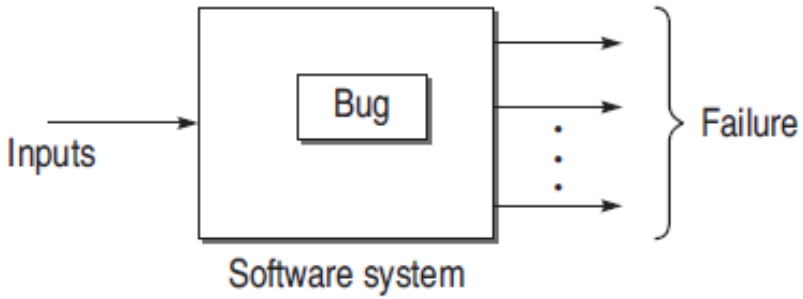
Software Testing Terminology

Failure:

- inability of a system or component to perform a required function according to its specification
- when results or behavior of the system under test are different as compared to specified expectations, then failure exists.
- Failure is the term which is used to describe the problems in a system on the output side

FAULT/DEFECT/BUG

- *Fault* is a condition that causes a system to produce failure
- Defect and Bug are synonymous to fault
- failures are manifestation of bugs.
- Some bugs are hidden

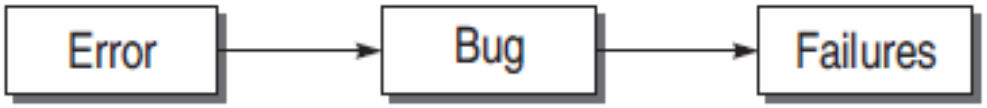


ERROR

- 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.

- Error causes a bug and the bug in turn causes failures



example

Consider the following module in a software:

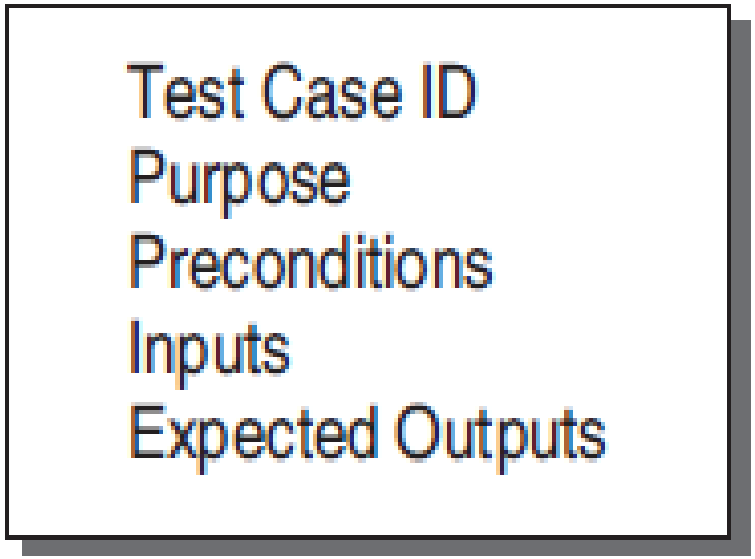
```
Module A()
{
….
while(a > n+1);
{
…
print(“The value of x is”, x);
}
….
}
```

- Printing the value of x which is critical for the use of software.
- But x is not printed by execution of this module.
- This is the failure of the program.
- A condition is preventing the body of while loop to be executed.
- This is known as *bug/defect/fault*

TEST CASE

- well-documented procedure designed to test the functionality of a feature in the system
- It has an identity and is associated with a program behavior
- Primary purpose of designing a test case is to find errors in the system.
- provide a set of inputs and its corresponding expected outputs to design a test case

Test case template



- ***Test case ID*** is the identification number given to each test case.
- ***Purpose*** defines why the case is being designed.
- ***Preconditions*** for running the inputs in a system can be defined, if required, in a test case.
- ***Inputs*** should not be hypothetical. Actual inputs must be provided, instead of general inputs
- ***Expected outputs*** are the outputs which should be produced when there is no failure.

- **Testware** The documents created during testing activities are known as *testware*
- **Incident :** An *incident* is the symptom(s) associated with a failure that alerts the user about the occurrence of a failure
- **Test oracle** An *oracle* is the means to judge the success or failure of a test, i.e. to judge the correctness of the system for some test

LIFE CYCLE OF A BUG

- life cycle of a bug can be classified into two phases:

  (i) bugs-in phase and

  (ii) bugs-out phase.

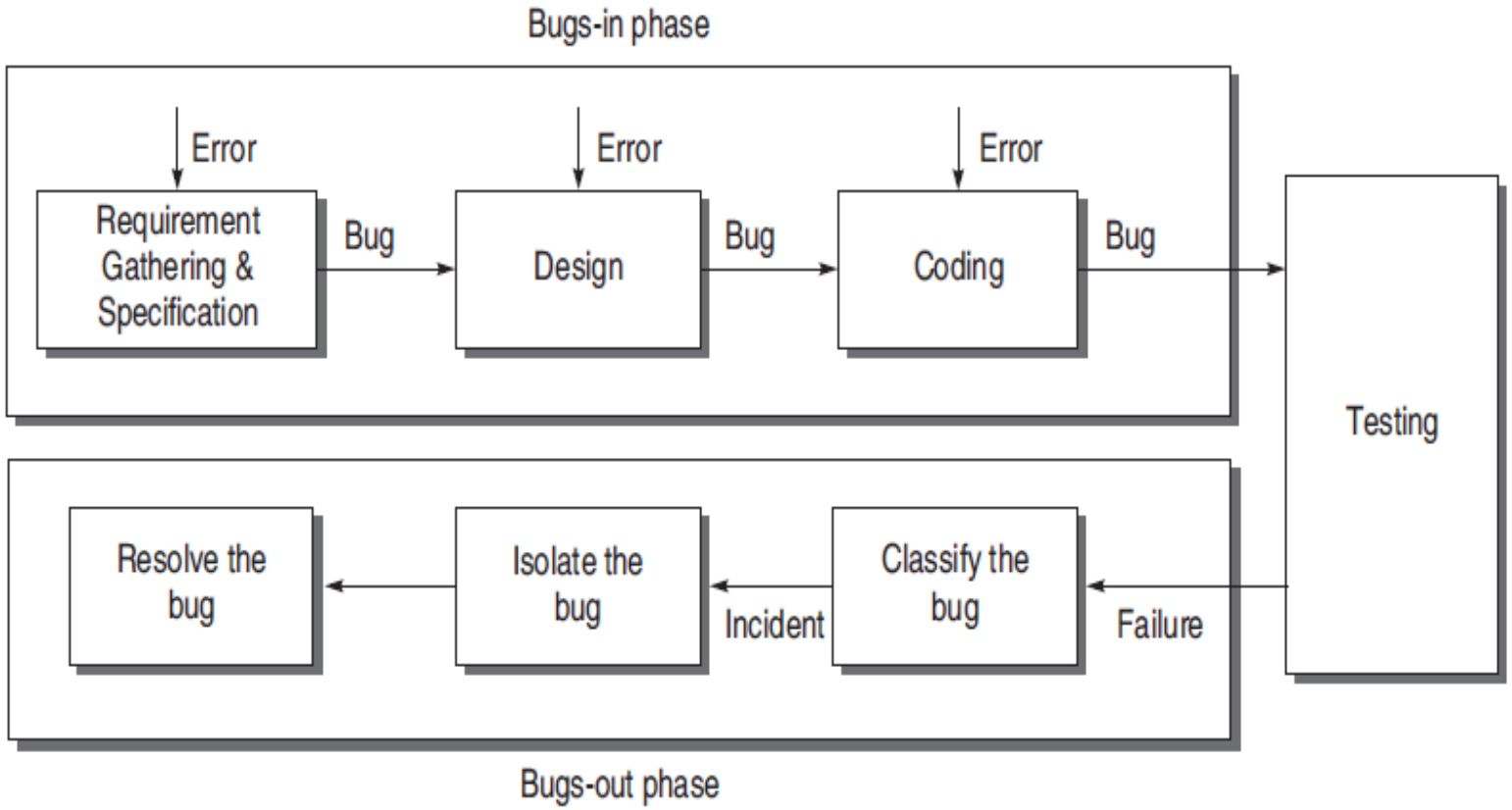
Bugs-In Phase

- This phase is where the errors and bugs are introduced in the software.
- Whenever we commit a mistake, it creates errors on a specific location of the software and consequently, when this error goes unnoticed, it causes some conditions to fail, leading to a bug in the software. This bug is carried out to the subsequent phases of SDLC, if not detected. Thus, a phase may have its own errors as well as bugs received from the previous phase.
- If you are not performing verification on earlier phases, then there is no chance of detecting these bugs.

Bugs-Out Phase

- If failures occur while testing a software product, we come to the conclusion that it is affected by bugs.
- However, there are situations when bugs are present, even though we don’t observe any failures.
- In this phase, when we observe failures, the following activities are performed to get rid of the bugs.

Life Cycle of a Bug



STATES OF A BUG

- ***New*** The state is new when the bug is reported first time by a tester.
- ***Open*** The new state does not verify that the bug is genuine. When the test leader approves that the bug is genuine, its state becomes open.
- ***Assign*** An open bug comes to the development team where the development team verifies its validity. If the bug is valid, a developer is assigned the job to fix it and the state of the bug now is ‘ASSIGN

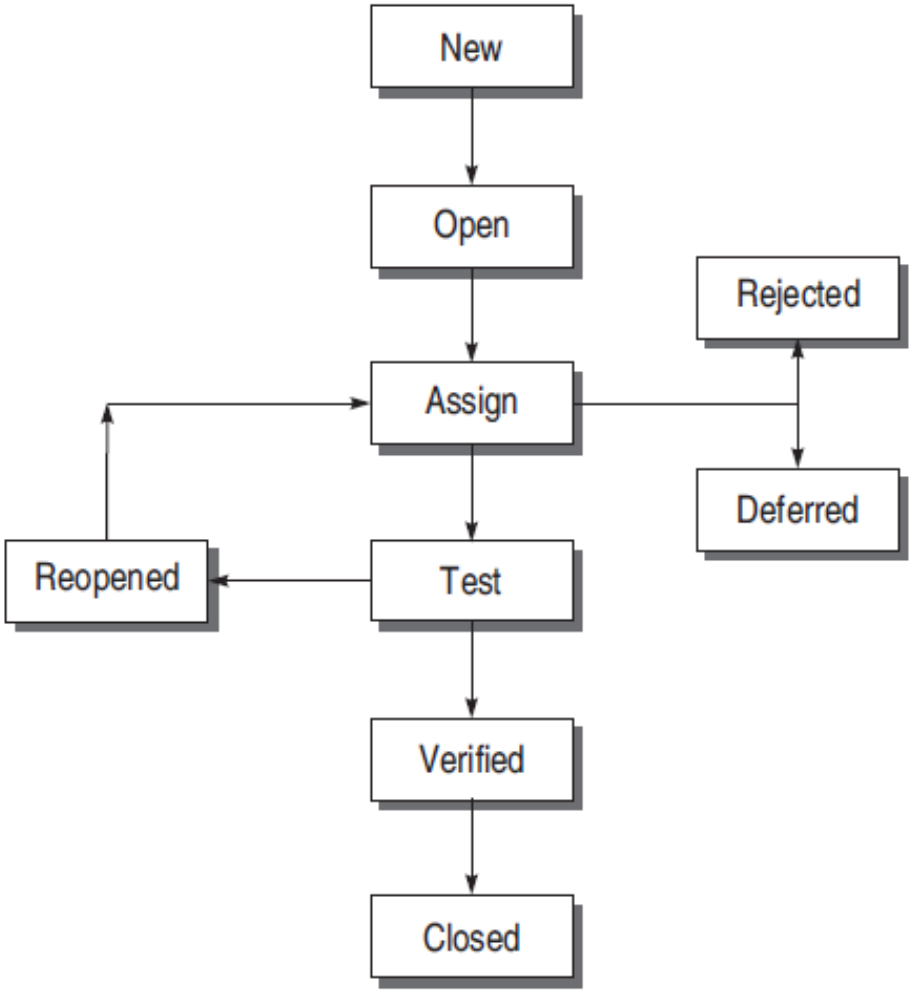
- ***Deferred*** The developer who has been assigned to fi x the bug will check its validity and priority. If the priority of the reported bug is not high or there is not sufficient time to test it or the bug does not have any adverse effect on the
- software, then the bug is changed to deferred state which implies the bug is expected to be fixed in next releases.
- ***Rejected*** It may be possible that the developer rejects the bug after checking its validity, as it is not a genuine one.

- ***Test*** After fixing the valid bug, the developer sends it back to the testing team for next round of checking. Before releasing to the testing team, the developer changes the bug's state to 'TEST'. It specifies that the bug has been fixed by the development team but not tested and is released to the testing team.
- ***Verified/fixed*** The tester tests the software and verifies whether the reported
- bug is fixed or not. After verifying, the developer approves that the bug is fixed and changes the status to 'VERIFIED'

- ***Reopened*** If the bug is still there even after fi xing it, the tester changes its status to ‘REOPENED’. The bug traverses the life cycle once again.
- In another case, a bug which has been closed earlier may be reopened if it appears again. In this case, the status will be REOPENED instead of OPEN.

- ***Closed*** Once the tester and other team members are confirmed that the bug is completely eliminated, they change its status to ‘CLOSED

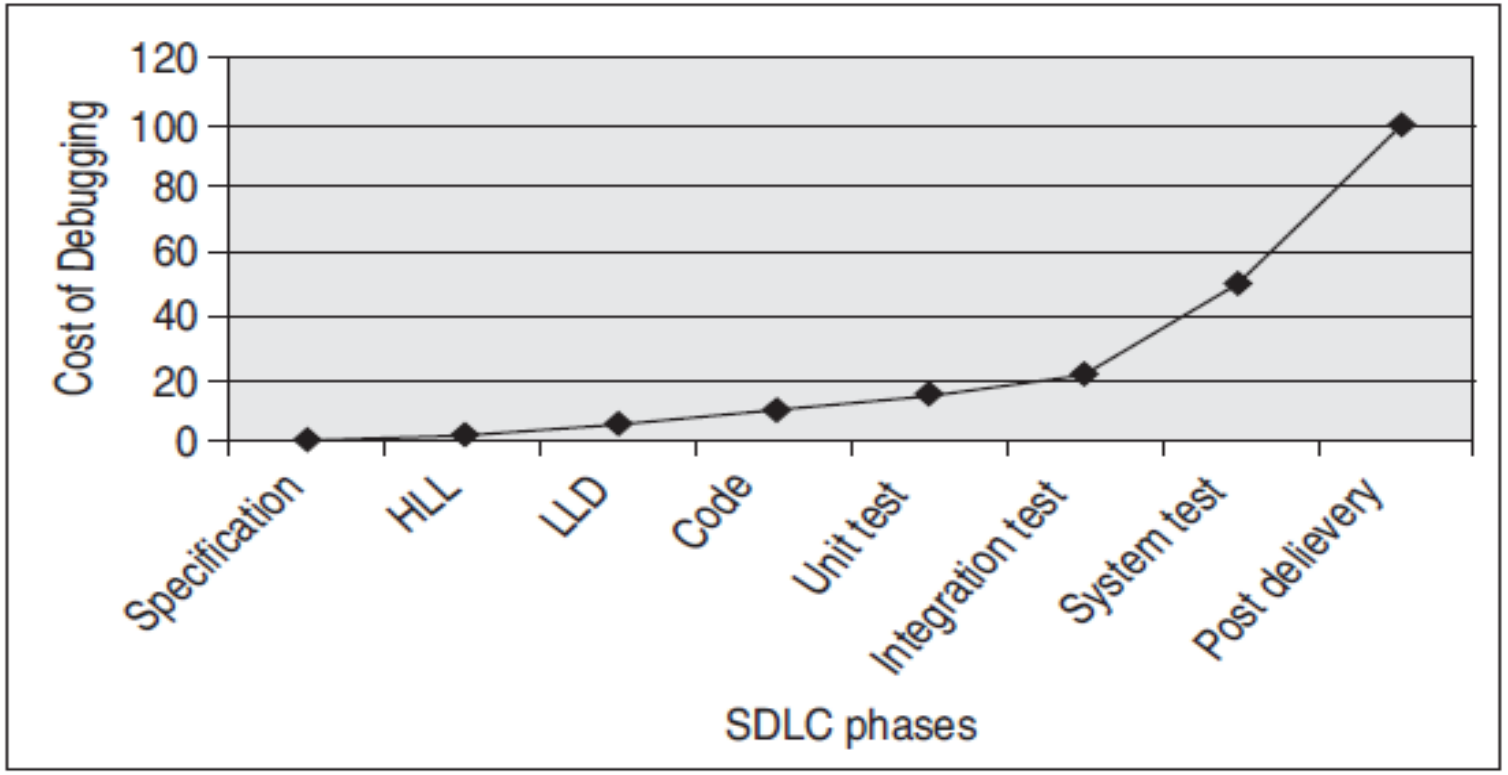
STATES OF A BUG



BUGS AFFECT ECONOMICS OF SOFTWARE TESTING

- Testing prior to coding is 50% effective in detecting errors
- After coding, it is 80% effective
- 10 times as costly to correct an error after coding
- 100 times as costly to correct a production error

- If the bugs embedded in earlier stages go undetected, it is more difficult to detect them in later stages
- *cost of a bug = Detection Cost + Correction Cost*



BUG CLASSIFICATION BASED ON CRITICALITY

Critical Bugs

- has the worst effect such that it stops or hangs the normal functioning of the software

Major Bug

- does not stop the functioning of the software
- It causes a functionality to fail to meet its requirements as expected

Medium Bugs

- less critical in nature as compared to critical and major bugs.
- outputs are not according to the standards or conventions
- Ex : redundant or truncated output

Minor Bugs

- do not affect the functionality of the software
- occur without any effect on the expected behavior or continuity of the software
- Ex: typographical error or misaligned

BUG CLASSIFICATION BASED ON SDLC

Requirements and Specifications Bugs

- most of the bugs appear in this phase
- If these bugs go undetected, they propagate into subsequent phases
- specified requirements may be incomplete, ambiguous, or inconsistent
- Specifi cation problems lead to wrong missing, or superfl uous features

Design Bugs

- Design bugs may be the bugs from the previous phase
- errors which are introduced in the present phase

1. ***Control flow bugs -*** unreachable paths

2. Logic bugs - improper layout of cases, missing cases, improper combination of cases

3. ***Processing bugs -*** arithmetic error, incorrect conversion, ignoring overflow, improper use of logical operators

4. ***Data flow bugs –*** data flow anomaly errors like un-initialized data, initialized in wrong format, data initialized but not used, data used but not initialized, redefined without any intermediate use

5. ***Error handling bugs -*** If the system fails, then there must be an error message or the system should handle the error in an appropriate way

6. Race condition bugs **-** Race conditions also lead to bugs. Sometimes these bugs are irreproducible

7. ***Boundary-related bugs*** **-** boundaries in loop, time, memory

8. ***User interface bugs*** **-** inappropriate functionality of some features; not doing what the user expects; missing, misleading, or confusing information; wrong content in the help text; inappropriate error messages,

Coding Bugs

- undeclared data, undeclared routines, dangling code, typographical errors, documentation bugs

Interface and Integration Bugs

External interface bugs - invalid timing or sequence assumptions related to external signals, misunderstanding external input and output formats, and user interface bugs.

Internal interface bugs - input and output format bugs, inadequate protection against corrupted data, wrong subroutine call sequence, call parameter bugs

Integration bugs - inconsistencies or incompatibilities between modules

Bugs in data transfer and data sharing between the modules.

System Bugs

- bugs while testing the system as a whole based on various parameters like performance, stress, compatibility, usability
- stress testing is very important
- **system bugs** - system is put under maximum load at every factor like maximum number of users, maximum memory limit, etc. and if it fails

Testing Bugs

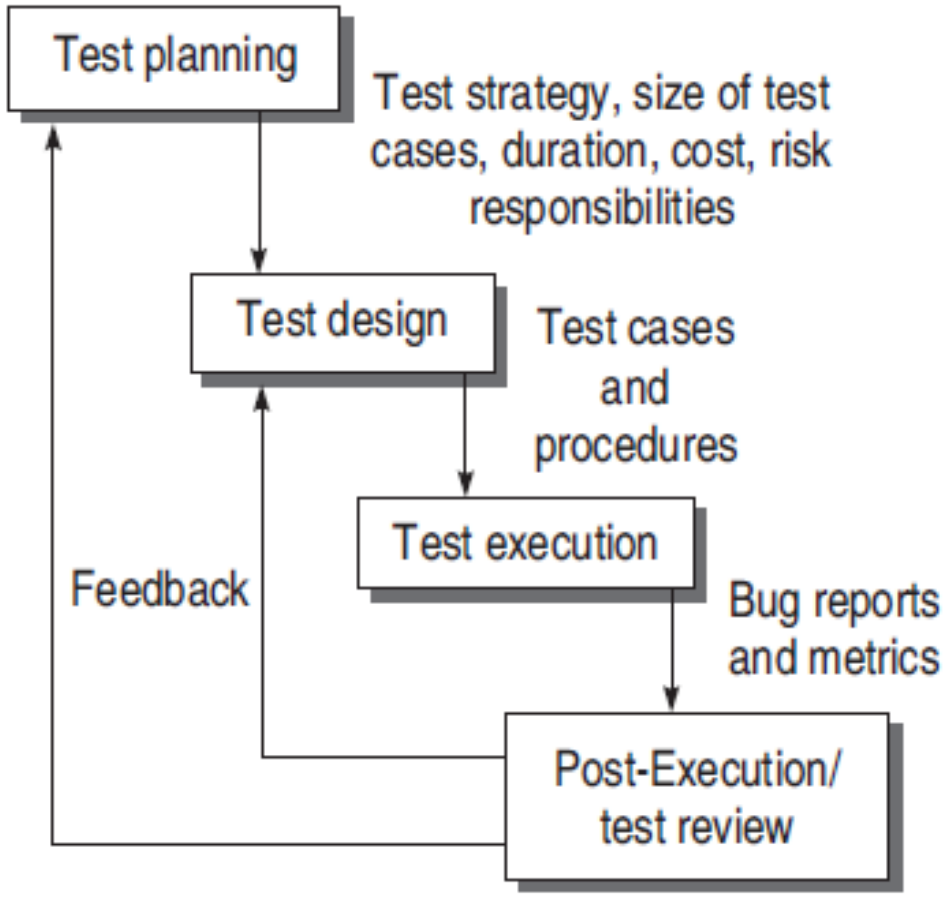
- Testing is also performed by testers – humans.
- failure to notice/report a problem,
- failure to use the most
- promising test case,
- failure to make it clear how to reproduce the problem,
- failure to check for unresolved problems just before the release,
- failure to verify fixes,
- failure to provide summary report

TESTING PRINCIPLES

- ***Effective testing, not exhaustive testing***
- ***Testing is not a single phase performed in SDLC***
- ***Destructive approach for constructive testing***
- ***Early testing is the best policy***
- ***Probability of existence of an error in a section of a program is proportional to the number of errors already found in that section***

- ***Testing strategy should start at the smallest module level and expand towards the whole program***
- ***Testing should also be performed by an independent team***
- ***Everything must be recorded in software testing***
- ***Invalid inputs and unexpected behavior have a high probability of finding an error***
- ***Testers must participate in specifi cation and design reviews***

SOFTWARE TESTING LIFE CYCLE (STLC)



Test Planning - Activities

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

- The major output of test planning is the *test plan document*.
- Test plans are developed for each level of testing.

Activities performed:

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

Test Design

- It is a well-planned process and important

Activities:

Determining the test objectives and their prioritization

Preparing list of items to be tested

Mapping items to test cases - A matrix can be created

This matrix will help in:

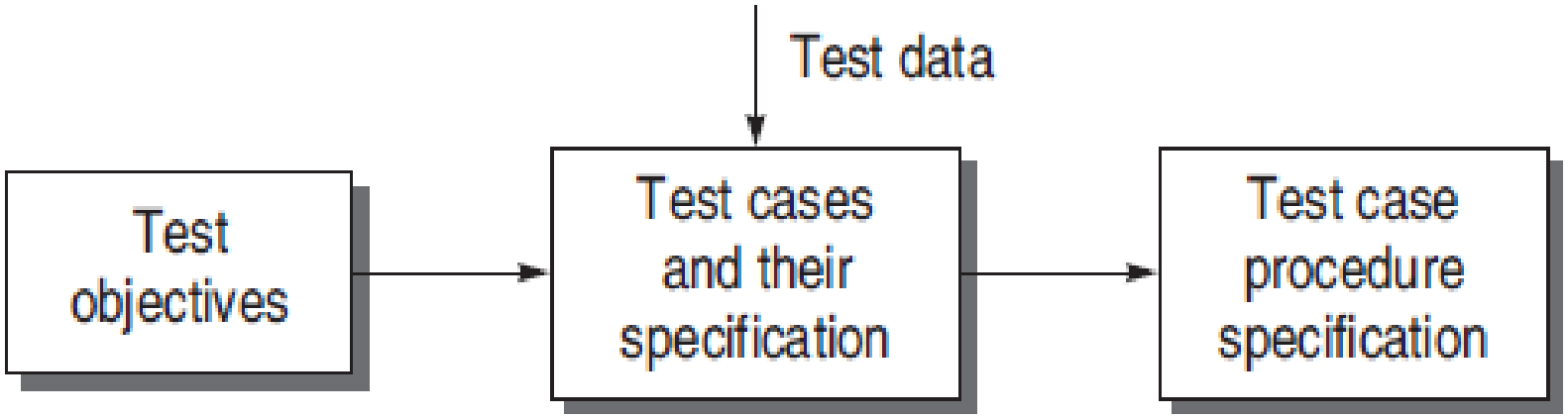
(a) Identifying the major test scenarios.

(b) Identifying and reducing the redundant test cases.

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

- Designing the test cases demands a prior analysis of the program at functional or structural level

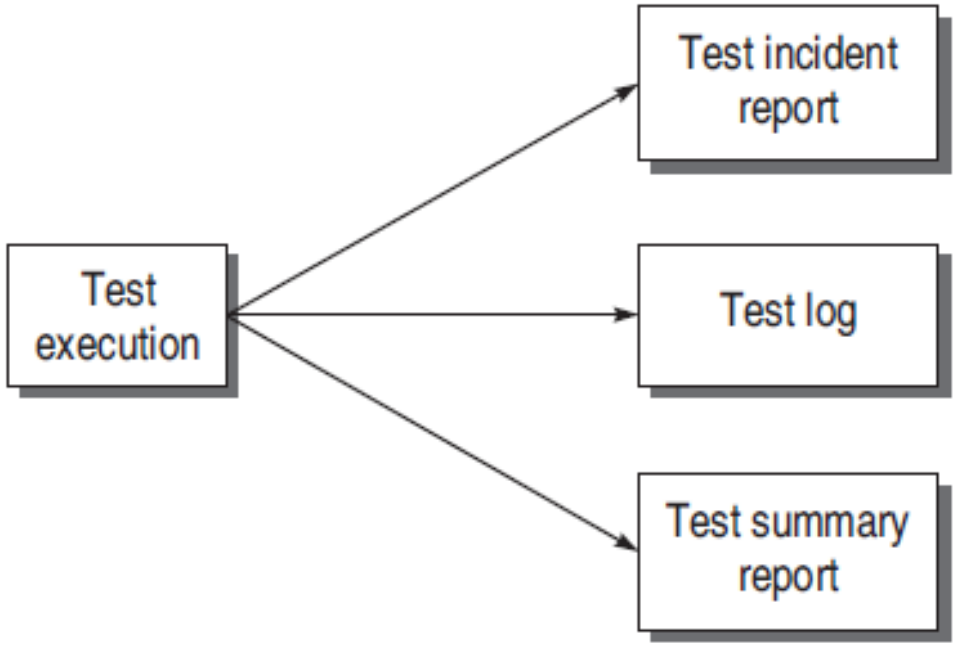
- ***Creating test cases and test data***
- ***Setting up the test environment and supporting tools***
- ***Creating test procedure specification***



Test Execution

- Test cases are executed including verification and validation
- Opt for automation or manual execution
- Test results are documented in the test incident reports, test logs, testing status, and test summary reports

Documents in Test Execution



Responsibilities at various levels for execution of the test cases

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

Post-Execution/Test Review

- Analyze bug-related issues and get feedback so that maximum number of bugs can be removed

Activities performed by developer

- ***Understanding the bug*** The developer analyses the bug reported and builds an understanding of its whereabouts

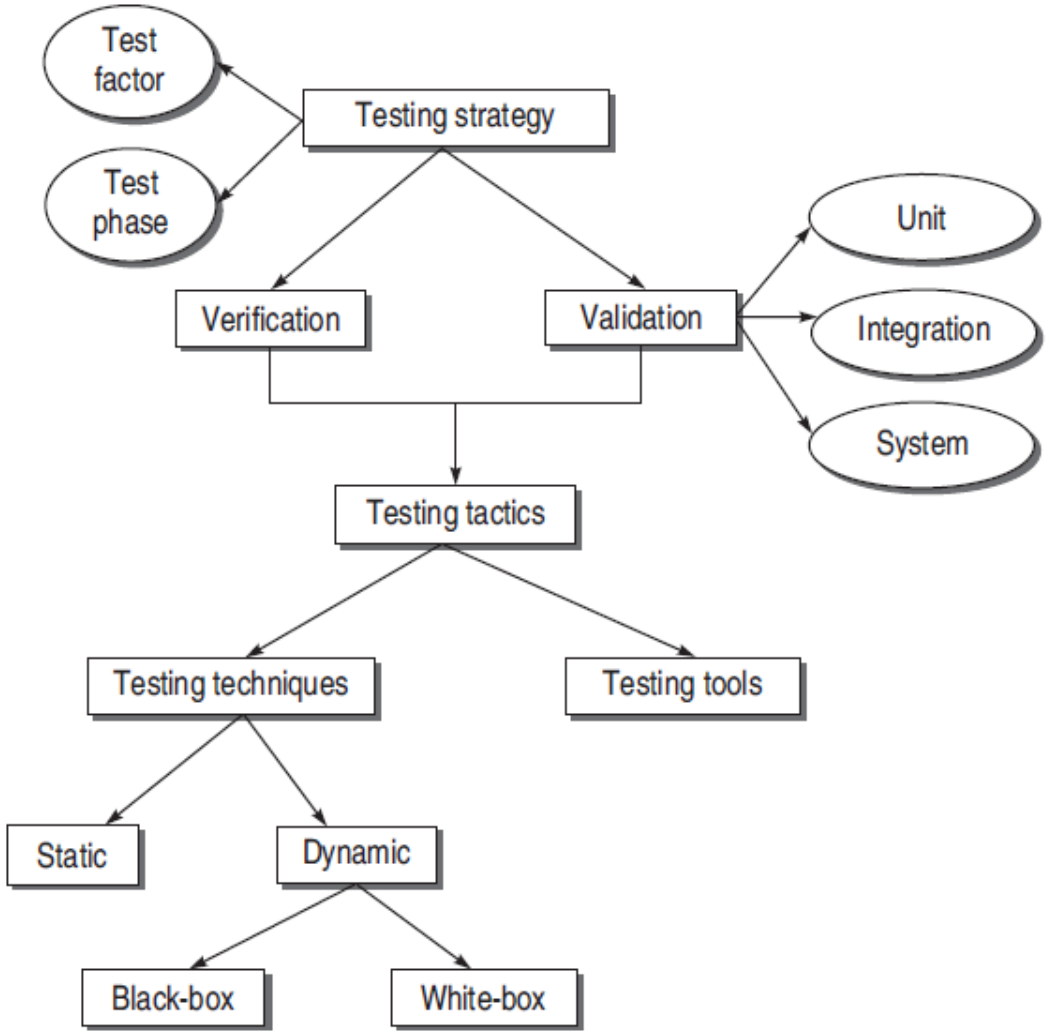
- ***Reproducing the bug*** Next, he confirms the bug by reproducing the bug and the failure that exists. This is necessary to cross-check failures. However, some bugs are not reproducible which increases the problems of developers
- ***Analyzing the nature and cause of the bug -*** A developer starts debugging its symptoms and tracks back to the actual location of the error in the design

- After fixing the bug, the developer reports to the testing team and the modified portion of the software is tested once again
- Final bug report and associated metrics are reviewed and analyzed for overall testing process.

Activities :

- *Reliability analysis*
- *Coverage analysis*
- *Overall defect analysis*

SOFTWARE TESTING METHODOLOGY



SOFTWARE TESTING STRATEGY

- Planning of the whole testing process into a well-planned series of steps.

Test Factors

- Test factors are risk factors or issues related to the system under development.
- Risk factors need to be selected and ranked according to a specific system under development.
- The testing process should reduce these test factors to a prescribed level

Test Phase

- This is another component on which the testing strategy is based. It refers to the phases of SDLC where testing will be performed.
- Testing strategy may be different for different models of SDLC, e.g. strategies will be different for waterfall and spiral models

TEST STRATEGY MATRIX

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

Steps to prepare this matrix:

- ***Select and rank test factors***
- ***Identify system development phases***
- ***Identify risks associated with the system under development***

Test strategy matrix

Test Factors	Test Phase					
	Requirements	Design	Code	Unit test	Integration test	System test

Example test strategy matrix

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

Suppose a new operating system has to be designed, which needs a test strategy.

Steps used:

- ***Select and rank test factors*** A critical factor to be considered for the development of an operating system is portability. This is the effort required to transfer a program from one hardware configuration to another. This factor matters the most, as the operating system has to be compatible with most hardware configurations.

- ***Identify the test phases*** In this step, all test phases affected by the selected test factors are identified.
- ***Identify the risks associated with each test factor and its corresponding test phase*** All the risks are basic concerns associated with each factor in a phase and are expressed in the form of a question
- ***Plan the test strategy for every risk identified*** After identifying the risks, it is required to plan a strategy to tackle them. It helps testers to start working on testing so that risks are mitigated

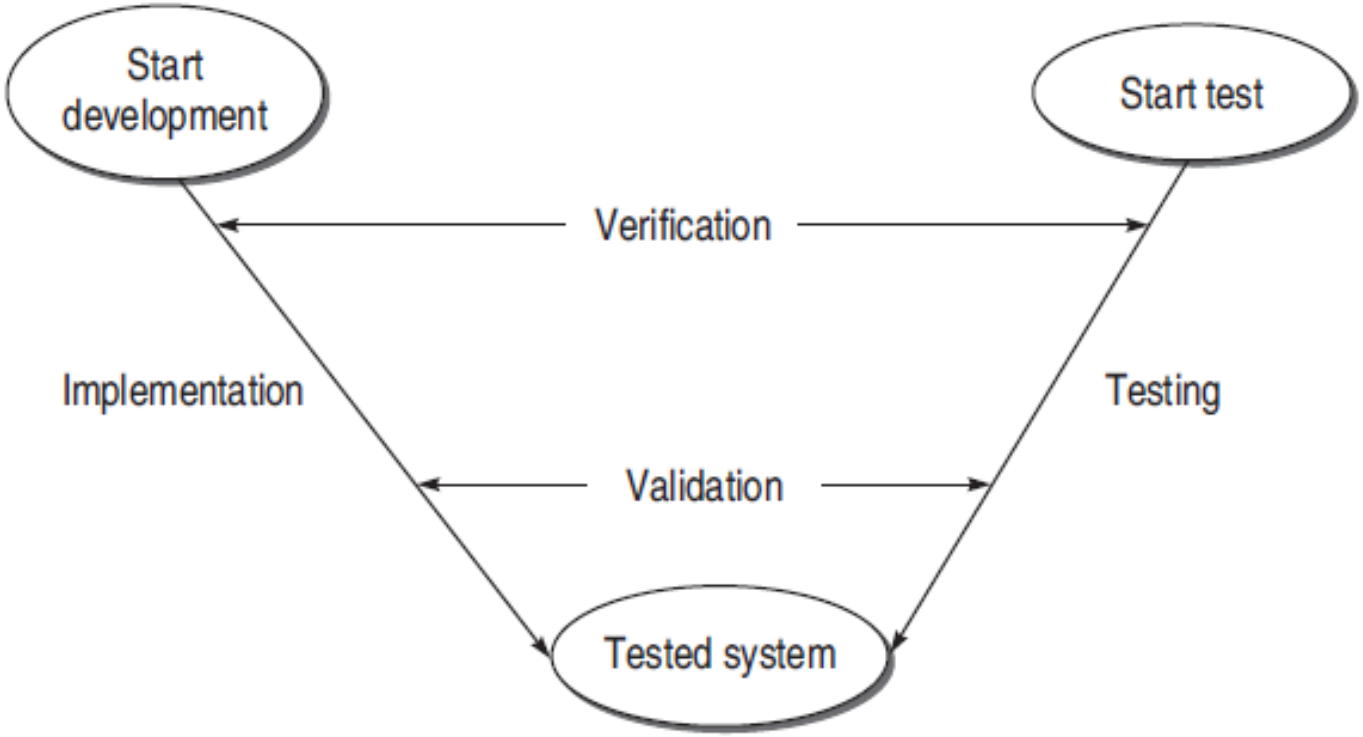
DEVELOPMENT OF TEST STRATEGY

- Testing strategy should start at the component level and finish at the integration of the entire system
- ***Verification is 'Are we building the product right?'***
- ***Validation is 'Are we building the right product?'***

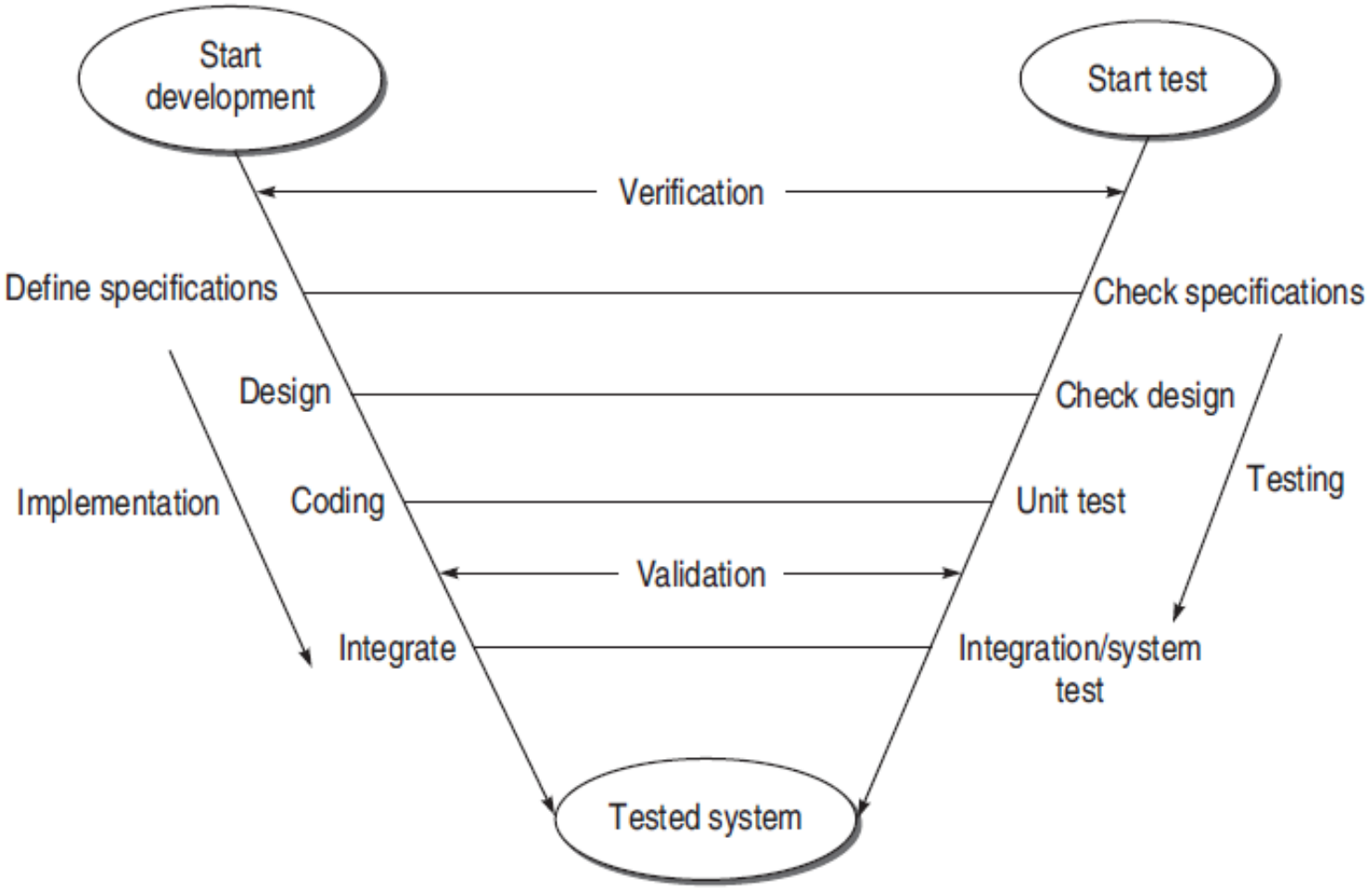
TESTING LIFE CYCLE MODEL

- Verification and validation (V&V) are the building blocks of a testing process
- Life cycle involves continuous testing of the system during the development process
- Life cycle testing is dependent upon the completion of predetermined deliverables at a specified point in the development life cycle

V-Testing Life Cycle Model



Expanded V-testing model



VALIDATION ACTIVITIES

Three levels of validation testing :

Unit Testing

- It is a major validation effort performed on the smallest module of the system.
- If avoided, many bugs become latent bugs and are released to the customer.
- Unit testing is a basic level of testing which cannot be overlooked, and
- Confirms the behavior of a single module according to its functional specifications

Integration Testing

- validation technique which combines all unit-tested modules and performs a test on their aggregation
- interfacing with other modules remain untested
- When one module is combined with another in an integrated environment, interfacing between units must be tested

System Testing

- focuses on testing the entire integrated system
- It incorporates many types of testing, as the full system can have various users in different environments.
- test the validity for specific users and environments.
- validity of the whole system is checked against the requirement specifications.

TESTING TACTICS

Software Testing Techniques

- *Given constraints on time, cost, computer time, etc., the key issue of testing becomes — What subset of all possible test cases has the highest probability of detecting the most errors?*