and software requirement list for the Test Environment.
- Setup test Environment and test data
- Perform smoke test on the build

Entry Criteria

- Environment setup is working as per the plan and checklist
- Test data setup is complete.

Deliverables

- Environment ready with test data set up
- Smoke Test Results.

STLC : TEST EXECUTION

- ✓ the process of executing the code and comparing the expected and actual results. When test execution begins, the test analysts start executing the test scripts based on test strategy allowed in the project.

Activities

- Execute tests as per plan
- Document test results, and log defects for failed cases
- Map defects to test cases in RTM
- Retest the Defect fixes
- Track the defects to closure

Entry Criteria

- Test cases
- Test scripts

Deliverables

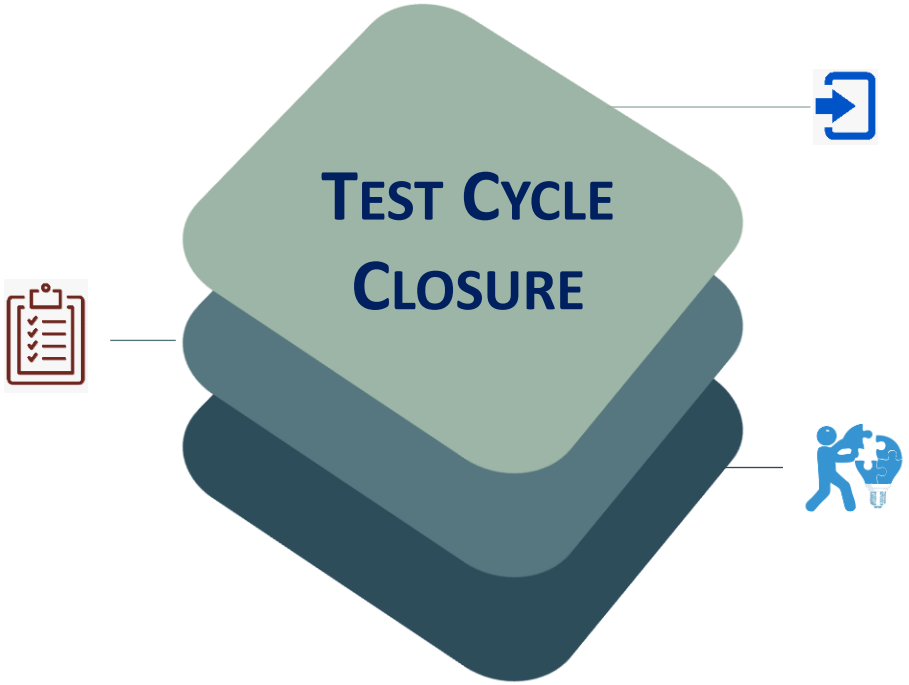
- Completed RTM with the execution status
- Test cases updated with results
- Defect reports

STLC : Test Cycle Closure

- ✓ involves calling out the testing team member meeting & evaluating cycle completion criteria based on Test coverage, Quality, Cost, Time, Critical Business Objectives, and Software.

Activities

- Evaluate cycle completion criteria based on Time, Test coverage, Cost, Software, Critical Business Objectives, Quality
- Prepare test metrics based on the above parameters.
- Document the learning out of the project
  - Prepare Test closure report
- Test result analysis to find out the defect distribution by type and severity.



Entry Criteria

- Test closure condition
- Test summary report

Deliverables

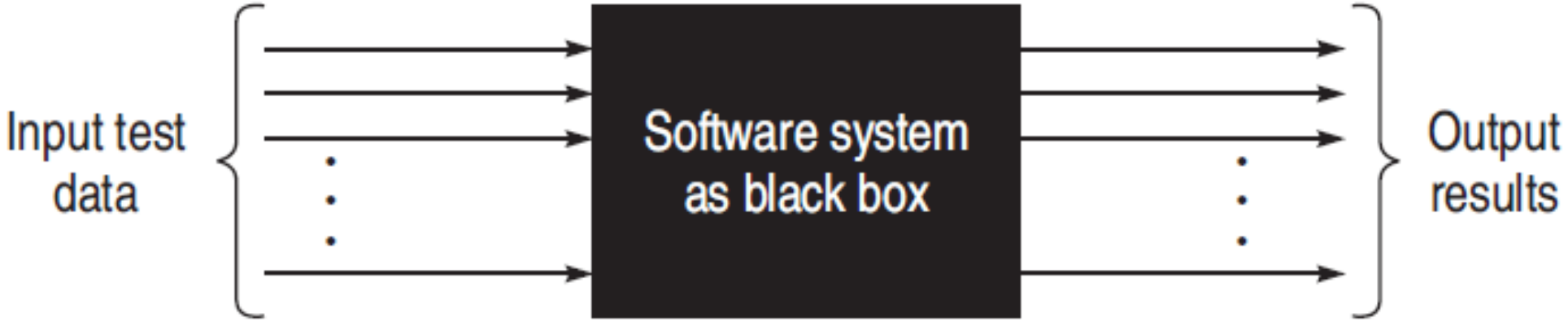
- Test Closure report
- Test metrics

TESTING CATEGORY & TECHNIQUES

Testing Category	Techniques
Dynamic testing: Black-Box	Boundary value analysis, Equivalence class partitioning, State table-based testing, Decision table-based testing, Cause-effect graphing technique, Error guessing.
Dynamic testing: White-Box	Basis path testing, Graph matrices, Loop testing, Data flow testing, Mutation testing.
Static testing	Inspection, Walkthrough, Reviews.
Validation testing	Unit testing, Integration testing, Function testing, System testing, Acceptance testing, Installation testing.
Regression testing	Selective retest technique, Test prioritization.

BLACK-BOX TESTING

- ✓A “black-box” means that one cannot see the internals of the system or inside the box.
- ✓A test or a testing activity that is considered “black-box”, only checks that **given some input to the SW system, there is expected output.**
- ✓This technique considers only the functional requirements of the software or module.



- ✓An example of such a test is User Acceptance Testing (UAT) where the person doing the testing has no access to the code, is not concerned how the system was implemented, but rather is testing that the system meets the user requirements and works as expected for the typical input that the system needs to work with.

LOOK AT EXAMPLE

Let's assume that we are building a small program that employees at a company can use to determine how much their insurance will cost. Here are the requirements for the application.

Requirement #1 An employee must have health insurance to get vision or dental insurance.

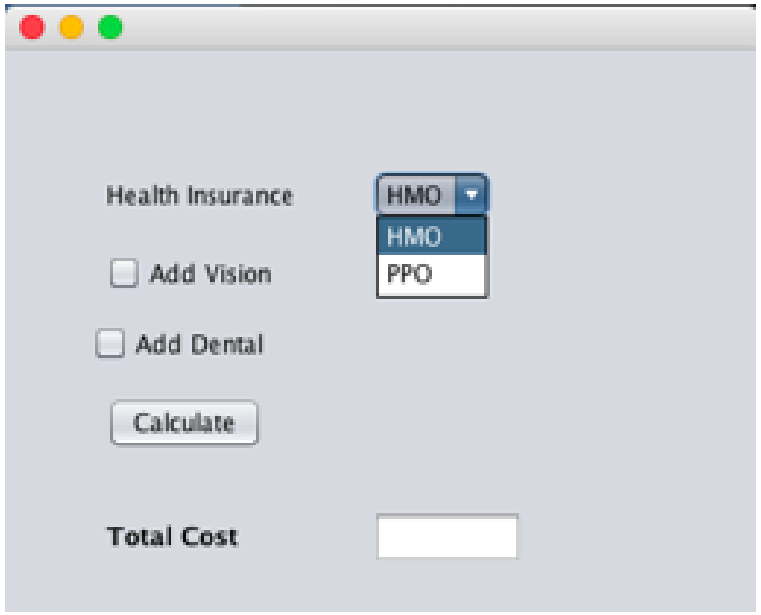
Requirement #2 There are two types of health insurance: HMO costs $100 a month; PPO costs $150 a month.

Requirement #3 Vision costs $25 a month

Requirement #4 Dental costs $20 a month

LOOK AT EXAMPLE

The user interface might look like this. The user selects a health insurance plan and might check boxes to get vision or dental insurance. Then the user clicks the Calculate button. The calculated cost appears in the text field beside the label “Total Cost”.

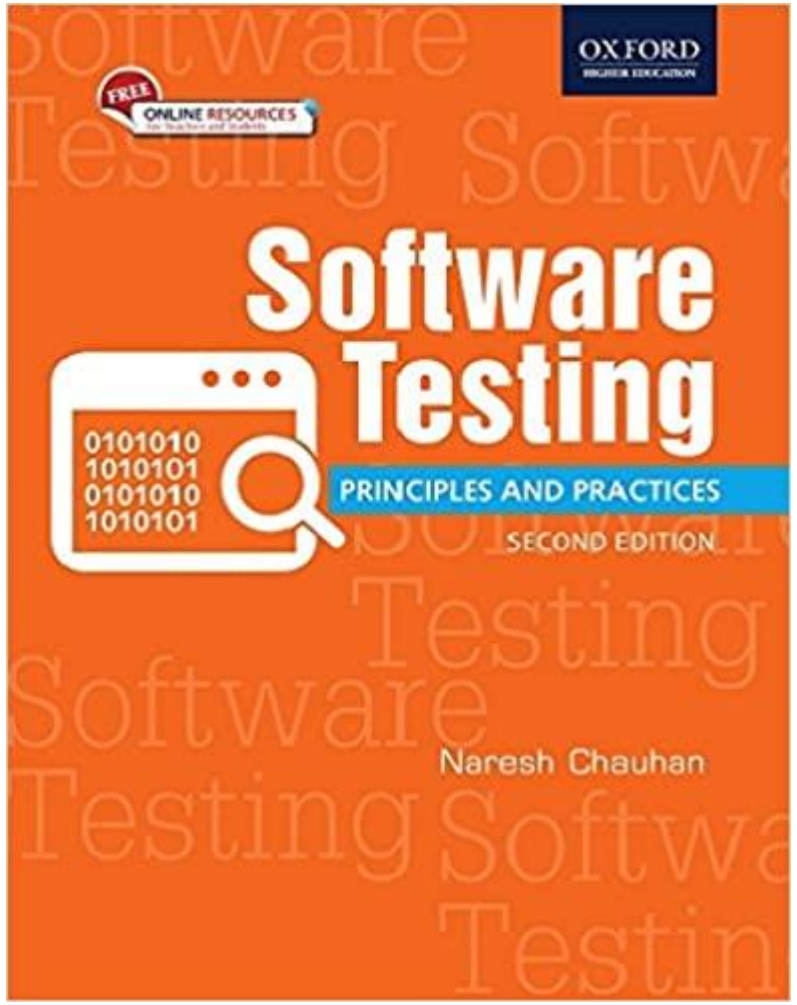


BLACK-BOX TESTING

- ✓ Black-box testing attempts to find errors in the following categories:
  - To test the modules independently
  - To test the functional validity of the software so that incorrect or missing functions can be recognized
  - To look for interface errors
  - To test the system behavior and check its performance
  - To test the maximum load or stress on the system
  - To test the software such that the user/customer accepts the system within defined acceptable limits

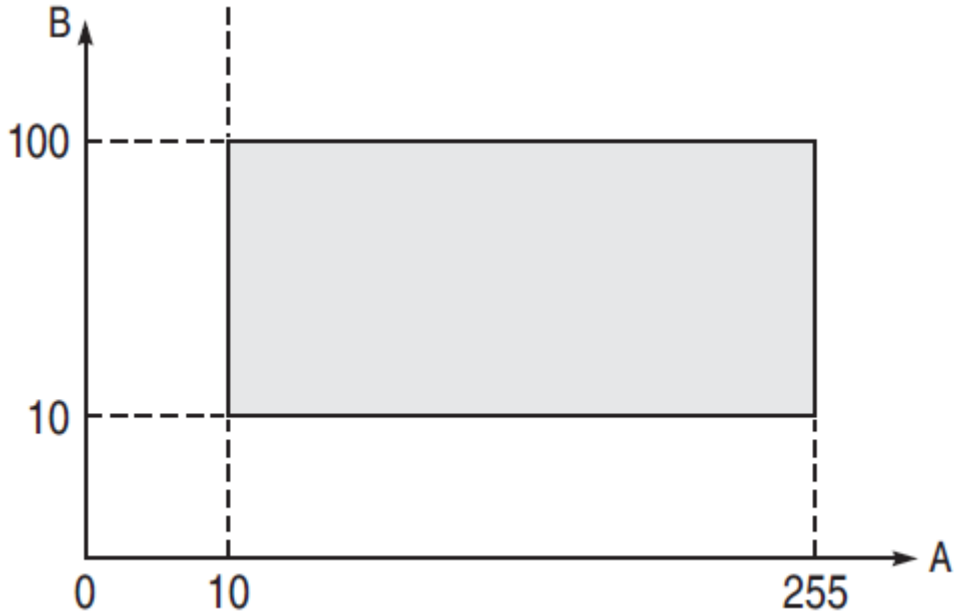
Black-Box Testing : Techniques

1. Boundary Value Analysis (BVA)
2. Equivalence Class Testing
3. State Table-Based Testing
4. Decision Table-Based Testing
5. Cause-Effect Graphing Based Testing
6. Error Guessing



BOUNDARY VALUE ANALYSIS (BVA)

- ✓ Test cases designed with boundary input values have a high chance to find errors.
- ✓ means that most of the failures crop up due to boundary values.
- ✓ boundary means the maximum or minimum value taken by the input domain.
- ✓ For example, if A is an integer between 10 and 255, then boundary checking can be on 10(9,10,11) and on 255(256,255,254). Similarly, B is another integer variable between 10 and 100, then boundary checking can be on 10(9,10,11) and 100(99,100,101)



Boundary Value Analysis (BVA)

BVA offers several methods to design test cases:

- ✓ Boundary Value Checking (BVC)
- ✓ Robustness Testing Method
- ✓ Worst-case Testing Method

BVA: Boundary Value Checking

- ✓ the 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:

  (a) Minimum value (Min)

  (b) Value just above the minimum value (Min+)

  (c) Maximum value (Max)

  (d) Value just below the maximum value (Max−).

BVA: Boundary Value Checking

Let us take the example of two variables, A and B. If we consider all the above combinations with nominal values, then following test cases can be designed:

1. A_{nom}, B_{min}
2. A_{nom}, B_{min+}
3. A_{nom}, B_{max}
4. A_{nom}, B_{max-}
5. A_{min}, B_{nom}
6. A_{min+}, B_{nom}
7. A_{max}, B_{nom}
8. A_{max-}, B_{nom}
9. A_{nom}, B_{nom}

It can be generalized that for ***n*** variables in a module, ***4n* + 1** test cases can be designed with boundary value checking method.

BVA: Robustness Testing Method

- ✓ Is an extension of BVC such that boundary values are exceeded as:
- ✓ The variable at its extreme value can be selected at:

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

  (b) A value just less than Minimum value **(Min−)**

So if we consider the previous example then we can add following cases :

10. A_{max+}, B_{nom}
11. A_{min-}, B_{nom}
12. A_{nom}, B_{max+}
13. A_{nom}, B_{min-}

It can be generalized that for ***n*** variables in a module, **6*n* + 1** test cases can be designed with Robustness Testing method.

BVA: Worst-Case Testing Method

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

So if we consider the previous example then we can add following cases to the list of 9 test cases designed in BVC as:

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-}

It can be generalized that for ***n*** variables in a module, $\mathbf{5^n}$ test cases can be designed with Worst-Case Testing method.

EXAMPLE

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

Solution:

(a)Test cases using BVC

Total # of test case will be 4n+1 = 5 [as n=1 here]

Min value = 1
Min^{+} value = 2
Max value = 100
Max^{-} value = 99
Nominal value = 50–55

min and max values

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

Test cases

EXAMPLE

(b) Test cases using Robust testing

Total # of test case will be 6n+1 = 7 [as n=1 here]

Min value = 1
Min^- value = 0
Min^+ value = 2
Max value = 100
Max^- value = 99
Max^+ value = 101
Nominal value = 50–55

min and max values

Test Case ID	Integer Variable	Expected Output
1	0	Invalid input
2	1	Not a prime number
3	2	Prime number
4	100	Not a prime number
5	99	Not a prime number
6	101	Invalid input
7	53	Prime number

Test cases

EXAMPLE

(c) Test cases using worst-case testing

Total # of test case will be $5^n = 5$ [as n=1 here]

So, the number of test cases will be same as BVC.

EXAMPLE

A program computes ab where a lies in the range [1,10] and b within [1,5].Design test cases for this program using **BVC**, **robust testing**, and **worst-case testing** methods.

Solution:

(a)Test cases using BVC : Total # of test case will be 4n+1 = 9 [as n=2]

	a	b
Min value	1	1
Min^+ value	2	2
Max value	10	5
Max^- value	9	4
Nominal value	5	3

min and max values

Test Case ID	a	b	Expected Output
1	1	3	1
2	2	3	8
3	10	3	1000
4	9	3	729
5	5	1	5
6	5	2	25
7	5	4	625
8	5	5	3125
9	5	3	125

Test cases

EXAMPLE

(b) Test cases using Robust testing: Total # of test case will be 6n+1 = 13 [as n=2]

	a	b
Min value	1	1
Min^- value	0	0
Min^+ value	2	2
Max value	10	5
Max^+ value	11	6
Max^- value	9	4
Nominal value	5	3

min and max values

Test Case ID	a	b	Expected output
1	0	3	Invalid input
2	1	3	1
3	2	3	8
4	10	3	1000
5	11	3	Invalid input
6	9	3	729
7	5	0	Invalid input
8	5	1	5
9	5	2	25
10	5	4	625
11	5	5	3125
12	5	6	Invalid input
13	5	3	125

Test cases

EXAMPLE

(c) Test cases using worst-case testing

Total # of test case will be $5^n = 25$ [as n=2]

Test Case ID	a	b	Expected Output
19	9	4	6561
20	9	5	59049
21	10	1	10
22	10	2	100
23	10	3	1000
24	10	4	10000
25	10	5	100000
17	9	2	81
18	9	3	729

	a	b
Min value	1	1
Min^+ value	2	2
Max value	10	5
Max^- value	9	4
Nominal value	5	3

min and max values

DECISION TABLE-BASED TESTING

- ✓ Decision table is another useful method to represent the information in a tabular method.
- ✓ It has the specialty to consider complex combinations of input conditions and resulting actions..

Formation of decision Table:

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

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

DECISION TABLE-BASED TESTING

Guidelines to develop a decision table:

- ✓ List all actions that can be associated with a specific procedure (or module).
- ✓ List all conditions (or decision made) during execution of the procedure.
- ✓ Associate specific sets of conditions with specific actions, eliminating impossible combinations of conditions; alternatively, develop every possible permutation of conditions.
- ✓ Define rules by indicating what action occurs for a set of conditions.

DECISION TABLE-BASED TESTING

For designing test cases from a decision table, following interpretations should be done:

- ✓ 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.
- ✓ If there are k rules over n binary conditions, there are at least k test cases and at the most 2n 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 test cases using decision table testing..

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		

DECISION TABLE-BASED TESTING

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

DECISION TABLE-BASED TESTING

Example:

A university is admitting students in a professional course subject to the following conditions:

(a) Marks in Java ≥ 70

(b) Marks in C++ ≥ 60

(c) Marks in OOAD ≥ 60

(d) Total in all three subjects ≥ 220 OR Total in Java and C++ ≥ 150

If the aggregate mark of an eligible candidate is more than 240, he will be eligible for scholarship course, otherwise he will be eligible for normal course.

The program reads the marks in the three subjects and generates the following outputs:

(i) Not eligible

(ii) Eligible for scholarship course

(iii) Eligible for normal course

Design test cases for this program using decision table testing.

DECISION TABLE-BASED TESTING

Solution:

	ENTRY									
	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10
C1: marks in Java ≥ 70	T	T	T	T	F	I	I	I	T	T
C2: marks in C++ ≥ 60	T	T	T	T	I	F	I	I	T	T
C3: marks in OOAD ≥ 60	T	T	T	T	I	I	F	I	T	T
C4:Total in three subjects ≥ 220	T	F	T	F	I	I	I	F	T	T
C5:Total in Java & C++ ≥ 150	F	T	F	T	I	I	I	F	T	T
C6:Aggregate marks > 240	F	F	T	T	I	I	I	I	F	T
A1: Eligible for normal course	X	X							X	
A2: Eligible for scholarship course			X	X						X
A3: Not eligible					X	X	X	X		

DECISION TABLE-BASED TESTING

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

Test Case ID	Java	C++	OOAD	Aggregate Marks	Expected Output
1	70	75	60	224	Eligible for normal course
2	75	75	70	220	Eligible for normal course
3	75	74	91	242	Eligible for scholarship course
4	76	77	89	242	Eligible for scholarship course
5	68	78	80	226	Not eligible
6	78	45	78	201	Not eligible
7	80	80	50	210	Not eligible
8	70	72	70	212	Not eligible
9	75	75	70	220	Eligible for normal course
10	76	80	85	241	Eligible for scholarship course

Dynamic Testing:
White-Box Testing Techniques

WHITE-BOX TESTING

- ✓A “white-box” or “clear-box” mean that one can see the internals of the system, or in other words, see inside the box.
- ✓So a test or a testing activity that is considered “white-box” is one where the tester has access to the code and uses that knowledge to write tests or conduct the testing activity.
- ✓example of “white-box” test is unit test where developers write test cases, often automated, to verify each unit of code such as a function, method, or class.
- ✓white-box testing ensures that the internal parts of the software are adequately tested.

WHITE-BOX TESTING : NECESSITY

- ✓ white-box testing techniques are used for testing the module for initial stage testing.
- ✓ Since white-box testing is complementary to black-box testing, there are categories of bugs which can be revealed by white-box testing, but not through black-box testing.
- ✓ Errors which have come from the design phase will also be reflected in the code, therefore we must execute white-box test cases for verification of code (unit verification).
- ✓ Some typographical errors are not observed and go undetected and are not covered by black-box testing techniques.
- ✓ White-box testing explores and confirms the testing of logical paths.

White-Box Testing : Logic Coverage Criteria

basic forms of logic coverage:

- ✓ Statement Coverage : is assumed that if all the statements of the module are executed once, every bug will be notified.
- ✓ Decision or Branch Coverage: each branch direction must be traversed at least once.
- ✓ Condition Coverage: Condition coverage states that each condition in a decision takes on all possible outcomes at least once.

STATEMENT COVERAGE

What is Statement Coverage?

is a white box test design technique which involves execution of all the executable statements in the source code at least once.

$$\textbf{Statement Coverage} = \frac{\textit{Number of executed statements}}{\textit{Total number of statements}} \times 100$$

The goal of Statement coverage is to cover all the possible statement in the code.

Unless a statement is executed, we have no way of knowing if an error exists in that statement.

Let's understand this with an example, how to calculate statement coverage.

EXAMPLE STATEMENT COVERAGE

```
scanf ("%d", &x);
scanf ("%d", &y);
while (x != y)
{
      if (x > y)
            x = x - y;
      else
            y = y - x;
 }
 printf ("x = ", x);
 printf ("y = ", y);
```

If we want to cover every statement of this code, then the following test cases must be designed:

Test case 1: x = y = n, where n is any number

Test case 2: x = n, y = n', where n and n' are different numbers.

Test case 1 just skips the while loop

Test case 2, the loop is also executed. However, every statement inside the loop is not executed.

So, two more cases need to be designed to cover all statements: designed:

Test case 3: x > y

Test case 4: x < y

Test case 3 and 4 are sufficient to execute all the statements in the code. But, if we execute only test case 3 and 4, then conditions and paths in test case 1 will never be tested and errors will go undetected.

EXAMPLE STATEMENT COVERAGE

```
1 Prints (int a, int b) {
2     int result = a+ b;
3     If (result> 0)
4          Print ("Positive", result)
5     Else
6          Print ("Negative", result)
7     }
```

Take input of two values of a and b

Find the sum of these two values.

If the sum is greater than 0, then print "This is the positive result."

If the sum is less than 0, then print "This is the negative result."

Now, let's see the two different scenarios and calculation of the percentage of Statement Coverage for given source code.

EXAMPLE STATEMENT COVERAGE

Scenario 1:
If a = 3, b = 9

```
1  Prints (int a, int b) {
2  int result = a+ b;
3  If (result> 0)
4      Print ("Positive", result)
5  Else
6      Print ("Negative", result)
7  }
```

The statements marked in yellow color are those which are executed as per the scenario.
Number of executed statements = 5, Total number of statements = 7
Statement Coverage: 5/7 = 71%

EXAMPLE STATEMENT COVERAGE

Scenario 2:
If a = -3, b = -9

```
1  Prints (int a, int b) {
2  int result = a+ b;
3  If (result> 0)
4      Print ("Positive", result)
5  Else
6      Print ("Negative", result)
7  }
```

The statements marked in yellow color are those which are executed as per the scenario.
Number of executed statements = 6, Total number of statements = 7
Statement Coverage: 6/7 = 85.20%

BRANCH COVERAGE

- ✓ “Branch” in a programming language is like the “IF statements”. An IF statement has two branches: True and False.
- ✓ So in Branch coverage (also called Decision coverage), we validate whether each branch is executed at least once.
- ✓ In case of an “IF statement”, there will be two test conditions:
  - One to validate the true branch and,
  - Other to validate the false branch.
- ✓ Hence, in theory, Branch Coverage is a testing method which is when executed ensures that each and every branch from each decision point is executed.

EXAMPLE BRANCH COVERAGE

```
scanf ("%d", &x);
scanf ("%d", &y);
while (x != y)
{
      if (x > y)
            x = x - y;
      else
            y = y - x;
 }
 printf ("x = ", x);
 printf ("y = ", y);
```

while and *if* statements have two outcomes: True and False. So test cases must be designed such that both outcomes for *while* and *if* statements are tested

Test case 1: x = y
Test case 2: x != y
Test case 3: x < y
Test case 4: x > y

CONDITION COVERAGE

consider the statement: while ((I ≤5) && (J < COUNT))

In this loop statement, two conditions are there. So test cases should be designed such that both the conditions are tested for True and False outcomes.

The following test cases are designed:

Test case 1: I ≤ 5, J < COUNT

Test case 2: I ≤ 5, J > COUNT

Condition coverage in a decision does not mean that the decision has been covered. If the decision

if (A && B)

multiple condition coverage requires that we should write sufficient test cases such that all possible combinations of condition outcomes in each decision and all points of entry are invoked at least once

Test case 1: A = True, B = True
Test case 2: A = True, B = False
Test case 3: A = False, B = True
Test case 4: A = False, B = False

BASIS PATH TESTING

- ✓ Based on the control structure, a flow graph is prepared and all the possible paths can be covered and executed during testing..
- ✓ is a more general criterion as compared to other coverage criteria and useful for detecting more errors.
- ✓ The objective of basis path testing is to define the number of independent paths, so the number of test cases needed can be defined explicitly to maximize test coverage.
- ✓ Problem: programs that contain loops may have an infinite number of possible paths and it is not practical to test all the paths.

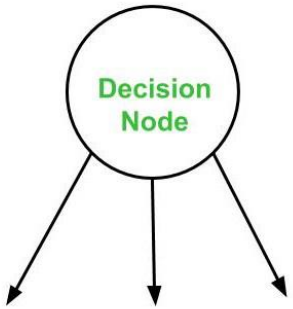
BASIS PATH TESTING

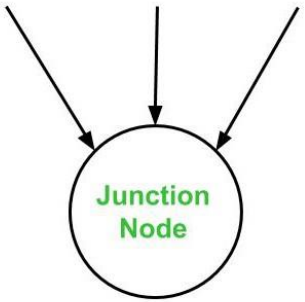
Steps for Basis Path testing

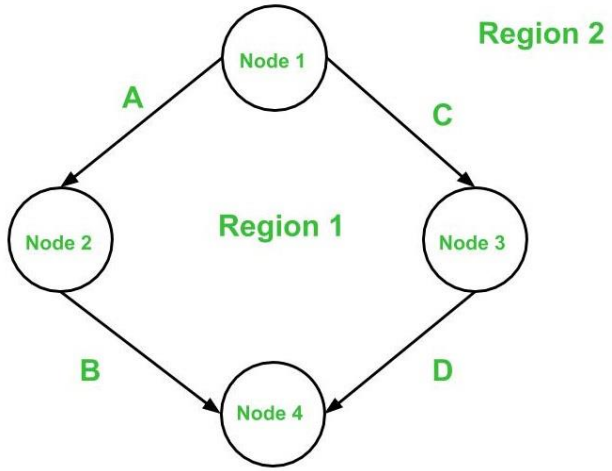
- ✓ Draw a **control flow graph** (to determine different program paths)
- ✓ Calculate **Cyclomatic complexity** (metrics to determine the number of independent paths)
- ✓ Find a basis set of paths (**independent path**)
- ✓ Generate test cases to exercise each path

BASIS PATH TESTING : CONTROL FLOW GRAPH

- ✓ A control flow graph (or simply, flow graph) is a directed graph which represents the control structure of a program or module.
- ✓ A control flow graph (V, E) has V number of **nodes/vertices** and E number of **edges** in it. A control graph can also have :
  - **Junction Node** – a node with more than one arrow entering it.
  - **Decision Node** – a node with more then one arrow leaving it.
  - **Region** – area bounded by edges and nodes (area outside the graph is also counted as a region.).







BASIS PATH TESTING : CYCLOMATIC COMPLEXITY

- ✓ The cyclomatic complexity V(G) is said to be a measure of the logical complexity of a program. It can be calculated using three different formulae :

Formula based on edges and nodes

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

Where,
e is number of edges,
n is number of vertices,
P is number of connected components.

Formula based on Decision Nodes

V(G) = d + P

where,
d is number of decision nodes,
P is number of connected nodes.

Formula based on Regions

V(G) = # of regions (R) in the graph

Calculating the number of decision nodes for Switch-Case/Multiple If-Else

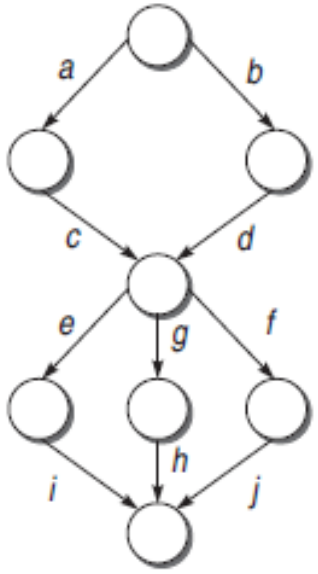
- ✓ If a decision node has exactly two arrows leaving it, then it is counted as one decision node. However, if there are more than 2 arrows leaving a decision node, it is computed using this formula :

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

BASIS PATH TESTING : INDEPENDENT PATH

- ✓ An independent path in the control flow graph is the one which introduces at least one new edge that has not been traversed before the path is defined

- ✓ cyclomatic complexity gives the number of independent paths present in a flow graph.



In the graph, there are six possible paths: *acei, acgh, acfh, bdei, bdgh, bdfj.*
In this case, of the **six** possible paths, only **four** are independent, as the other two are always a linear combination of the other four paths.

If we calculate the cyclometric complexity, it will be same.

BASIS PATH TESTING : EXAMPLE

```
     main()
     {
          int number;
          int fact();
1.        clrscr();
2.        printf("Enter the number whose factorial is to be found out");

3.        scanf("%d", &number);
4.        if(number <0)
5.           printf("Facorial cannot be defined for this number);
6.        else
7.           printf("Factorial is %d", fact(number));
8.   }
```

```
     int fact(int number)
     {
          int index;
1.        int product =1;
2.        for(index=1; index<=number; index++)
3.           product = product * index;
4.        return(product);
5.   }
```

A program for calculating the factorial of a number. It consists of main() program and the module fact().

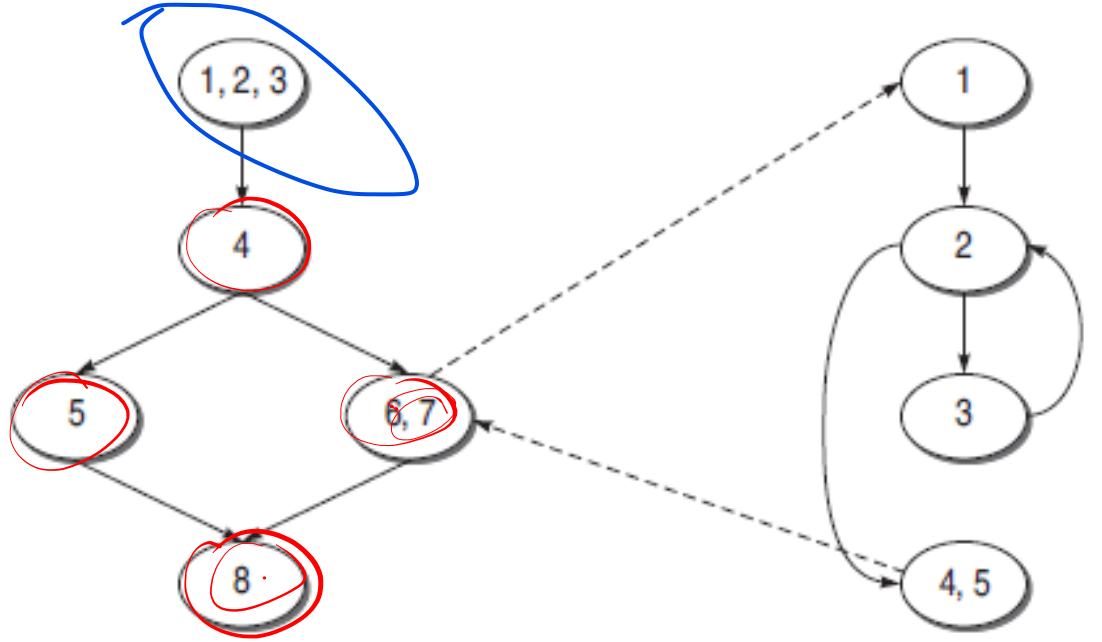
How to Calculate the Cyclomatic complexity?

If a program P consist of multiple 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

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

BASIS PATH TESTING : EXAMPLE

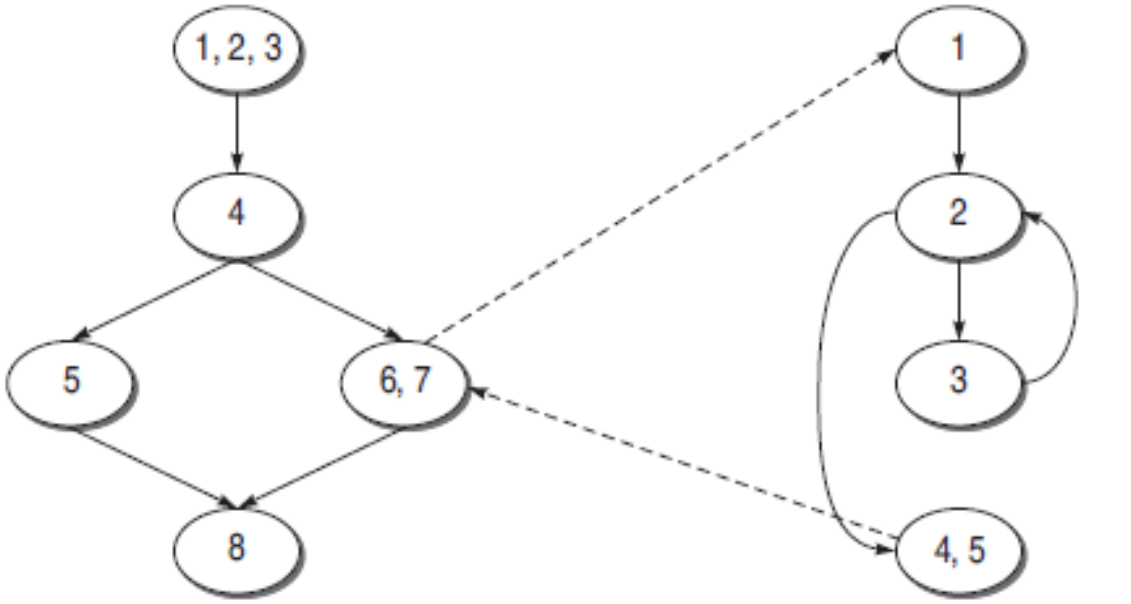
```
    main()
    {
        int number;
        int fact();
1.      clrscr();
2.      printf("Enter the number whose factorial is to be found out");

3.      scanf("%d", &number);
4.      if(number <0)
5.          printf("Facorial cannot be defined for this number);
6.      else
7.          printf("Factorial is %d", fact(number));
8.  }

    int fact(int number)
    {
        int index;
1.      int product =1;
2.      for(index=1; index<=number; index++)
3.          product = product * index;
4.      return(product);
5.  }
```



(a) Flow graph for main ()

(b) Flow graph for fact ()

BASIS PATH TESTING : EXAMPLE



(a) Flow graph for main ()

(b) Flow graph for fact ()

Cyclomatic complexity of main()

(a) $V(M) = e - n + 2p$
$= 5 - 5 + 2$
$= 2$

(b) $V(M) = d + p$
$= 1 + 1$
$= 2$

(c) $V(M)$ = Number of regions = 2

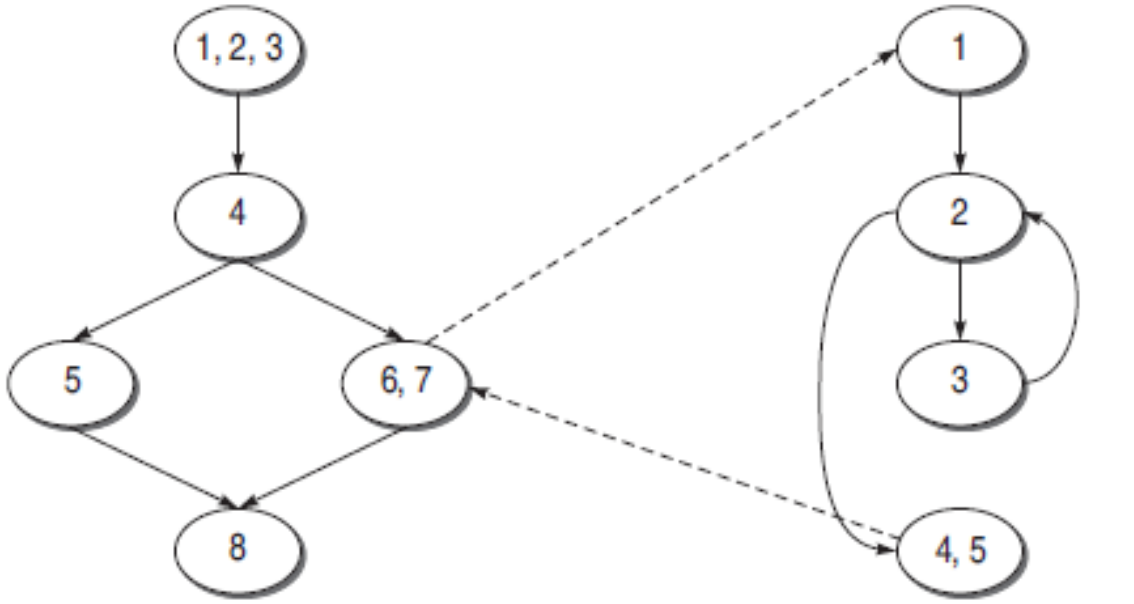
Cyclomatic complexity of fact()

(a) $V(R) = e - n + 2p$
$= 4 - 4 + 2$
$= 2$

(b) $V(R)$ = Number of predicate nodes + 1
$= 1 + 1$
$= 2$

(c) $V(R)$ = Number of regions = 2

BASIS PATH TESTING : EXAMPLE



(a) Flow graph for main ()

(b) Flow graph for fact ()

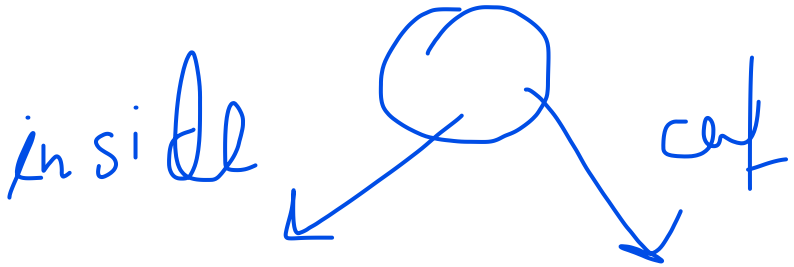
Cyclomatic complexity of the whole graph considering the full program

(a) V (G) = e – n + 2p
= 9 – 9 + 2 * 2
= 4
= V (M) + V (R)

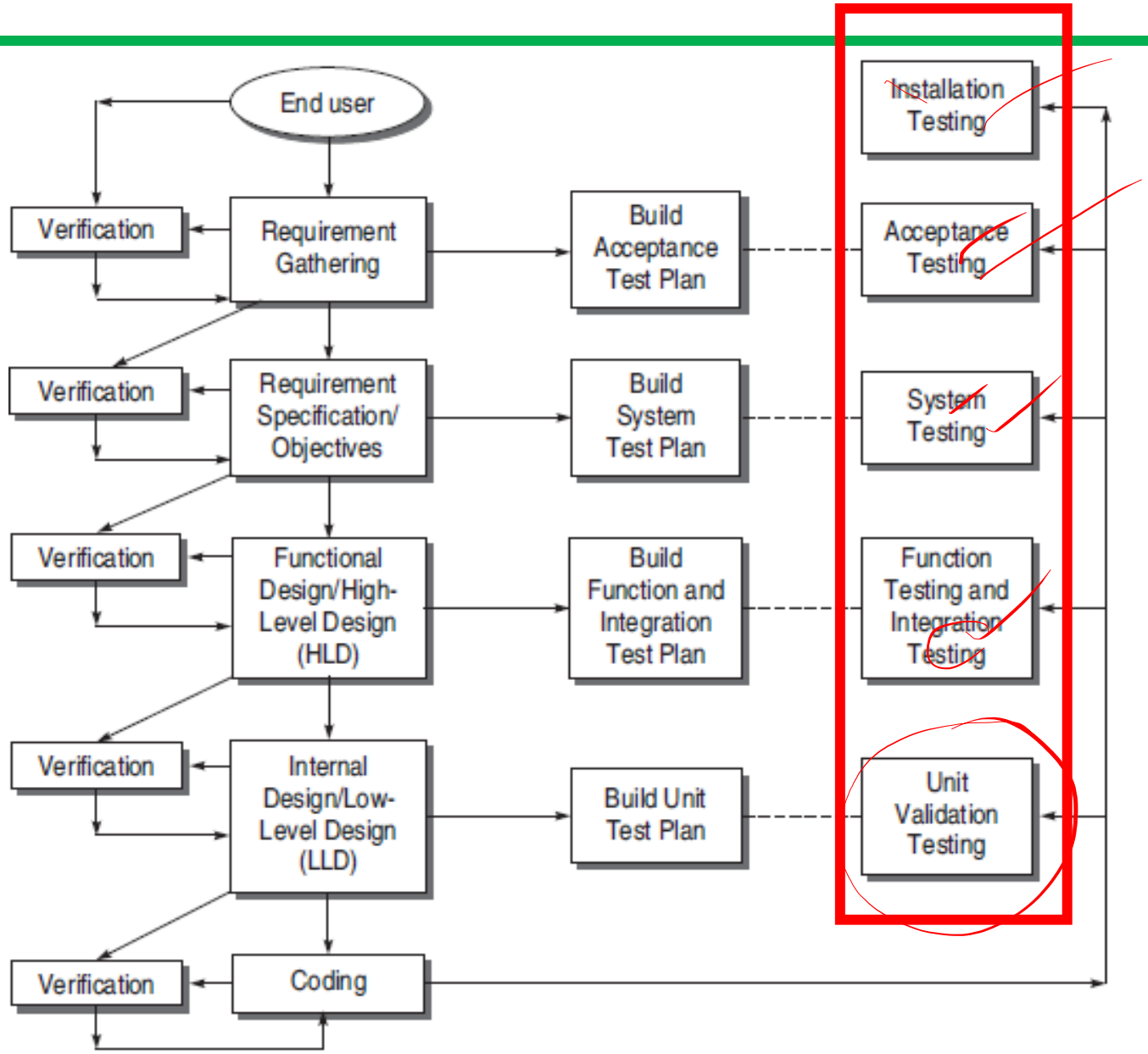
(b) V (G) = d + p
= 2 + 2
= 4
= V (M) + V (R)

(c) V (G) = Number of regions
= 4
= V (M) + V (R)

Pathao parcel service announces overnight delivery anywhere inside Dhaka and two days delivery scheme for outside Dhaka (within Bangladesh). The delivery fee is 2 Tk per gram for letters in Dhaka (5 Tk outside of Dhaka), and 10 Tk per gram for parcels in Dhaka (15 Tk outside of Dhaka). The maximum weight they deliver is 200 grams for a letter and 2000 grams for a parcel.



VALIDATION ACTIVITIES



INTEGRATION TESTING

In **integration testing** smaller units are integrated into larger units and larger units into the overall system.

This differs from unit testing in that units are no longer tested independently but in groups, the focus shifting from the individual units to the interaction between them.

The purpose of integration testing is to verify the functional, performance, and reliability between the modules that are integrated.

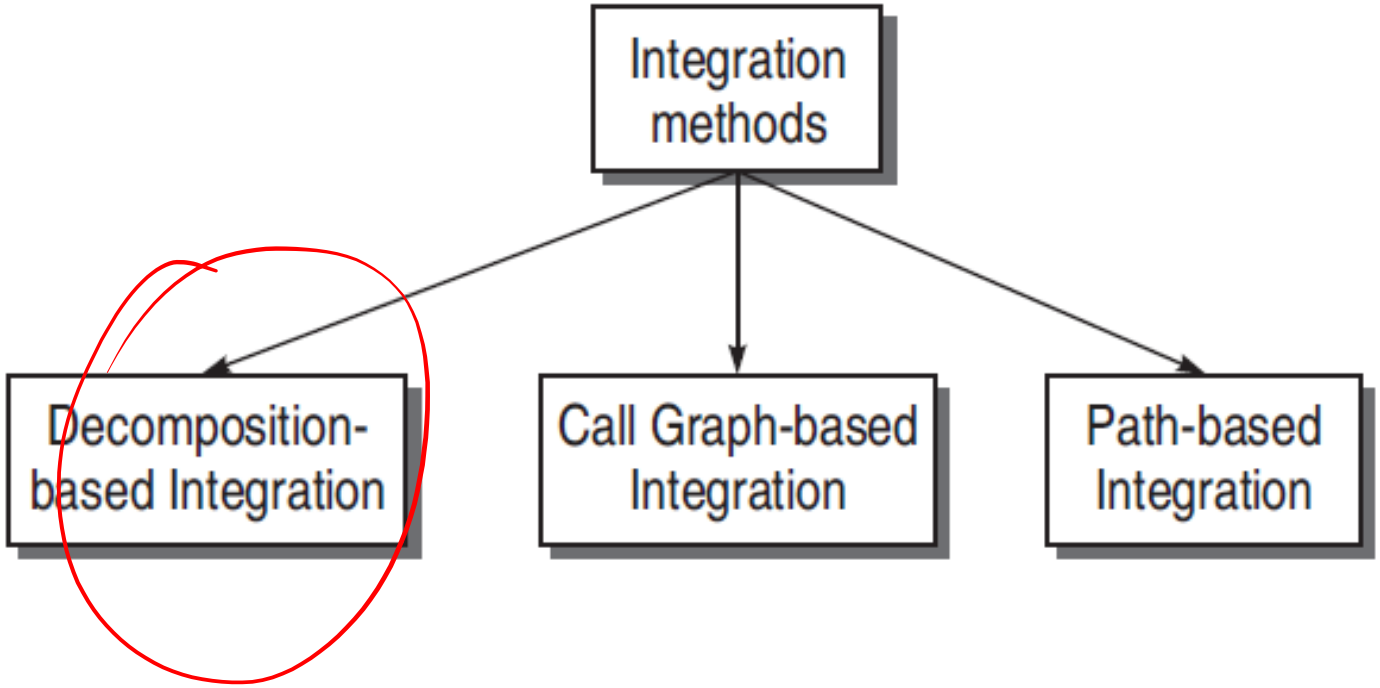
INTEGRATION TESTING

Why do we do integration testing?

- ✓ Unit tests only test the unit in isolation
- ✓ Many failures result from faults in the interaction of subsystems
- ✓ Often many Off-the-shelf components are used that cannot be unit tested
- ✓ Without integration testing the system test will be very time consuming
- ✓ Failures that are not discovered in integration testing will be discovered after the system is deployed and can be very expensive.

INTEGRATION TESTING

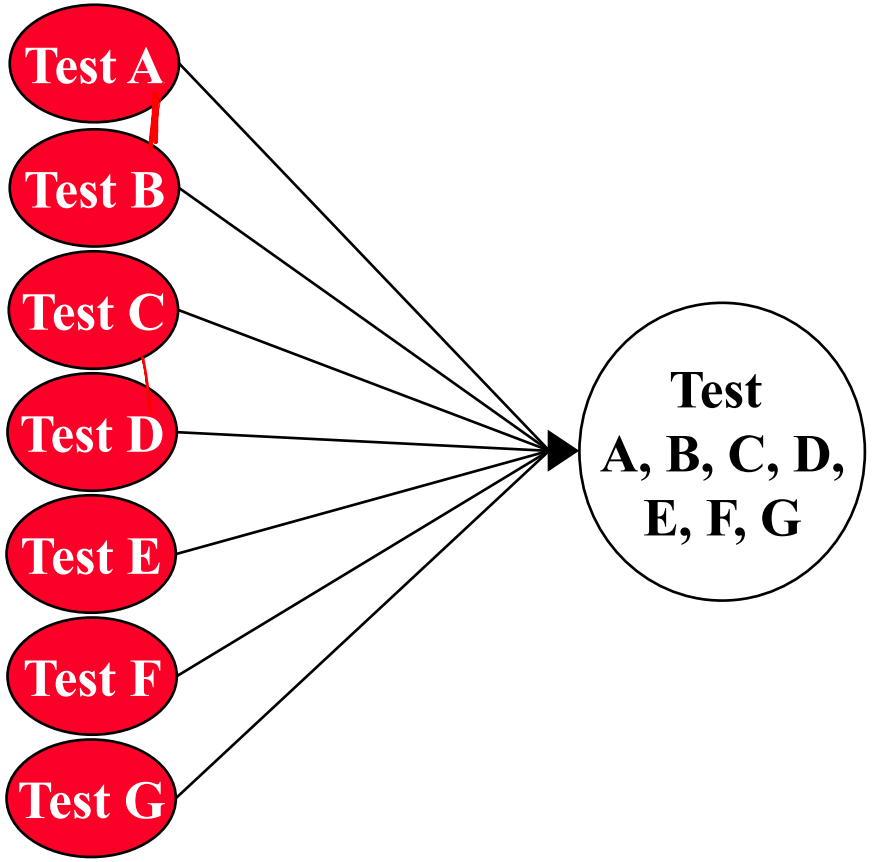
There are three approaches for integration testing.

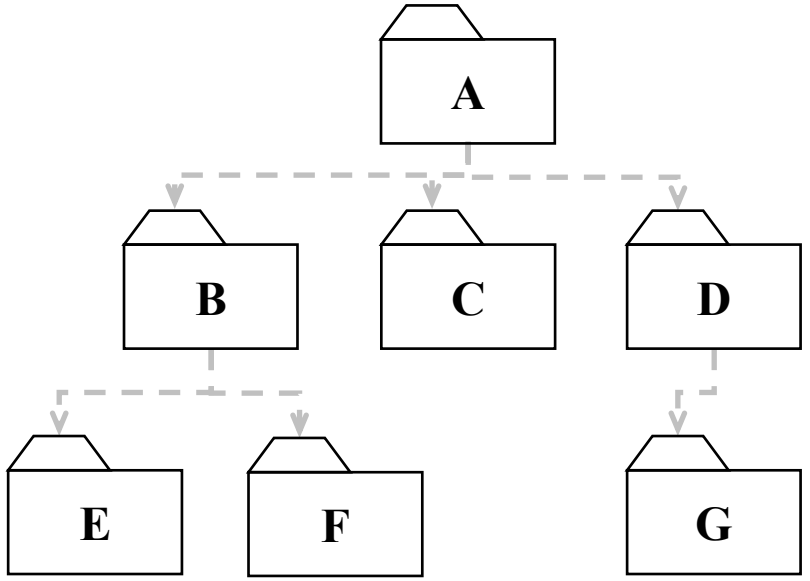


INTEGRATION TESTING : DECOMPOSITION BASED

Big-Bang Approach

- ✓ This unit tests each of the subsystems, and then does one massive integration test, in which all the subsystems are immediately tested together.
- ✓ Don‘t try this!! Why: The interfaces of each of the subsystems have not been tested yet.

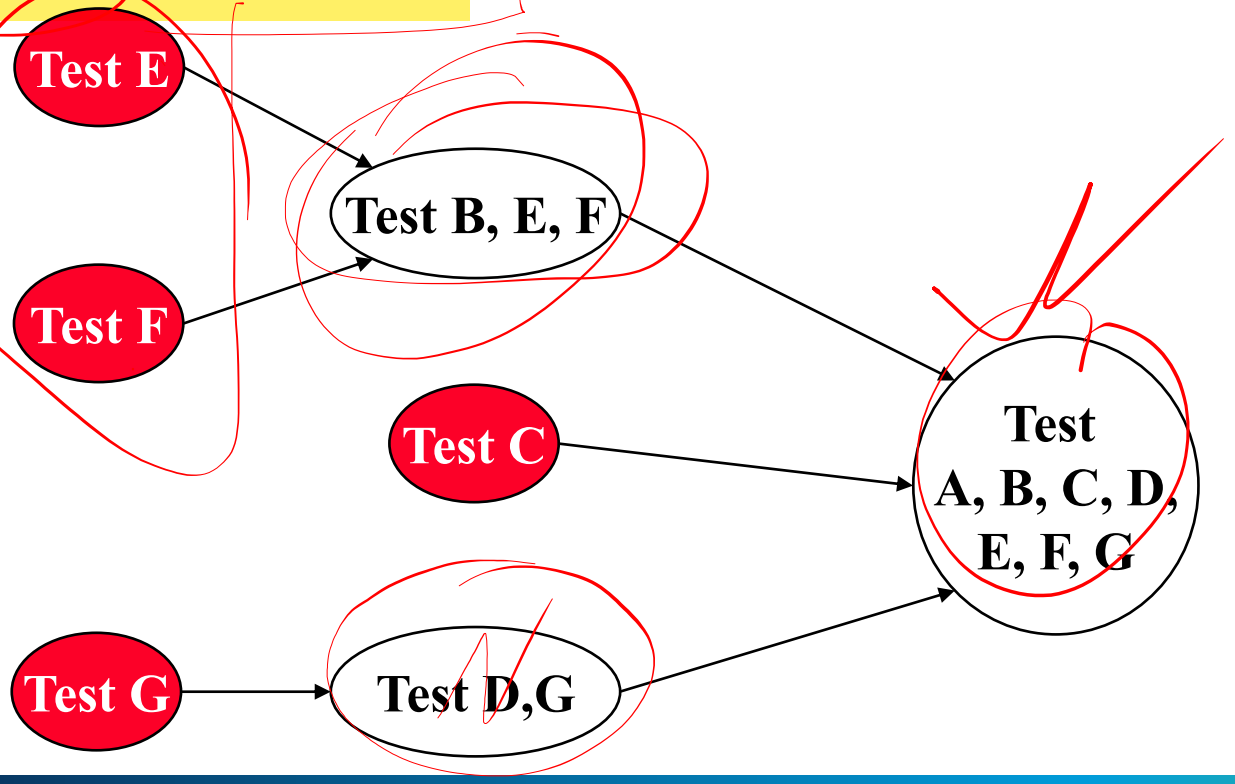


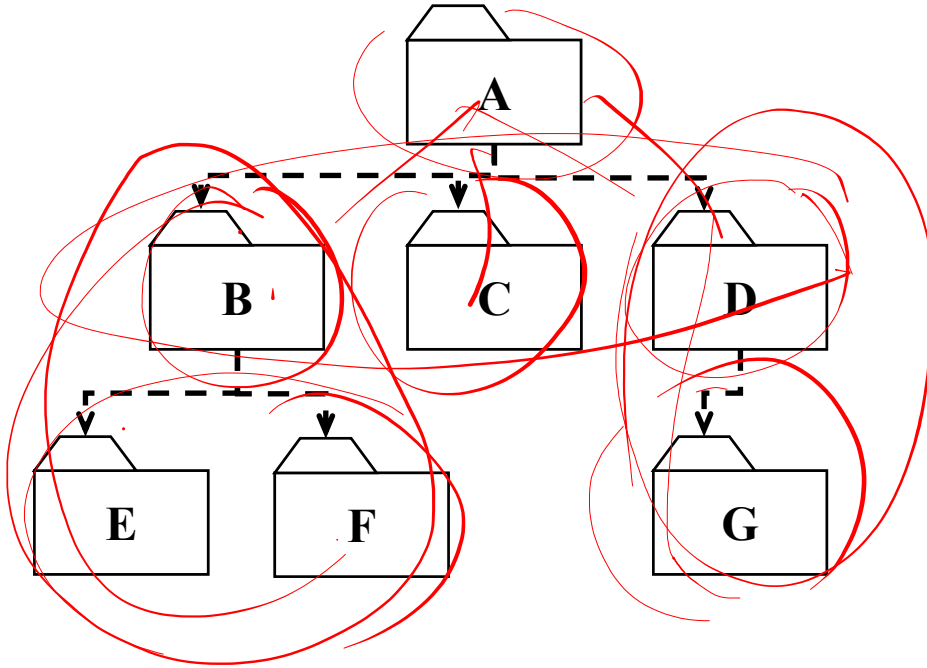


INTEGRATION TESTING : DECOMPOSITION BASED

Bottom-up Testing Strategy

- ✓ The subsystems in the lowest layer of the call hierarchy are tested individually
- ✓ Then the next subsystems are tested that call the previously tested subsystems
- ✓ This is repeated until all subsystems are included
- ✓ Drivers are needed.

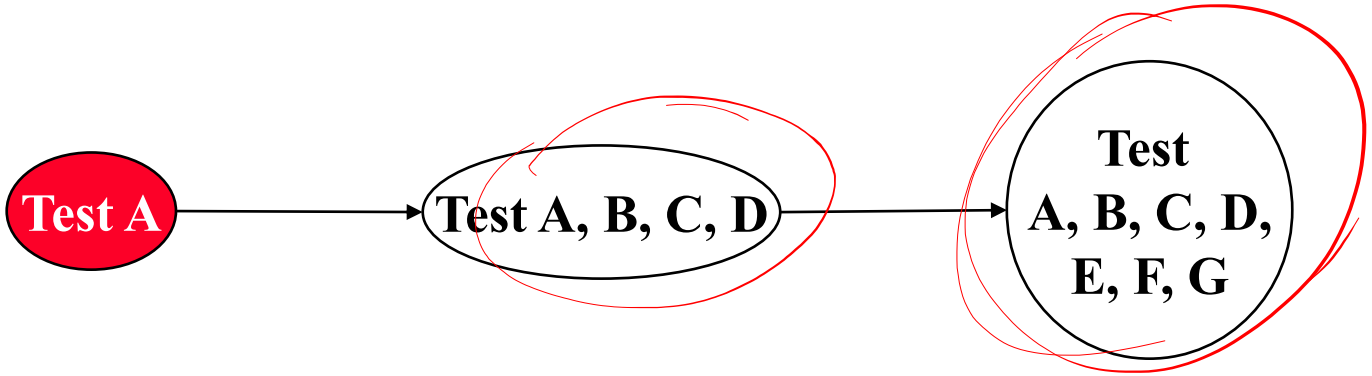


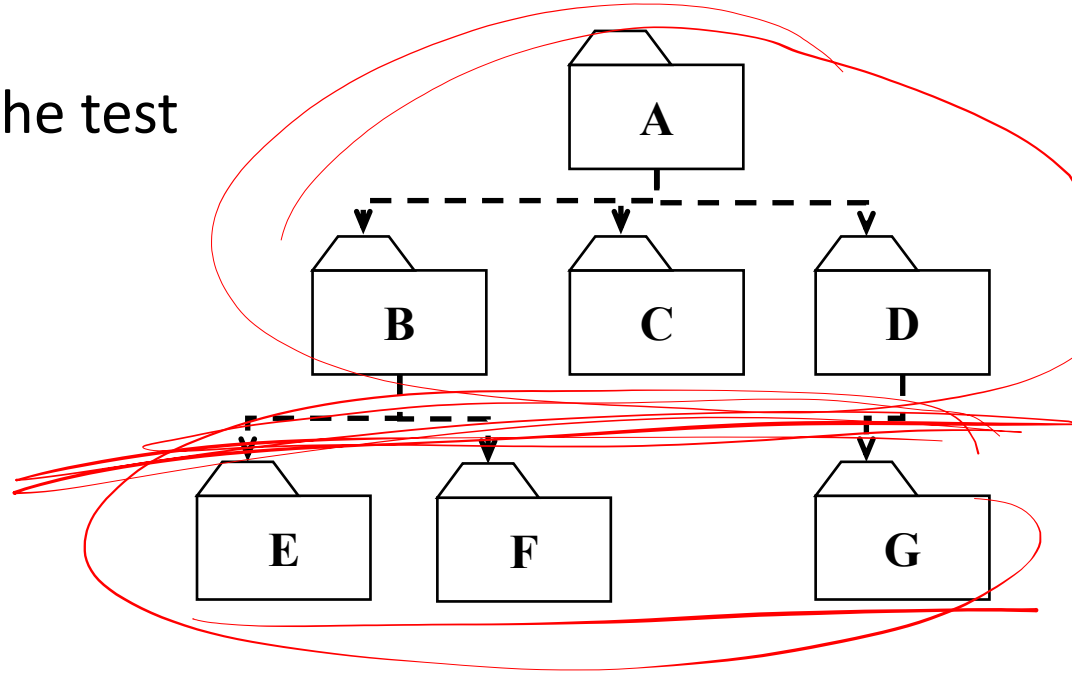


INTEGRATION TESTING : DECOMPOSITION BASED

Top-Down Testing Strategy

- ✓ Test the top layer or the controlling subsystem first
- ✓ Then combine all the subsystems that are called by the tested subsystems and test the resulting collection of subsystems
- ✓ Do this until all subsystems are incorporated into the test
- ✓ Stubs are needed to do the testing.





INTEGRATION TESTING : DECOMPOSITION BASED

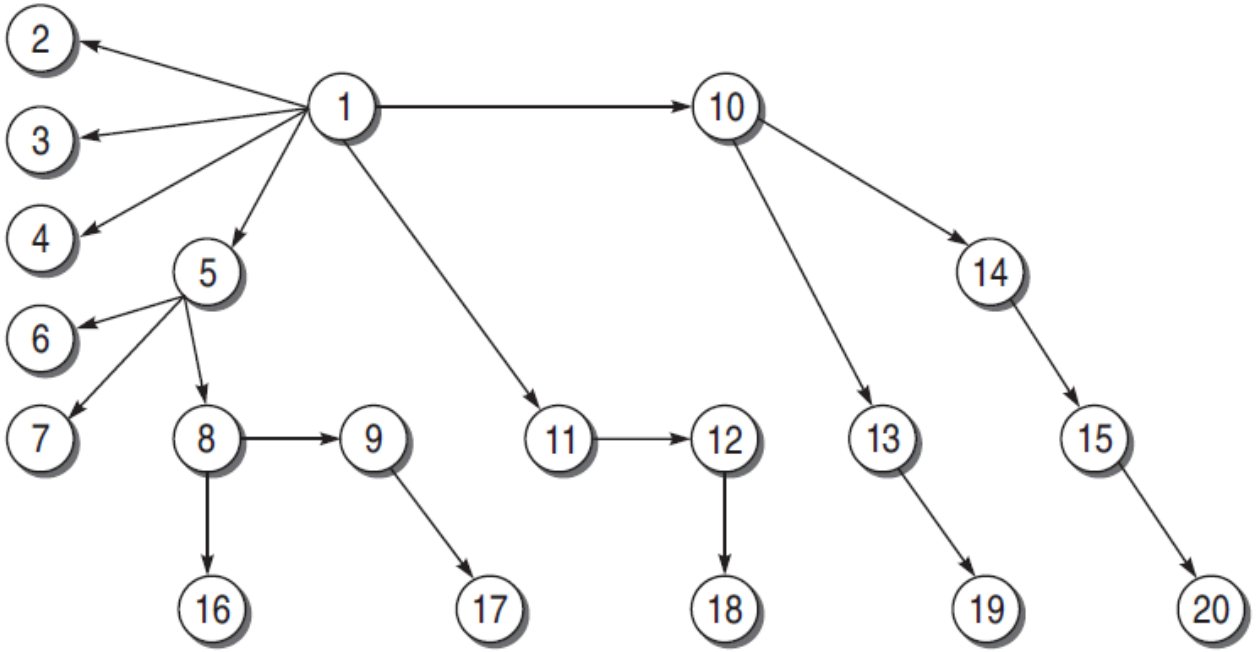
Comparison between top-down and bottom-up testing

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.

INTEGRATION TESTING : CALL GRAPH BASED

- ✓ A call graph is a directed graph, wherein the nodes are either modules or units, and a directed edge from one node to another means one module has called another module.
- ✓ The call graph can be captured in a matrix form which is known as the adjacency matrix.
- ✓ The idea behind using a call graph for integration testing is to avoid the efforts made in developing the stubs and drivers.

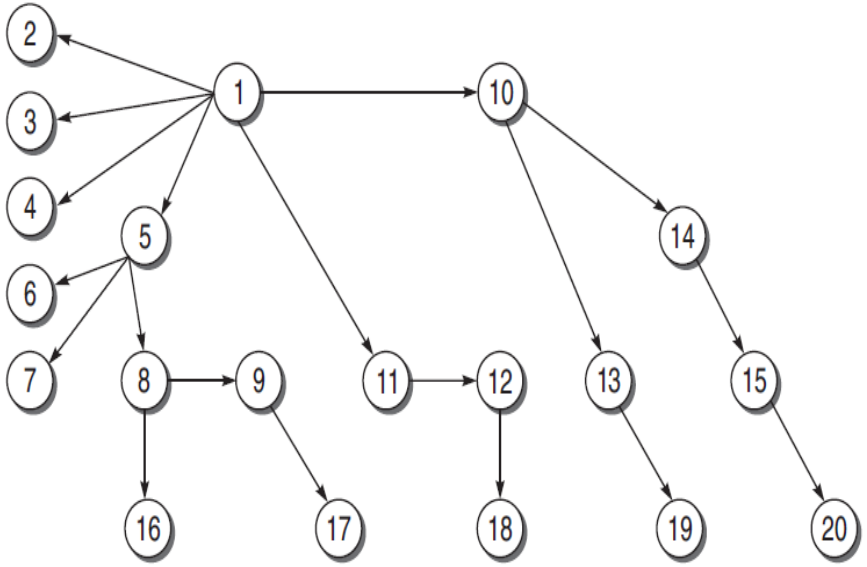
- ✓ If we know the calling sequence, and if we wait for the called or calling function, if not ready, then call graph-based integration can be used



Call Graph

INTEGRATION TESTING : CALL GRAPH BASED

Adjacency matrix

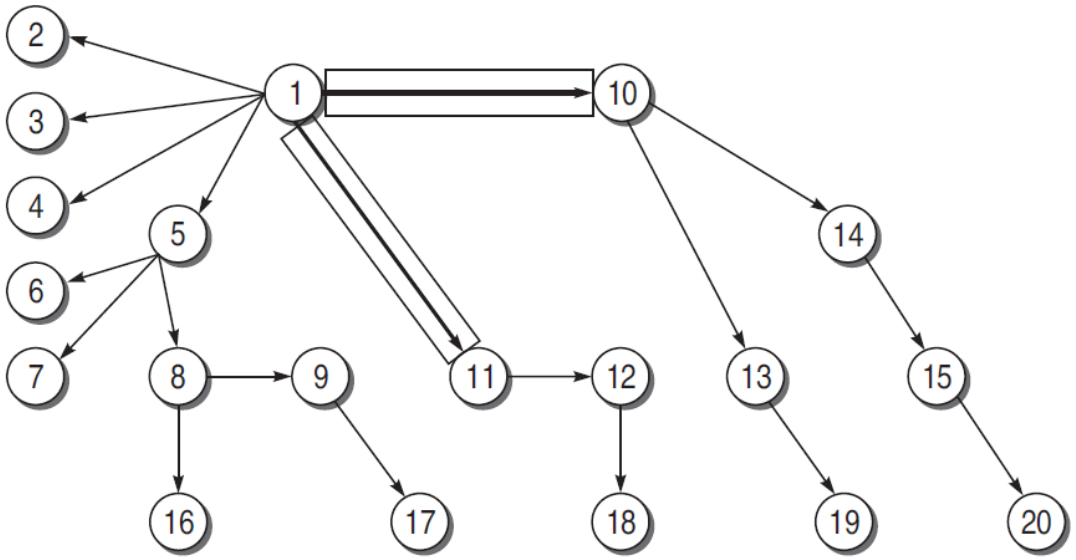


1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	×	×	×	×					×	×									
2																			
3																			
4																			
5					×	×	×												
6																			
7																			
8								×							×				
9																×			
10												×	×						
11											×								
12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

INTEGRATION TESTING: CALL GRAPH BASED

There are two types of integration testing based on call graph:

- ✓ Pair-wise Integration
- ✓ Neighborhood Integration



Pair-wise Integration

A form of integration testing that targets pairs of components that work together, as shown in a call graph.

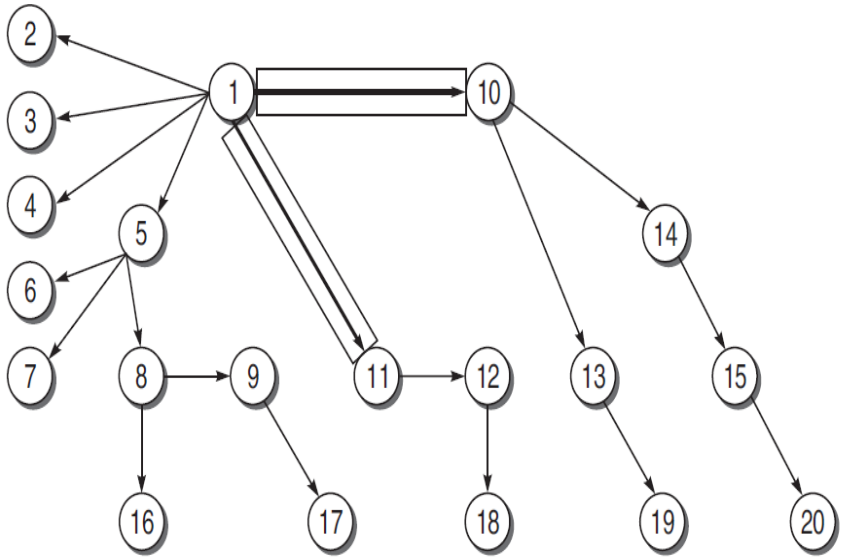
If we consider only one pair of calling and called modules, then we can make a set of pairs for all such modules, as shown in Fig., for pairs 1–10 and 1–11. The resulting set will be the total test sessions which will be equal to the sum of all edges in the call graph.

For example, in the call graph shown in Fig., the number of test sessions is 19 which is equal to the number of edges in the call graph.

INTEGRATION TESTING: CALL GRAPH BASED

There are two types of integration testing based on call graph:

- ✓ Pair-wise Integration
- ✓ 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 total test sessions in neighborhood integration can be calculated as:

Neighborhoods = nodes – sink nodes

= 20 – 10

= 10

where **sink node** is an instruction in a module at which the execution terminates.

INTEGRATION TESTING: PATH BASED

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.

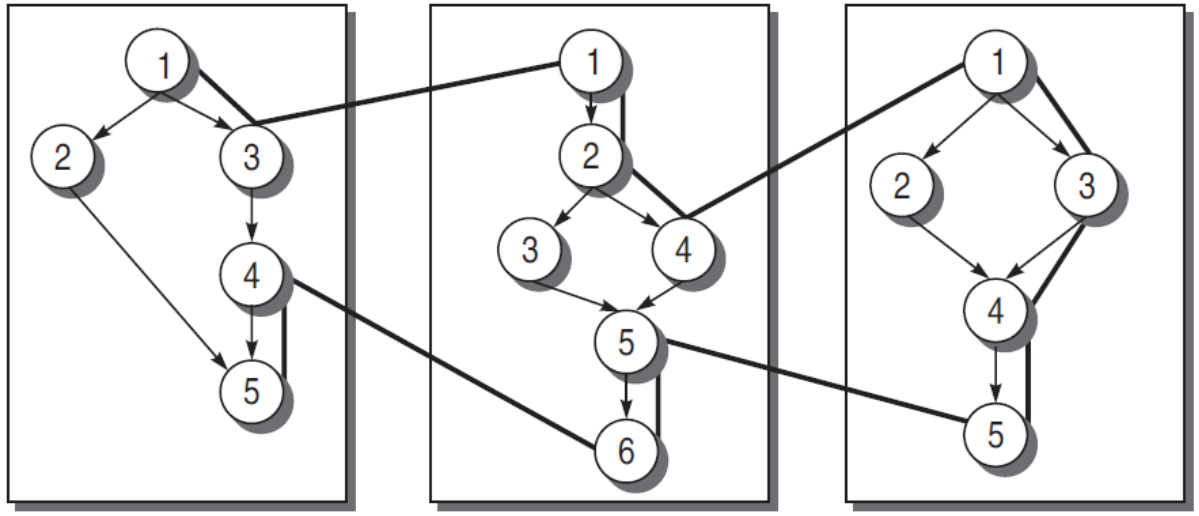
Module execution path (MEP) It is a sequence of statements that begins with a source node and ends with a sink node, with no intervening sink nodes. .

message is a programming language mechanism by which one unit transfers control to another unit, and acquires a response from the other unit.

MM-Path It is a path consisting of MEPs and messages. The path shows the sequence of executable statements.

MM-path graph It can be defined as an extended flow graph where nodes are MEPs and edges are messages.

EXAMPLE

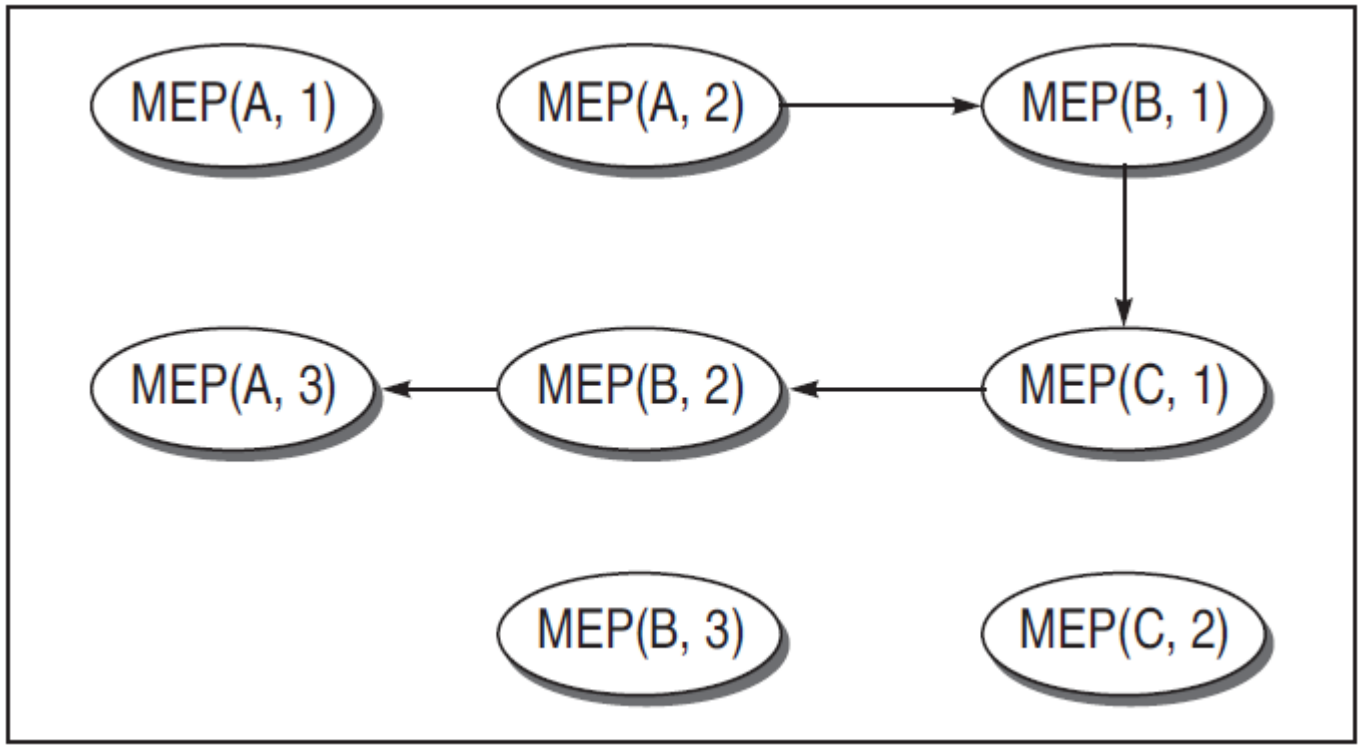


Unit A Unit B Unit C

	Source Nodes	Sink Nodes	MEPs
Unit A	1,4	3,5	MEP(A,1) = <1,2,5> MEP(A,2) = <1,3> MEP(A,3) = <4,5>
Unit B	1,5	4,6	MEP(B,1) = <1,2,4> MEP(B,2) = <5,6> MEP(B,3) = <1,2,3,4,5,6>
Unit C	1	5	MEP(C,1) = <1,3,4,5> MEP(C,2) = <1,2,4,5>

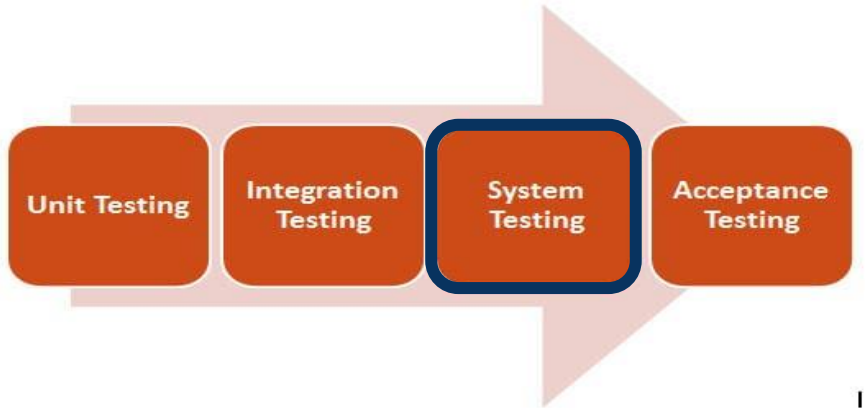
EXAMPLE

The MM-path graph for this example is



SYSTEM TESTING

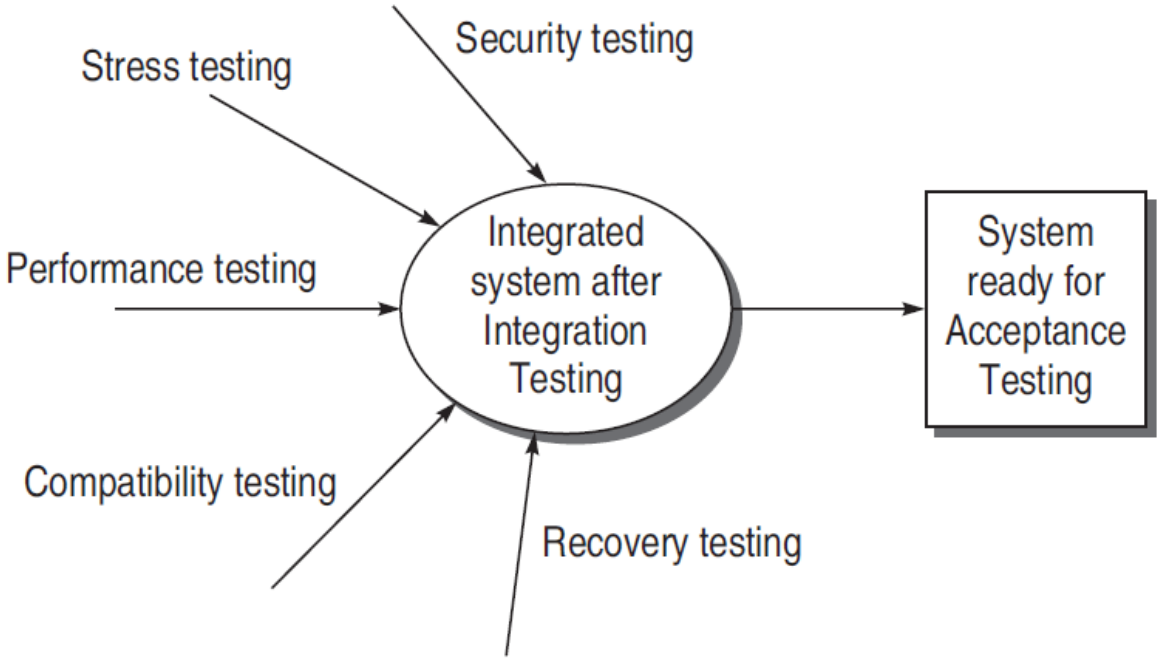
- ✓ is a series of different tests to test the whole system on various grounds (functional & non functional) where bugs have the probability to occur. The ground can be performance, security, maximum load, etc.
- ✓ **SYSTEM TESTING** is a level of testing that validates the complete and fully integrated software product.
- ✓ The purpose of a system test is to evaluate the end-to-end system specifications.
- ✓ System Testing (ST) is a **black box** testing technique performed to evaluate the complete system.
- ✓ is usually carried out by a team that is independent of the development team in order to measure the quality of the system unbiased.



System Testing

System testing includes:

- ✓ Functional Testing
  - Validates functional requirements
- ✓ Performance Testing
  - Validates non-functional requirements

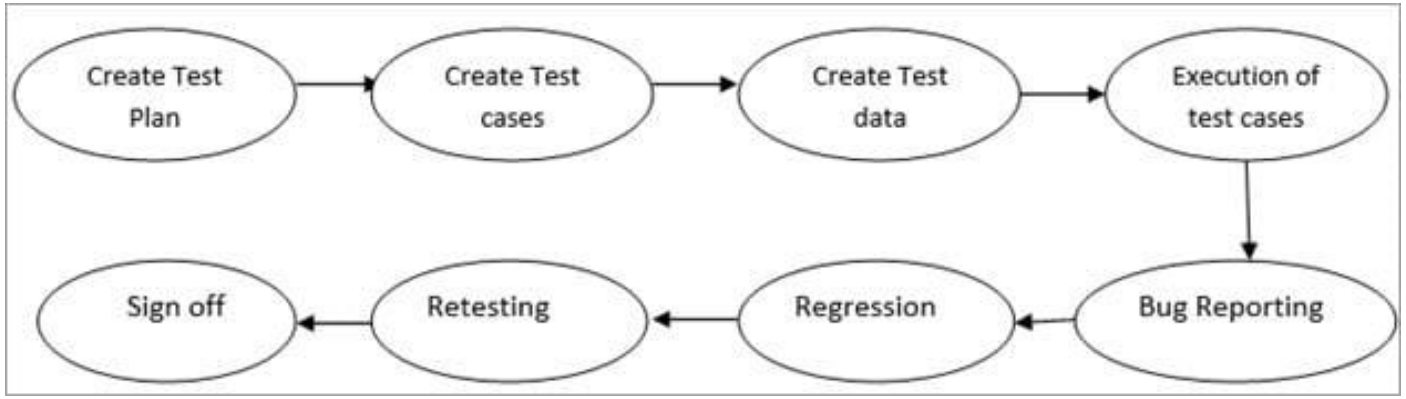


When we are system testing, we are testing all subsystems together

System Testing

Steps in system Testing

1. The very first step is to create a Test Plan.
2. Create System Test Cases and test scripts.
3. Prepare the test data required for this testing.
4. Execute the system test cases and script.
5. Report the bugs. Re-testing the bugs once fixed.
6. **Regression** testing to verify the impact of the change in the code.
7. Repetition of the testing cycle until the system is ready to be deployed.
8. Sign off from the testing team.



Acceptance Testing

USER ACCEPTANCE TESTING(UAT)

Testing is performed by the Client of the application to determine whether the application is developed as per the requirements specified by him/her.

Acceptance testing is the formal testing conducted to determine whether a software system satisfies its acceptance criteria and to enable clients to determine whether to accept the system or not.

User Acceptance Testing (UAT)

UAT is a type of testing performed by the **end user** or **the client** to verify/accept the software system before moving the software application to the production environment.

UAT is done in the final phase of testing after functional, integration and system testing is done.

The main purpose of UAT is to validate the end to end business flow. It does NOT focus on Cosmetic errors, Spelling mistakes or System testing.

User Acceptance Testing is carried out in a separate testing environment with production-like data setup.

It is a kind of black box testing where two or more end-users will be involved. The Full Form of UAT is User Acceptance Testing.

User Acceptance Testing(UAT)

Need of UAT:

- ✓ Developers code software based on requirements document which is their "own" understanding of the requirements and **may not actually be what the client needs from the software**.
- ✓ Requirements changes during the course of the project may not be communicated effectively to the developers.

1
- Developers have included features on their "own" understanding

2
- Requirements changes "not communicated" effectively to the developers

Running acceptance tests ensures that no requirement change has happened in the meantime and that everything is as it should be to satisfy the customer.

User Acceptance Testing (UAT)

Entry Criteria

- ✓ System testing is complete and defects identified are either fixed or documented.
- ✓ Acceptance plan is prepared and resources have been identified.
- ✓ Test environment for the acceptance testing is available.

Exit Criteria

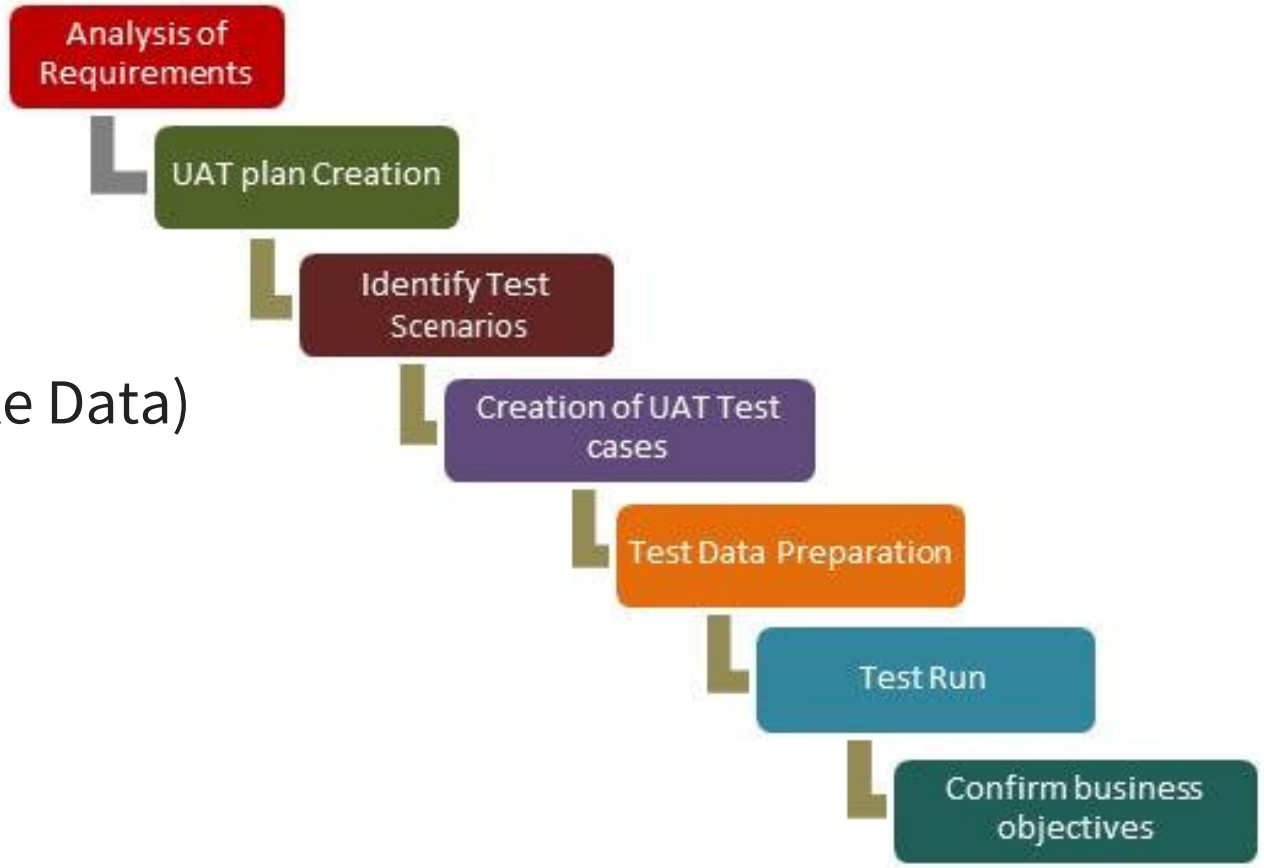
- Acceptance decision is made for the software.
- In case of any warning, the development team is notified.

User Acceptance Testing (UAT)

UAT Process:

Once Entry criteria for UAT are satisfied, following are the tasks need to be performed by the testers:

- ✓ Analysis of Business Requirements
- ✓ Creation of UAT test plan
- ✓ Identify Test Scenarios
- ✓ Create UAT Test Cases
- ✓ Preparation of Test Data(Production like Data)
- ✓ Run the Test cases
- ✓ Record the Results
- ✓ Confirm business objectives



User Acceptance Testing (UAT)

Types of Acceptance Testing

Acceptance testing is classified into the following two categories:

- ✓ **Alpha Testing** Tests are conducted at the development site by the end users. The test environment can be controlled a little in this case.

- ✓ **Beta Testing** Tests are conducted at the customer site and the development team does not have any control over the test environment.

ALPHA TESTING

Alpha testing is one of the most common software testing strategy used in software development. Its specially used by product development organizations.

- ✓This test takes place at the developer's site. Developers observe the users and note problems.
- ✓Alpha testing is testing of an application when development is about to complete.
- ✓Minor design changes can still be made as a result of alpha testing.
- ✓Alpha testing is typically performed by a group that is independent of the design team, but still within the company, e.g. in-house software test engineers, or software QA engineers.

ALPHA TESTING

Entry Criteria to Alpha

- ✓ All features are complete/testable (no urgent bugs).
- ✓ High bugs on primary platforms are fixed/verified.
- ✓ 50% of medium bugs on primary platforms are fixed/verified.
- ✓ All features have been tested on primary platforms.
- ✓ Performance has been measured/compared with previous releases (user functions).
- ✓ Usability testing and feedback (ongoing).
- ✓ Alpha sites are ready for installation.

Exit Criteria from Alpha

- ✓ Get responses/feedbacks from customers.
- ✓ Prepare a report of any serious bugs being noticed.
- ✓ Notify bug-fixing issues to developers.
- ✓ Make sure that no more additional features can be included
- ✓ Sign off on Alpha testing

ALPHA TESTING

Advantages of Alpha Testing:

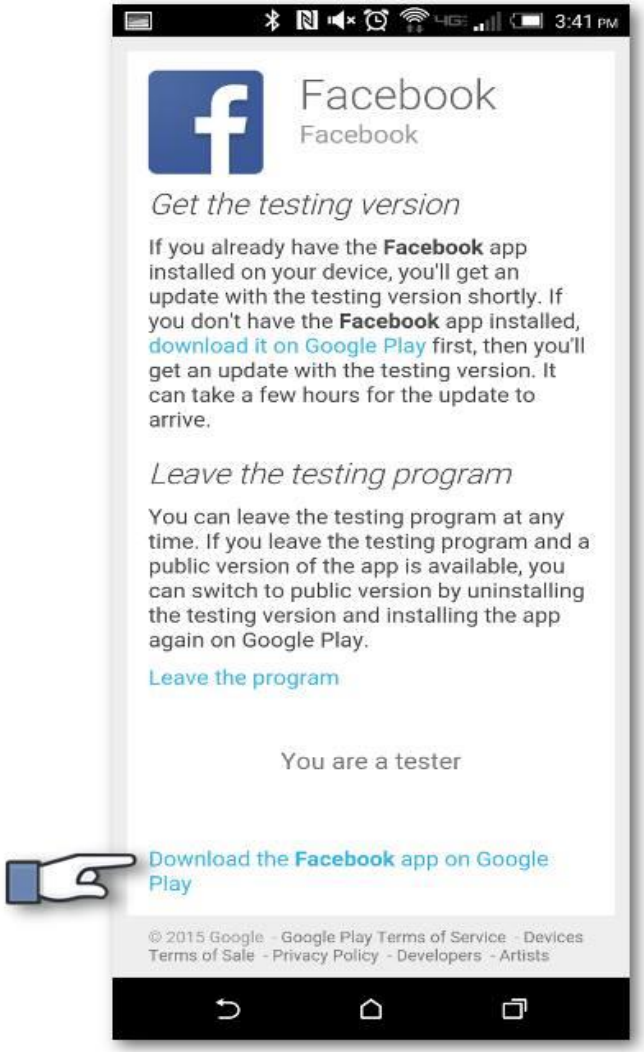
- ✓Provides better view about the reliability of the software at an early stage
- ✓Helps simulate real time user behavior and environment.
- ✓Detect many showstopper or serious errors

Disadvantages of Alpha Testing:

- ✓In depth functionality of the software cannot be tested as it is still under development stage.

BETA TESTING

- ✓Beta Testing, also known as “field testing”.
- ✓Takes place in the customer’s environment and involves some extensive testing by a group of customers who use the system in their environment.
- ✓Beta Testing 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.
- ✓Beta Test provides a complete overview of the true experience gained by the end users while experiencing the product



*Real People, Real Environment, Real Product are the three **R's** of Beta Testing and the question that arises here in Beta Testing is **"Do Customers like the Product?".***

BETA TESTING

Entrance criteria for Beta Testing:

- ✓ Positive responses from alpha sites.
- ✓ Customer bugs in alpha testing have been addressed.
- ✓ There are no fatal errors which can affect the functionality of the software.
- ✓ Secondary platform compatibility testing is complete.
- ✓ Regression testing corresponding to bug fixes has been done.
- ✓ Beta sites are ready for installation.

Exit Criteria for Beta Testing:

- ✓ Get responses/feedbacks from the beta testers.
- ✓ Prepare a report of all serious bugs.
- ✓ Notify bug-fixing issues to developers.

BETA TESTING

Guidelines for Beta Testing:

- ✓ Don’t expect to release new builds to beta testers more than once every two weeks.
- ✓ Don’t plan a beta with fewer than four releases.
- ✓ If you add a feature, even a small one, during the beta process, the clock goes back to the beginning of eight weeks and you need another 3–4 releases..

BETA TESTING

Advantages of Beta Testing:

- ✓ Reduces product failure risk via customer validation.
- ✓ Beta Testing allows a company to test post-launch infrastructure.
- ✓ Improves product quality via customer feedback
- ✓ Cost effective compared to similar data gathering methods

Disadvantages of Beta Testing:

- ✓ Doesn’t allow any control over the testing as it is carried out in real environment and not under the lab environment.
- ✓ Finding the right beta users and maintaining their participation could be a challenge

Alpha Testing Vs Beta testing:

Alpha Testing	Beta Testing
performed by Testers who are usually internal employees of the organization	performed by Clients or End Users who are not employees of the organization
performed at developer's site	performed at client location or end user of the product
Reliability and Security Testing are not performed in-depth Alpha Testing	Reliability, Security, Robustness are checked during Beta Testing
Long execution cycle may be required for Alpha testing	Only few weeks of execution are required for Beta testing

Smoke Testing

SMOKE TESTING

Assume you order an item from **Daraz**.

What will you do after getting the parcel form the Daraz delivery man?

1. check that the parcel is addressed to you
2. and then make sure parcel is intact and not torn.
3. Next, you open the parcel and see the item is what you ordered and also make sure it is new, not old.

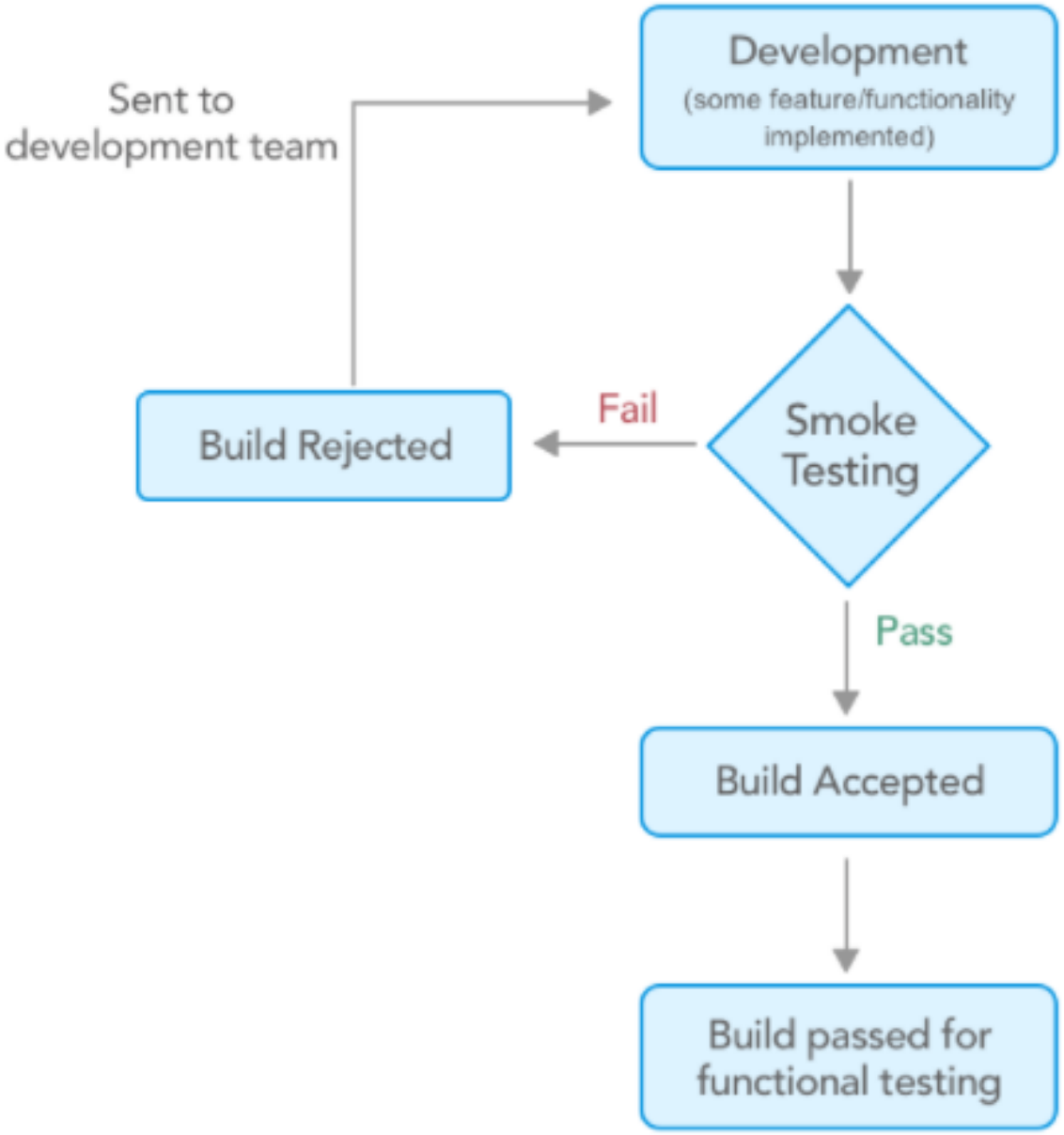
Well, you just completed your smoke testing on the parcel.

In case of SW testing, Smoke testing is performed on the 'new' build given by developers to QA team to verify if the basic functionalities are working or not.

It is the first test to be done on any new build.

SMOKE TESTING

- ✓ In smoke testing, the test cases chosen cover the most important functionality or component of the system.
- ✓ The objective is not to perform exhaustive testing, but to verify that the critical functionality of the system is working fine.



SMOKE TESTING

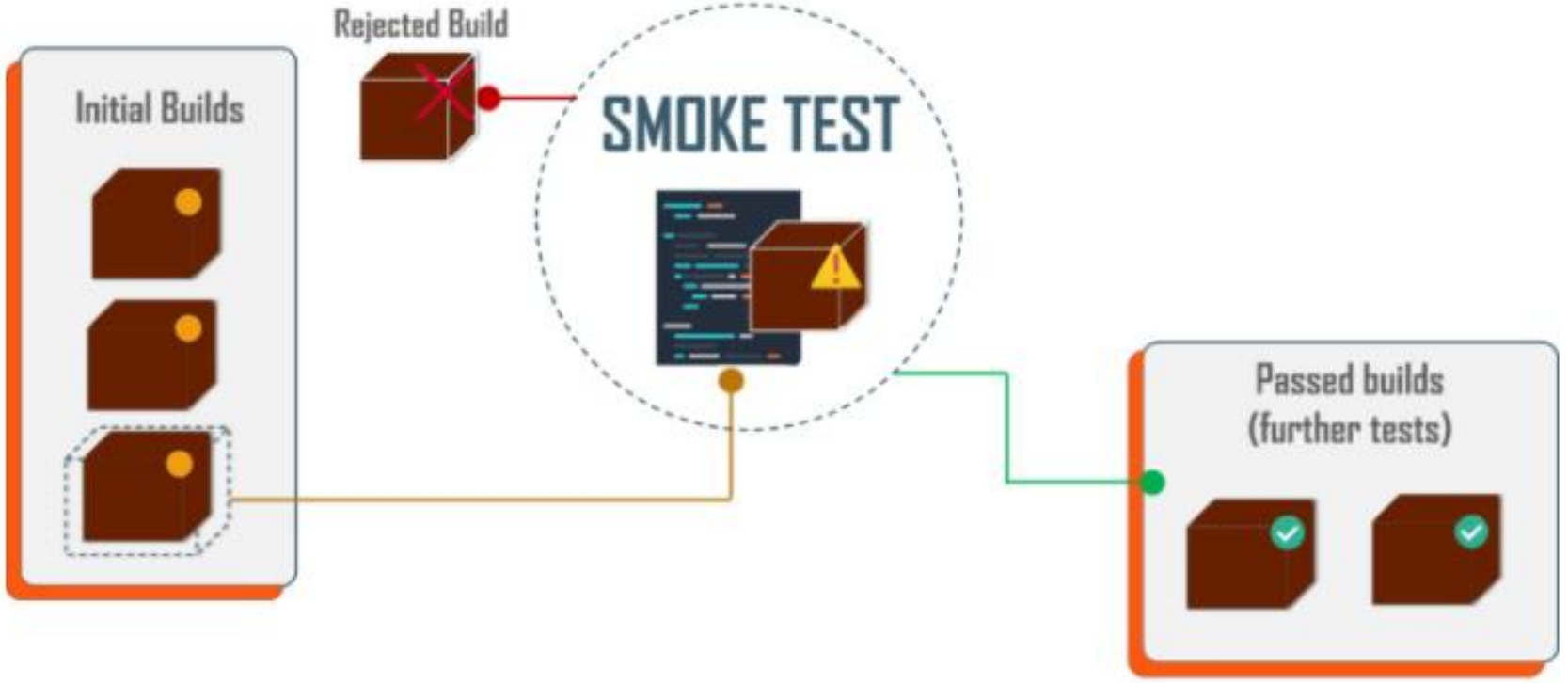
Why do we need Smoke Testing

- ✓ Just imagine a situation where you have a testing team consisting of 10 members.
- ✓ all the 10 testers start to test the application and raise the defects for failures they find.
- ✓ Now, at the end of the day, the development team may come back say, sorry this is not the right build or the QA team may stop the testing saying there are too many issues.
- ✓ But again 10 people have already wasted their 8 hours for this which means 80 hours of productivity is lost.
- ✓ This is the reason why we need to have a smoke test done, before jumping into a full-fledged testing cycle.

SMOKE TESTING

When and How Often do We Need Smoke Testing?

- ✓ Smoke Testing normally takes a maximum of 60 minutes and should be done for every new build.
- ✓ Once the product is stable, you can even think about automating the smoke tests.
- ✓ a smoke test is very critical, because it will prevent an unstable or broken build from being pushed into production.



SMOKE TESTING

What are Scenarios that need to be included in a Smoke Test?

- ✓ **Build Verification:** The first and foremost step in a smoke test is to verify the build, the build number, and environment availability. .

- ✓ **Account Creation:** If your application involves a user creation, then you should try to create a new user and check if the system is successfully allowing you to do that

- ✓ **Log in Logout:** If applicable in your SUT (System Under Test), as part of the smoke test you should try to successfully login with old and newly created credentials. also test logout

- ✓ **Business Critical Features:** This is very important. For all the major or business-critical features, we should a simple test to ensure that the most commonly used functionalities are not broken.

SMOKE TESTING

- ✓ **Integration Scenarios:** This is the most important part of a smoke test. The effectiveness of this part depends upon the understanding of the system integrations by the tester. For example, if the tester knows that there is some data that flows from system A to system B, then he must make it a point to check that as part of the smoke test (any 1 value).

- ✓ **Add/Edit/Delete:** Data is always saved in a database. The three basic operations in a database are added the record, edit record and delete a record.

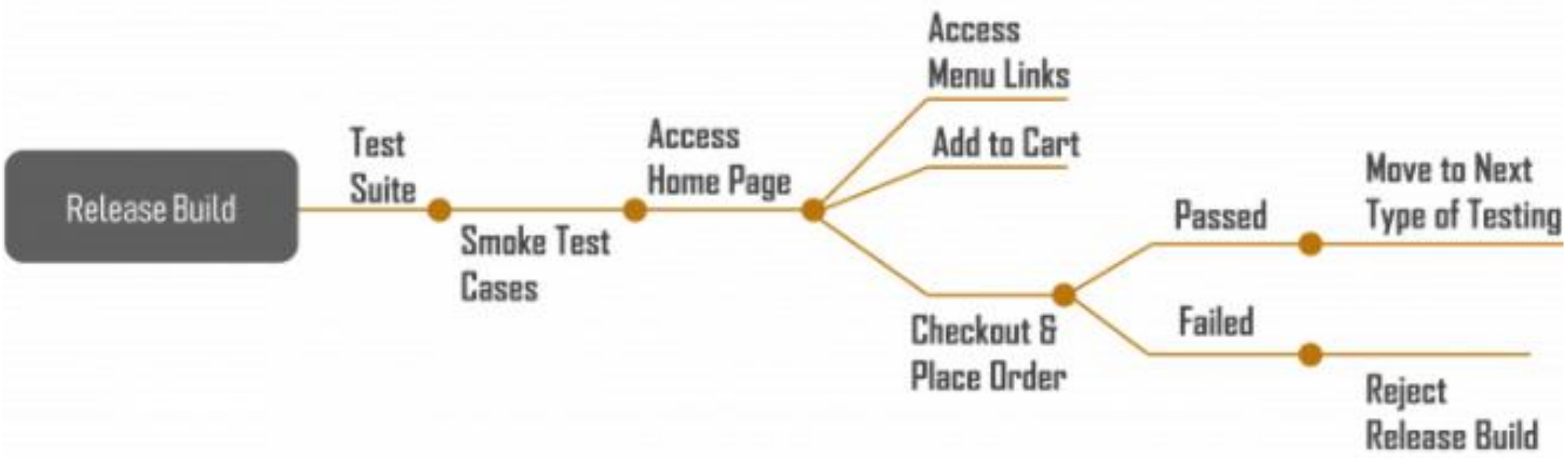
- ✓ **Overall Navigations:** The last part is overall navigation.

SMOKE TESTING

Who will Perform the Smoke Test?

- ✓ Usually QA lead is the one who performs smoke testing. Once the major build of software has been done, it will be tested to find out it's working well or not..

- ✓ The entire QA team sits together and discusses the main features of the software and the smoke test will be done to find out its condition..



SMOKE TESTING

Advantages:

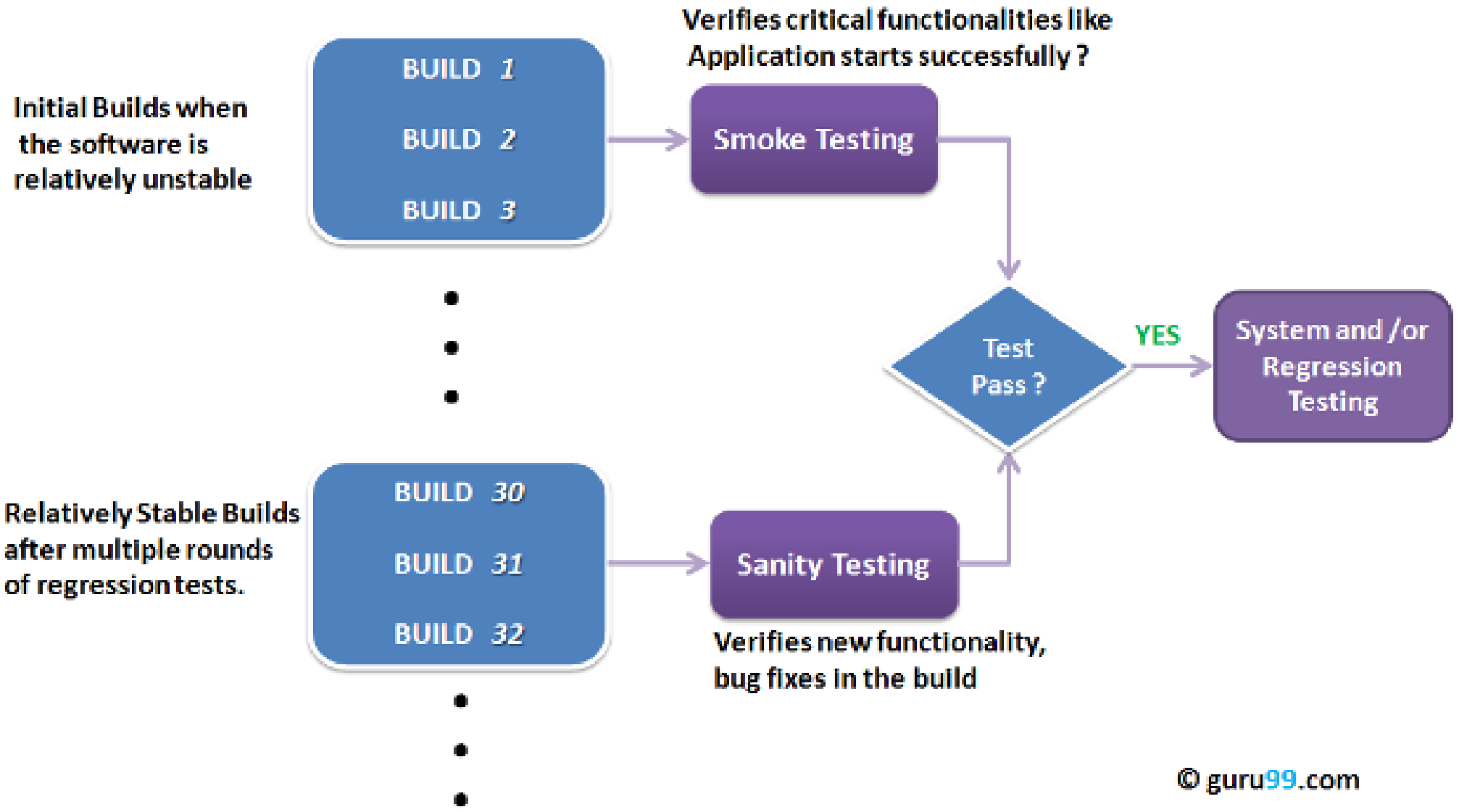
- ✓ It helps to find faults earlier in the product lifecycle.
- ✓ It saves the testers time by avoiding testing an unstable or wrong build
- ✓ It provides confidence to the tester to proceed with testing
- ✓ It helps to find integration issues faster
- ✓ Major severity defects can be found out
- ✓ Detection and rectification will be an easy process
- ✓ Since execution happens quickly, there will be a faster feedback

Sanity Testing

SANITY TESTING

- ✓ When a new build is received with minor modifications, instead of running a thorough regression test suite we perform a sanity test.
- ✓ It determines that the modifications have actually fixed the issues and no further issues have been introduced by the fixes.
- ✓ Sanity testing is generally a subset of regression testing and a group of test cases executed that are related to the changes made to the product.

Many testers get confused between sanity testing and smoke testing


Verifies critical functionalities like Application starts successfully ?
Initial Builds when the software is relatively unstable
BUILD 1
BUILD 2
BUILD 3
Smoke Testing
Test Pass ?
YES
System and /or Regression Testing
Relatively Stable Builds after multiple rounds of regression tests.
BUILD 30
BUILD 31
BUILD 32
Sanity Testing
Verifies new functionality, bug fixes in the build

Regression Testing

REGRESSION TESTING

Is a type of software testing to confirm that a recent program or code change has not adversely affected existing features.

Is nothing but a **full or partial** selection of **already executed** test cases which are **re-executed** to ensure existing functionalities work fine.

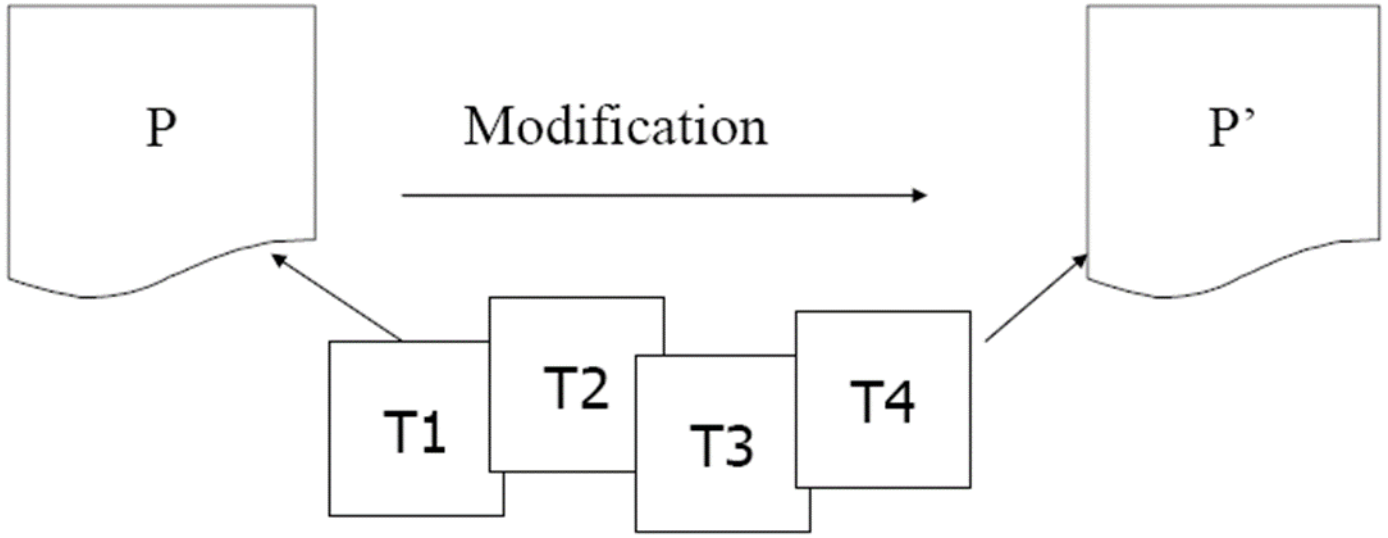
Regression test cases - **carefully selected** to cover as much of the system as possible that can **in any way have been affected** by a maintenance change or any kind of change.

REGRESSION TESTING

Regression testing is the execution of a set of test cases on a program in order to ensure that its **revision** does not produce **unintended faults**, does not "regress" - that is, become less effective than it has been in the past

In a more formal way-

Given program **P**, its modified version **P'**, and a test set **T** that was used to previously test **P**, find a way to utilize **T** to gain sufficient confidence in the correctness of **P'.**

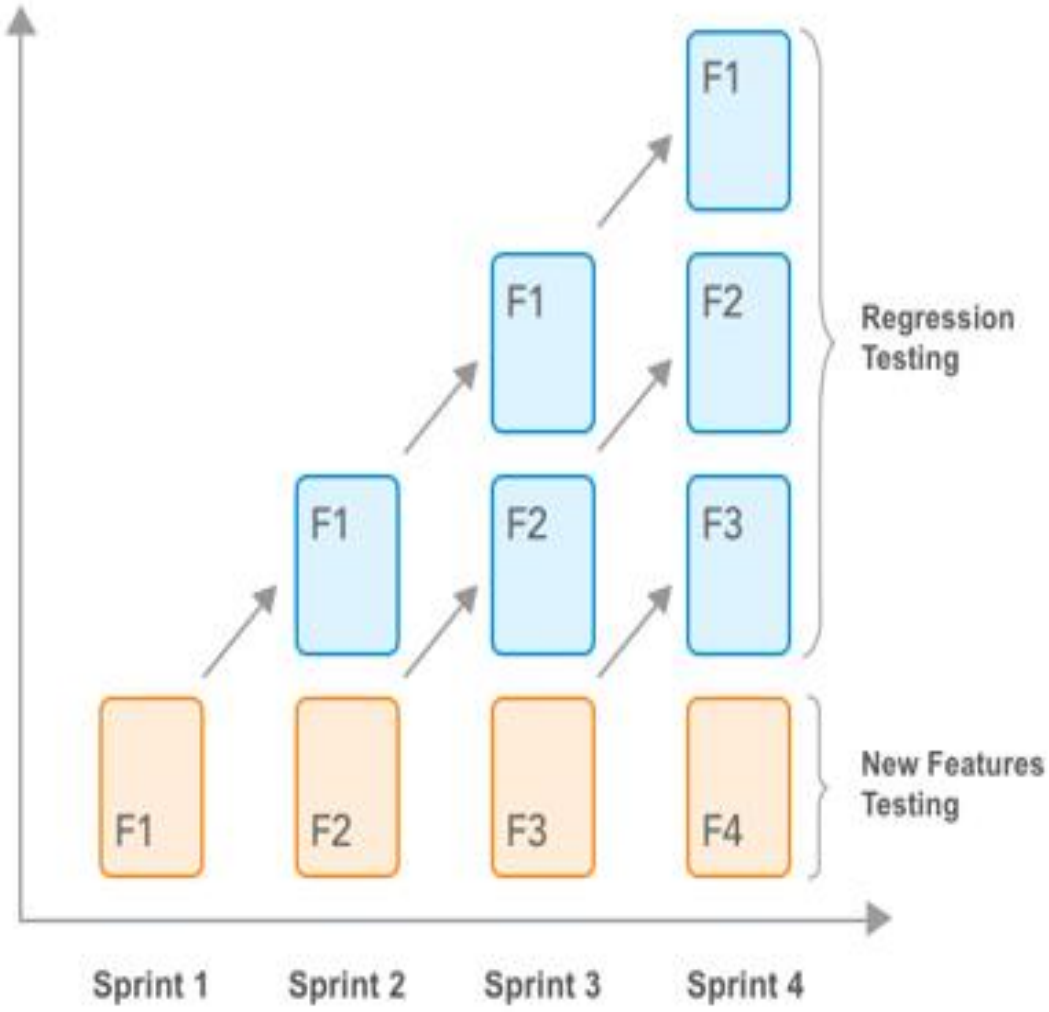


REGRESSION TESTING

- ✓ Regression testing is crucial during maintenance.
- ✓ It is a good idea to automate regression testing so that all test cases are run after each modification to the software
- ✓ When you find a bug in your program you should write a test case that exhibits the bug.
- ✓ Then using regression testing you can make sure that the old bugs do not reappear

REGRESSION TESTING

- ✓ Regression testing is performed to make sure that a change or addition hasn't broken any of the existing functionality.
- ✓ Its purpose is to find bugs that may have been accidentally introduced into the existing build and to ensure that previously removed bugs continue to stay dead.
- ✓ regression testing is the process of re-testing the modified parts of the software and ensuring that no new errors have been introduced into previously tested source code due to these modifications.



REGRESSION TESTING

Regression testing is the process of re-testing the modified parts of the software and ensuring that no new errors have been introduced into previously tested source code due to these modifications.

Let us check what are the causes to modify a software…….

(i) Some errors may have been discovered during the actual use of the software.

(ii) The user may have requested for additional functionality.

(iii) Software may have to be modified due to change in some external policies and principles.

(iv) restructuring work may have to be done to improve the efficiency/performance of the SW.

(v) Software may have to be modified due to change in existing technologies.

REGRESSION TESTING

When and How Often do We Need Regression Testing?

✓ We typically think of regression testing as a software maintenance activity

Few cases when we need to perform regression test:

- Addition of new functionalities
- Change requirements
- After bug fix
- Performance issue
- Environment change

Challenges:

✓ Time Consuming

✓ Expensive

✓ Complex

REGRESSION TESTING

Selecting test cases for regression testing

Effective Regression Tests can be done by selecting the following test cases -

- ✓Test cases which have frequent defects
- ✓Functionalities which are more visible to the users
- ✓Test cases which verify core features of the product
- ✓Test cases which has undergone more and recent changes
- ✓All Integration Test Cases
- ✓All Complex Test Cases
- ✓Boundary value test cases
- ✓A sample of Successful test cases
- ✓A sample of Failure test cases

Non Functional Testing

NON-FUNCTIONAL TESTING

The Major Non-Functional Testing techniques include:

- ✓ Performance testing
- ✓ Load testing
- ✓ Stress testing
- ✓ Scalability testing
- ✓ Stability testing
- ✓ Compatibility testing
- ✓ Compliance testing
- ✓ Recovery testing
- ✓ Security testing
- ✓ Usability testing

Performance Testing

PERFORMANCE TESTING

- ✓ is a type of software testing that ensures that the software applications will perform well under their expected workload.
- ✓ The goal of performance testing is not to find bugs but to eliminate performance bottlenecks.
- ✓ Performance testing measures the quality attributes of the system, such as scalability, reliability and resource usage.

PERFORMANCE TESTING

Some of the factors that governs Performance testing:

- ✓Throughput
- ✓Response Time
- ✓Tuning
- ✓Benchmarking

PERFORMANCE TESTING

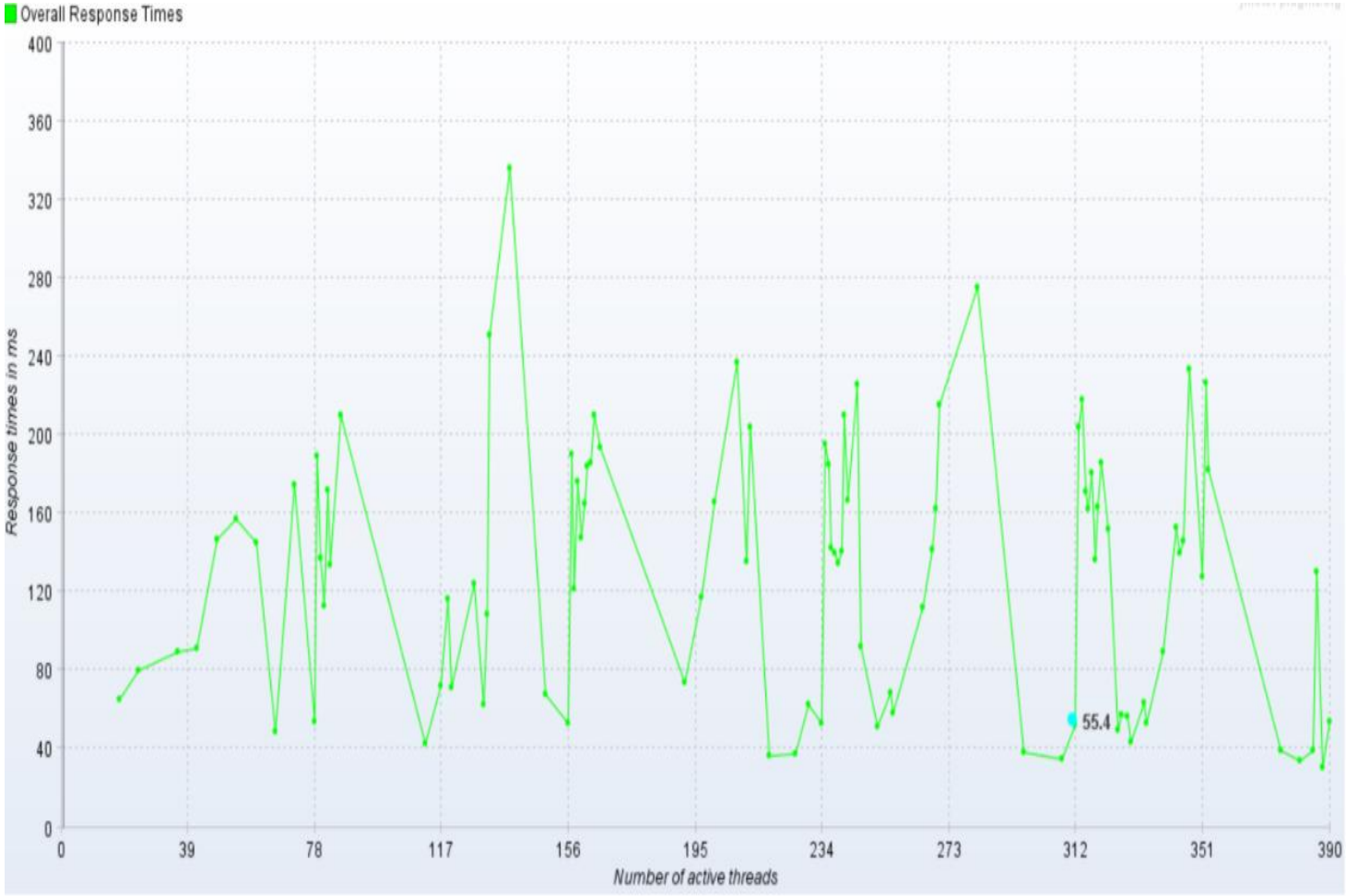
Throughput:

- ✓ Capability of a product to handle multiple transactions in a give period.
- ✓ Throughput represents the number of requests/business transactions processed by the product in a specified time duration [typically measured as total no. of transaction or requests in a given time or TPS (transaction per second].
- ✓ It is one of the significant indicator that helps in evaluating the performance of application.

PERFORMANCE TESTING

Response Time:

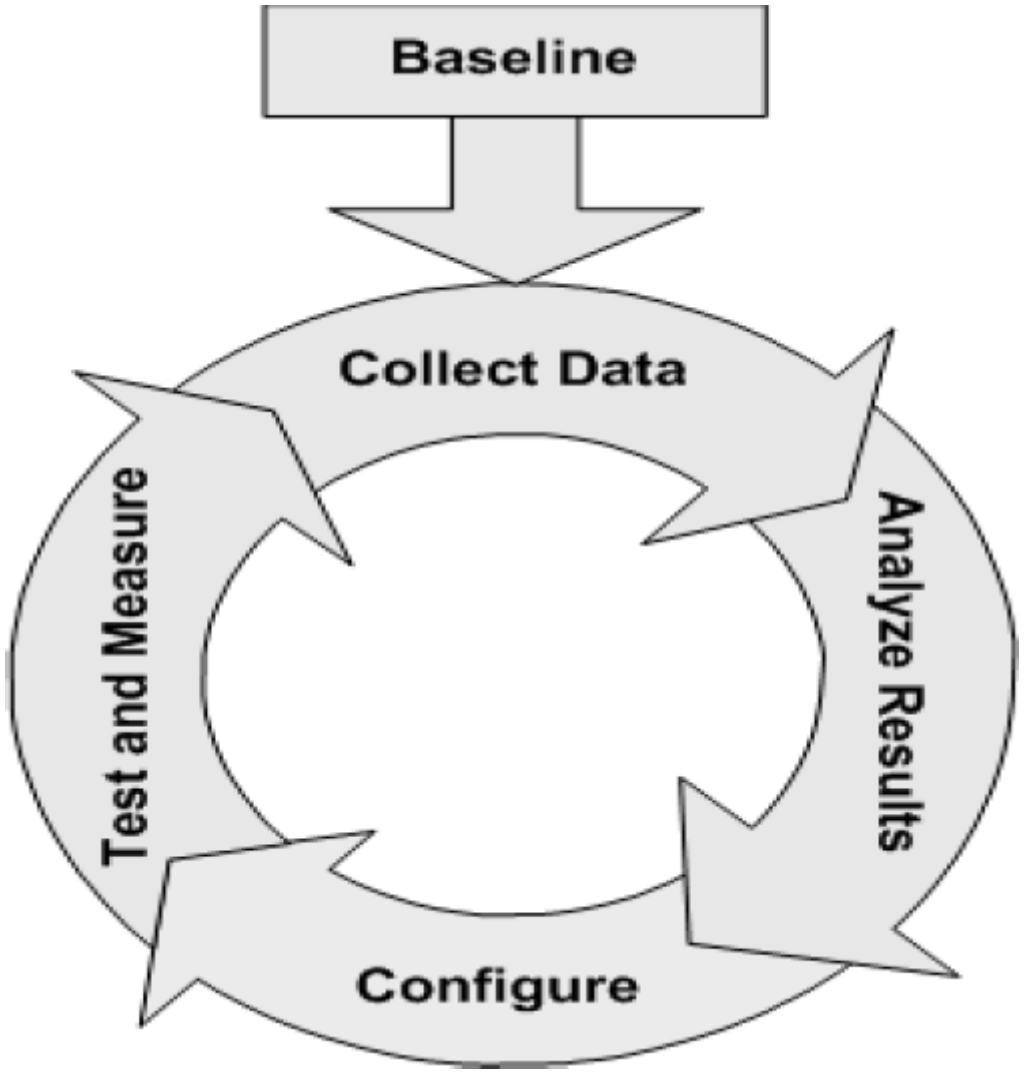
- ✓ It is equally important to find out how much time each of the transactions took to complete.
- ✓ Response time is defined as the delay between the point of request and the first response from the product.
- ✓ The response time increases proportionally to the user load.



PERFORMANCE TESTING

Tuning:

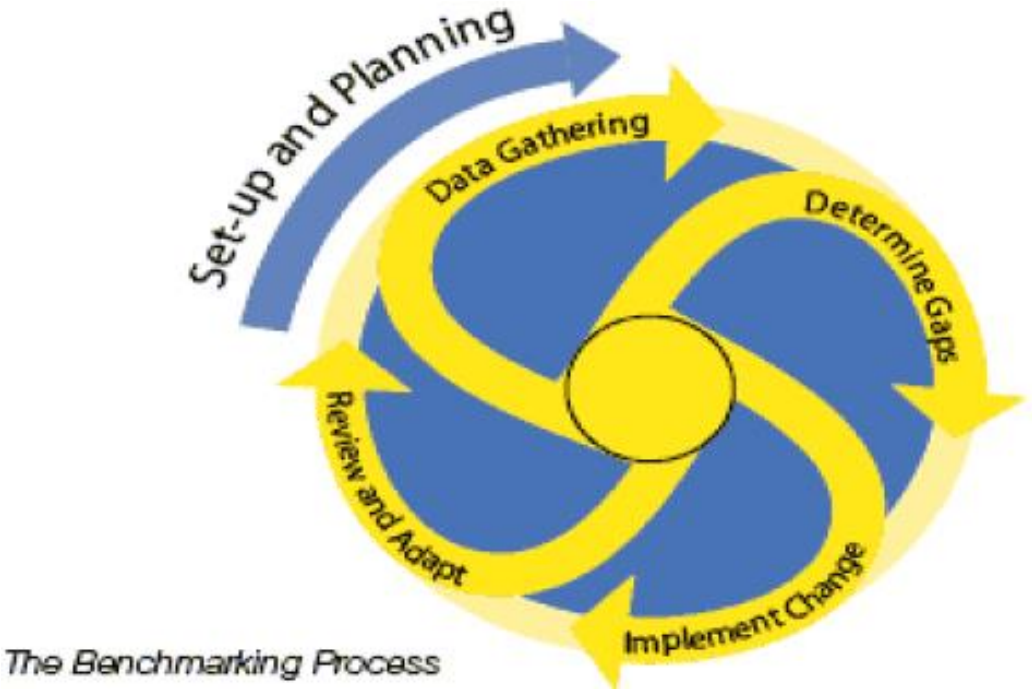
- ✓ Tuning is the procedure by which product performance is enhanced by setting different values to the parameters of the product, operating system and other components.
- ✓ Tuning improves the product performance without having to touch the source code of the product.



PERFORMANCE TESTING

Benchmarking:

- ✓ A very well-improved performance of a product makes no business sense if that performance does not match up to the competitive products.
- ✓ A careful analysis is needed to chalk out the list of transactions to be compared across products so that an **apple-apple comparison** becomes possible.



The Benchmarking Process

PERFORMANCE TESTING

Finally

- ✓ The testing to evaluate the response time (speed), throughput and utilization of system to execute its required functions in comparison with different versions of the same product or a different competitive product is called Performance Testing.
- ✓ Tuning is performed until the system under test achieves the expected levels of performance.

PERFORMANCE TESTING

Why Performance testing???

PERFORMANCE TESTING

When is it required????

Design Phase:

- ✓ Pages containing lots of images and multimedia for reasonable wait times. Heavy loads are less important than knowing which types of content cause slowdowns.

Development Phase:

- ✓ To check results of individual pages and processes, looking for breaking points, unnecessary code and bottlenecks.

Deployment Phase:

- ✓ To identify the minimum hardware and software requirements for the application.

PERFORMANCE TESTING

What should be tested???

High frequency transactions:

- ✓ The most frequently used transactions have the potential to impact the performance of all of the other transactions if they are not efficient.

Mission Critical transactions:

- ✓ The more important transactions that facilitate the core objectives of the system should be included, as failure under load of these transactions has, by definition, the greatest impact.

Read Transactions:

- ✓ At least one READ ONLY transaction should be included, so that performance of such transactions can be differentiated from other more complex transactions.

Update Transactions:

- ✓ At least one update transaction should be included so that performance of such transactions can be differentiated from other transactions.

PERFORMANCE TESTING

The Process:

1. Identify performance scenarios

Firstly, we will identify the performance scenarios based on these below factors:

- Most commonly scenarios
- Most critical scenarios
- Huge data transaction

2. Plan and design performance test script

In this step, we will install the tools in the Test Engineer Machine and access the test server and then we write some script according to the test scenarios and run the tool.

3. Configure the test environment & distribute the load

After writing the test scripts, we will arrange the testing environment before the execution. And also, manage the tools, other resources and distribute the load according to the "Usage Pattern" or mention the duration and stability.

PERFORMANCE TESTING

The Process:

4. Execute test scripts Once we are done with distributing the load, we will execute, validate, and monitor the test scripts.

5. Result After executing the test scripts, we will get the test result. And check that the result meeting the goal in the given response time or not, and the response time could be maximum, average, and minimum.

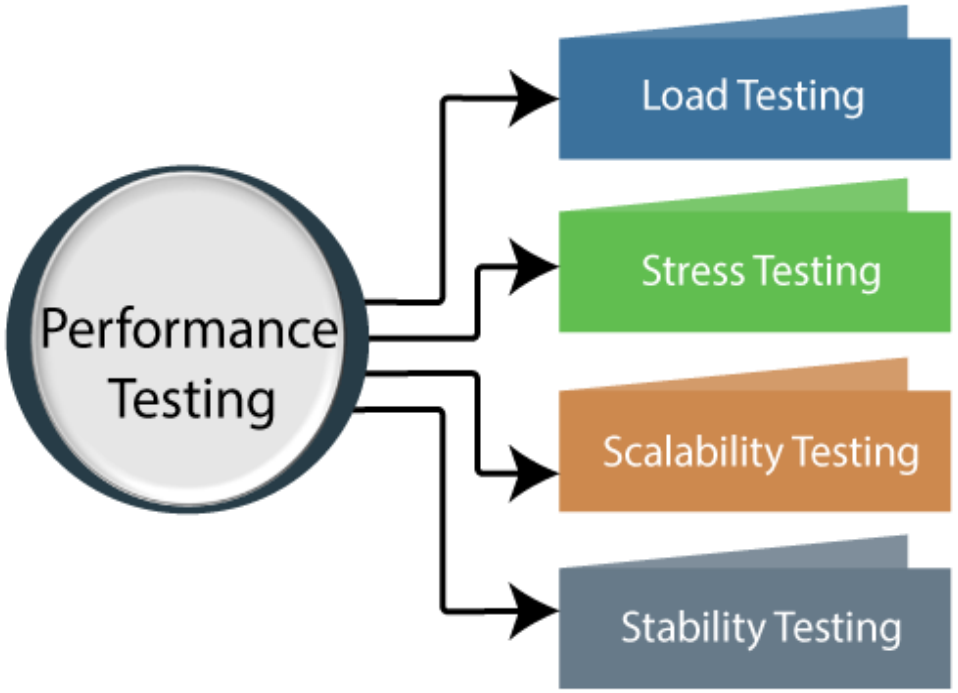
6. Analysis result First, we will analyze the test result whether it meets with the response time or not.

7. Identify the Bottleneck After that, we will identify the bottleneck (bug or performance issue). And the bottleneck could occur because of these aspects like the problem in code, hardware issue (hard disk, RAM Processor), network issues, and the software issue (operating system). And after finding the bottleneck, we will perform tuning (fix or adjustment) to resolve this bottleneck.

8. Re-run test Once we fix the bottlenecks, re-run the test scripts and checks the result whether it meets the required goal or not.

PERFORMANCE TESTING

Types of Performance Testing:



PERFORMANCE TESTING

Load Testing

- ✓ Load testing is performance testing technique using which the response of the system is measured under various load conditions.
- ✓ The load testing is performed for normal and peak load conditions.
- ✓ is used to check the performance of an application by applying some load which is either less than or equal to the **desired load**.

PERFORMANCE TESTING

Stress Testing:

- ✓ Stress testing a Non-Functional testing technique that is performed as part of performance testing.
- ✓ During stress testing, the system is monitored after subjecting the system to overload to ensure that the system can sustain the stress.
- ✓ That is with stress testing we check the behavior of an application by applying load greater than the desired load.
- ✓ The recovery of the system from such phase (after stress) is very critical as it is highly likely to happen in production environment.

PERFORMANCE TESTING

Reasons for conducting Stress Testing:

- ✓ It allows the test team to monitor system performance during failures/heavy load.
- ✓ To verify if the system has saved the data before crashing or NOT.
- ✓ To verify if the system prints meaning error messages while crashing or did it print some random exceptions.
- ✓ To verify if unexpected failures do not cause security issues.

PERFORMANCE TESTING

Scalability Testing:

- ✓ Checking the performance of an application by increasing or decreasing the load in particular scales (no of a user) is known as scalability testing.
- ✓ Scalability testing is divided into two parts which are as follows:

  - **Upward scalability testing:** It is testing where we increase the number of users on a particular scale until we get a crash point. We will use upward scalability testing to find the maximum capacity of an application.

  - **Downward scalability testing:** The downward scalability testing is used when the load testing is not passed, then start decreasing the no. of users in a particular interval until the goal is achieved. So that it is easy to identify the bottleneck (bug).

PERFORMANCE TESTING

Stability Testing:

- ✓ Checking the performance of an application by applying the load for a particular duration of time is known as Stability Testing.

PERFORMANCE TESTING

Tools used for Performance Testing

Open Source	Commercial
OpenSTA	LoadRunner
Diesel Test	Silk Performer
TestMaker	Qengine
Grinder	Empirix e-Load
LoadSim	
Jmeter	
Rubis	

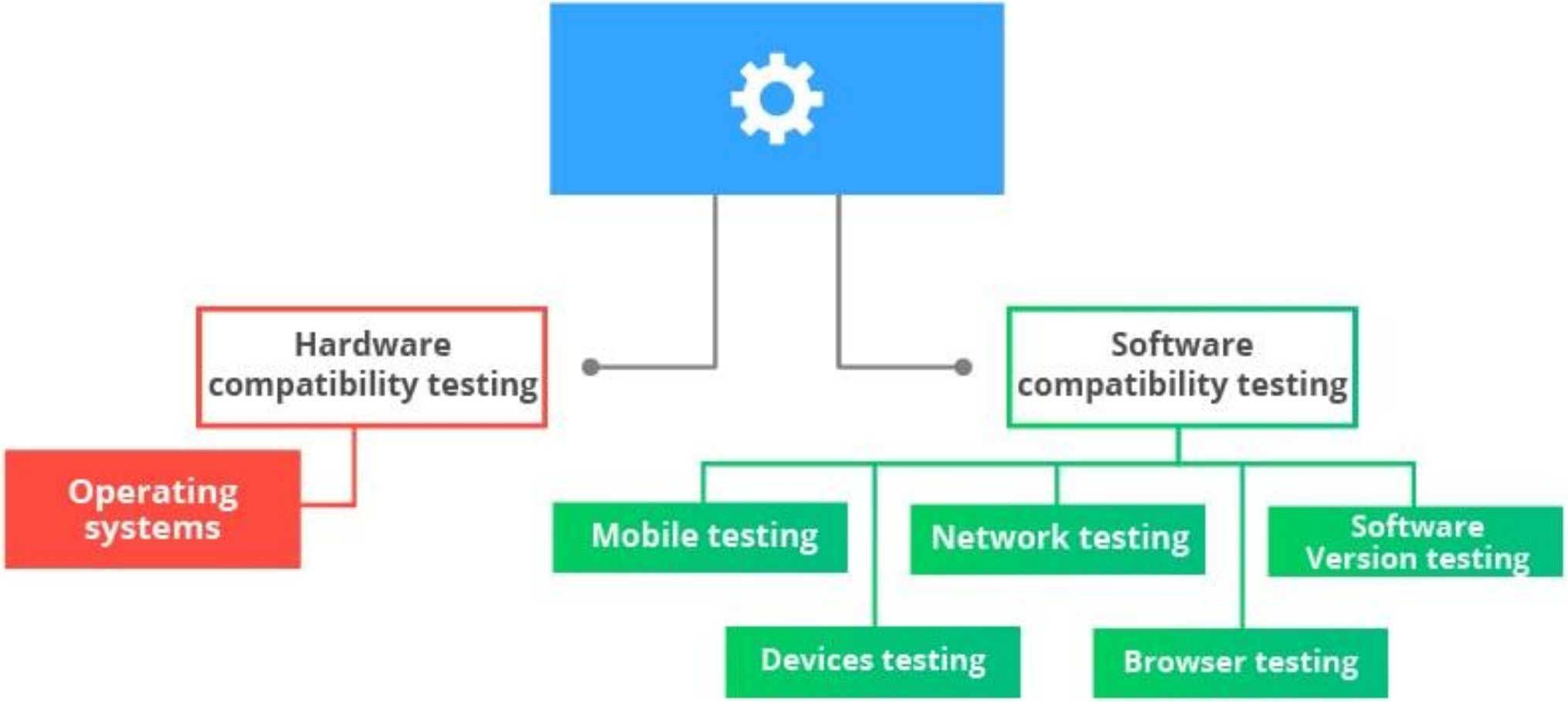
These are well Documented, Should check those with interest…

COMPATIBILITY TESTING

- ✓ Sometimes known as platform testing
- ✓ Compatibility Testing is a type of Non-functional 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.

COMPATIBILITY TESTING

Types of Compatibility Tests



COMPATIBILITY TESTING

There are two types of version checking in Compatibility Testing :

is a technique to verify the behavior and compatibility of the developed hardware or software with their older versions of the hardware or software. Backward compatibility testing is much predictable as all the changes from the previous versions are known.

is a process to verify the behavior and compatibility of the developed hardware or software with the newer versions of the hardware or software. Forward compatibility testing is a bit hard to predict as the changes that will be made in the newer versions are not known.

COMPATIBILITY TESTING

Benefits of Compatibility Testing:

Enhances Software Development Process

- Easy to validate application's stability, usability, and scalability across various platforms and deliver feedback.
- helps to analyze defects if any in the development stage itself.

Identifies Errors before Production

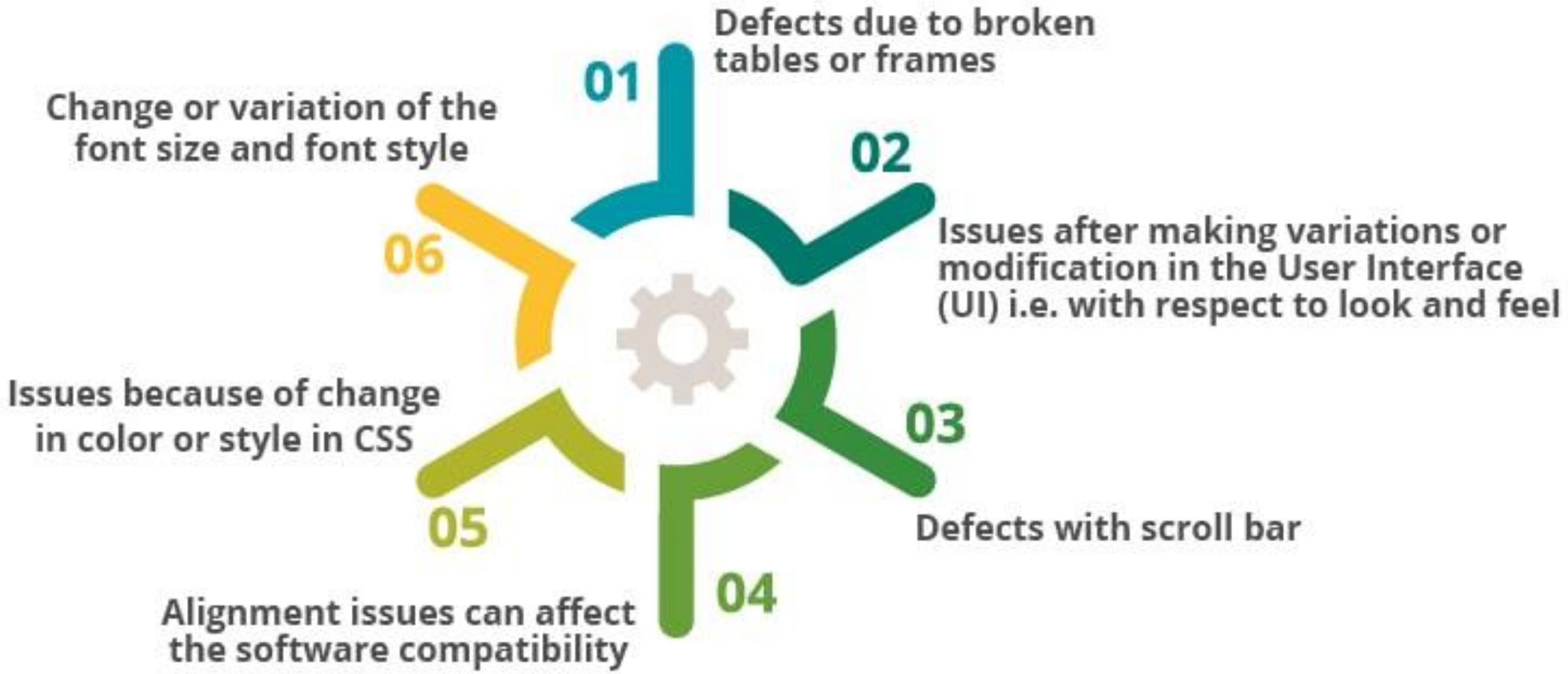
- ensures to meet the fundamental expectation of users.
- effectively performs system compatibility tests on different browsers, platforms, configurations, OS, hardware, etc.
- Thus, as compatibility tests are performed on the app, it ensures to deliver best customer experience.

Delivers & Meets Customer Expectations

- capable of timely identification of the defects
- Since defects are identified earlier before production, the testing practice gives enough time for the developer teams to fix them at the earliest.

COMPATIBILITY TESTING

Common Defects encountered during Compatibility Testing:



COMPLIANCE TESTING

- ✓ Compliance Testing is performed to maintain and validate the compliant state for the life of the software. Every industry has a regulatory and compliance board that protects the end users.
- ✓ The software used in the pharmaceutical industry, the Food and Drug Administration (FDA) regulation comes into the picture.
- ✓ Checklists for Compliance Testing:
  - ✓ Professionals, who are knowledgeable and experienced, who understand the compliance must be retained.
  - ✓ Understanding the risks and impacts of being non-compliant
  - ✓ Document the processes and follow them
  - ✓ Perform an internal audit and follow with an action plan to fix the issues

RECOVERY TESTING

- ✓Recovery testing is a type of non-functional testing technique performed in order to determine how quickly the system can recover after it has gone through system crash or hardware failure.
- ✓Recovery testing is the forced failure of the software to verify if the recovery is successful.

RECOVERY TESTING

Recovery Plan – Steps

1. Determining the feasibility of the recovery process.
2. Verification of the backup facilities.
3. Ensuring proper steps are documented to verify the compatibility of backup facilities.
4. Providing Training within the team.
5. Demonstrating the ability of the organization to recover from all critical failures.
6. Maintaining and updating the recovery plan at regular intervals.

SECURITY TESTING

- ✓ Security testing is a testing technique to determine if an information system protects data and maintains functionality as intended.
- ✓ Process intended to reveal flaws in the security mechanisms of an information system
- ✓ Finding out the potential loopholes & weakness of the system
- ✓ To check whether there is an information leakage

SECURITY TESTING

Aims to verify 6 basic principles as listed below:

- Confidentiality – Is my secret safe…
- Integrity – Is my data tampered..
- Authentication – Who am I? Something you know!! Something you have!
- Authorization – What can I do..
- Availability –Do the information ready for use when expected..
- Non-repudiation - prevent the later denial that an action happened, or a communication that took place

USABILITY TESTING

What is Usability?

Usability Is a measure of how easy it is to use something:

- ✓ How easy the SW will be for a typical user to understand, learn, and operate
- ✓ e.g., “user-friendliness”

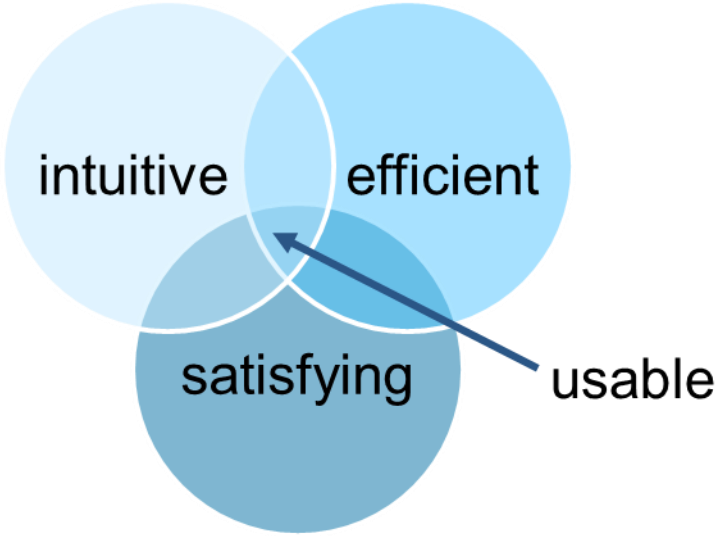
Usability is difficult to evaluate and measure

USABILITY TESTING

- ✓ Usability testing, a non-functional testing technique that is a measure of how easily the system can be used by end users.

ISO 9241-11 (3.1):Definition of Useability

'the extent to which a product can be used by specified users to achieve goals with effectiveness, efficiency and satisfaction in a specified context of use'



USABILITY TESTING

- ✓ It is difficult to evaluate and measure but can be evaluated based on the below parameters:
  - ✓ Level of Skill required to learn/use the software. It should maintain the balance for both novice and expert user.
  - ✓ Time required to get used to in using the software.
  - ✓ The measure of increase in user productivity if any.
  - ✓ Assessment of a user's attitude towards using the software.

USABILITY TESTING

Why Usability testing?

Uncover Problems
in the design

Discover Opportunities
to improve the design

Learn about Users
behavior and preferences

USABILITY TESTING

Usability Testing Process