Unit-5

CODING AND UNIT TESTING

Programming Principles and Guidelines

The main task before a programmer is to write quality code with few bugs in it. The additional constraint is to write code quickly. Writing source code is a skill that can only be acquired by practice. However, based on experience, some general rules and guidelines can be given for the programmer. Good programming (producing correct and simple programs) is a practice independent of the target programming language, although well-structured programming languages make the programmer's job simpler.

1.Common Coding Errors

Software errors (we will use the terms errors, defects and bugs inter changeably in our discussion here; precise definitions are given in the next chapter)are a reality that all programmers have to deal with. Much of effort in developing software goes in identifying and removing bugs. There are various practices that can reduce the occurrence of bugs, but regardless of the tools or methods we use, bugs are going to occur in programs. Though errors can occur in a wide variety of ways, some types of errors are found more commonly.

Here we give a list of some of the commonly occurring bugs.

Memory Leaks

A memory leak is a situation where the memory is allocated to the program which is not freed subsequently. This error is a common source of software failures which occurs frequently in the languages which do not have automatic garbage collection (like C, C++). They have little impact in short programs but can have catastrophic effect on long running systems.

Freeing an Already Freed Resource

In general, in programs, resources are first allocated and then freed. For example, memory is first allocated and then de allocated. This error occurs when the programmer tries to free the already freed resource. The impact of this common error can be catastrophic.

NULL Dereferencing

This error occurs when we try to access the contents of a location that points to NULL. This is a commonly occurring error which can bring a software system down. It is also difficult to detect as it the NULL dereferencing may occur only in some paths and only under certain situations. Often improper initialization in the different paths leads to the NULL reference statement.

Lack of Unique Addresses

Aliasing creates many problems, and among them is violation of unique addresses when we expect different addresses. For example in the string concatenation function, we expect source and destination addresses to be different.

Synchronization Errors

In a parallel program, where there are multiple threads possibly accessing some common resources, then synchronization errors are possible [43, 55].These errors are very difficult to find as they don't manifest easily. But when they do manifest, they can cause serious damage to the system. There are different categories of synchronization errors, some of which are:

1. Deadlocks

2. Race conditions

3. Inconsistent synchronization

Array Index Out of Bounds

Array index often goes out of bounds, leading to exceptions. Care needs to be taken to see that the array index values are not negative and do not exceed their bounds.

Arithmetic exceptions

These include errors like divide by zero and floating point exceptions. There sult of these may vary from getting unexpected results to termination of the program.

Off by One

This is one of the most common errors which can be caused in many ways. For example, starting at 1 when we should start at 0 or vice versa, writing<— N instead of < N or vice versa, and so on.

Enumerated data types

Overflow and underflow errors can easily occur when working with enumerated types, and care should be taken when assuming the values of enumerated data types.

Illegal use of &; instead of &;&:

This bug arises if non short circuit logic (like & or |) is used instead of short circuit logic (&& or ||). Non short circuit logic will evaluate both sides of the expression. But short circuit operator evaluates one side and based on the result, it decides if it has to evaluate the other side or not.

String handling errors

There are a number of ways in which string handling functions like strcpy, sprintf, gets etc can fail. Examples are one of the operands is NULL, the string is not NULL terminated, or the source operand may have greater size than the destination. String handling errors are quite common.

Buffer overflow

Though buffer overflow is also a frequent cause of software failures, in todays world its main impact is that it is a security flaw that can be exploited by a malicious user for executing arbitrary code. When a program takes an input which is being copied in a buffer, by giving a large (and malicious) input, a malicious user can overflow the buffer on the stack. By doing this, the return address can get rewritten to what ever the malicious user has planned. So, when the function call ends, the control goes to where the malicious user has planned, which is typically some malicious code to take control of the computer or do some harmful actions.

2.Structured Programming

 Structured programming started in the70s, primarily against indiscriminate use of control constructs like gotos

 Goal was to simplify program structure so it is easier to argue about programs

 Is now well established and followed. A program has a static structure which is the ordering of statements in the code – and this is a linear ordering

 A program also has dynamic structure –order in which statements are executed

 Both dynamic and static structures are ordering of statements

 Correctness of a program must talk about the dynamic structure

 To show a program as correct, we must show that its dynamic behavior is as expected

 But we must argue about this from the code of the program, i.e. the static structure

 I.e program behavior arguments are made on the static code

 This will become easier if the dynamic and static structures are similar

 Closer correspondence will make it easier to understand dynamic behavior from static structure

 This is the idea behind structured programming

 Goal of structured programming is to write programs whose dynamic structure is same as static

 I.e. statements are executed in the same order in which they are present in code

 As statements organized linearly, the objective is to develop programs whose control flow is linear

Meaningful programs cannot be written asset of simple statements

 To achieve the objectives, structured constructs are to be used

 These are single-entry-single-exit constructs

 With these, execution of the statements can be in the order they appear in code

 The dynamic and static order becomes same

 Main goal was to ease formal verification of programs

 For verification, the basic theorem to be shown for a program S is of the form P {S} Q

 P – precondition that holds before S executes

 Q – post condition that holds after S has executed and terminated

The most commonly used single-entry and single-exit statements are:

Selection: if B then SI else S2

if B then SI

Iteration: While B do S

repeat S until B

Sequencing: SI; S2; S3;...

3.Information Hiding

 Software solutions always contain data structures that hold information

 Programs work on these DS to perform the functions they want

 In general only some operations are performed on the information, i.e. the data is manipulated in a few ways only

 E.g. on a bank's ledger, only debit, credit, check cur balance etc are done Information hiding – the information should be hidden; only operations on it should be exposed

 I.e. data structures are hidden behind the access functions, which can be used by programs

 Info hiding reduces coupling

 This practice is a key foundation of OO and components, and is also widely used today

4.Some Programming Practices

The concepts discussed above can help in writing simple and clear code with few bugs. There are many programming practices that can also help towards that objective. We discuss here a few rules that have been found to make code easier to read as well as avoid some of the errors. Some of these practices are from.

Control Constructs: As discussed earlier, it is desirable that as much as possible single-entry, single-exit constructs be used. It is also desirable to use a few standard control constructs rather than using a wide variety of constructs, just because they are available in the language.

Gotos: Gotos should be used sparingly and in a disciplined manner. Only when the alternative to using gotos is more complex should the gotos be used. In any case, alternatives must be thought of before finally using a goto. If a goto must be used, forward transfers (or a jump to a later statement) is more acceptable than a backward jump.

Information Hiding: As discussed earlier, information hiding should be supported where possible. Only the access functions for the data structures should be made visible while hiding the data structure behind these functions.

User-Defined Types: Modern languages allow users to define types like the enumerated type. When such facilities are available, they should be exploited where applicable. For example, when working with dates, a type can be defined for the day of the week. Using such a type makes the program much clearer than defining codes for each day and then working with codes.

Nesting: If nesting of if-then-else constructs becomes too deep, then the logic become harder to understand. In case of deeply nested if-then-elses, it is often difficult to determine the if statement to which a particular else clause is associated. Where possible, deep nesting should be avoided, even If it means a little inefficiency. For example, consider the following construct of nested if-then-elses:

if CI then SI

else if C2 then S2

else if C3 then S3

else if C4 then S4;

If the different conditions are disjoint (as they often are), this structure can be converted into the following structure:

if CI then SI

if C2 then S2

if C3 then S3

if C4 then S4

Module Size: We discussed this issue during system design. A programmer should carefully examine any function with too many statements(say more than 100). Large modules often will not be functionally cohesive. There can be no hard-and-fast rule about module sizes the guiding principle should be cohesion and coupling.

Module Interface: A module with a complex interface should be carefully examined. As a rule of thumb, any module whose interface has more than five parameters should be carefully examined and broken into multiple modules with a simpler interface if possible.

Side Effects: When a module is invoked, it sometimes has side effects of modifying the program state beyond the modification of parameters listed in the module interface definition,

for example, modifying global variables. Such side effects should be avoided where possible, and if a module has side effects, they should be properly documented.

Robustness: A program is robust if it does something planned even for exceptional conditions. A program might encounter exceptional conditions in such forms as incorrect input, the incorrect value of some variable, and overflow. If such situations do arise, the program should not just "crash" or" core dump"; it should produce some meaningful message and exit gracefully.

Switch case with default: If there is no default case in a "switch" statement, the behavior can be unpredictable if that case arises at some point of time which was not predictable at development stage. Such a practice can result in a bug like NULL dereference, memory leak, as well its other types of serious bugs. It is a good practice to always include a default case.

Give Importance to Exceptions: Most programmers tend to give less attention to the possible exceptional cases and tend to work with the mainflow of events, control, and data. Though the main work is done in the main path, it is the exceptional paths that often cause software systems to fail. To make a software system more reliable, a programmer should consider all possibilities and write suitable exception handlers to prevent failures or loss when such situations occur.

5.Coding Standards

Programmers spend far more time reading code than writing code. Over the life of the code, the author spends a considerable time reading it during debugging and enhancement. People other than the author also spend considerable effort in reading code because the code is often maintained by someone other than the author. In short, it is of prime importance to write code in a manner that it is easy to read and understand. Coding standards provide rules and guidelines for some aspects of programming in order to make code easier to read. Most organizations who develop software regularly develop their own standards.

Naming Conventions

Some of the standard naming conventions that are followed often are:

• Package names should be in lower case (e.g., mypackage, edu.iitk.maths)

• Type names should be nouns and should start with uppercase (e.g.,

Day, DateOfBirth, EventHandler)Variable names should be nouns starting with lower case (e.g., name,

amount)Constant names should be all uppercase (e.g., PI, MAXJTERATIONS)

• Method names should be verbs starting with lowercase (e.g., getValue())

• Private class variables should have the _ suffix (e.g., "private int value_").(Some standards will require this to be a prefix.)

• Variables with a large scope should have long names; variables with a small scope can have short names; loop iterators should be named i, j ,k, etc.

Files

There are conventions on how files should be named, and what files should contain, such that a reader can get some idea about what the file contains. Some examples of these conventions are:

• Java source files should have the extension .Java—this is enforced by most compilers and tools.

• Each file should contain one outer class and the class name should be same as the file name.

• Line length should be limited to less than 80 columns and special characters should be avoided. If the line is longer, it should be continued and the continuation should be made very clear.

Statements

These guidelines are for the declaration and executable statements in the source code. Some examples are given below. Note, however, that not everyone will agree to these. That is why organizations generally develop their own guidelines that can be followed without restricting the flexibility of programmers for the type of work the organization does.

• Variables should be initialized where declared, and they should be declared in the smallest possible scope.

• Declare related variables together in a common statement. Unrelated variables should not be declared in the same statement.

• Class variables should never be declared public.

• Use only loop control statements in a for loop.

• Loop variables should be initialized immediately before the loop.

• Avoid the use of break and continue in a loop.

• Avoid the use of do ... while construct.

• Avoid complex conditional expressions—introduce temporary Boolean variables instead.

• Avoid executable statements in conditionals.

Commenting and Layout

Comments are textual statements that are meant for the program reader to aid the understanding of code. The purpose of comments is not to explain in English the logic of the program—if the logic is so complex that it requires comments to explain it, it is better to rewrite and simplify the code instead.

In general, comments should explain what the code is doing or why the code is there, so that the code can become almost standalone for understanding

the system. Comments should generally be provided for blocks of code, andin many cases, only comments for the modules need to be provided.

Incrementally Developing Code

 Coding starts when specs for modules from design is available

 Usually modules are assigned to programmers for coding

 In top-down development, top level modules are developed first; in bottom-up lower levels modules

 For coding, developers use different processes; we discuss some here

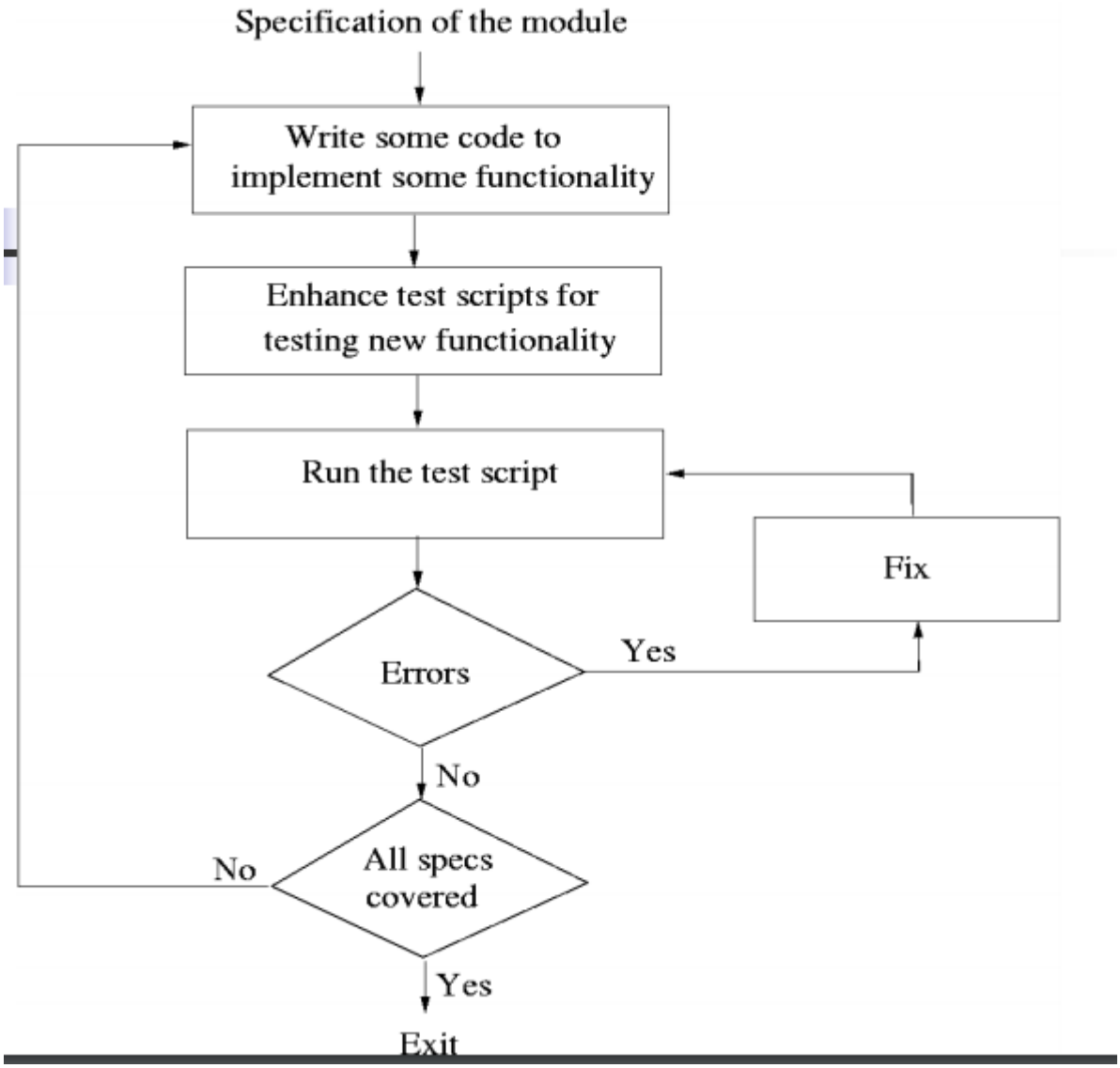
An Incremental Coding Process

 Basic process: Write code for the module, unit test it, fix the bugs

 It is better to do this incrementally –

write code for part of functionality, then test it and fix it, then proceed

 I.e. code is built code for a module Incrementally



Test Driven Development

 This coding process changes the order of activities in coding

 In TDD, programmer first writes the test scripts and then writes the code to pass the test cases in the script

 This is done incrementally

 Is a relatively new approach, and is apart of the extreme programming (XP)

 In TDD, you write just enough code to pass the test

 I.e. code is always in sync with the tests and gets tested by the test cases

 Not true in code first approach, as test cases may only test part of functionality

 Responsibility to ensure that all functionality is there is on test case design, not coding

 Help ensure that all code is testable Focus shifts to how code will be used as test cases are written first

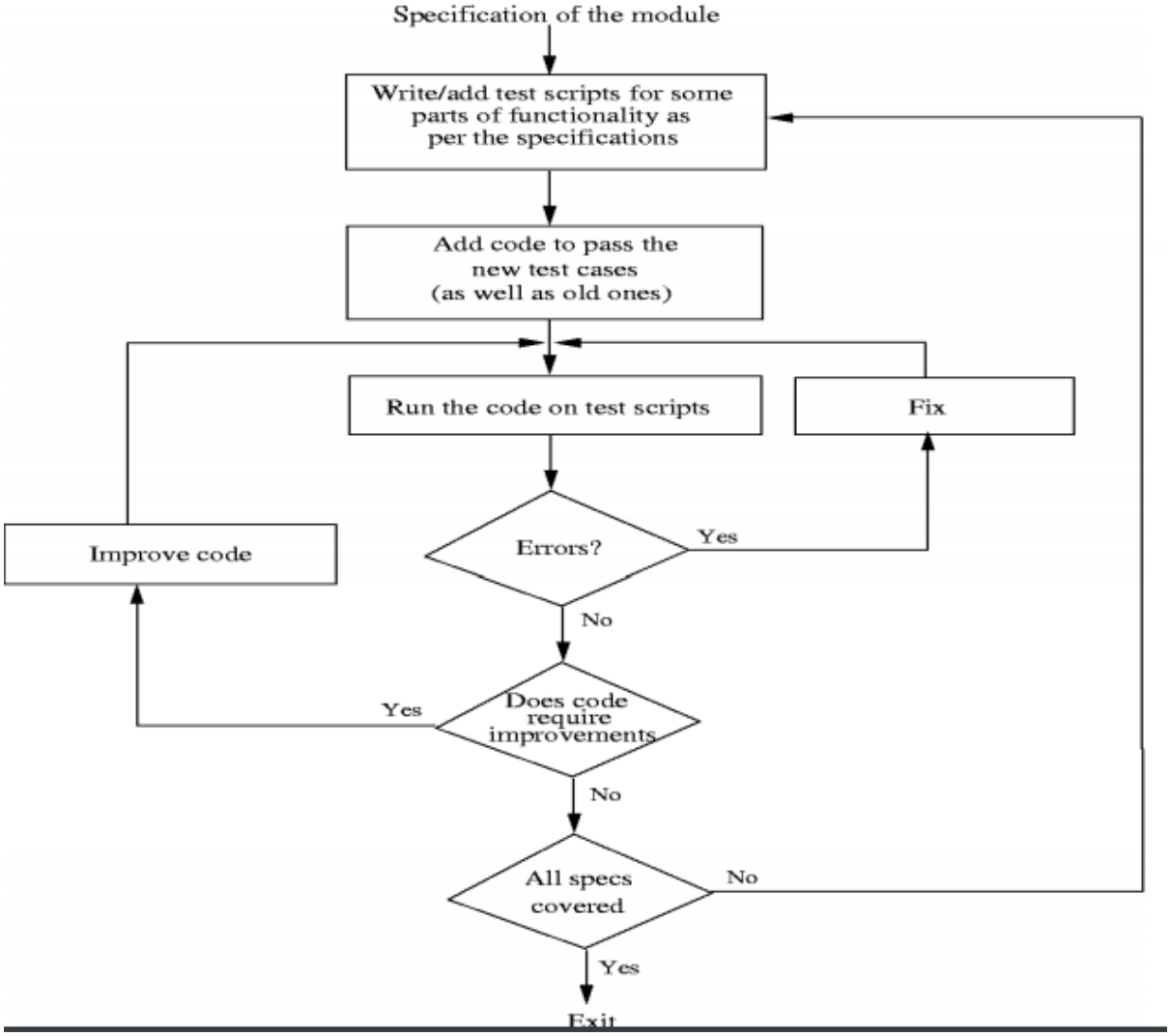
 Helps validate user interfaces specified in the design

 Focuses on usage of code

 Functionality prioritization happens naturally

 Has possibility that special cases for which test cases are not possible get left out

 Code improvement through refactoring will be needed to avoid getting a messy code



Pair Programming

 Also a coding process that has been proposed as key practice in XP

 Code is written by pair of programmers rather than individuals

 The pair together design algorithms, data structures, strategies, etc.

 One person types the code, the other actively reviews what is being typed

 Errors are pointed out and together solutions are formulated

 Roles are reversed periodically

PP has continuous code review, and reviews are known to be effective

 Better designs of algos/DS/logic/…

 Special conditions are likely to be dealt with better and not forgotten

 It may, however, result in loss of productivity

 Ownership and accountability issues are also there

 Effectiveness is not yet fully known

Managing Evolving Code:

During coding process, code written by a programmer evolves. Code by different programmers have to be put together to form the system. Besides normal code changes, requirement changes also cause change. Evolving code has to be managed

1) Source Code Control and Built:

Source code control is an essential step programmers have to do

Generally tools like CVS, VSS are used. A tool consists of repository, which is aν controlled directory structure. The repository is the official source for all the code files. System build is done from the files in the repository only. Tool typically provides many commands to programmers.

- Checkout a file: by this a programmer gets a local copy that can be modified.
- Check in a file: changed files are uploaded in the repository and change is then available to all.
- Tools maintain complete change history andν all older versions can be recovered.

- Source code control is an essential tool forν developing large projects and for coordination.

2) Refactoring:

Refactoring is a technique to improve existing code by improving its design (i.e. the internal structure).In TDD, refactoring is a key step. Refactoring is done generally to recue coupling or increase cohesion. Involves changing of code to improve some design property. No new functionality is added. To mitigate risks associated with refactoring two golden rules.

Refactor in small steps. Have test scripts available to test that the functionality is preserved. With refactoring code is continually improving; refactoring cost is paid by reduced main effort later. There are various refactoring patternsν that have been proposed.

"Bad smells" that suggest that refactoring may be desired

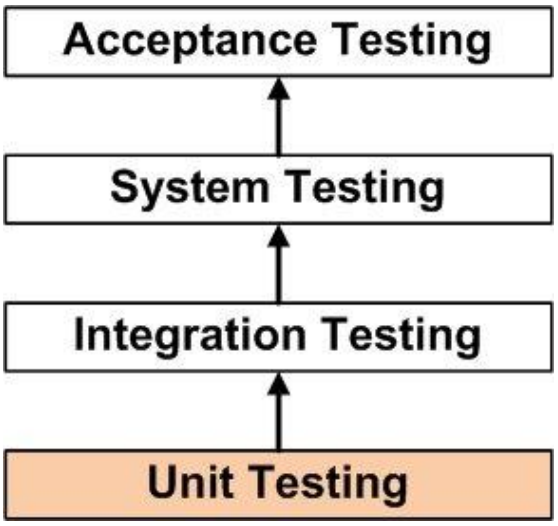
- Duplicate code
- Long method
- Long class
- Long parameter list
- Switch statement
- Speculative generality
- Too much communication between objects, etc

What is Unit Testing?

Unit testing, is a testing technique using which individual modules are tested to determine if there are any issues by the developer himself. It is concerned with functional correctness of the standalone modules.

- The main aim is to isolate each unit of the system to identify, analyze and fix the defects.
- A unit is the smallest testable part of any software. It usually has one or a few inputs and usually a single output.
- In procedural programming, a unit may be an individual program, function, procedure, etc.

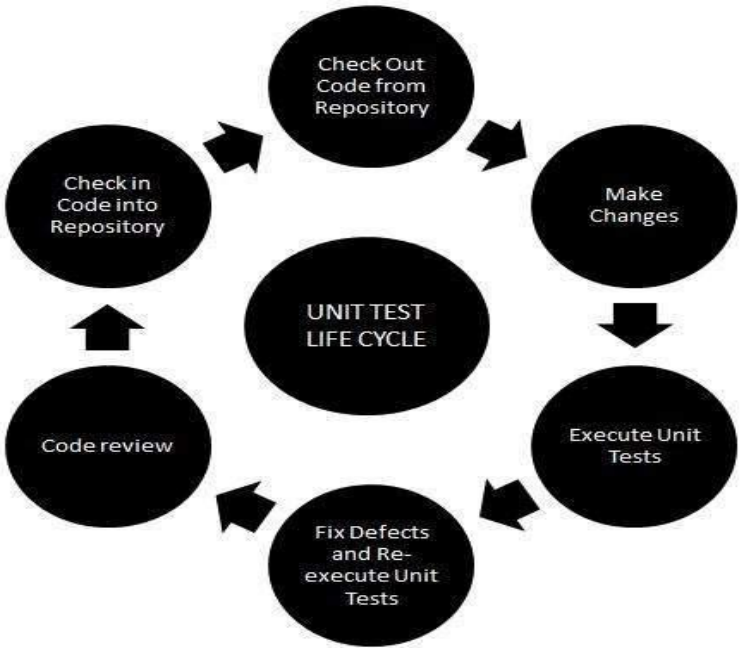
- In object-oriented programming, the smallest unit is a method, which may belong to a base/ super class, abstract class or derived/ child class. Unit testing frameworks, drivers, stubs, and mock/ fake objects are used to assist in unit testing.



Unit Testing - Advantages:

- Reduces Defects in the newly developed features or reduces bugs when changing the existing functionality.
- Reduces Cost of Testing as defects are captured in very early phase.
- Improves design and allows better refactoring of code.
- Unit Tests, when integrated with build gives the quality of the build as well.

Unit Testing Life Cycle:



Unit Testing Techniques:

- **Black Box Testing -** Using which the user interface, input and output are tested.
- **White Box Testing -** used to test each one of those functions behaviour is tested.
- **Gray Box Testing -** Used to execute tests, risks and assessment methods.

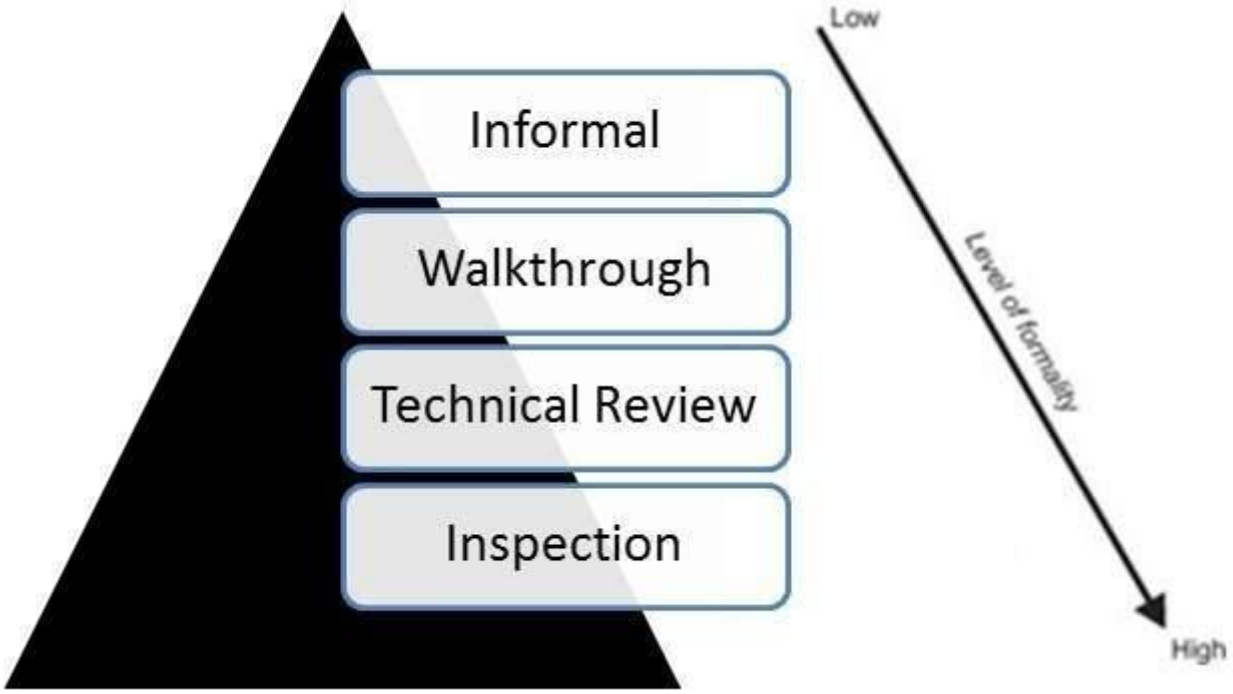
What is code Inspection?

Code Inspection is the most formal type of review, which is a kind of static testing to avoid the defect multiplication at a later stage.

- The main purpose of code inspection is to find defects and it can also spot any process improvement if any.
- First proposed by Fagan in 70s
- An inspection report lists the findings, which include metrics that can be used to aid improvements to the process as well as correcting defects in the document under review.
- Preparation before the meeting is essential, which includes reading of any source documents to ensure consistency.
- Inspections are often led by a trained moderator, who is not the author of the code.

- The inspection process is the most formal type of review based on rules and checklists and makes use of entry and exit criteria.
- It usually involves peer examination of the code and each one has a defined set of roles.
- After the meeting, a formal follow-up process is used to ensure that corrective action is completed in a timely manner.
- It is a structured process with defined roles for the participants. The focus is on identifying problems, not resolving them. Review data is recorded and used for monitoring the effectiveness.

Where Code Inspection fits in?



Advantages:

- It is very effective for finding the defects from the code.

Disadvantages:

- The code inspection process is time consuming.
- A larger number of people are involved in the code inspection process, it can be turned out to be costly process.

- Due to these drawbacks of the code inspection, a single person may carry out the code inspection and code reviews.

METRICS

A) Metrics for Size:

1)KLOC (Thousands (kilo) of lines of code) is a measure of the size of a computer program.

2) LOC are "artifact" of all software development project that can be easily counted.

3) LOC measures are programming language dependent. They penalize well-designed but shorter programs. They cannot easily accommodate non-procedural languages.

4)When used in estimation, LOC requires may be difficult to determine; planner must estimate the LOC long before analysis and design have been completed

5)**Halstead's Volume**

n1: no of distinct operators

n2: no of distinct operands

N1: total occurrences of operators

N2: Total occurrences of operands

Vocabulary, n = n1 + n2

Length, N = N1 + N2

Volume, V = N log2 (n)

B) **Metrics for Complexity**

1) Cyclomatic complexity is software metric, used to indicate the **complexity** of a program.

2) Represent the program by its control flow graph with e edges, n nodes, and p parts Cyclomatic complexity is defined as $V(G) = en+p\nu$

3)This is same as the number of linearly independent cycles in the graph And is same as the number of decisions (conditionals) in the program plus one

TESTING

Detecting defects in Testing:

During testing, software under testv (SUT) executed with set of test cases .

Failure during testing => defects are present No failure => confidence grows, but cannot say "defects are absent" To detect defects, must cause failuresv during testing.

TESTING CONCEPTS:

Error,Fault and Failure:

Fault : It is a condition that causes the software to fail to perform its required function.

Examples: – Software bug – Random hardware fault – Memory bit "stuck" – Omission or commission fault in data transfer etc

Error : Refers to difference between Actual Output and Expected output.

A fault may lead to an error, i.e., error is a mechanism by which the fault becomes apparent

Example: – memory bit got stuck but CPU does not access this data – Software "bug" in a subroutine is not "visible" while the subroutine is not called

Failure : It is the inability of a system or component to perform required function according to its specification.

One of the goals of safety-critical systems is that error should not result in system failure

Test case, Test suite and Test harness:

What is Test case?

A test case is a document, which has a set of test data, preconditions, expected results and post conditions, developed for a particular test scenario in order to verify compliance against a specific requirement.

Test Case acts as the starting point for the test execution, and after applying a set of input values, the application has a definitive outcome and leaves the system at some end point or also known as execution post condition.

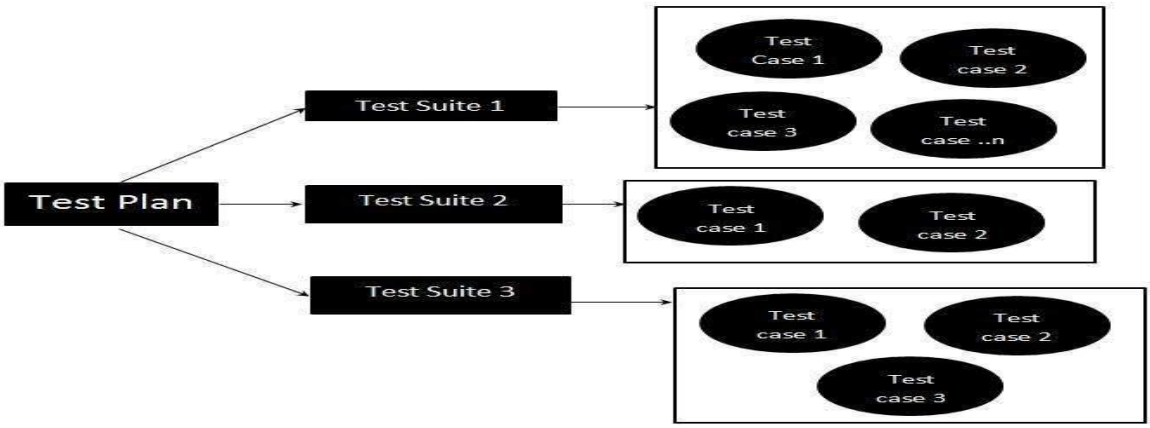
What is a Test Suite?

Test suite is a container that has a set of tests which helps testers in executing and reporting the test execution status. It can take any of the three states namely Active, Inprogress and completed.

A Test case can be added to multiple test suites and test plans. After creating a test plan, test suites are created which in turn can have any number of tests.

Test suites are created based on the cycle or based on the scope. It can contain any type of tests, - functional or Non-Functional.

Test Suite - Diagram:



What is Harness?

Test Harness, also known as automated test framework mostly used by developers. A test harness provides stubs and drivers, which will be used to replicate the missing items, which are small programs that interact with the software under test.

Test Harness Features:

- To execute a set of tests within the framework or using the test harness
- Provide a flexibility and support for debugging
- To capture outputs generated by the software under test
- To record the test results(pass/fail) for each one of the tests
- Helps the developers to measure code coverage at code level.

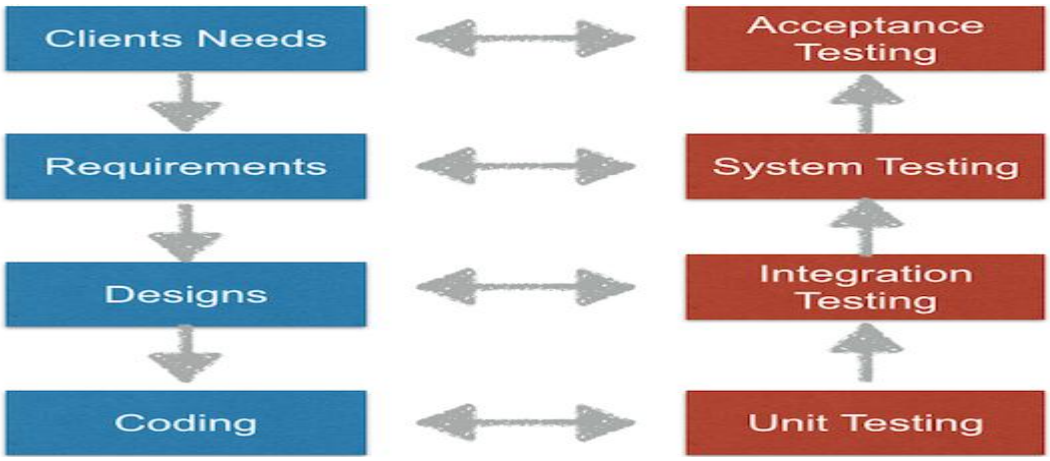
Psychology of Testing:

Psychology of testing is a type of testing which fully depends on the mindset of developers and tester. When we are building the software, we working positively towards the software never think about negative things. The mindset should be different while testing and reviewing developing software. With the correct mindset, the programmer can test their own code. At a certain point independence tester often makes the tester more effective to finding defects. That's why independence tester highly recommendable. Because they are testing specialists or professional testers.

Levels of Independence Tester in Psychology of testing

- Tests designed by the person who wrote the software under test
- Tests designed by another person
- Tests designed by a person from a different organizational group or test specialists
- Tests designed by a person from a different organization or company

Levels of Testing: Levels Of Testing:



Unit/component testing

Unit testing aims to verify each part of the software by isolating it and then perform tests to demonstrate that each individual component is correct in terms of fulfilling requirements and the desired functionality.

This type of testing is performed at the earliest stages of the development process, and in many cases it is executed by the developers themselves before handing the software over to the testing team.

The advantage of detecting any errors in the software early in the day is that by doing so the team minimises software development risks, as well as time and money

Integration testing

Integration testing aims to test different parts of the system in combination in order to assess if they work correctly together. By testing the units in groups, any faults in the way they interact together can be identified.

Testers can adopt either a bottom-up or a top-down integration method.
In bottom-up integration testing, testing builds on the results of unit testing by testing higher-level combination of units (called modules) in successively more complex scenarios.

It is recommended that testers start with this approach first, before applying the top-down approach which tests higher-level modules first and studies simpler ones later.

System testing

The next level of testing is system testing. As the name implies, all the components of the software are tested as a whole in order to ensure that the overall product meets the requirements specified.

System testing enables testers to ensure that the product meets business requirements, as well as determine that it runs smoothly within its operating environment. This type of testing is typically performed by a specialized testing team.

Acceptance testing

Finally, Acceptance Testing is the level in the software testing process where a product is given the green light or not. The aim of this type of testing is to evaluate whether the system complies with the end-user requirements and if it is ready for deployment.

By performing acceptance tests, the testing team can find out how the product will perform when it is installed on the user's system. There are also various legal and contractual reasons why acceptance testing has to be carried out.

Hierarchy: The four levels of tests shouldn't only be seen as a hierarchy that extends from simple to complex, but also as a sequence that spans the whole development process from the early to the later stages. Note however that later does not imply that acceptance testing is done only after say 6 months of development work. In a more agile approach, acceptance testing can be carried out as often as every 2-3 weeks

Testing Process:

Testing is a process rather than a single activity. This process starts from test planning then designing test cases, preparing for execution and evaluating status till the test closure. It is a task intended to detect defects in software by contrasting a computer program's expected results with its actual results for a given set of inputs.

Test Plan

Test planning, the most important activity to ensure that there is initially a list of tasks and milestones in a baseline plan to track the progress of the project. It also defines the size of the test effort.

It is the main document often called as master test plan or a project test plan and usually developed during the early phase of the project.

Test Planning Activities:

- To determine the scope and the risks that need to be tested and that are NOT to be tested.
- Documenting Test Strategy.
- Making sure that the testing activities have been included.
- Deciding Entry and Exit criteria.
- Evaluating the test estimate.

Test Case Design Techniques:

With the assistance of test case design techniques, one can effortlessly test various components of the software, such as its internal structure, codes, design, test cases, and more. Moreover, they enable software developers and testers to create and design test cases that simplify the process of testing and help them execute test cases effortlessly. Therefore, if you

want to ensure the quality, effectiveness, reliability and consistency of your software product, it is vital for you to choose the right test case design technique.

Things to Consider while Choosing Test Design Techniques:

- **Risks & Objectives:** The objective and purpose of each software testing process differs from another, hence their requirements for test design technique also differs from each other.

- **Time & Budget:** Without time & budget constraints, testers can easily experiment with multiple techniques. However, these constraints force them to choose a technique that enables them to get accurate outputs within a limited period of time.

Following are some design techniques:

- Boundary Value Analysis (BVA)
- Equivalence Partitioning (EP)
- Decision Table Testing
- State Transition Diagrams
- Use Case Testing

Test Case Execution:

Test execution is the process of executing the code and comparing the expected and actual results.

Following factors are to be considered for a test execution process:

1) Based on a risk, select a subset of **test** suite to be executed for this cycle.

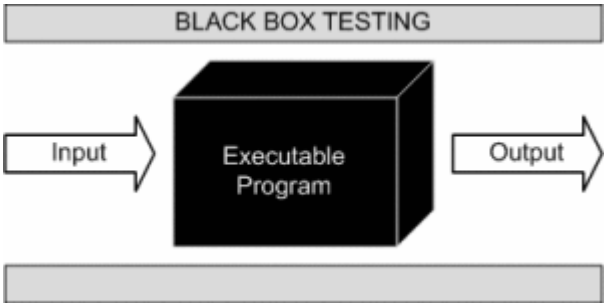
2) Assign the test cases in each test suite to testers for execution.

Executing test cases may require drivers or stubs to be written; some tests can be auto, others manual. A separate test procedure document may be prepared. Test summary report is often an output – gives aν summary of test cases executed, effort, and defects found, etc

Monitoring of testing effort is important to ensure that sufficient time is spent Computer time also is an indicator of how testing isν proceeding

Black box testing:

Black-box testing is a method of software testing that examines the functionality of an application without peering (looking) into its internal structures or workings. This method of test can be applied virtually to every level of software testing: unit, integration, system and acceptance. It is sometimes referred to as specification-based testing.



This method attempts to find errors in the following categories:

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external database access
- Behavior or performance errors
- Initialization and termination errors

Advantages / Pros of Black Box Testing

- Unbiased tests because the designer and tester work independently
- Tester is free from any pressure of knowledge of specific programming languages to test the reliability and functionality of an application / software
- Facilitates identification of contradictions and vagueness in functional specifications
- Test is performed from a user's point-of-view and not of the designer's
- Test cases can be designed immediately after the completion of specifications

Disadvantages / Cons of Black Box Testing

- Tests can be redundant if already run by the software designer
- Test cases are extremely difficult to be designed without clear and concise specifications
- Testing every possible input stream is not possible because it is time-consuming and this would eventually leave many program paths untested
- Results might be overestimated at times
- Cannot be used for testing complex segments of code

Even though black box testing does not test a system comprehensively, still, it can help one achieve a user's expectation from an application / software.

White box testing:

WHITE BOX TESTING (also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing) is a software testing method in which the internal structure/design/implementation of the item being tested is known to the tester.

It is the testing based on an analysis of the internal structure of the component or system.

Advantages

- Testing can be commenced at an earlier stage. One need not wait for the GUI to be available.
- Testing is more thorough, with the possibility of covering most paths.

Disadvantages

- Since tests can be very complex, highly skilled resources are required, with a thorough knowledge of programming and implementation.
- Test script maintenance can be a burden if the implementation changes too frequently.
- Since this method of testing is closely tied to the application being tested, tools to cater to every kind of implementation/platform may not be readily available.

Software Metric:

Software metric is a standard of measure of a degree to which a software system or process possesses some property. Even if a metric is not a measurement (metrics are functions, while measurements are the numbers obtained by the application of metrics), often the two terms are used as synonyms.

The goal is obtaining objective, reproducible and quantifiable measurements, which may have numerous valuable applications in schedule and budget planning, cost estimation, quality assurance, testing, software debugging, software performance optimization, and optimal personnel task assignments.

Limitations

As software development is a complex process, with high variance on both methodologies and objectives, it is difficult to define or measure software qualities and quantities and to determine a valid and concurrent measurement metric, especially when making such a prediction prior to the detail design. Another source of difficulty and debate is in determining which metrics matter, and what they mean. The practical utility of software measurements has therefore been limited to the following domains:

- Scheduling
- Software sizing
- Programming complexity
- Software development effort estimation
- Software quality

A specific measurement may target one or more of the above aspects, or the balance between them, for example as an indicator of team motivation or project performance.