ARUNAI ENGINEERING COLLEGE

(Affiliated to Anna University)
Velu Nagar,Thiruvannamalai-606 603
www.arunai.org

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BACHELOR OF ENGINEERING

THIRD YEAR

SIXTH SEMESTER

IT8076 - SOFTWARE TESTING

IT8076-SOFTWARE TESTING

UNIT I INTRODUCTION

Testing as an Engineering Activity – Testing as a Process – Testing Maturity Model- Testing axioms – Basic definitions – Software Testing Principles – The Tester's Role in a Software Development Organization – Origins of Defects – Cost of defects – Defect Classes – The Defect Repository and Test Design –Defect Examples- Developer/Tester Support of Developing a Defect Repository.

UNIT II TEST CASE DESIGN STRATEGIES

Test case Design Strategies – Using Black Box Approach to Test Case Design – Boundary Value Analysis – Equivalence Class Partitioning – State based testing – Cause-effect graphing – Compatibility testing – user documentation testing – domain testing – Random Testing – Requirements based testing – Using White Box Approach to Test design – Test Adequacy Criteria – static testing vs. structural testing – code functional testing – Coverage and Control Flow Graphs – Covering Code Logic – Paths – code complexity testing – Additional White box testing approaches- Evaluating Test Adequacy Criteria.

UNIT III LEVELS OF TESTING

The need for Levels of Testing – Unit Test – Unit Test Planning – Designing the Unit Tests – The Test Harness – Running the Unit tests and Recording results – Integration tests – Designing Integration Tests – Integration Test Planning – Scenario testing – Defect bash elimination System Testing – Acceptance testing – Performance testing – Regression Testing – Internationalization testing – Ad-hoc testing – Alpha, Beta Tests – Testing OO systems – Usability and Accessibility testing – Configuration testing –Compatibility testing – Testing the documentation – Website testing.

UNIT IV TEST MANAGEMENT

People and organizational issues in testing – Organization structures for testing teams – testing services – Test Planning – Test Plan Components – Test Plan Attachments – Locating Test Items – test management – test process – Reporting Test Results – Introducing the test specialist – Skills needed by a test specialist – Building a Testing Group- The Structure of Testing Group.The Technical Training Program

UNIT V TEST AUTOMATION

Software test automation – skills needed for automation – scope of automation – design and architecture for automation – requirements for a test tool – challenges in automation – Test metrics and measurements – project, progress and productivity metrics

IT8076
SOFTWARE TESTING

UNIT-1 INTRODUCTION

UNIT - I

INTRODUCTION

Testing as an Engineering Activity:

Software Engineering: is a discipline that produces error free sw within a time & budget

* Software Engineering is defined as the application of systematic disciplined and quantifiable approach to develop an effective and efficient software

Software Testing:

* Testing can be described as a process Used for revealing (script) defects in software and for establishing that the software has attained a specified degree of quality with respect to selected attributes.

Engineering Discipline:

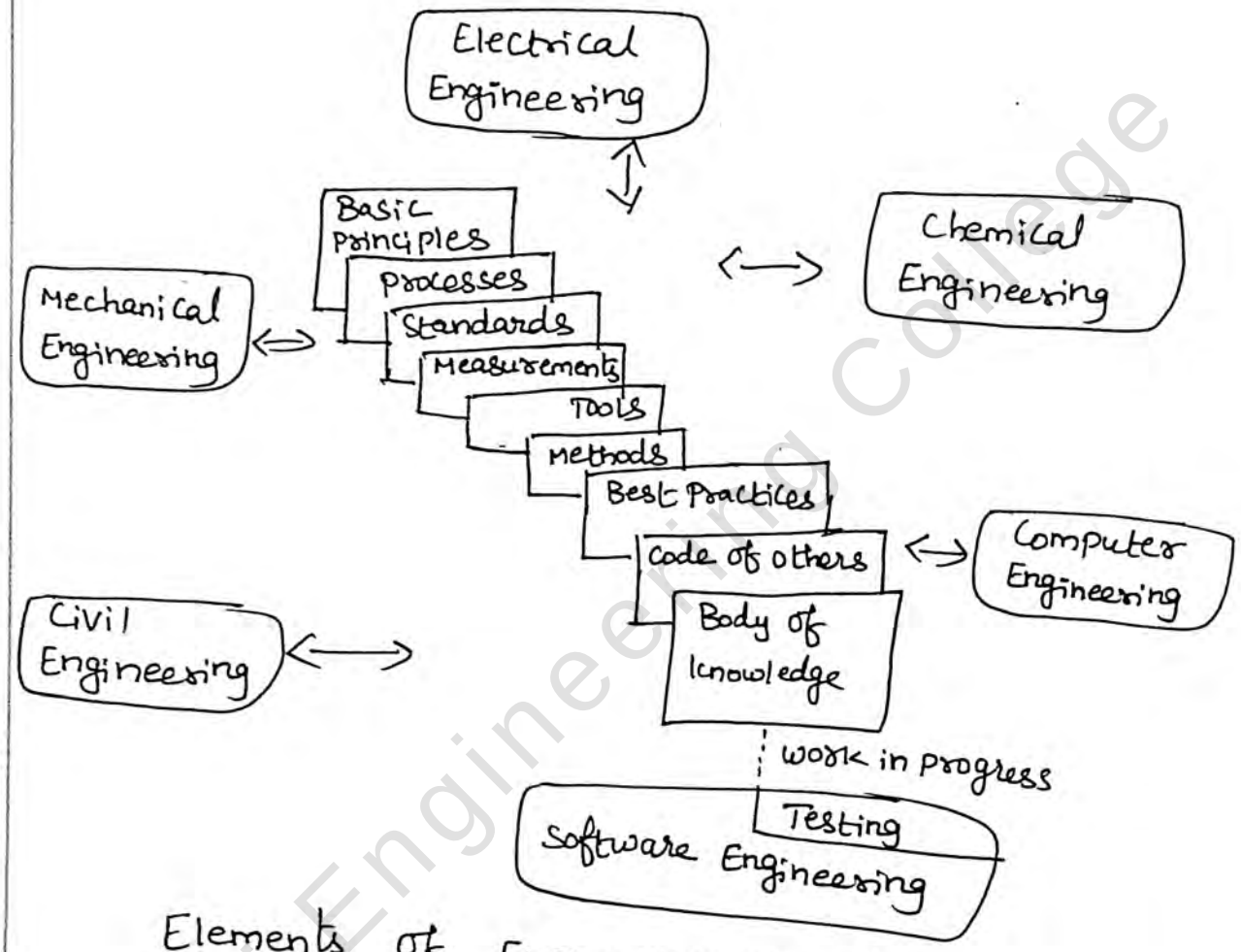
* The Education and training of engineers in each engineering discipline is based on the teaching of related scientific principles, engineering processes, standards, methods, tools, measurement and best practices

* The Software Engineering is represented as the formal Engineering discipline. It related to other engineering discipline, It holds

✓ A defined body of knowledge

✓ A code of Ethics

✓ A certification process.



Elements of Engineering Disciplines

* Validation and Verification process play a key role in quality Determination

* Engineers have proper Education, training and certification

* A well trained and well education domain (ie testing expert) is called "test specialist".

* A test specialist must known, Test related principles, processes, measurements, standards, plans, tools

*

The Role of process in Software Quality:

The need for software products of high quality has pressured in the profession

1) TO identify and quantify quality factors such as usability, testability, maintainability and reliability

2) TO identify engineering practices that support the production of quality products having these favorable attributes.

* In practice, the development of high-quality software are project planning, requirements management, development of formal specification.

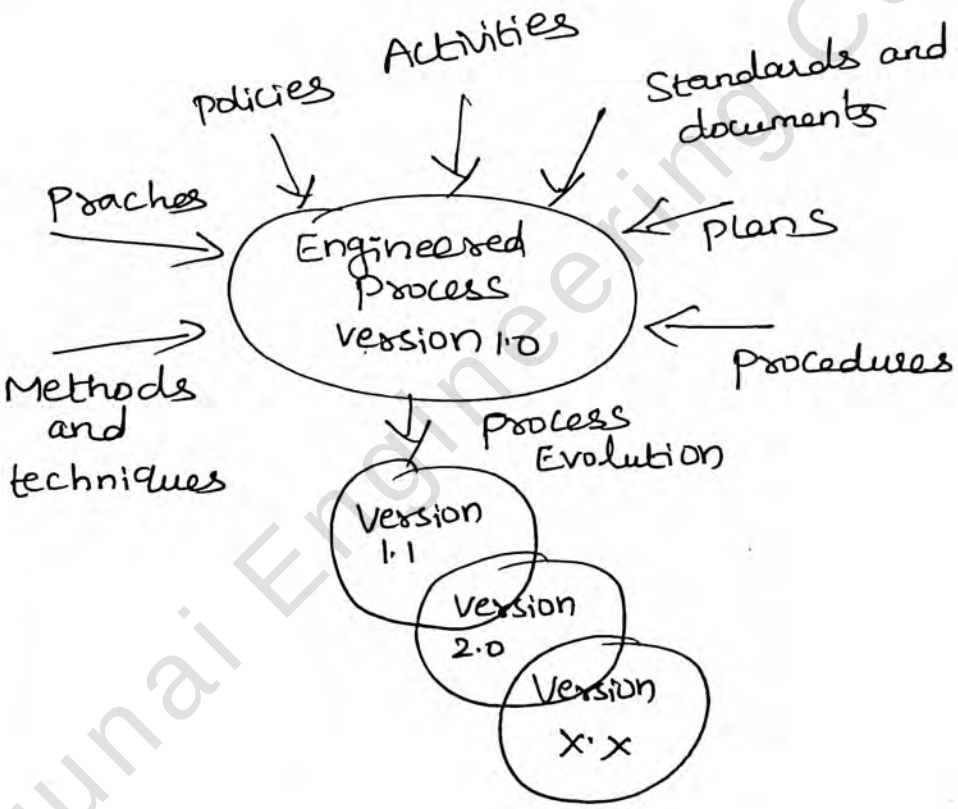
* Structured design with use of information hiding and encapsulation, design and code reuse, inspections and reviews, Product and process measures, education and training of software professions

* Development and application of CASE tools, use of effective testing techniques and integration of testing activities into the entire life cycle.

* process in the software Engineering domain, is the set of methods, practices, standards, documents, activities policies and procedures.

* Software Engineers use to develop and maintain a software system and its associated artifacts, such as project and test plans, design documents Code and manuals.

Components of an Engineered process

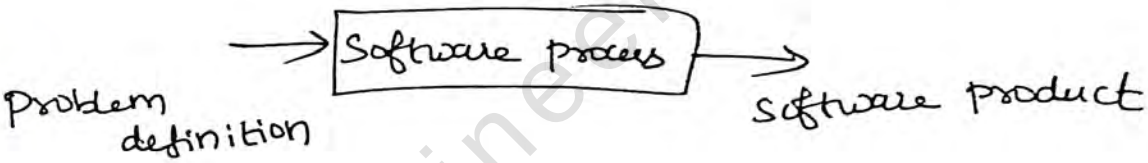


Testing as a process:

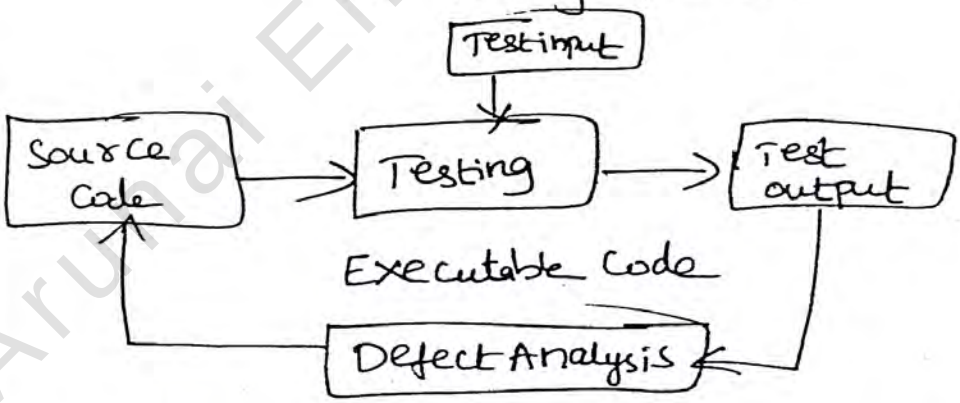
*Testing is a integral part of the software development process

* Testing process means "To find the defect before the end user to find!"

* The purpose of testing is to cover the defect in the system.

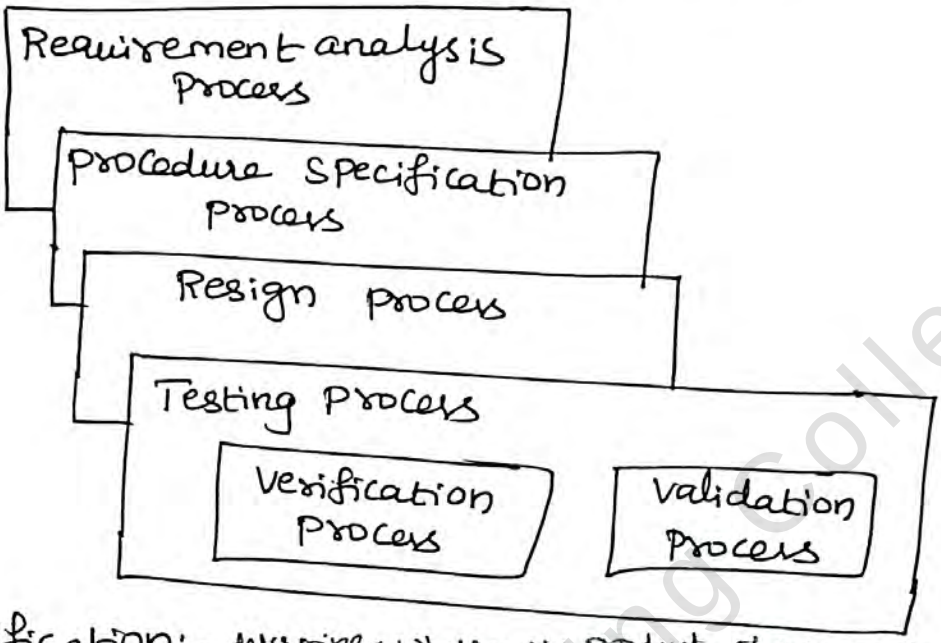


The Testing process



* Testing itself is related to two other processes called Verification and validation.

Software Development processes



Verification: - determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase

* "whether a System is right or wrong"
* Boehm suggests
  "Are we building the product rights?"
* It confirms Software with specification
* It checks all the functional & non functional requirements

Validation: Evaluating a software s/y or component during or at the end of development cycle to determine whether it satisfies specified requirements

* "whether a system is right system or not"
* Boehm suggests
  " Are we building the right products"?
* Its checking, whether expected functionality of a product is achieved or not

Testing Maturity Model:-

* When a software is tested, there are so many processes which are followed in order to attain maximum quality and minimizing defects or errors.

* Test Maturity Model is one of such model which has a set of structured levels.

* The main aim of this model is to optimize the testing processes.

* The testing maturity model or TMM contains five levels They are

Level 1: Initial

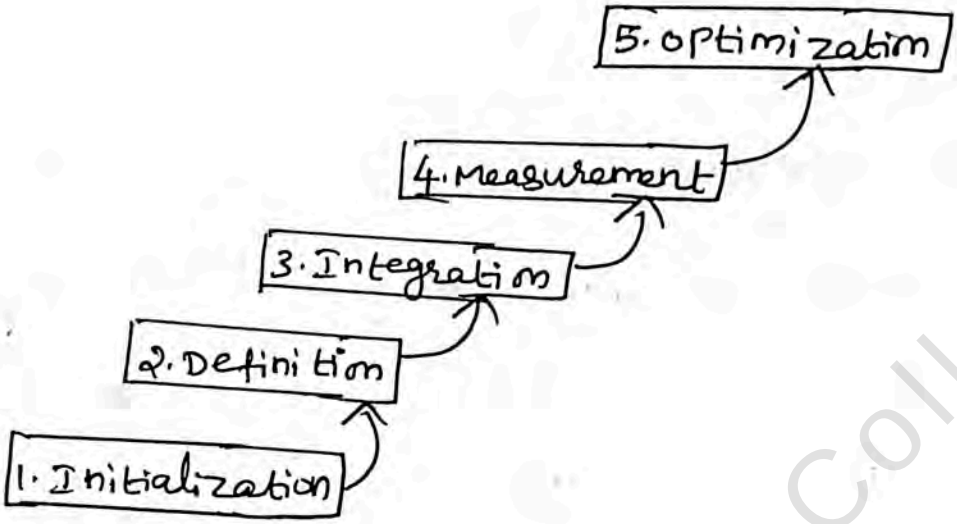
Level 2: Phase Definition

Level 3: Integration

Level 4: Measurement

Level 5: Optimization / Defect Prevention and Quality control

Testing Maturity Model :-



1. Initialization:

* In this level, there is no defined testing process. The main aim of this level is to make sure that the software is running fine and there are no road blocks.

* There are no quality checks before delivering the product.

2. Definition:

* As name suggests this level is all about defining the requirements.

* In this level test case, test strategies and test plans are developed according to the requirements given by the client

to build a software.

* The main aim of this level is to make sure that the product runs according to the requirements and to achieve that test cases and test plan documents are created and followed.

3. Integration:-

* In this level testing is integrated with the software life cycle and becomes a part of it.

Eg: 'V' model has both development and testing phases.

* Testing comes after the development is over and the software under test is handed over to the professional testing team.

* Testing is carried out independently. The whole testing objectives are based on the risk management.

4. Management and Measurement:

The requirements are take care in this level of TMM.

* Testing becomes the part of all the activities in software life cycle

* From reviewing the gathered requirements and design of the software to deciding the quality criteria, is included.

* This builds a clear picture for the organization which in turn helps them to achieve the required Quality.

5. optimization:

* The level is responsible for optimizing the test process itself.

* Testing processes are tested and measures are taken to improve them iteration by iteration.

* In this level defects are prevented by improving the processes throughout the software life cycle so main focus is defect prevention rather than defect detection in each phase

Testing Axioms:

* Axioms means "An established rule, principle or law".
* It is also called "truism".

Types of Axioms

* Axioms are derived from "grand tester master theory"

1. The Stake Holder Axiom - Identify & engage the people or organization who will use & benefit from test
2. The Test Basis Axiom - Identify & agree of goals, requirements visions
3. The test oracle Axiom - Based to be used to determine expected behavior
4. The Scope Management Axiom - Identify items in & out of scope & managing
5. The Coverage Axiom - Define a target for quantify of testing
6. The Delivery Axiom - media & format
7. The Environment Axiom - used for testing, include manage & environment
8. The Event Axiom - Events planned or unplanned test
9. The prioritization Axiom - priority test
10. The Execution Sequence Axioms - sequence of test to ensure "Most important tests"
11. The design Axiom - Identify, adopt & agree model
12. The Respect test Axiom - Retesting & regression testing

Conclusion of Axioms: (Basic Axioms)

Axiom 1

Testing is not possible to test a program Completely

Axiom 2 do not test for all i/p's

↳ Software testing is a risk-based exercise → financial loss, security, loss of money

Test too much – high cost
too little – s/w failure increases

Axiom 3

↳ Testing cannot show the absence of bugs

Axiom 4

↳ The more bugs you find, the more bugs there one

↳ Programmers can do same mistakes

Axiom 5

↳ Not all bugs found will be fixed

Axiom 6

↳ It is difficult to say when a bug is needed a bug

Axiom 7

↳ Specifications are never final

Axiom 8

↳ Software tester is not a expert in any time

Axiom 9

↳ Software testing is a disciplined and technical profession.

Basic Definitions:

Errors:

*An error is a mistake, misconception or misunderstanding on the part of a software developer.

Faults (Defects) - It is a condition that causes the SW to fail to perform its required function.

* A fault (defect) is introduced into the software as the result of an error.

Failures: -

* A failure is the inability of a software system or component to perform its required functions within specified performance requirements

Test cases: How to be tested [Test Scenario - what to be tested

* The usual approach to detecting defects in a pieces of software is for the tester to select a set of input data and than execute the software with the input data under a particular set of condition.

* The tester bundles this information into an item called a test cases.

* A test case in a practical sense is a test-related item which contains the following information

1. A set of test inputs
2. Execution condition - required to running the test
3. Expected outputs

Test:

A test is a group of related test cases, or a group of related test cases and test procedures.

Test oracle:

A test oracle is a document or piece of software that allows testers to determine whether a test has been passed or failed.

Test Bed:

A test bed is an environment that contains all the hardware and software needed to test a software component or a software system

Software Quality:

1. Quality relates to the degree to which a system, system component or process meets customer or user needs or expectations or specified requirements

Software Quality Assurance (SQA) Group:

* SQA group is a team of people with the necessary training and skills to ensure that all necessary actions are taken during the development process so that the resulting S/w confirms to established technical requirements.
* A review is a group meeting whose purpose is to evaluate a S/w artifact or a set of S/w artifacts

Software Testing Principles:

* Testing principles are important to test specialists/ engineers because they provide the foundation for developing testing knowledge and acquiring testing skills.

* It is a foundation for
  a) developing testing knowledge
  b) Acquiring testing skills

* A principle can be defined as
  1. A general or fundamental, law, doctrine or assumption
  2. A rule or code of conduct
  3. The law or facts of nature

* In software domain, principles may also refer to rules or codes of conduct relating to professionals who design, develop, test and maintain software systems

Principles 1:

The testing is the process of exercising a Software Component using a selected set of test cases, with the intent of

i) revealing defects and
ii) Evaluating quality

* The principle supports testing as an execution-based activity to detect defects

* It also supports the separation of testing from debugging, since the intent of the latter to locate defects and repair the software.

* In the case of the latter, the tester executes the software using test cases to evaluate properties such as reliability, usability, maintainability and level of Performance

* Test results are used to compare the actual properties of the Software to those specified in the requirement document as a quality goal.

principle 2:

* when the test objective is to detect defects then a good test case is one that has a high Probability of revealing a yet-undetected error.

* It supports careful test design and provides a criterion with which to evaluate test case design and the effectiveness of the testing effort when the objective is to detect defects.

* The goal of the test is to prove/disprove the hypothesis (ie) determine if the specific defect is present/absent

Principle 3:

* Test results should be inspected meticulously.

* Testers need to carefully inspect and interpret tests results several erroneous and costly scenarios may occur if case is not taken.

Eg: A failure may be overlooked and the test may be granted a "pass" status when in reality the software has failed the test.

* A failure may be suspected when in reality non exists.

* The outcome of quality test may be misunderstood, resulting in unnecessary network, or oversight of a critical problem.

Principle 4:

A test case must contain the expected output or result.

* Expected outputs allow the tester to determine
  i) whether a defect has been revealed
  ii) pass/fail status for the test

* It is very important to have a correct statement of the output so that needless time is not spent due to misconceptions about the outcome of a test

Principle 5:

Test cases should be developed for both valid and invalid input conditions.

* A tester must not assume the software under test will always be provided with valid inputs
* Inputs may be incorrect for several reasons

Eg software users may have misunderstandings or lack information about the nature of the inputs

Principle 6:

The probability of the existence of additional defects in a software component is proportional to the number of defects already detected in the component.

* This principle is that the higher number of defects already detected in a component, the most likely it is to have additional defects when it undergoes further testing.

Eg Two components A and B and tester have found 20 defects in A and 3 defects in B, then the probability of the existence of additional defects in A is higher than B.

* This empirical observation may be due to several causes.

Principle 7:

Testing should be carried out by a group that is independent of the development group.

Tester must realize that

i) Developers have a great deal of pride of their work

ii) on a practical level it may be difficult for them to conceptualize where defects could be found.

Testing is not successful for

1. Misunderstanding of requirements
2. Misunderstanding of specifications relating to software

3. Hard to known the faults

4. Difficulty in locating the defects

It is solved by Independent testing Group (ITG) people.

1. These people one involved in an organization to find the defects

2. Its implemented as a completely separate functional entity in an organization.

3. The SQA group may hold the testers as their members

4. Testers should not play "gotcha" games with developers

5. There should be cooperation in the group, then only a software with highest quality will be produced

Principle 8:

Tests must be repeatable and reusable

* Testing repeatedly is termed as "Regression test"

* Repeat the tests after the defect is repaired

* Repeatable to test, avoid the duplicate test conditions

* It keeps exact information / conditions of the testing.

Principle 9:

Testing Should be planned

* The test plan describes the objective of testing.

* Its used to verify that whether enough time and resources are allocated for the testing tasks or not.

* Testing is easily monitored and managed through test plan.

* Test planning is accommodated with project Planning

* It helps to maintain mutual relationship between Project manager and test manager.

* After the Software is developed by the developer in the correct date, testercan test the software on Particular test, otherwise tester cannot test at the time

* The test risk has to be evaluated.

Principle 10:

Testing Activities Should be integrated into the software life cycle.

* It is no longer feasible to postpone testing activities until after the Code has been written.

* In addition to test planning, some other types of testing activities such as Usability testing can also be

carried out early in the life cycle by using prototypes
* These activities can continue on until the software is delivered to the users.

Principle 11:

Testing is a creative and challenging task.

Difficulties and challenges for the tester include the following

1. Tester needs to have comprehensive knowledge of the software engineering discipline.
2. A tester need to have knowledge from both experience and education as to
   i) How software is specified ii) Designed iii) Develope
3. A tester needs to be able to manage many details
4. Tester must know about test fault and its root cast
5. Testers must have hypothesis knowledge
6. Tester must understand what has to be tested?
7. Before testing, Test cases must be prepared.
8. Test procedures should be designed & recorded
9. Tester must Analyze the test results
10. Tester must have a knowledge about testing tool
11. Testers must be educated & trained in Particular area
12. Tester keep Good relationship wit Clients, Users, designers, developer, requirements engineer

Tester role in a Software Development organization

Tester important roles are

1. To reveal defects
2. To find weak points
3. To know the inconsistent behavior of system
4. To find the situations at which software does not work
5. To get more programming experience, it helps to
   * Understanding software/system
   * How code is developed
   * Possibilities of errors.
   * when error it may be occured
   * which situation error it may occured
6. Try to produce High Quality Software
7. Try to satisfy User Requirements and Needs
8. Tester combined work with Test Manager and Project Manager it helps to
   * To Prepare test plans
   * To maintain organizational Testing Standards, Policies, goals, Procedures

9. Responsible for Low Defect Software

10. TO minimize the Costs for support

11. TO Deliver the Software as per the customer Needs, also Reliable, Usable

12. Inform the Errors/ Defects to Developers

13. Tester needs

- ↳ Communication Skills
- ↳ Team working Skills
- ↳ Decision making Skills
- ↳ Scripting knowledge/ Coding Skills
- ↳ working experience

Origins of Defects:

* It is a variation between actual software requirement specification (SRS) and final executed build (ie exe file)

Types of defects

1. Defects from product specifications

* The product developed varies in the product specifications (ie SRS/ BRS/ CRS)

2. Variance from customer/user expectations.

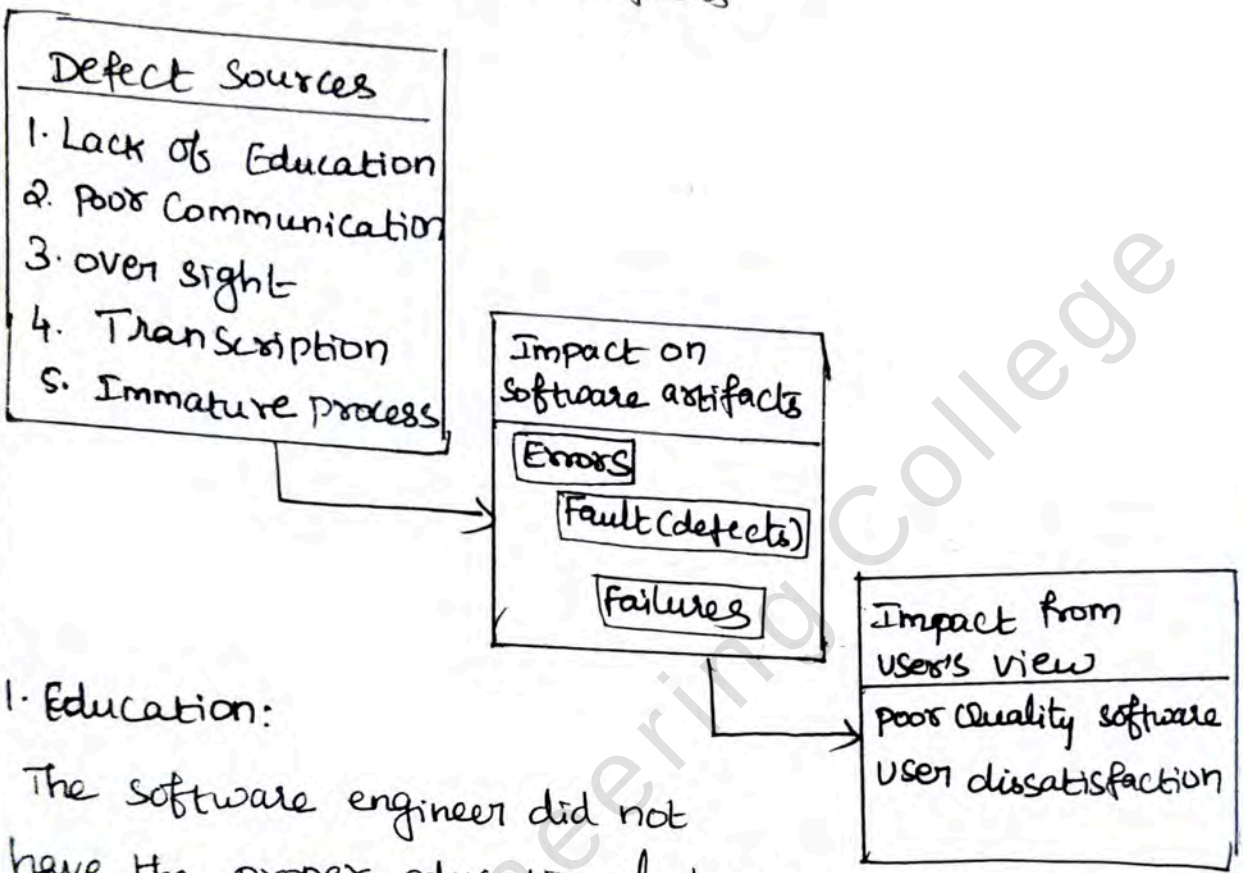
* Variance means difference

* This variance is something that the user wanted is not in the built product

* Defects have harmful affects on software user and software engineers work very hard to produce high-quality software with a low number of defects

Defects sources:

origins of defects



1. Education:

The software engineer did not have the proper educational background to prepare the software artifact

2. Communication:

* The software engineer must communicate with group members properly. Eg misunderstanding stupidly

3. oversight:

* The software engineer omitted to do something

Eg Initialization statement omitted

4. Transcription:

* The software engineer knows what to do, but makes a mistake in doing it

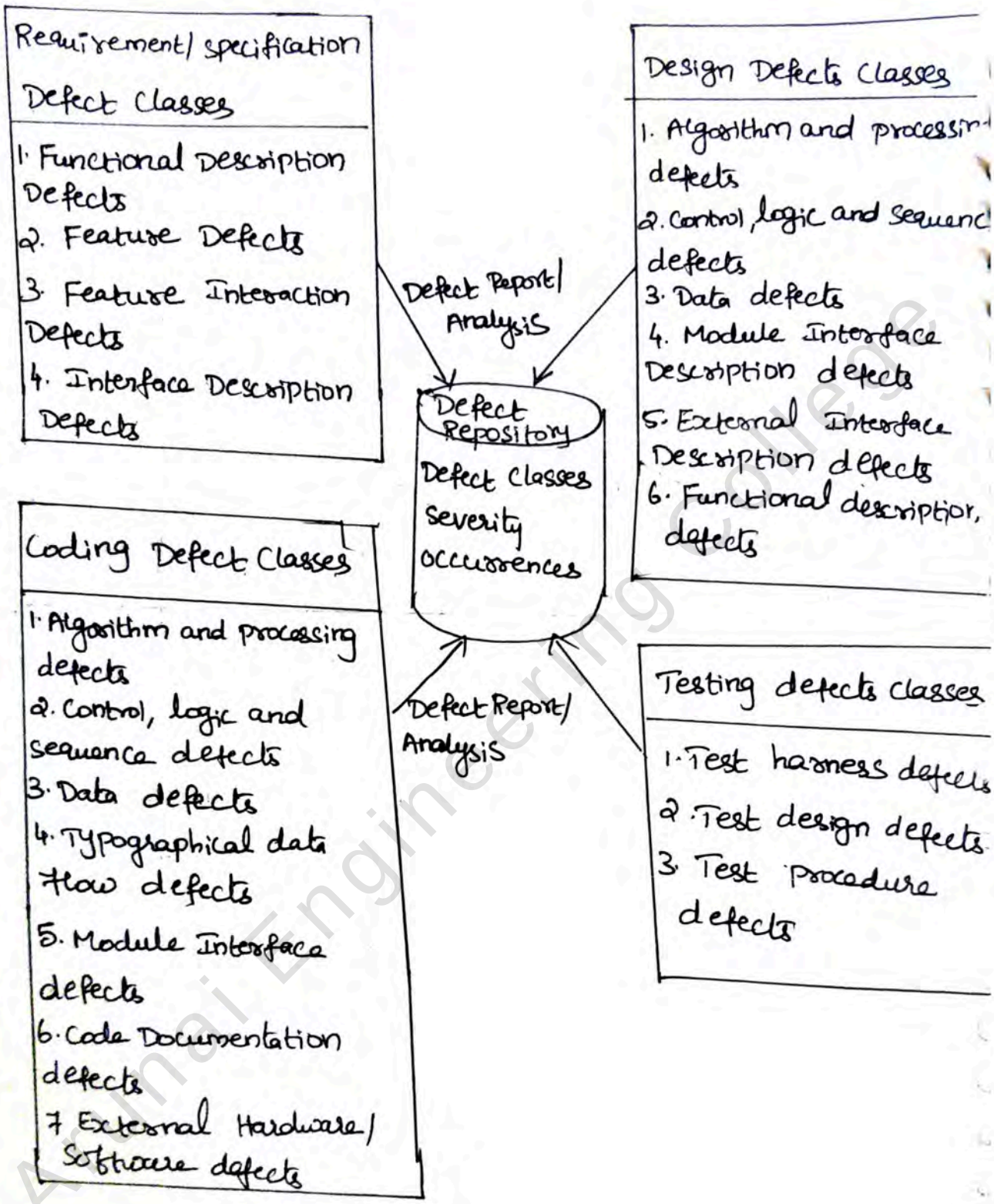
5. process:

* The process used by the software engineer misdirected his/her actions

Defect classes:

* Defects classified in many ways.
* A single classification scheme is necessary to adapt and apply for all projects
* Some defects fit into more than one classes or category.
* Developers, testers and SQA staff should try to be as consistent as possible when recording defect data
* The defect types and frequency of occurrence should be used to guide test planning and test design
* The defects are classified into 4 types, They are

1. Requirement and specification defects
2. Design Defects
3. Coding defects
4. Testing defects



Defect classes and the defect Repository

1. Requirement and Specification defects

* Defects injected in early phases is very difficult to remove in latter phase

* Requirement documents are written in natural language so it creates a chance for Ambiguous, contradictory, Unclear, redundant, Imprecise requirements

* Some of the Specification / requirments defects are

1. Functional Description Defects:

The overall description of what the system does, and how it should behave is

* Incorrect * Ambiguous * Incomplete

2. Feature Defects:

* Feature may be described as distinguish characteristics of a software component or system

* It describes Missing, incorrect, Incomplete, superfluous

3. Feature Interaction Defects:

* It describes, how the incorrect description interact in the feature.

* How the features interacts with another features

4. Interface Description Defects:

* It describes, how the target software is interfaced with

1. External Software 2. Hardware 4. Users.

2. Design Defects:

* It occur when System components, interaction between System components, interaction between the components and outside software/hardware and Interaction with Users.

* Defects may also occur in Algorithm design, Control, Logic representation, Data elements, Module interface descriptions and external hardware/Software/User Interface descriptions.

Design defects are

1. Algorithm and processing Defects:

* These occur when the processing steps in the algorithm as described by pseudo code are incorrect

2. Control, logic and Sequence Defects:

* Incorrectly developed pseudo code create control and logic defects

↳ Control defects - Poor logic flow in code

↳ Logic defects - Logic operator applied mistakenly

↳ Sequence defects - Conditions are not properly checked in the pseudocode

Eg Incorrect branching condition

3. Data Defects

* Due to poor data structure design, it create data defects

Eg Incorrect allocation of Memory, Lacking of field in a record

4. Module Interface Description Defects:

* Its derived from incorrect or inconsistent Parameter types, an incorrect number of parameter, Incorrect ordering of Parameter

5. Functional Description Defects:

* It included incorrect, missing and/or unclear Design element

Eg Poor Explanation of function

6. External Interface Description Defects.

* These defects are derived from incorrect design descriptions for the interface with

↳ COTS components ↳ Data base

↳ External software system ↳ Hardware devices

3. Coding Defects:

* when executing the code, if any errors occurred it is called "coding defects".

* It may be occurs on,

↳ Misunderstanding of programming languages and its construction

↳ Miscommunication with designes.

Coding Defects are

1. Algorithmic and processing Defects

* It occurs on Unchecked overflow/underflow conditions, Data Conversion, Missing Parenthesis, Precision loss, In-correct order of Parenthesis

2. Control, logic and Sequence Defects

* It created by,
  * Incorrect expression of case statements
  * Incorrect iteration of loops
  * Missing path and conditions

3. Typographical Defects

* These defects also called "Syntax error"

Eg Incorrect spelling of a variable name, that are usually detected by a compiler, self review & peer reviews

4. Initialization Defects:

* It occurs on, when initialization statements are omitted or in-correct

* It may occur because of

1) Misunderstanding or Lack of Communication between programmers
2) Misunderstanding of programming environment
3) Carelessness

5. Data flow defects

* Poor operational sequences create data flow defects

6. Data Defects:

* Due to poor data structure implementation, it create data defects

Eg Incorrect accessing of files

Module Interface Defects

* These defects are derived from
  - Inconsistent parameter types
  - Incorrect number of parameters
  - Improper Design
  - Incorrect sequence of calls

8. Code Documentation Defects:

* When the code documentation does not reflect what the program actually does or is incomplete or ambiguous this is called a code documentation defect

9. External Hardware, software Interfaces defects:

* These defects a arised from System calls, Database linking, Memory usage, Resources usage, Interrupts and Exception handling

4. Testing Defects:

*Test Plans, test cases, test harnesses and test Procedures can also contain defects.

*Testing defects are classified into two types

1. Test Harness Defects:

* It is also called the test harness or scaffolding code

* code is reusable code. Code is used again when the version of Software is released.

* Code must be correctly, Designed, Implemented Tested.

* Otherwise, defects occurs on unit and Integration levels. So, code should be maintain and support easily checking by the Software.

2. Test Case Design and Test Procedure Defects

* Test Case design arise some valid defects incorrect test cases, incomplete test cases, missing test cases, inappropriate Test cases and test Procedures

* These defects are best detected in test plan review.

Defect Examples: The Coin problem

Example: Coins to dollars Conversion Problem.

1. Specification for program Calculate-Coin-values

Requirements:

* This program or system finds total dollars and cents value for a set of coins

* User input may be pennies, nickles, dimes, quarters etc.

* The program outputs must be the total dollars and cent values of the coins to the user.

* Number of coins is an integer (input)

* Number of dollars is an integer (output)

* Number of cents is also integer (output)

Specification Defects:

1. Functional Description Defects:

* The functional description defects arise because the function description is ambiguous and incomplete.

* It does not state

i. Input & output ii) Number of coins iii) Number of cents and dollars iv) value is zero or great

v) Pre and post conditions

Eg: No. of coins >=0 (not to be negative)

Eg: No. of dollars >=0

vi) Accept invalid values or not

vii) pre and post conditions are to be solved by black box testing using

a) Boundary value Analysis
b) Negative value testing
c) NULL and database testing

2. Interface Description Defects:

* It is relate to

i) Poor education/ unexperience people
ii) Ambiguous nature
iii) Incomplete nature of specification

Design Description for program calculate-coin-values

```
Program Calc-Coin Value
        no-of-coins is integer
        total-coin-val- is integer
        no-of-cents is integer
        no-of-conts is integer
    coin-val is array of 6 integers
    coin-val initalized to: 1, 5, 10, 25, 25, 100 begin
        initialize total-coin-val to zero
```

```
initialize loop_count to one
while loop_count is less than 6 begin
  output "enter coins_count" read (no-of-coins)
total-coin-val = total-coin-val + no-of-coin *
        coin-val [loop_count]
  increment loop_count
end
  no-of-dollars = total-coin-val / 100
  no-of-cents = total-coin-val / 100 * no.of dollars
  output (no_of_dollars, no.of_cents)
end.
```

Design Defects:

1. Control, logic and sequencing Defects
   * The defect in this subclass arises from an incorrect "while" loop condition.
2. Algorithmic and processing defects:
   * These defects arise from
   1. Lack of error checks for incorrect and/or invalid inputs
   2. Lack of path. Ex. error inputs
   3. Rejection of error conditions checks like division by zero
3. Data Defects:
   * The wrong values to be entered
   * Coin-val read 1, 5, 10, 25, 50, 100

4. External Interface Description Defects:

* This defects arising from the absence of input messages or prompts that introduce the program to the user and the request input.

5) Control, logic and sequence Defects

* These include the loop variable increment step which is out of the scope of the loop.

* IL- Create logic defects

6. Algorithmic and processing Defects

* The division operator will create a problem.

7 Data flow defects

* The variable total-coin-val is not initialized its used before its defined

8 Data defects:

* Array coin-val is holding error when it is initialized. These errors are carried from designing to coding

9. External Hardware, Software Interface Defects:

* External function "Scanf" is correctly written

* "f" Symbol not included in "scanf" statement

10. code Documentation Defects:

* The Entire code is represented as a document which is incomplete and ambiguous

* The coding defects are solved by white box testing, logical, loop & branch testing, Control testing.

Developer and testers support for Development of defect Repository:

* A requirement for repository development should be a part of testing and/or debugging Policy statements.

* Forms and templates will need to be designed to collect the data.

Eg The test incident reports and defect fix reports, Test reports

* Defect and its relevant information are stored in defect repository

* Defect repository development is a essential part of testing and debugging.

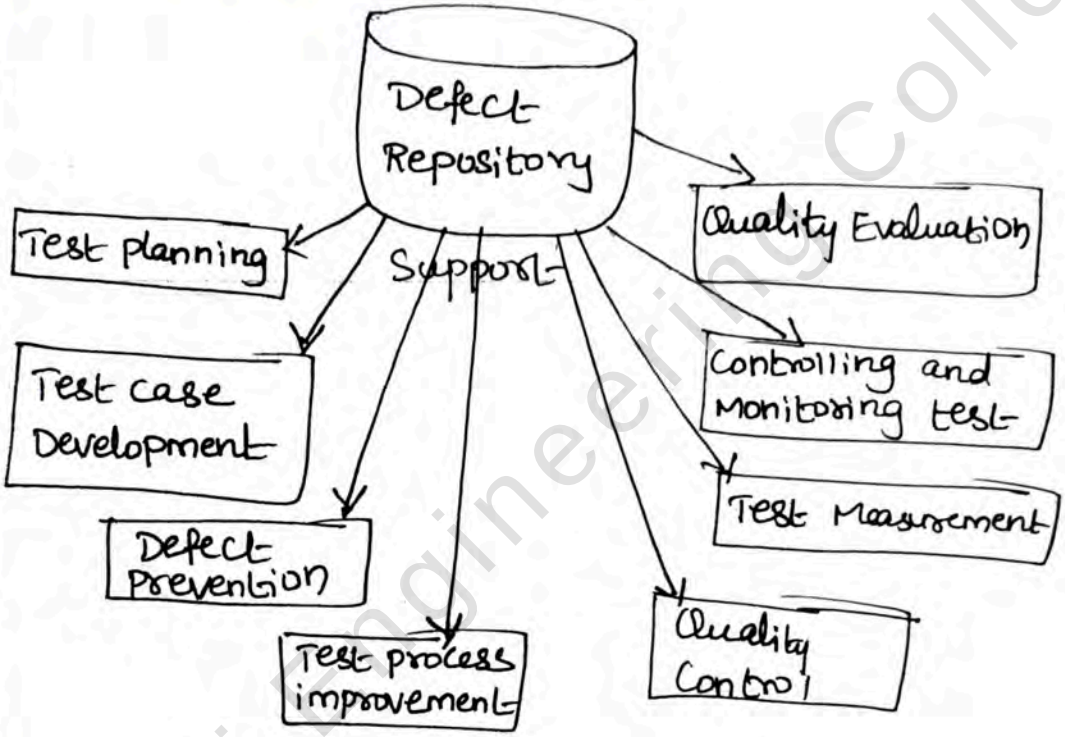
* Reports are to be recorded each defect and frequency of the occurrences for each defect type after testing.

* Every defects to be monitoring for each ongoing project

* When the tester/developer does changes in the process, the distribution of defects will also be changed.

The defect data is used for test planning. It helps.

1. To choose testing techniques
2. To design test cases (if required)
3. To allocate the number of resources (if needed)
4. To estimate testing schedule
5. To estimate testing cost



TMM Maturity goals

1. Controlling and monitoring test
2. Quality evaluation and control
3. Test measurement
4. Test process improvement
5. Defect prevention
6. Test planning
7. Test case development

IT8076-SOFTWARE TESTING

UNIT-2
TEST CASE DESIGN STRATEGIES

TEST CASE DESIGN

NTRODUCTION

- The Testing Maturity Model provides some answers to these questions. It can serve as a learning tool, or framework, to learn about testing. Support for this usage of the TMM lies in its structure. It introduces both the technical and managerial aspects of testing in a manner that allows for a natural evolution of the testing process, both on the personal and organizational levels.
- In this chapter we begin the study of testing concepts using theTMM as a learning framework. We begin the development of testing skills necessary to support achievement of the maturity goals at levels 2–3 of the Testing Maturity Model. TMM level 2 has three maturity goals, two of which are managerial in nature.
- Note that this goal is introduced at a low level of the TMM, indicating its importance as a basic building block upon which additional testing strengths can be built. In order to satisfy this maturity goal test specialists in an organization need to acquire technical knowledge basic to testing and apply it to organizational projects.

THE SMART TESTER

The smart tester is to understand the functionality, input/output domain and the environment for use of the code being tested. For certain types of testing the user must also understand in detail how the code is constructed.

Novice Tester

Novice testers, taking their responsibility seriously, might try to get test a module or component using all possible inputs and exercise all possible software structures. Using this approach, they reason, will enable them to detect all defects

Roles of a Smart Tester

- Reveal defects
- Can be used to evaluate software performance, usability & reliability.
- Understand the functionality, input/output domain and the environment for use of the code being tested

TEST CASE DESIGN STRATEGIES AND TECHNIQUES

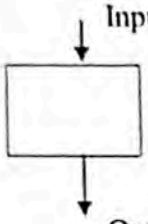
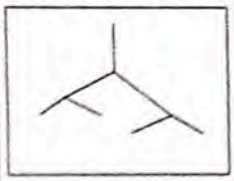
Test Strategies	Tester's View	Knowledge sources	Techniques / Methods
Black-box testing (not code-based) (sometimes called functional testing)		Requirements document Specifications User manual Models Domain knowledge Defect analysis data Intuition Experience	Equivalence class partitioning Boundary value analysis Cause effect graphing Error guessing Random testing State-transition testing Scenario-based testing
White-box testing (also called code-based or structural testing)		Program code Control flow graphs Data flow graphs Cyclomatic complexity High-level design Detailed design	Control flow testing/coverage: - Statement coverage - Branch (or decision) coverage - Condition coverage - Branch and condition coverage - Modified condition/ decision coverage - Multiple condition coverage - Independent path coverage - Path coverage Data flow testing/ coverage Class testing/coverage Mutation testing

Figure: Two basic Testing Strategies

USING THE BLACK BOX APPROACH TO TEST CASE DESIGN

- Given the black box test strategy where we are considering only inputs and outputs as a basis for designing test cases. How do we choose a suitable set of inputs from the set of all possible valid and invalid inputs?
- Keep in mind that infinite time and resources are not available to exhaustively test all possible inputs. This is prohibitively expensive even if the target software is a simple software unit. The goal for the smart tester is to effectively use the resources available by

developing a set of test cases that gives the maximum yield of defects for the time and effort spent.

- To help achieve this goal using the black box approach we can select from **several methods.** Very often combinations of the methods are used to detect different types of defects. Some methods have greater practicality than others.
    - **Random Testing**
    - **Equivalence Class Partitioning**
    - **Boundary Value Analysis**
    - **Other black box test design approaches**
        - **Cause-and-Effect Graphing**
        - **State Transition Testing**
        - **Error Guessing**

BLACK BOX TEST CASE DESIGN TECHNIQUES

RANDOM TESTING

Each software module or system has an input domain from which test input data is selected. If a tester randomly selects inputs from the domain, this is called random testing. For example, if the valid input domain for a module is all positive integers between 1 and 100, the tester using this approach would randomly, or unsystematically, select values from within that domain; for example, the values 55, 24, 3 might be chosen.

Issues in Random Testing:

- Are the three values adequate to show that the module meets its specification when the tests are run?
- Should additional or fewer values be used to make the most effective use of resources?
- Are there any input values, other than those selected, more likely to reveal defects? For example, should positive integers at the beginning or end of the domain be specifically selected as inputs?
- Should any values outside the valid domain be used as test inputs? For example, should test data include floating point values, negative values, or integer values greater than 100?

More structured approaches to black box test design address these issues.

Use of random test inputs may save some of the time and effort that more thoughtful test input selection methods require. However, the reader should keep in mind that according to many testing experts, selecting test inputs randomly has very little chance of producing an effective set of test data.

Illustrate with an Example the following black box testing techniques

i) Equivalence class partitioning

ii) Boundary value Analysis

i) Equivalence Class Partitioning : (ECP)

* Equivalence class partitioning is a software testing technique that evolves identifying a small set of representative input values that produce as many different output conditions as possible.

* The set of input values that generate one single expected output is called a partition.

* when the behavior of the software is the same for a set of values then the set is termed as an equivalence class (or) equivalence partition

* one sample from the partition is enough for testing as the result of picking up some more values from the set will be the same and will not yield any additional defects.

* An the values produce equal and same output they are termed as equivalence partition.

Testing by this techniques invalves.

1) Identifying all Partitions for the Complete set of input, output values for a Product.

2) Picking up one member value from each partition for testing to maximize complete coverage.

Advantages:

1) It gain good Coverage with a small number of test cases

2) Redundancy of tests is minimized by not repeating the same tests for multiple values in the same partition

The equivalence partition table consists of

1. Equivalence partition definition
2. Type of input
3. Representative data for that Partition
4. Expected Results

Each row is taken as a single test case and is executed.

The step to prepare an Ecp table are as follows.

1. Choose Criteria for doing the Ecp (range, list of value etc)
2. Identify the valid Ecp based on the above Criteria (number of ranges allowed values)
3. Select a sample data from the Partition
4. write expected result based on the requirement given.
5. Identify special values if any, and include them in the table.
6. Check to have expected results for all the cases Prepared.
7. If expected result is not clear for any particular test case, mark appropriately and escalate for corrective actions.

Example:

Life Insurance Premium rates:

* A life Insurance Company has base premium of RS.50 for all ages. Based on age group, an additional monthly premium has to be paid that is listed

Age group	Additional Premium
Under 35	Rs. 2
35 - 39	Rs. 3
60 +	Rs. 6

Equivalence classes for the life Insurance Premium

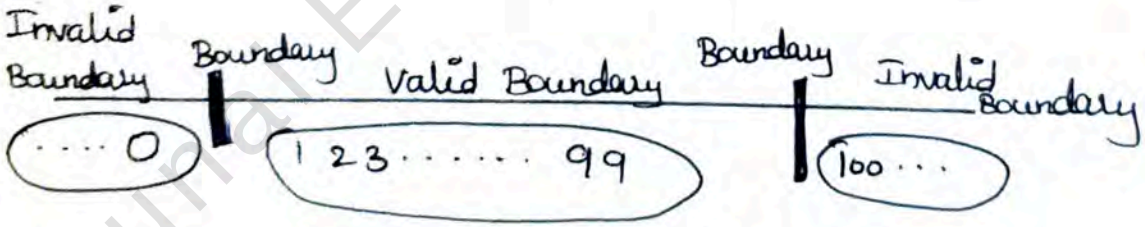
S.NO	Equivalence Partitions	Type of Input	Test data	Expected results
1.	Age below 35	valid	26, 12	Monthly Premium = Rs 50 + 2 = Rs. 52
2	Age 35-39	valid	37	Monthly Premium = Rs. 50 + 3 = Rs. 53
3.	Age above 60	valid	65, 90	Monthly Premium = Rs. 50 + 6 = Rs. 56
4	Negative age	Invalid	-23	warning Message - Invalid input
5	Age as 0	Invalid	0	warning Message - Invalid input

ii) Boundary value Analysis (BVA)

* Boundary value Analysis (BVA) is used to find the errors at boundaries of input domain rather than finding those errors in the center of input
* Test both valid boundaries and invalid boundaries
* It is based on boundaries between partitions
* It is a part of functionality testing but test engineers are giving special treatment to input domains of objects
* It is also called "INPUT DOMAIN TESTING".

Example:

* Consider a printer that has an input option of the number of copies to be made from 1 to 99.



In the Example valid / Invalid boundary values are

Min = 1 Max = 99

So the solution is

min	→1	Pass	Max+1	→100	fail
Min-1	→0	Fail	Max-1	→98	pass
Min+1	→2	Pass			
Max	→99	Pass			

* Boundary value analysis is chosen to detect the error.
* Most of the error/defect occurs on boundary, not in the center of the input domain
* It allows the selection of set of test cases and those test cases exercises the boundary values.
* It designs test cases at the edge of input domain.
* It also concentrates on the output domain

Guidelines for BVA:

1. BVA is to select input variables values for minimum, above minimum, normal value, below maximum, maximum
2. Failures occurs rarely as the result of the simultaneous occurrence of 2 or more faults
3. Variables are program dependent, language dependent, bounded discrete, unbounded discrete & logical variables.
4. Programs written in non-strongly typed language are more appropriate candidates for BVA
5. Boundary inequalities of 'n' input variables define a n-dimensional input space

Black box testing

i) Requirements based testing
ii) Positive and Negative testing
iii) State based testing
iv) User documentation and compatibility.

i) Requirements Based Testing:

* Requirements-based testing is a testing approach in which test cases, conditions and data are derived from requirements.

* It includes functional tests and also non functional attributes such as performance, reliability or usability.

It is a 12 step process

1. Validate requirements against objectives
2. Apply scenarios against requirements
3. Perform initial ambiguity review
4. Perform domain Expert review
5. Create Cause-effect graph
6. Logical Consistency check by RBT
7. Review of test cases by specification writers
8. Review of test cases by Users

9. Review of test cases by developers
10. Walk test cases through design
11. Walk test cases through code
12. Execute test cases against code

Requirements types:

1. Explicit requirements - It is stated and documented as part of the requirements specification

2. Implied or Implicit requirements - This is not documented but assumed to be incorporated in the system.

* The precondition for requirements testing is a detailed review of the requirements specification. It checks

↳ Consistency, Correctness, Completeness, Testability Clarity of requirements etc.

* All explicit requirements and implied requirements are collected and documented as "Test Requirement Specification" (TRs)

* Eg sample requirements specification for lock and key system

Sample requirements specification for lock & key system

S.NO	Requirements Identifier	Description	Priority (High, Medium, low)
1.	BR-01	Inserting the key numbered 123-456 and turning it clockwise should facilitate locking	H
2.	BR-02	Inserting the key numbered 123-456 and turning it anticlockwise should facilitate unlocking	H
3.	BR-04	No other object can be used to lock	M
4	BR-05	No other object can be used to unlock	M
5	BR-07	The lock & key must be made of metal & must weigh approximately 150 grams	L
6	BR-08	Lock and unlock directions should be changeable for usability of left-handers	L

* Requirements are tracked by Requirements Traceability Matrix (RTM). An RTM traces all the requirements from their genesis through design, development and testing.

* The "test conditions" Column lists the different ways of testing the requirements. These conditions can be grouped together to form a single test case.

* The "test case IDs" Column can be used to complete the mapping between test cases & the requirements

* RTM helps in identifying the relationship between the requirements and test cases.

1) one-to-one: For each requirement there is one test case
2) one to many: For each requirements there are many test case
3) Many to one: A set of requirements can be tested by one test cases
4) Many to many: Many requirements can be tested by many test cases
5) one to none: The requirements can have no test case

* The "phase of testing" column is used multiple Phases of testing - Unit, Component, integration and System testing

Sample requirements Traceability Matrix

SNO	Requirement Identifier	Description	Priority (H, M, L)	Test Conditions	Test Case IDS	Phase Of Testing
1.	BR-01	Inserting the key numbered 123-456 and turning it clockwise should facilitate locking	H	Use key 123-456	Lock 001	Unit, Component
2	BR-04	No other object can be used to lock	M	Use key 789-001 use hairpin use screw drivers	Lock -005, Lock -006, Lock -007	Integration
3	BR-07	Lock and key must be made of metal and must weigh approximately 150 grams	L	Use weighing	Lock -012	System

Role of RTM:

1) The RTM enables testers to prioritize the test cases execution to find the defects

2) Test conditions can be grouped to create test cases or can be represented as unique test cases.

3) TO find the adequate test cases/ high priority requirements

Metrics that can be collected or inferred from RTM Matrix are

1) Requirements addressed prioritywise

2) Number of test cases requirement wise

3) Total number of test cases prepared.

After the test cases are executed, the test results can be used to collect metrics such as

- Total number of test cases passed
- Total number of test cases failed
- Total number of defects in requirements
- Number of requirements completed
- Number of requirements pending.

ii) Positive and Negative Testing:

* The purpose of positive Testing is to prove that the product works as per specification and expectations.

* A product delivering an error when it is expected to given an error is also a part of positive testing

Example of positive Test cases

Requirements NO	Input 1	Input 2	Current State	Expected outcome
BR-01	key 123-456	Turn clockwise	Unlocked	Locked
BR-01	key 123-456	Turn clockwise	Locked	No change
BR-02	key 123-456	Turn anticlockwise	unlocked	No change
BR-03	key 123-456	Turn anticlockwise	Locked	Unlock
BR-04	Hairpin	Turn clockwise	Locked	No change

* Negative testing is done to show that the Product does not fail when an unexpected input is given.

* The Purpose of negative testing is to try and break the system. Negative testing covers scenarios for which the product is not designed and coded.

* Important for the tester to know the negative Situations that may occur at the end-user level so that the application can be tested and made fool proof

Negative Test cases

S.No	Input1	Input2	Current State	Expected output
1.	Some other lock's key	Turn Clockwise	Lock	Lock
2	some other lock's key	Turn clockwise	Unlock	Unlock
3.	Thin Piece of wire	Turn anticlockwise	Unlock	Unlock
4	Hit with a stone		Lock	Lock

* The difference between positive testing and negative testing is in the Coverage.

* For positive testing if all documented requirements and test conditions are covered, then Coverage can be considered to be 100 Percent.

* For negative testing requires a high degree of Creativity among the testers to Cover as many "unknowns" as possible to avoid failure at a customer site

iii) State based Testing (or) Graph Based Testing:

* It is black box testing techniques, in which output's are triggered by changes to the input conditions or changes to 'State' of the system.

* In other words, tests are designed to execute valid and invalid state transitions.

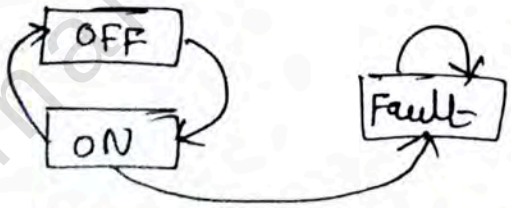
* State Based Testing Uses:

↳ when we have sequence of events that occur and associated conditions that apply to those events

↳ when the proper handling of a particular event depends on the events and conditions that have occurred in the past.

↳ It is used for real time systems with various states and transitions involved.

Eg: A system transition is represented



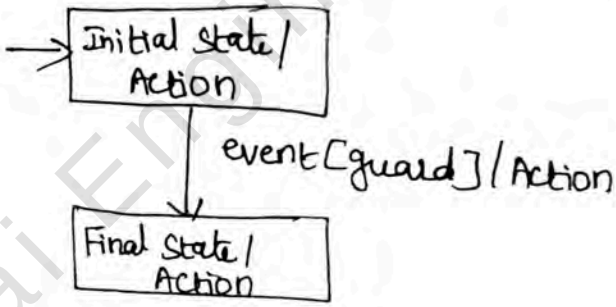
State & Transition Scenario

Tests	Test 1	Test 2	Test 3
Start state	off	on	on
Input	Switch on	Switch off	switch off
output	Light on	Light off	Fault
Finish state	ON	OFF	ON

State machine:

* Implementation independent Specification (model) of the dynamic behavior of the system.

1. State - Abstract situation in the life cycle of a system entity
2. Event - A particular input - (A message or method call)
3. Action - The result, output or operation that follows an event
4. Transition - An allowable two state sequence (ie) a change of state("firing") caused by event
5. Guard - Predicate Expression associated with an event stating a Boolean restriction for a transition to fire.



State Machine

State Transition Diagram:

* It is graphical representation of a state machine
* It is represented by a state graph having a finite number of states & a finite number of transitions between states
* It is also called "Graph based Testing"
* It is useful for both Procedural & object oriented implementation

* State or graph based testing is very useful language processor (Eg compiler) testing, workflow modeling, Data flow modelity etc.

Example 1:

Application in which a number is validated

Rules:

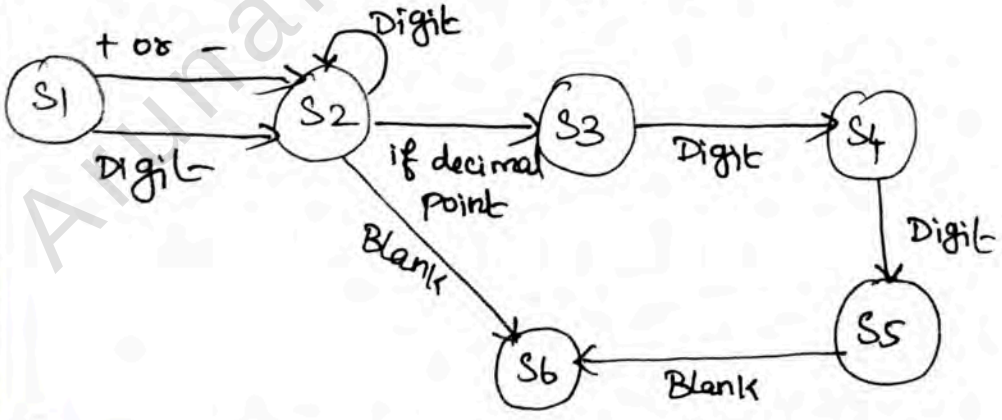
1) A number can start with an optional sign

2) The optional sign can be followed by any number of digits

3) The digits can be optionally followed by a decimal point, represented by period.

4) If there is a decimal point that there should be two digits after the decimal

5) Any number - whether or not, it has a decimal point should be terminated by the blank.



State Transition Diagram

State Transition Table

Input	Current State	Next State
Digit	1	2
+	1	2
-	1	2
Digit	2	2
Blank	2	6
Decimal Point	2	3
Digit	3	4
Digit	4	5
Blank	5	6

* The State transition table can be used to derive test cases to test valid and invalid numbers.

* valid test cases can be generated by

1) Start from the start state (State 1)
2) Choose a path that leads to the next State
3) If invalid input is encountered in a given state, generate an error condition test case.
4) Repeat the process till you reach the final State.

iv) User documentation and Compatibility:

* User documentation covers all the manuals, user guides, installation guides, set up guides, Read me files, software release notes, online help

* User documentation testing should have two objectives

1) TO check if what is stated in the document is available in the product

2) TO check if what is there in the product is explained correctly in the document

* when a product is upgraded, the corresponding product documentation should also get updated as necessary to reflect any changes that may affect a user.

* TO concentrates on the specification given in the document is Perfectly matching with product behavior or not.

Benefits of User documentation Testing

1) Removes Uncertainties
2) offer good training materials to freshers.
3) Good marketing strategy
4) Better customer satisfication
5) Easily ensure the problem during review.

Compatibility Testing:

* It is also called "portability Testing".

* It is a non functional testing conducted on the application to evaluate the application's compatibility with in different environments.

* During this test, test engineers validates that whether our application build run on customer expected platform or not?

Two types of Compatibility testing:

1. Forward compatibility
2. Backward compatibility

Compatibility Testing technique:

1) Horizontal combination:

* In this technique, all the value of parameters to execute test cases are combined into row in the compatibility matrix

* Machines are set to every row and the set of Product characteristics are tested.

2) Intelligent Sampling:

* In this technique, Combinations of infrastructure parameters, set of features are combines and tested.

* Intelligent Sample are generated based on the data collected on the set of dependencies of the product with Parameters.

* If the product result are less dependent on a set of Parameters then they are taken out from the collection of intelligent samples.

*Backward Compatibility Testing:

* The testing that ensures the current Version of the product continues to work with the older versions of the same product is called backward compatibility testing.

* The Product parameters required for the backward compatibility matrix and are tested.

*Forward compatibility testing:

* There are some provisions for the product to work with later versions of the product and other infrastructure components keeping future requirements in mind.

DOMAIN TESTING

- White box testing required looking at the program code. Black box testing perform testing without looking at the program code but looking at the specifications. Domain testing can be considered as the next level of testing in which we do not look even at the specifications of a software product but are testing the product, purely based on domain knowledge and expertise in the domain of application.
- Domain testing is the ability to design and execute test cases that relate to the people who will buy and use the software. It helps in understanding the problems they are trying to solve and the ways in which they are using the software to solve them.
- Domain testing involves testing the product, not by going through the logic built into the product. The business flow determines the steps, not the software under test. This is also called "business vertical testing". Test cases are written based on what the users of the software do on a typical day.

Example:

Cash withdrawal functionality in an ATM. The user performs the following actions.

Step 1 : Goto the ATM

Step 2 : Put ATM card inside

Step 3 : Enter correct PIN

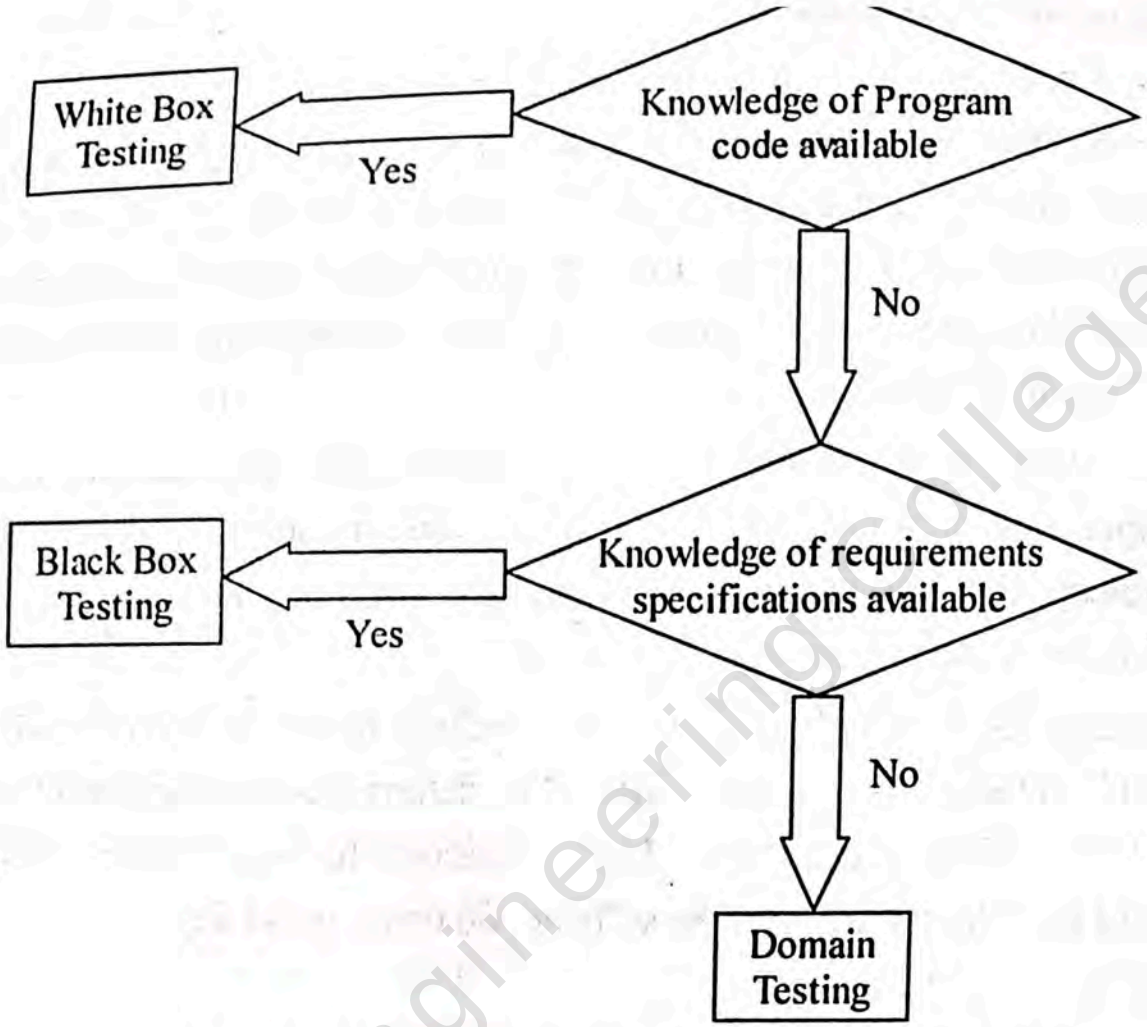
Step 4 : Choose cash withdrawal

Step 5 : Enter amount

Step 6 : Take the cash

Step 7 : Exit and retrieve the card

- Domain testing requires domain knowledge rather than the knowledge of what the software specification contains or how the software is written. Thus domain testing can be considered as an extension of black box testing.
- The test engineers performing the domain testing are selected in such a way that they have in-depth knowledge of the business domain. This reduces the effort and time required for training the testers in domain testing and also increases the effectiveness of domain testing.



Context of white box, black box and domain testing

- In the above example, a domain tester is not concerned about testing everything in the design; rather, he or she is interested in testing everything in the business flow. However, when you are testing as an end user in domain testing, all you are concerned with is whether you got the right amount or not.
- When the test case is written for domain testing, you would find the intermediate steps missing. Just because those steps are missing does not mean they are not important. These "missing steps" (such as checking the denominations) are expected to be working before the start of domain testing.
- Generally, domain testing is done after all components are integrated and after the product has been tested using other black box.

BLACK BOX TEST CASE DESIGN TECHNIQUES

RANDOM TESTING

Each software module or system has an input domain from which test input data is selected. If a tester randomly selects inputs from the domain, this is called random testing. For example, if the valid input domain for a module is all positive integers between 1 and 100, the tester using this approach would randomly, or unsystematically, select values from within that domain; for example, the values 55, 24, 3 might be chosen.

Issues in Random Testing:

- Are the three values adequate to show that the module meets its specification when the tests are run?
- Should additional or fewer values be used to make the most effective use of resources?
- Are there any input values, other than those selected, more likely to reveal defects? For example, should positive integers at the beginning or end of the domain be specifically selected as inputs?
- Should any values outside the valid domain be used as test inputs? For example, should test data include floating point values, negative values, or integer values greater than 100?

More structured approaches to black box test design address these issues.

Use of random test inputs may save some of the time and effort that more thoughtful test input selection methods require. However, the reader should keep in mind that according to many testing experts, selecting test inputs randomly has very little chance of producing an effective set of test data.

White box testing:

* White box testing is a way of testing the external functionality of the code by Examining and testing the program code that realizes the external functionality

* This is also known as Clear box, or glass box or open box testing.

* White box testing takes into account the program code, code structure and internal design flow.

* A number of defects come about because of incorrect translation of requirements and design into program code

* Some other defects are created by programming errors and programming language unconventional behaviour.

Testing Techniques:

1. Statement Coverage - This technique is aim to exercising all programming Statement with minimal test.

2. Path Coverage - Every statement in program is correctly partipate in the program or not.

3. Program technique Coverage - It checks CPU cycles during execution, less number of memory cycles etc

4. Condition Coverage

5. Loop coverage

6. Function Coverage.

* It is a coding level testing strategy.

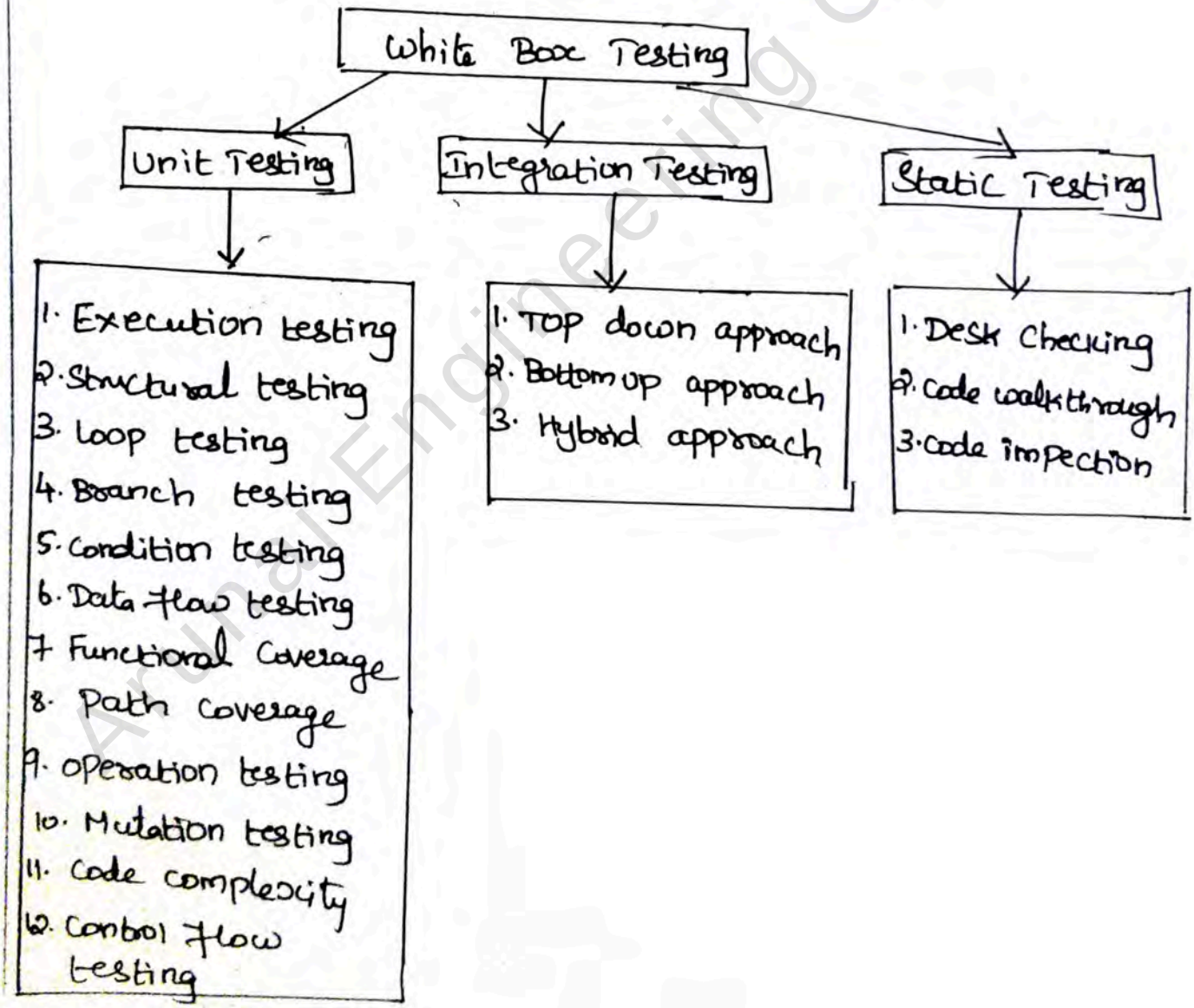
* It ensures all the statements & conditions have been executed at least once

* It verify that the software design is valid and also whether it was build according to the specified design or not.

* Software developers Verifies interms of internal logic with the help of test case design.

* The tester's aim is to find it all the logical & data elements in the software unit are functionating properly or not
* It helps to test small components/ modules
* This strategy is implemented for design & code base control

White Box Testing Technique



Test Adequacy Criteria:

Testers need a framework

* For deciding which structural elements to select as the focus of testing.
* For choosing the appropriate test data.
* Test data adequacy criterion is a stopping rule.

Application scope of adequacy criteria:

1) Helping testers to select properties of a program to focus on during test.
2) Helping testers to select a test data set for a program based on the selection properties
3) Indicating to testers whether or not testing can be stopped for that program.

Program-based adequacy criterion:

* If the test data adequacy criterion deals structural properties then it is known as "program based adequacy criterion".
* It is used in white box testing. They use either logic & control structures, Data flow program test or faults

Specification-based test data adequacy criteria:

* It focus on program specifications

* Adequacy criteria are usually expressed as statements that depict the property or feature of interest, and the conditions under which testing can be stopped.

Coverage Analysis:

* Coverage analysis is used to set testing goals and to develop and evaluate test data.

* when a coverage related testing goal is expressed as a percent it is often called the "degree of coverage".

* The planned degree of coverage is given in the test plan.

* when the test adequacy is fulfilled then planned degree of coverage 100 %.

* Some time degree of coverage not reach 100%. Due to

1) Nature of Unit => * statements not be reachable,
* Does not give safety, Unnecessary statement

2) Lack of resources => * Not enough time to set the Complete Coverage * Lack of tester * Lack of proper tools

3) Project related issues => Timing, Scheduling, Marketing constraints etc.

Static Testing VS structural Testing

Static Testing, a software testing technique in which the software is tested without executing the code. It has two parts as listed below:

- Review - Typically used to find and eliminate errors or ambiguities in documents such as requirements, design, test cases, etc.
- Static analysis - The code written by developers are analysed (usually by tools) for structural defects that may lead to defects.

Types of Reviews:

The types of reviews can be given by a simple diagram:

Static Analysis - By Tools:

Following are the types of defects found by the tools during static analysis:

- A variable with an undefined value
- Inconsistent interface between modules and components
- Variables that are declared but never used
- Unreachable code (or) Dead Code
- Programming standards violations
- Security vulnerabilities
- Syntax violations

Structural Testing

Structural testing, also known as glass box testing or white box testing is an approach where the tests are derived from the knowledge of the software's structure or internal implementation.
The other names of structural testing includes clear box testing, open box testing, logic driven testing or path driven testing.
Structural Testing Techniques:

- **Statement Coverage -** This technique is aimed at exercising all programming statements with minimal tests.
- **Branch Coverage -** This technique is running a series of tests to ensure that all branches are tested at least once.
- **Path Coverage -** This technique corresponds to testing all possible paths which means that each statement and branch are covered.

Calculating Structural Testing Effectiveness:

Statement Testing = (Number of Statements Exercised / Total Number of Statements) x 100 %
Branch Testing = (Number of decisions outcomes tested / Total Number of decision Outcomes) x 100 %
Path Coverage = (Number paths exercised / Total Number of paths in the program) x 100 %

Advantages of Structural Testing:

- Forces test developer to reason carefully about implementation
- Reveals errors in "hidden" code
- Spots the Dead Code or other issues with respect to best programming practices.

Disadvantages of Structural Box Testing:

- Expensive as one has to spend both time and money to perform white box testing.
- Every possibility that few lines of code is missed accidentally.
- In depth knowledge about the programming language is necessary to perform white box testing.

Code Coverage Testing:

* Logic based (white box based) test design and use of test data adequacy (Criteria Coverage) concepts provide 2 major Payoffs for the testers

1. Quantative coverage goal propose
2. commercial tool support is readily available to facilitate the tester's work.

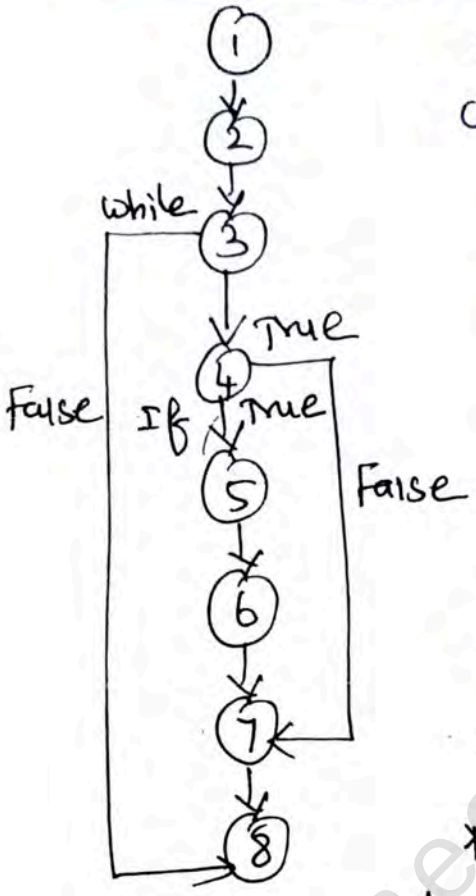
* when the goal is to satisfy the statement adequacy criterion, then a set of test cases can be developed.

* If the module gets executed, all of the statement in the module are executed at least once.

Code Sample with branch & loop

```
1   Sum (a, num, sum)
2    sum =0
3    int i=1
4    while (i<=num)
5       if (a[i]>0)
6           sum= sum+a[i]
7       end if
8          i= i+1
9       end while
10    end sum
```

Control flow graph



* A tester to prepare test cases which checks the nodes from 1 to 8 in the flow graph.

* when test reaches this goal, the statement adequate Criterion is satisfied by the test data and also other logic structures associated with the adequate criterion.

* Test cases should be designed to get 100% Coverage.

* Every decision element in the code executes with all possible outcomes atleast once.

* Complete decision coverage goal is the Stronger Coverage goal than Statement coverage

* If the statement coverage is so weak that is not considered to be very useful for revealing defects

* Input values must confirm the execution of true/false possibilities for the decision in line 3 and 4

Possible Test cases

Decision / Branch	Value of Variable	Value of Predicate	Test case: Value of a, num
			a = 1, -45, 3 num = 3
while	1 4	True False	
if	1 2	True False	

The test should fulfill both

* Branch adequacy Criterion
* Statement adequacy Criterion

Consider a statement

If (a<min and b>max and (not int c))

This statement has 3 conditions are predicate

1. a<min 2. b>max 3. not in c

Decision Coverage:

It must checks all the possible condition atleast once for all the branch/loop predicates.

Condition Coverage:

* It must needs atleast one execution of all possible conditions and combinations of decision.

* Coverage Criterion can be arranged in a hierarchy of strengths from weakest to strongest

1. Statement
2. Decision
3. Decision/ Condition

Example:

```
If (age<65 and married==true)
   do x
   do y...
else
   do z
```

Condition 1: Age less than 65

Condition 2: Married is true

Test cases for simple decision Coverage

Value for age	Value for married	Decision outcome (compound predicate)	Test cases ID
30	True	True	1
75	True	False	2

Test cases for condition Coverage

Value for age	Value for married	Condition1 outcome	Condition2 outcome	Test cases ID
75	True	False	True	2
30	False	True	False	3

Test cases for Decision condition Coverage

Value for age	Value for married	condition 1 outcome	condition 2 outcome	Decision outcome (compound predicate)	Test cases ID
30	True	True	True	True	1
75	True	False	True	False	2
30	False	True	False	False	3

* The large the number of test cases that must be developed to insure complete coverage.

* "Multiple condition coverage" or "Multiple decision conditions", the complexity of test case design increases with the strength of the coverage criterion.

* The test decide the criterion based on the
  - ↳ code
  - ↳ reliability requirement
  - ↳ Available resources.

Control Flow Graphs:

* "The logic elements most commonly considered for coverage are based on the flow of control in a unit of code".

Eg: 1) program statements
2) Decisions / branches
3) conditions
4) combinations of decisions and conditions
5) paths

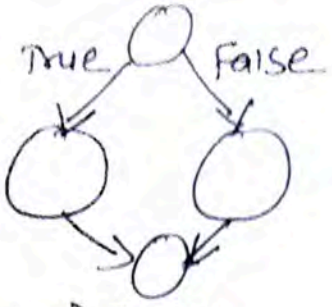
* These logical elements are rooted in the concept of a program prime.

* A program prime is an automatic programming unit.

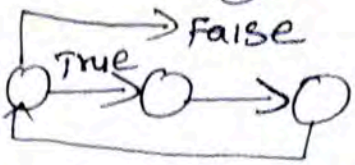
* All structural programs can be built from three basic primes

1) Sequence

2) Decision / condition



3) Iteration



* Using the concept of a Prime and the ability to use combinations of primes to develop structural code, a flow diagram for the software unit under test can be developed.

* The flow graph can be used by the testers to evaluate the code with respect to its testability as well as to develop white box test cases.

* A control flow graph describes in which the different instructions of a program gets executed.

* Control flow graph also says how the control flow through the program.

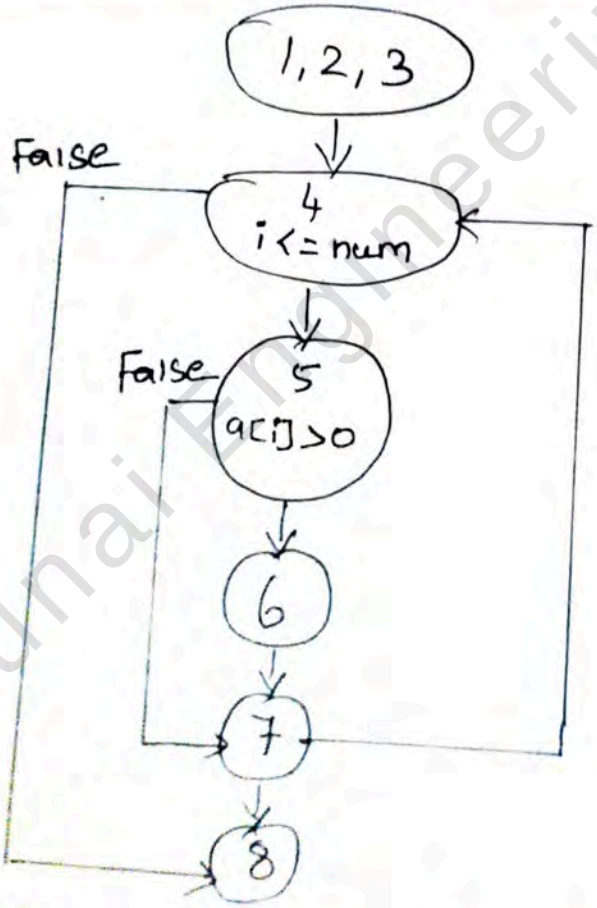
* Code Sample with branch and loop

```
POS_SUM (a, num, sum)
  Sum = 0
  int i=1
  while(i<=num)
      if(a[i]>0)
```

```
        Sum = Sum + a[i]
    endif
        i = i + 1
    end while
end POS_SUM
```

* In the flow graph the nodes represent sequential statements, as well as decision and looping predicates.

A control flow graph representation for the code



* Sequential statements are combined as a block
* Edges in the graph represent transfer of control

* An edge from one node to another node exists if the execution of the statement representing the first code can result in transfer of control to the other code.

* Each Circle is said to be flow graph node

* The direction of the transfer depends on the outcome of the condition in the predicate (true or false)

* Arrow on the flow graph represent edges or links

* Commercial tools are used to generate control flow graphs from code and in some cases from pseudo code.

* It support to generate control flow graphs for complex code.

* It helps to design white box test cases.

Cyclomatic Complexity

* The cyclomatic complexity is a software attribute and it is very useful to a tester.

* The complexity value is usually calculated from the control flow graph (G_1) by the formula

$$\boxed{V(G_1) = E - N + 2} \quad \text{or} \quad V(G_1) = E - N + P$$

where

E - Number of edges in the control flow graph
N - Number of nodes in the flow graph
P - Number of nodes that have exit points

The cyclomatic complexity value is useful to the tester in several ways.

1) It provide an approximation of the number of test cases needed for branch coverage in a module of structural code.

2) The tester can use the value of $V(G_1)$ along with past project data to approximate the testing time and resources required to test a software module.

3) Cyclomatic complexity value & control flow graph introduces another tool, it is called "path".

Covering Code Logic

Logic-based white box-based test design and use of test data adequacy/ coverage concepts provide two major payoffs for the tester:

(i) quantitative coverage goals can be proposed, and

(ii) commercial tool support is readily available to facilitate the tester‘s work

Testers can use these concepts and tools to decide on the target logic elements (properties or features of the code) and the degree of coverage that makes sense in terms of the type of software, its mission or safety criticalness, and time and resources available. For ex ample, if the tester selects the logic element program statements, this indicates that she will want to design tests that focus on the execution of program statements. If the goal is to satisfy the statement adequacy/ coverage criterion, then the tester should develop a set of test cases so that when the module is executed, all (100%) of the statements in the module are executed at least once. In terms of a flow graph model of the code, satisfying this criterion requires that all the nodes in the graph are exercised at least once by the test cases.

/* pos_sum nds the sum of all positive numbers (greater than zero) stored in an integer

array a. Input parameters are num_of_entries, an integer, and a, an array of integers with num_of_entries elements. The output parameter is the integer sume */

1. pos_sum(a, num_of_entries, sum)

2. sum 0

3. inti 1

4. while (i < num_of_entries)

5. if a[i] > 0

6. sum sum a[i]

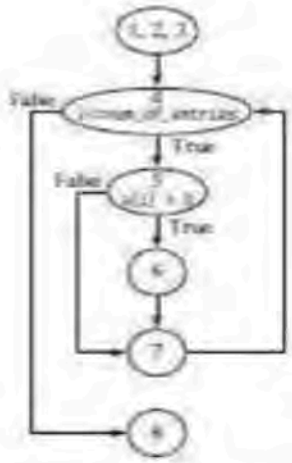
endif

7. i = i + 1

end while

8. end pos_sum

Code sample with branch and loop.



A control flow graph representation for the code

we have simple predicates or conditions in the branch and loop instructions. However, decision statements may contain multiple conditions, for example, the statement

If (x MIN and y MAX and (not INT Z))

has three conditions in its predicate: (i) x MIN, (ii) y MAX, and (iii) not INT Z.

Decision coverage only requires that we exercise at least once all the possible outcomes for the branch or loop predicates as a *whole, not for each individual condition contained in a compound predi*cate. There are other coverage criteria requiring at least one execution of the all possible conditions and combinations of decisions/conditions. The names of the criteria reflect the extent of condition/decision coverage.

Below is a simple example showing the test cases for a decision statement with a compound predicate.

Decision or branch	Value of variable i	Value of predicate	Test case: Value of a, num_of_entries
			a=1, −45,3 num_of_entries = 3
while	1	True	
	4	False	
if	1	True	
	2	False	

A test case for the code that stratifies the decision coverage criterion.

if(age <65 and married true) do X

do Y .....

else

do Z

Condition 1: Age less than 65
Condition 2: Married is true

Test cases for simple decision coverage

Value for age	Value for married	Decision outcome (compound predicate as a whole)	Test case ID
30	True	True	1
75	True	False	2

Note that these tests would not exercise the possible outcome for married as false. A defect in the logical operator for condition 2, for example, may not be detected. Test cases 2 and 3 shown as follows would cover both possibilities.

Test cases for condition coverage

Value for age	Value for married	Condition 1 outcome	Condition 2 outcome	Test case ID
75	True	False	True	2
30	False	True	False	3

Note that the tests result in each condition in the compound predicate taking on a true/false outcome. However, all possible outcomes for the decision as a whole are not exercised so it would not satisfy decision/condition coverage criteria. Decision/condition coverage requires that every condition will be set to all possible outcomes and the decision as a whole will be set to all possible outcomes. A combination of test cases 1, 2, and 3 would satisfy this criterion.

Test cases for decision condition coverage

Value for age	Value for married	Condition 1 outcome	Condition 2 outcome	Decision outcome (compound predicate as a whole)	Test case ID
30	True	True	True	True	1
75	True	False	True	False	2
30	False	True	False	False	3

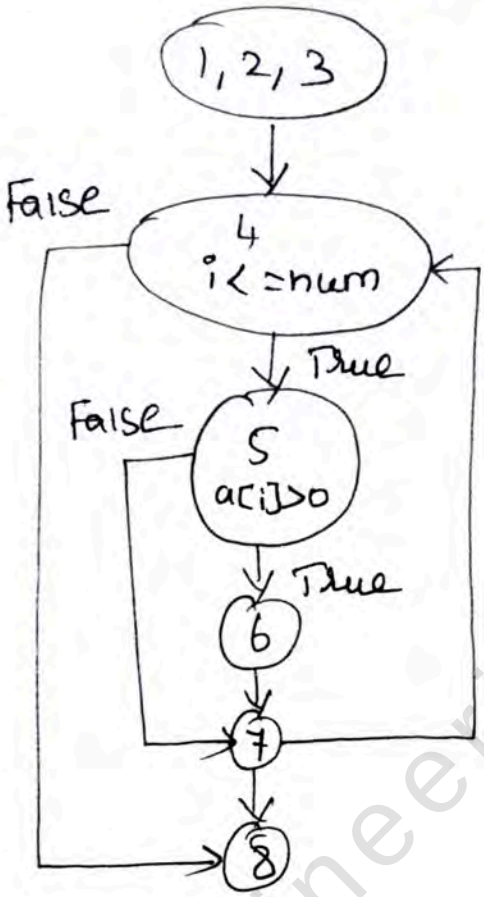
The criteria described above do not require the coverage of all the possible combinations of conditions. This is represented by yet another criterion called multiple condition coverage where all possible combinations of condition outcomes in each decision must occur at least once when the test cases are executed. That means the tester needs to satisfy the following combinations for the example decision statement:

Condition 1	**Condition 2**
True	True
True	False
False	True
False	False

In most cases the stronger the coverage criterion, the larger the number of test cases that must be developed to insure complete coverage. For code with multiple decisions and conditions the complexity of test case design increases with the strength of the coverage criterion. The tester must decide, based on the type of code, reliability requirements, and resources available which criterion to select, since the stronger the criterion selected the more resources are usually required to satisfy it.

Path:

Control flow graph



E = 7, N = 6

Cyclomatic complexity

V(G) = 7-6+2

V(G) = 3

Path:

* A path is a sequence of control flow nodes usually beginning from the entry node of a graph through to the exit node.

* Paths are denoted by

1 - 2 - 3 - 4 - 8

"-" dashes represents edges between two nodes.

* Cyclomatic complexity is a measure of the number of "independent" paths in the graph

Independent path:

* Deriving a set of independent paths using a flow graph can support a tester in identifying the control flow features in the code and in setting coverage goals

* "The independent paths are defined as any new path through the graph that introduces a new edge that has not be traversed before the path is defined".

Set of independent paths for flow graph

i) 1 - 2 - 3 - 4 - 8

ii) 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8

iii) 1 - 2 - 3 - 4 - 5 - 7 - 4 - 8

Basic set:

* A set of independent paths for a graph is called a basis set.

* The number of independent paths in a basis set is equal to the cyclomatic complexity of the graph.

* Identifying the independent paths provides useful support for achieving decision coverage goals.

Complete Path Coverage:

* Every path in a module must be exercised by the test set at least once.

* Even in a small and simple unit of code there may be many paths between the entry and exist node.

* Every loop multiplies the number of paths based on number of possible iteration of the loop.

* Complete Path Coverage for even a single module may not be practical & for large and complex modules it is not feasible.

* Some paths in a program may be unachievable (ie) they cannot be executed no matter what combinations of input data are used.

* So, Complete Coverage for path cannot be obtained It is impossible to work

1. The basis set is a special set of paths
2. It does not represent all the paths in the module.
3. This is used as a tool to help the tester in the process of getting decision coverage.

Code Complexity Testing:

* Cyclomatic Complexity is a source code complexity measurement that is being correlated to a number of coding errors.

* It is calculated by developing a control flow graph of the code that measures the number of linearly independent paths through a program module.

* These complexity measure helps to derive the test cases. By using this test cases the basic set of execution paths are used.

Steps to convert flowchart into flowgraph:

1. Find the decision points/predicates in the program.
2. check the Predicates are simple or not
3. Combine all the sequential statement into one.
4. If a set of sequential statement are followed by a simple predicated then, group all the sequential & predicate statements into node.

* They have two edges coming from one node. These are called "Predicate nodes"

5. Check whether all edges ended at some node.
6. Add a node to represent all the set of sequential statements at the end of the program.

Calculating cyclomatic complexity for code:

1. Draw the flow graph from given set of program statements

2. From flow graph, by using following formula

$V(G) = E - N + 2$

or

$V(G) =$ Edges $-$ Nodes $+ 2$

or

$V(G) =$ Edges $-$ Nodes $+$ number of exit points

3. other complexity formula

$V(G) = P + 1$

$V(G) =$ complexity P - Number of Predicate nodes

Eg:

Number of independent paths = 2

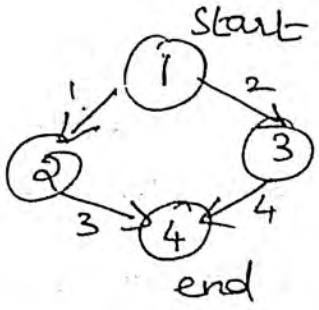
1 - 2 - 3

1 - 3 - 4

Number of nodes(N) = 4

1, 2, 3, 4

Number of edge(E) = 4



* cyclomatic complexing $V(G) = E - N + 2$

$= 4 - 4 + 2 = 2$

$V(G) = (2)$

* Number of predicate nodes(P) = 1

* cyclomatic complexity $V(G) = P + 1 = 1 + 1 = 2$

Additional White Box Test Design Approaches

In addition to methods that make use of software logic and control structures to guide test data generation and to evaluate test completeness there are alternative methods that focus on other characteristics of the code. One widely used approach is centered on the role of variables (data) in the code. Another is fault based. The latter focuses on making modifications to the software, testing the modified version, and comparing results. These will be described in the following sections of this chapter.

Data Flow and White Box Test Design

In order to discuss test data generation based on data flow information, some basic concepts that define the role of variables in a software component need to be introduced.

We say a variable is defined in a statement when its value is assigned or changed.

For example in the statements

```
Y = 26 * X
Read (Y)
```

the variable Y is defined, that is, it is assigned a new value. In data flow notation this is indicated as a def for the variable Y.

We say a variable is used in a statement when its value is utilized in a statement. The value of the variable is not changed.

A more detailed description of variable usage is given by Rapps and Weyuker [4]. They describe a predicate use (p-use) for a variable that indicates its role in a predicate. A computational use (c-use) indicates the variable's role as a part of a computation. In both cases the variable value is unchanged. For example, in the statement

Y =26*X

the variable X is used. Specifically it has a c-use. In the statement if (X >98)

Y= max

X has a predicate or p-use. There are other data flow roles for variables such as undefined or dead, but these are not relevant to the subsequent discussion.

Loop Testing

Loops are among the most frequently used control structures. Experienced software engineers realize that many defects are associated with loop constructs. These are often due to poor programming practices and lack of reviews. Therefore, special attention should be paid to loops during testing. Beizer has classified loops into four categories: simple, nested, concatenated, and unstructured

Loop testing strategies focus on detecting common defects associated with these structures. For example, in a simple loop that can have a range of zero to n iterations, test cases should be developed so that there are:

(i) zero iterations of the loop, i.e., the loop is skipped in its entirely;

(ii) one iteration of the loop;

(iii) two iterations of the loop;

(iv)k iterations of the loop where k n;

(v n 1 iterations of the loop;

(vi)n 1 iterations of the loop (if possible).
Mutation Testing

Mutation testing is another approach to test data generation that requires knowledge of code structure, but it is classified as a fault-based testing approach. It considers the possible faults that could occur in a software component as the basis for test data generation and evaluation of testing effectiveness.

Mutation testing makes two major assumptions:

1. The competent programmer hypothesis. This states that a competent programmer writes programs that are nearly correct. Therefore we can assume that there are no major construction errors in the program; the code is correct except for a simple error(s).

2. The coupling effect. This effect relates to questions a tester might have about how well mutation testing can detect complex errors since the changes made to the code are very simple. DeMillo has commented on that issue as far back as 1978 [10]. He states that test data that can distinguish all programs differing from a correct one only by simple errors are sensitive enough to distinguish it from programs with more complex errors.

Mutation testing starts with a code component, its associated test cases, and the test results.

The original code component is modified in a simple way to provide a set of similar components that are called mutants.

Each mutant contains a fault as a result of the modification. The original test data is then run with the mutants. If the test data reveals the fault in the mutant (the result of the modification) by producing a different output as a result of execution, then the mutant is said to be killed. If the

mutants do not produce outputs that differ from the original with the test data, then the test data are not capable of revealing such defects. The tests cannot distinguish the original from the mutant. The tester then must develop additional test data to reveal the fault and kill the mutants. A test data adequacy criterion that is applicable here is the following:

A test set T is said to be mutation adequate for program P provided that for every in equivalent mutant Pi of P there is an element t in T such that Pi(t) is not equal to P(t).

The term T represents the test set, and t is a test case in the test set. For the test data to be adequate according to this criterion, a correct program must behave correctly and all incorrect programs behave incorrectly for the given test data.

Evaluating Test Adequacy Criteria:

* Testers are often faced with the decision of which criterion to apply to a given item under test given the nature of the item and the constraints of the test environment (time, costs, resources)

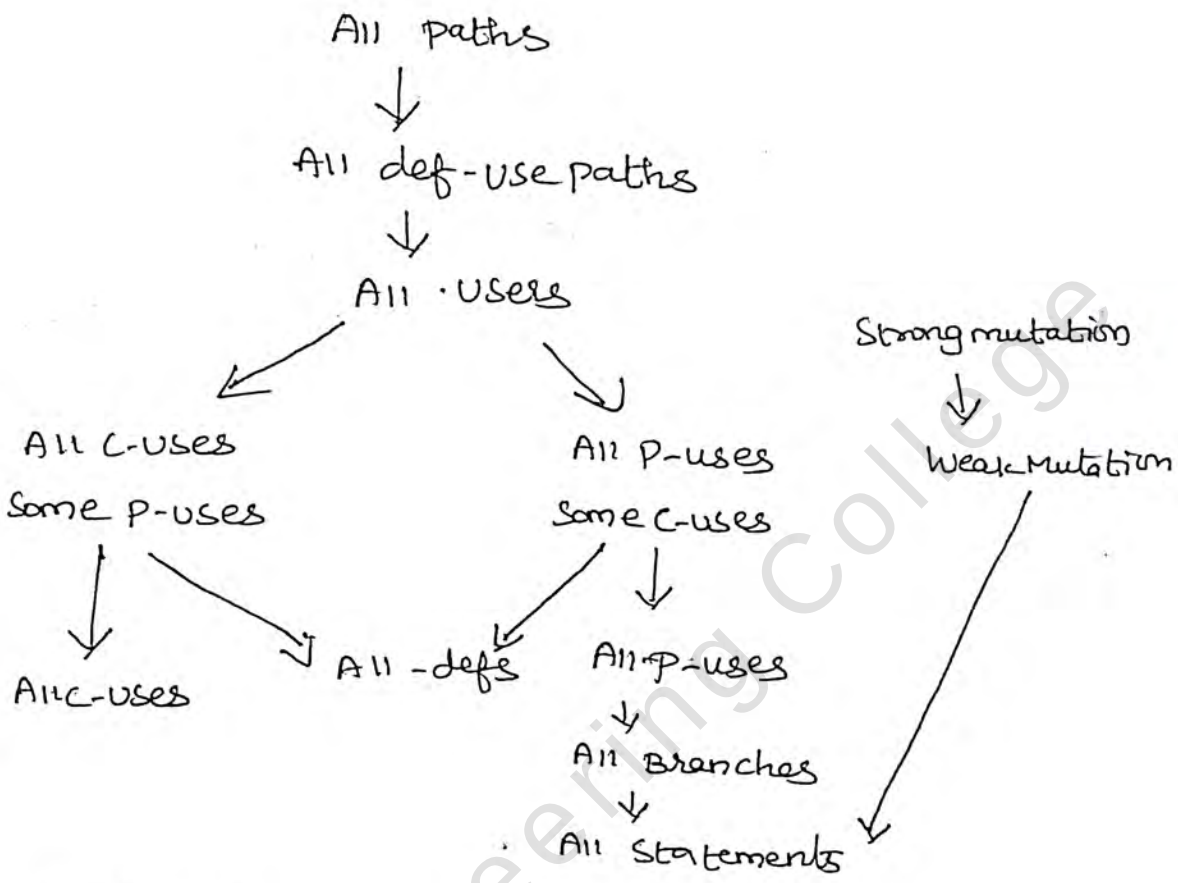
* one source of information the tester can use to select an appropriate criterion is the test adequacy criterion hierarchy.

* Test adequacy hierarchy describes a subsumes relationship among the criteria.

* Satisfying an adequacy criterion at the higher levels of the hierarchy implies a greater thoroughness in testing.

* Test adequacy criteria is useful to

1. Test selection Approaches
2. Revealing missing tests
3. often in combination
4. Define a notion of "Thoroughness" in a test suite
5. Defined interms of "covering" some information
6. May be used for selection as well as measurement.



Test adequacy criterion hierarchy

Testers can use the axioms to

1) Recognize both strong and weak adequacy criteria
2) properties of selected test data adequacy criteria
3) It help to choose a suitable criterion
4) Stimulate thought for the development of new criteria.

the weyuker's eleven axioms that allow testers to evaluate test adequacy criteria

Axioms:

It is a rule it helps to tester in order to

1. Recognize adequacy criteria
2. Properties of selected test data adequacy Criteria
3. It help to choose a suitable criterion
4. Stimulate thought for the development of new criteria.

Axioms are based on the following set of assumptions

1. Programs are written in a structured programming language.
2. Programs are single entry/exit
3. All input statements appear at the beginning of the program.
4. All output statements appear at the end of the program

Weyuker axioms/properties:

1. Applicability property

* For every program there exists an adequate test set

2. No exhaustive applicability property

For a program P and a test set T, P is adequately tested by the test set T, and T is not an exhaustive test set

3. Monotonicity property:

If a test set T is adequate for program P and if T is equal to or a subset of T' then T' is adequate for program P.

4. Inadequate empty set

An empty test set is not an adequate test for any program.

5. Anti extensionality property:

These are programs P and Q such that P is equivalent to Q and T is adequate for P, but T is not adequate for Q

6. General multiple change property:

These are programs p and q that have the same shape and there is a test set T such that T is adequate for P, but is not adequate for Q.

7. Anti decomposition property:

There is a program P and a component Q such that T is adequate for P, T' is a set of vectors of values that variables can assume on entrance to Q for some L in T, and T' is not adequate for Q.

8. Anticomposition property:

* These are programs P and Q, and test set T such that T is adequate for P. and the set of vectors of values that variables can assume on entrance to Q for inputs in T is adequate for Q, but T is not adequate for P.

9. Renaming Property:

* If P is a renaming of Q, then T is adequa for P only if T is adequate for Q.

* A program P is a renaming of Q if P is identical to Q expect for the fact that all instanc of an identifier, let us say "a" in Q have been replaced in P by an identifier.

* Let us say b where "b" does not occur in Q, or if there is a set of such renamed identifiers.

10. Complexity Property:

* For every n, there is a program P such that P is adequately tested by a size n test set, but not by any size n-1 test set

11. Statement Coverage Property

* If the test set T is adequate for P, then T causes every executable statement of P to be executed.

IT8076-SOFTWARE TESTING

UNIT-3
LEVELS OF TESTING

NEED FOR LEVELS OF TESTING

Execution based software testing, especially large systems, is usually carried out at different levels mostly 3-4 levels.

Major Phases of testing:

- Unit Testing
- Integration Testing
- System Testing

A Principal goal is to detect functional and structural defects in the unit.

At the integration level several components are tested as group, and tester investigates component interaction.

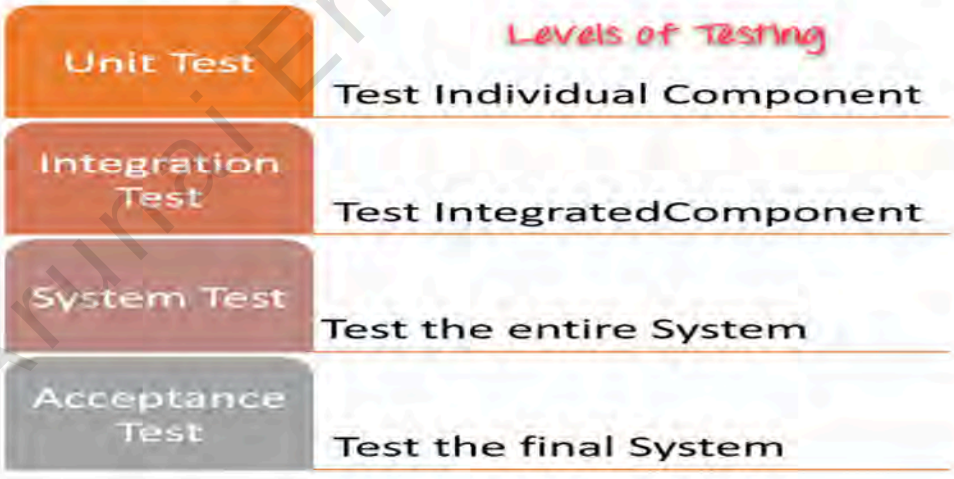
At the system level the system as a whole is tested and a principal goal is to evaluate attribute such as ability, reliability and performance

Level of Testing and Software Development Paradigms

The approach used to design and develop a software system has an impact on how testers plan and design suitable tests.

The TWO major approaches to system development are **Bottom up and Top down approaches.** These approaches are supported by two major types of programming languages- procedure Oriented and Object Oriented.

Level of Testing



The different nature of the code produced requires testers to use different strategies to identify and test components and component groups.

Systems developed with procedural languages are generally viewed as being composed of passive data and active procedures

When test cases are developed the focus is on generating input data to pass to the procedures (or functions) in order to reveal defects.

Object oriented systems are viewed as being composed of active data along with allowed operations on that data, all encapsulated within a unit similar to abstract data type.

Unit Test:

* Unit Testing is a testing in which the individual unit of the software are tested in isolation from other parts of a program.

* A unit is the smallest possible testable software components.

* A unit in a typical procedure-oriented software system.

a) Performs a single cohesive function

b) Can be compiled separately

c) Is a task in a work breakdown structure

d) Contains code that can fit on a single page (or) screen.

* A unit is said to be procedure (or) function, by written / using a procedural programming languages.

Some Components Suitable for Unit Test

procedures and Functions	Classes/objects and Methods	procedure sized reusable components (small-sized COTS components or components from an in-house reuse library

* The principal goal for Unit Testing is insure that each individual software unit is functioning according to its specification.

* To Prepare for unit test the developer/tester must perform several tasks. They are

1) Plan the general approach to unit testing.
2) Design the test cases, and test procedures.
3) Define relationships between the tests.
4) Prepare the auxiliary code necessary for unit test.

Unit Test Planning:

* It define "what to test", "How to test" "When to test" and "who to test"

* Test plan is a project level document

* It may be prepared as a component of the
  i) Master test plan or
  ii) Stand-alone plan

* It should be developed in conjunction with the master test plan and the Project plan for each project.

* Documents that provide inputs for the unit test plan are
  a) project plan b) Requirement
  c) specification d) Design Documents that describe the target unit

* set of development Phases for Unit test Planning are
  Phase1: Describe Unit test approach and risks
  Phase2: Identify unit features to be tested
  Phase3: Add levels of detail to the plan.

* In each Phase a set of activities is assigned based on IEEE Unit test standard.

Phase1: Describe Unit test approach and risks

The general approach to unit testing is outlined. The test Planner:
  a) Identifies test risks
  b) Describes techniques to be used for designing the test cases for the units
  c) Describes techniques to be used for data validation and recording of test results.

d) Describes the requirements for test harness and other software that interfaces with the Units to be tested.

* The planner identifies completeness requirements such as states, functionality control and data flow patterns.

* The planner also identifies termination conditions for the Unit tests. This includes coverage requirements and special cases.

* Special cases may result in abnormal termination of Unit test. Strategies for handling these special cases need to be documented.

* The planner estimates resources needed for Unit test, such as hardware, software and staff and develops a tentative schedule under the constraints identified at that time.

Phase 2: Identify Unit features to be tested:

* This phase requires information from the Unit specification and detailed design description.

* The planner determines which features of each Unit will be tested.

Eg: Functions, Performance requirements, States, State transitions control structures, messages and data flow patterns.

* If Some features will not be covered by the tests, they should be mentioned and the risks of not testing them be assesed.

* Input/output characteristics associated with each unit should also be identified, such as variables with an allowed ranges of values and performance at a certain level.

Phase 3: Add levels of detail to the plan

* The planner refines the plan as produced in the previous two phases.

* The planner adds new details to the approach, resource and scheduling portions of the unit test plan.

Eg: Existing test cases that can be reused for this project can be identified in this phase.

* Unit availability and Integration scheduling information should be included in the revised version of the test plan

* The planner must be sure to include a description of how test results will be recorded.

Designing the Unit Tests:

* Part of the preparation work for unit test involves unit test design. It is important to specify

a) The test cases

b) The test procedures

Test case data should be tabularized for ease of use and reuse.

Test Case Specification notation:

* Arrange the components of a test case into a semantic network with parts, object-ID, Test-Case-ID, purpose and List-of-Test-case-steps.

* Test design specification includes lists of relevant states, messages, exceptions and interrupts.

* Developer/tester should describe the relationship between the tests.

* Test suites can be defined that bind related tests together as a group. All of this test design information is attached to the unit test plan.

* Test cases, test procedures and test suites may be reused from past project if it has been careful to store, so that they are easily retrievable & reusable.

* Test case design at the unit level can be based on use of the black and white box test strategies.

* Both of these approaches are useful for designing test cases for functions and procedures.

The test harness:

* In addition to developing the test cases, supporting code must be developed to exercise each unit and to connect it to the outside world.

* The tester is considering a stand-alone function/ procedure/ class, rather than a complete system, code will be needed to call the target unit and also to represent modules that are called by the the target unit. This code is called test harness.

* The auxiliary code developed to support testing of units and components is called a test harness.

* The harness consists of drivers that call the target code and stubs that represent modules it calls

* The development of drivers and stubs requires testing resources.

* The drivers and stubs must be tested themselves to insure they are working properly and they are reusable for subsequent releases of the software.

Eg A driver could have the following options and combinations of options.

a) Call the target Unit

b) do 1, and Pass inputs parameters from the table

c) do 1, 2 and display Parameters.

d) do 1, 2, 3 and display results.

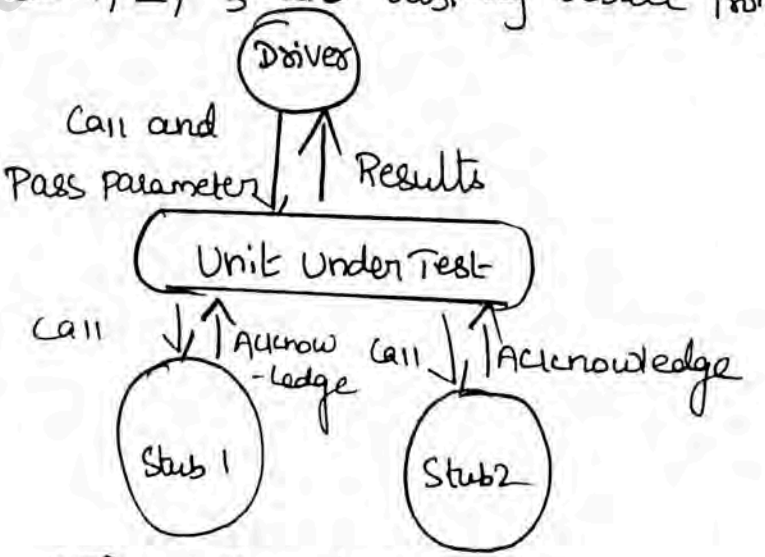
The stubs could also exhibit different levels of functionality. A stub could:

a) display a message that it has been called by the target unit

b) do 1 and display any input parameters passed from the target unit

c) do 1, 2 and pass back a result from a table

d) do 1, 2, 3 and display result from table



The Test harness

Running the Unit test and Recording Results.

Unit tests can begin when

a) The Units becomes available from the developers

b) The test cases have been designed and reviewed.

c) The test harness and any other supplemental supporting tools are available.

* The testers then proceed to run the tests and record results

* Test logs can be used to record the results of specific tests.

* The status of the test effort for a unit, and a summary of the test results could be recorded in a simple format.

Summary work sheet for unit test results

Unit Test Worksheet			
Unit Name: _____			
Unit Identifier: _____			
Tester: _____			
Date: _____			
Test case ID	Status (run / not run)	Summary of results	Pass/fail

* The tester have to carefully record, review and check test results at any level of testing.

* The tester must determine from the results whether the unit has passed or failed the test.

* If the test is failed, the nature of the problem should be recorded in a test incident report.

* When a unit test fails a test there may be several reasons for the failure. The most likely reason is a fault in the unit implementation (code)

Other reasons are

1) A fault in the test case specification
2) A fault in test procedure execution
3) A fault in the test environment
4) A fault in the unit design.

* The causes of the failure should be recorded in a test summary report, which is a summary of testing activities for all the units covered by unit test.

* When a unit test has been completely tested & finally passes all of the required tests its ready for integration.

* Test summary report is a valuable document for the group responsible for integration & system tests.

Integration Testing :-

* Integration is defined as the set of interactions among components.

* Testing the interaction between the modules and interaction with other systems externally is called integration testing.

* "At the integration level several components are tested as a group and the tester investigates components interactions."

Goals:

↳ Integration test for procedural code has two major goals.

1. To detects defects that occur on the interfaces of units

2. To assemble the individual units into working sub systems and finally a complete

system that is ready for system test.

* Integrating testing works best as an iterative process in procedural-oriented system. one unit at a time is integrated into a set of previously integrated modules which have passed a set of integration tests.

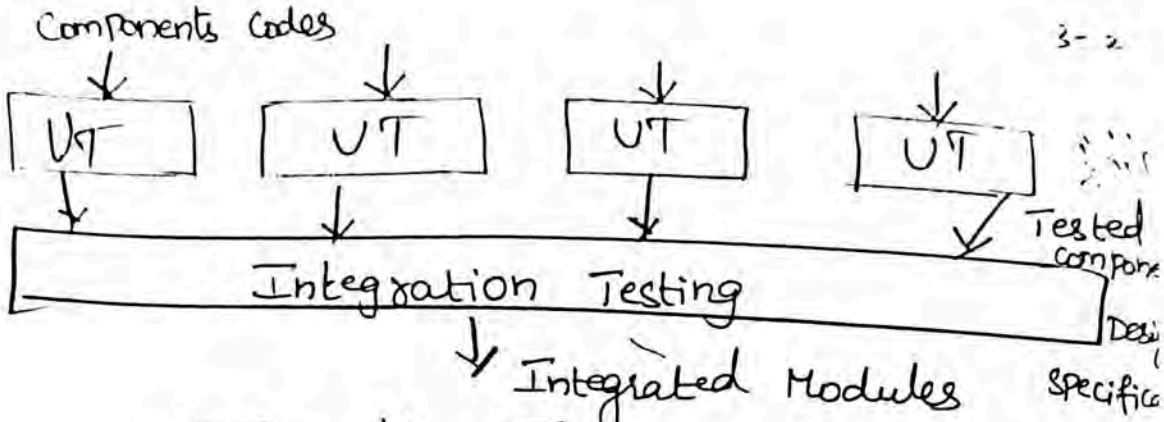
* The interfaces and functionality of the new unit in combination with the previously integrated units is tested.

Integration Strategies for Procedures and Functions:

For procedual and functional oriented system these are major integration strategies.

1) TOP-down integration
2) Bottom-up integration
3) Bi-directional integration
4) System integration.

TOP-down and Bottom-UP integration Strategies only one module at a time is added to the growing subsystem



Integration Testing

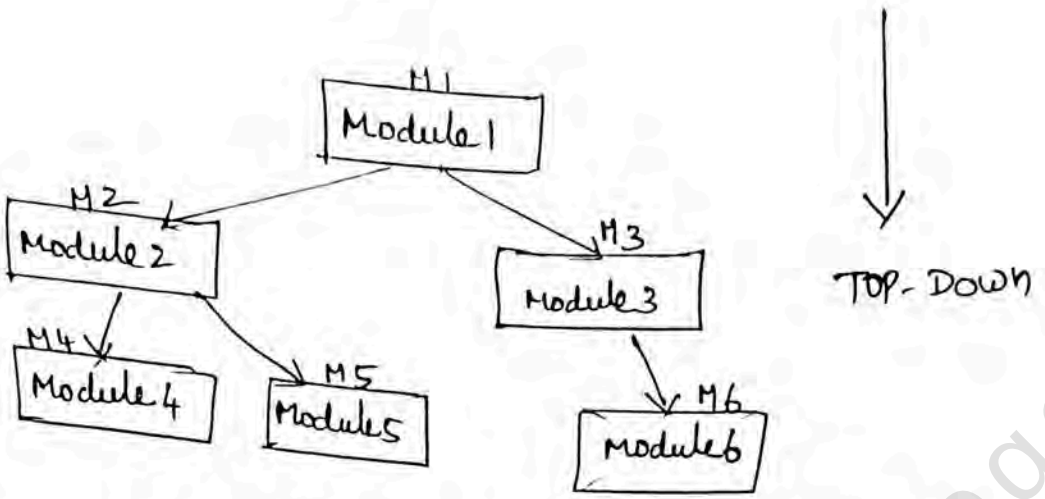
1) Top-Down Integration:

* Integration testing involves testing the top most component interface with other components in same order as navigate from top to bottom till cover all the components

* If a set of components and their related interfaces can deliver functionality without expecting the presence of other components or with minimal interface requirements in the software/ product then the set of components and their related interfaces is called as a "Sub-systems".

* Each sub-system in a product can work independently with or without other sub-systems

* This makes the integration testing easier and focus on required interfaces rather than getting worried about each & every combination of components.



* Begin top-down integration with module1 Create two stubs to represents module 2 and Module3. when the tests are passed, then replace the two stubs by the actual module one at a time.

* one can traverse the structure chart and integrate the modules in a depth (or) Breadth-first manner.

Depth-first Manner (approach):

↳ order of integration M_1, M2, M4, M5, M3, M6

Broadth-first approach:

↳ order of integration M_1, M2, M3, M4, M5, M6

Advanteges:

↳ The upper-level modules are tested early in integration. If they are complex and need to be redesigned there will be more time to do so.

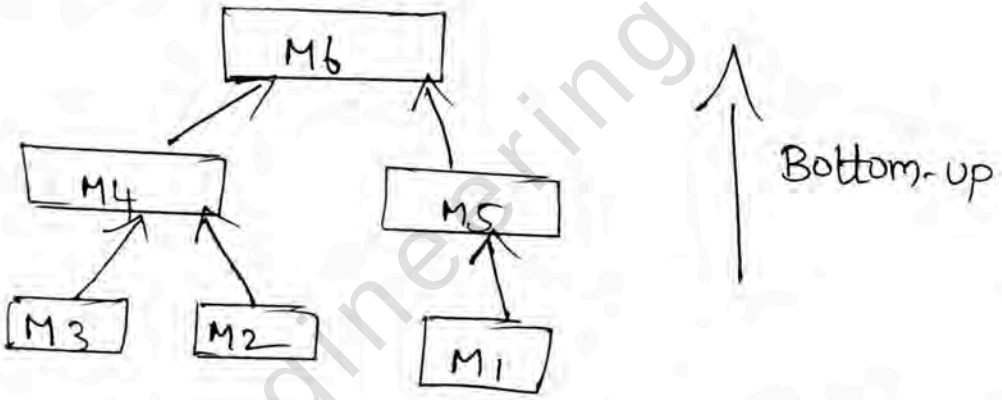
Disadvantages:

* Top-down integration requires the development of complex Stubs to drive significant data upward.

* Basic functionality is tested at the end of cycle.

2) Bottom-Up Integration:

* Bottom-Up Integration is just the opposite of top-down integration.



* Bottom-up integration of the modules begins with testing the lowest modules, those at the bottom of the structure Chart (M3, M2, M1, M4, M5, M6)

* Components or systems are substituted by drivers. Developer used a temporary program, instead of main module construction, It is called "Drivers"

* Drivers are needed to test these modules. The next step is to integrate modules on the next upper levels of the structure chart whose subordinate modules have already been tested.

Eg: M3 and M2 are tested, then select M_4 and integrate it with M3 and M2. the actual M_4 replaces the drivers for these modules.

* Main difference between Top-down & Bottom-up is the arrows from top to bottom (downward-pointing arrows) indicate interaction or control flow.

* The arrows from bottom to top (upward-pointing arrows) indicate integration approach or integration path.

* Top-down integration approach is best suited for waterfall & V models.

* Bottom-up integration approach for the iterative and agile methodologies.

* In practical scenario the approach selected for integration depends more on the design and architecture of a product and on associated priorities.

Advantage:

* The low-level modules are usually well tested early in the integration process. This is important if these modules are candidates for reuse.

Disadvantage:

* It is required to create the test drivers for modules at all levels, except the top control

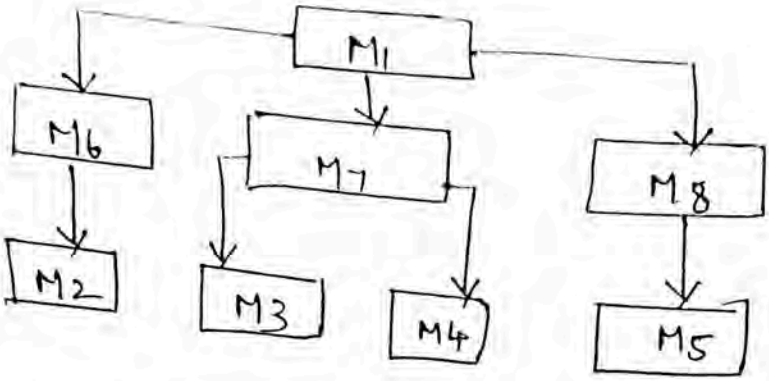
3) Bi-Directional Integration:

* Bi-directional integration is a combination of the top-down and bottom-up integration approach used together to derive integration steps.

* Let us assume the software components become available in the order mentioned by the component numbers.

* The individual components 1, 2, 3, 4 & 5 are tested separately and bi-directional integration is performed initially with the use of stubs and drivers.

* Drivers are used to provide upstream connectivity while Stubs provide downstream connectivity



Bi-directional Integration

* A driver is a function which redirects the requests to some other component and Stubs simulate the behavior of a missing component.

* After the functionality of these integrated Components are tested, the drivers and Stubs are discarded.

* once components 6, 7 and 8 become available the integration methodology then focuses only on those components, as these are the components which need focus and are new. This approach is also called "sandwich integration".

Steps for integration Using sandwich Testing

Step	Interfaces tested
1	6-2
2	7-3-4
3	8-5
4	(1-6-2)-(1-7-3-4)-(1-8-5)

Eg 1-3 use a bottom-up integration approach and Step 4 use a top-down integration approach

Advantage:

* Easily Combine modules (Sub modules and main module)
* It helps to developers effectively.

Disadvantage:

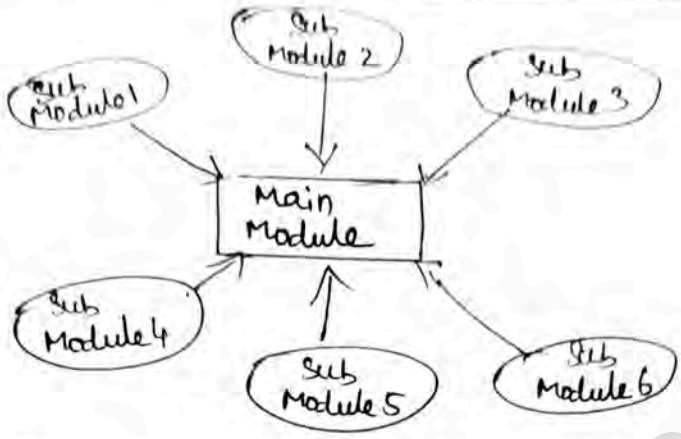
* It is temporary one & Unstructured one.

4) System Integration:

* System integration means that all the components of the system are integrated and tested as a single Unit.
* Integration testing, which is testing of interfaces, can be divided into two types.
  - ↳ Components or sub-system integration
  - ↳ Final integration testing or system integration
* Instead of integrating component by component and testing, this approach waits till all components arrive and one around of integration testing is done. This approach is also called "Big bang integration"

* It reduces testing effort and removes duplication in testing.

Big - Bang Approach



* System integration using the big bang approach is well suited in a product development scenario where the majority components are already available and stable and very few components get added or modified.

* In this case, instead of testing components interface one by one, it make sense to integrate all the components at one go and test once, saving effort and time for the multi-step component integration.

Advantage:

* Saves time and effort

Disadvantage:

* Difficult to trace the case of failure.

Integration Test planning:-

* Integration test must be planned. planning can begin when high-level design is complete so that the system architecture is defined.

* Integration test planning required following documents.

1. Requirement document (SRS)
2. User manuals (help documents)
3. working sequence (module description)
4. Design document
5. Design test plan

The Documents provides/gives,

1. Structure chart 2. State Chart 3. Data dictionaries 4. Cross reference tables. 5. Module interface descriptions. 6. Data flow Descriptions 7. Message and events descriptions. 8. Unit description / procedure oriented System units 9. Integration test Strategy.

10. Cluster - Cluster test plan, language description of functionality, Collection of classes in the cluster, cluster test cases are included.

* The main objective of integration test is to construct the working subsystems.

* The selection of an integration strategy depends on,

- ✓ Software characteristics
- ✓ project schedule
- ✓ Testing resources etc.

These critical/risk modules are identified by

1. It require more software requirements
2. High level of control
3. complex
4. Error prone
5. Poor performance etc.

Integration Test plan

Integration Test plan (ITP)
1. Test items
2. Features to be tested i) Integration with dispatcher software ii) Integration with client software iii) Integration with agent software
3. Test deliverables
4. Testing tasks
5. Environment needs
6. Test case pass/fail Criteria

SCENARIO TESTING

Scenario testing is defined as a "set of realistic user activities that are used for evaluating the products" .It is also defined as testing involving customer scenarios.

There are two methods to evolve scenario

1. **System Scenario**
2. **Use case Scenario/ Role Based Scenarios.**

System Scenario:-

System Scenario is a method where by the set of activities used for scenario testing covers several components in the system.

The **following approaches** can be used to develop system scenarios.

Story-line:

Develop a story-line that combines various activities of the product

Life-cycle / state transitions:

Consider an object, derive the different transitions / modification that happen to the object and derive scenarios to cover them

Deployment / implementation details from customer:

Develop a scenario from a known customer Deployment / implementation details and create a set of activities by various users in that implementation

Business verticals:

Visualizing how a product / software will be applied to different business verticals and create a set of activities as scenarios (e.g., insurance, life sciences)

Battle-ground scenarios:

Create some scenarios to justify "the product works" and some scenarios to "try and break the system" to justify "the product doesn't work."

Use Case Scenarios:-

- ✓ ***A use case Scenario is a stepwise procedure on how a user intends to use a system, with different user roles and associated parameters.***
- ✓ A use case scenario can include stories, pictures and deployment details.
- ✓ Use cases are useful for explaining customer problems and how the software can solve those problems without any ambiguity

Example:-

The scenario below is an example of withdrawing a cash from a bank. A customer fills up a cheque and gives it to an official in the bank. The official verifies the balance in the account from the computer and gives the required cash to the customer .The customer in this example is a actor, the clerk the agent , and the response given by the computer which gives the balance in the account , is called the system response.

Actor	System Response
User would like to withdraw cash and inserts the card in ATM machine	Request for Password or PIN (Personal identification number)
User Fill-in the Password or PIN	
	Validate the password or PIN
	Give a list containing types of accounts
User selects an account type	
	Ask the user for amount
User Fill-in the amount of cash required	
	Check availability of funds Update account balance Prepare receipt Dispense cash
Retrieve cash from ATM	
	Print receipt

Actor and System Response in Use Case for ATM cash withdrawal

DEFECT BASH ELIMINATION SYSTEM TESTING:

• Defect bash is an ad hoc testing, done by people performing different roles to bring out all types of defects.

• It is very popular among application development companies, where the products can be used by people who perform different roles.

• The testing by all the participants during the defect bash is not based on written test cases.

• Defect bash brings together plenty of good practices that are popular in testing industry. They are as follows :-

1. Enabling people to "cross boundaries and test beyond assigned area"
2. Bringing different people performing different roles together in the organization for testing - "Testing isn't for testers alone"
3. Let everyone in organization use the product before delivery - "Eat your own dog food"
4. Bringing fresh pairs of eyes to uncover new defects – "Fresh eyes have less bias"
5. Bringing in people who have different levels of product understanding, to test the product together randomly – "Users of software are not the same"
6. Testing doesn't wait for the time taken for documentation – "Does testing wait till all documentation is done?"
7. Enabling people to say the "system works" as well as enabling them to "break the system" – "Testing isn't to conclude that the system works or doesn't work

Even though it is said that defect bash is an ad hoc testing, not all activities of defects bash are unplanned. All the activities in the defect bash are planned activities, except for what to be tested .

It involves several steps:-

1. Choosing the frequency and duration of defect bash.
2. Selecting the right product build.
3. Communicating the objectives of each defect bash to everyone
4. Setting up and monitoring the lab for defect bash.
5. Taking action and fixing issues.
6. Optimizing the effort involved in defect bash.

1. Choosing frequency and duration

• Too frequent or too few rounds may not meet objective

• Optimize duration involved

2. Selecting right product build

• Good-quality product • Regression tested build

• Too many defects spoil confidence

3. Communication objective of defect bash

• Purpose & objective has to be clear

• Areas of focus to be communicated

• Defects that can be found easily by test team shouldn't be objective

4. Setting up and monitoring lab

• Right configuration and resources

• Easy install & set-up help

• Optimized for both functional & non-functional defects

• Monitor all resources (RAM, disk, CPU, network)

5. Taking actions and fixing issues

• Duplicate defects

• Not possible to look at each defect alone due to volume

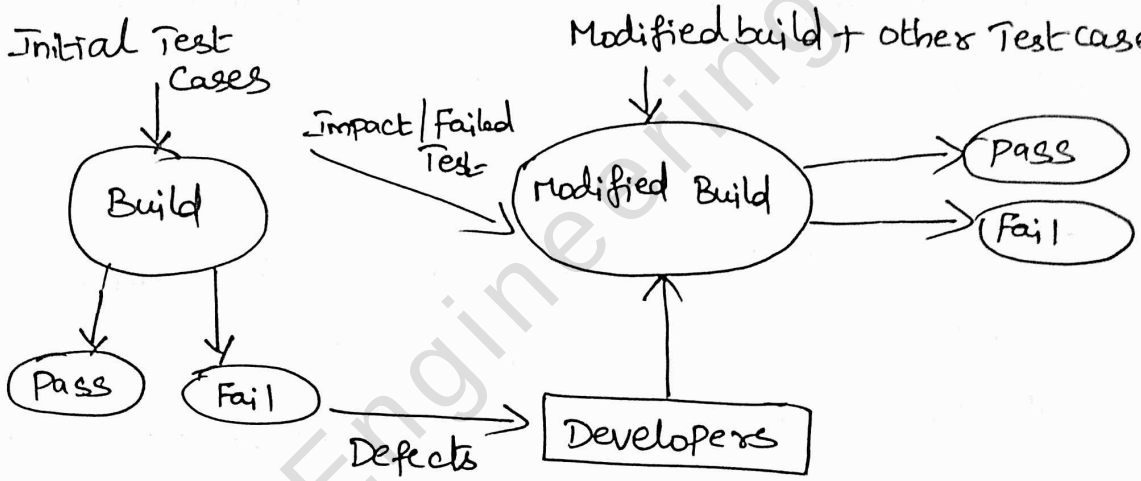
• Code reviews and inspections

Regression Testing :-

Definition:

"The re-execution of tests on modified build to ensure bug-fix works and possibilities of side effects occurrence" is called regression Testing.

Regression Testing



Purpose:

* The purpose of regression testing is to confirm that a recent program or code change has not adversely affected existing features.

Needs for regression Testing:

* Regression testing is required when there is a,

1. Change in requirements and code is modified according to the requirement.

2. New feature is added to the software.
3. Defect fixing.
4. Performance issue fix

Types:

1. Normal / regular regression tests
2. Final regression Tests

1. Normal / regular regression Tests:

* It is performed to verify if the build has not broken any other parts of the application by the recent code changes for defect fixing for enhancement.

2. Final regression Tests

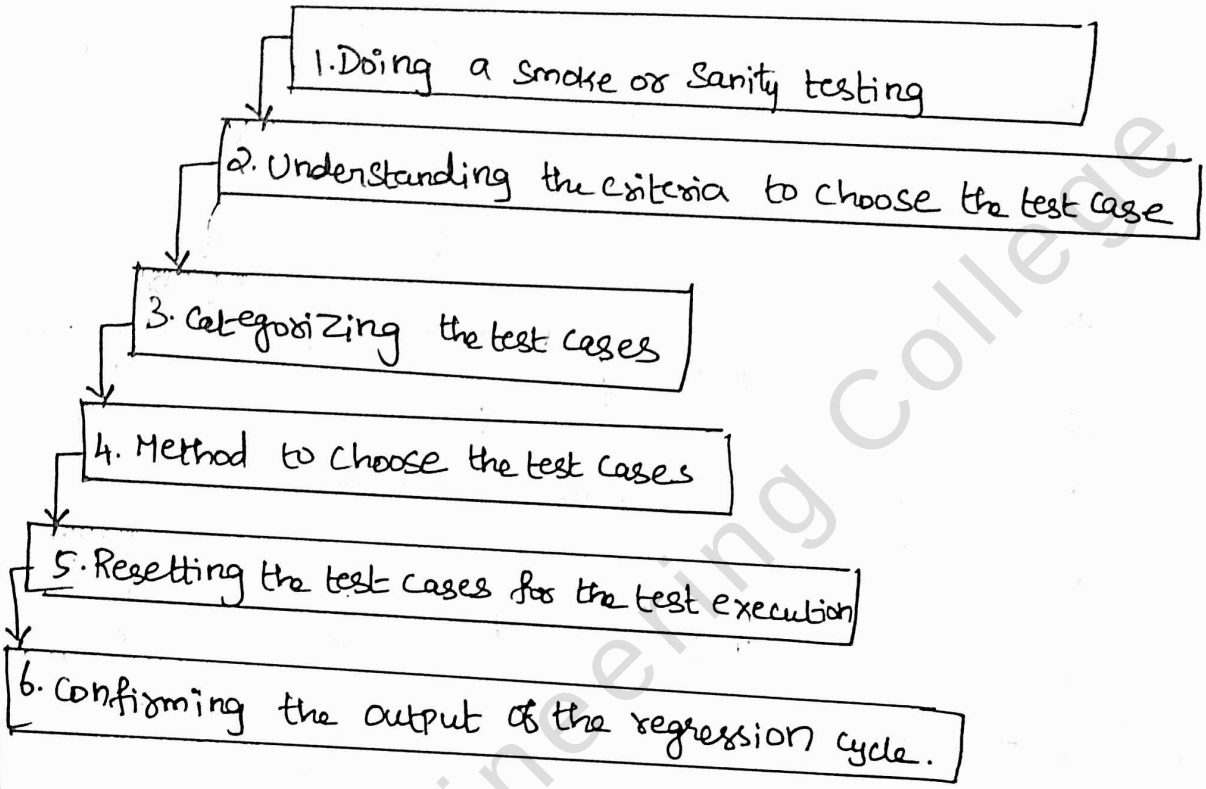
* It is performed to validate the build that has not changed for a period of time.
* This build is deployed or shipped to customers.

When to perform regression testing?

1. when the buy/defect is reopens?
2. Particular amount of defect found and repeated.
3. Fixed defects create additional defects / bugs.
4. whenever new functionality is added in the system.

Regression testing steps:

* Regression tests are the ideal cases of automation which results is better Return on Investment (RoI)



Regression Testing - Process

Step 1: Doing a Smoke or Sanity test

Smoke Testing:

* It is performed to ascertain that the critical functionalities of the program is working fine or not.
* It is to verify the "stability" of the system in order to proceed with more rigorous testing.
* It is usually documents/scripted.
* It is subset of regression testing.

Sanity Testing:

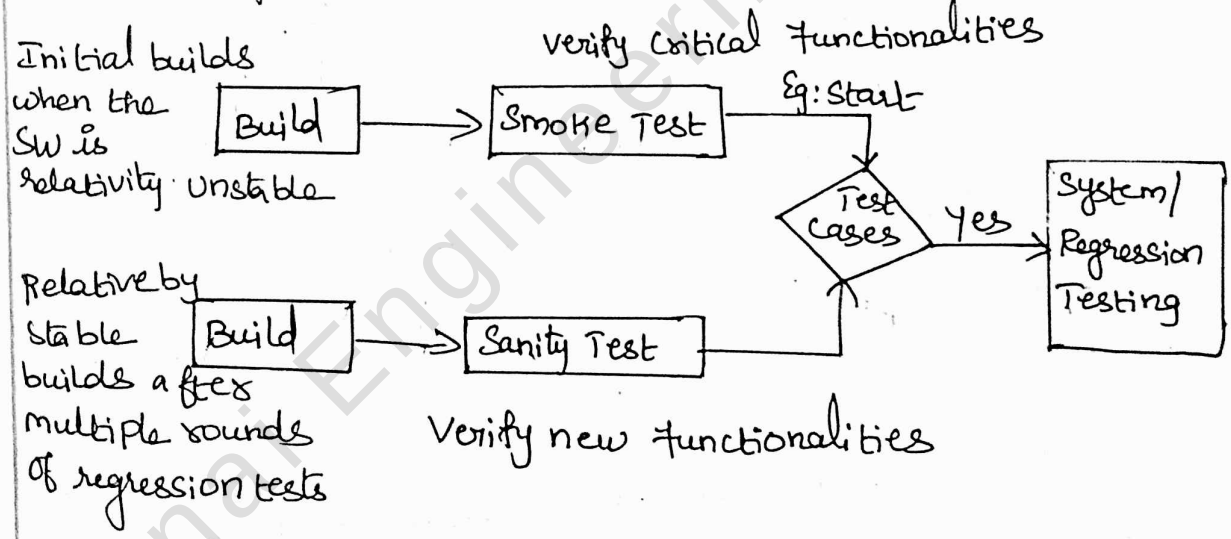
* It is done to check the new functionality / bugs have been fixed.

* It is verify the "rationality" of the system in order to proceed with more rigorous testing.

* It is a subset of acceptance testing.

* The above step helps, whether the product is built properly or not?

* It helps to find basic functionality and critical functionality error.



Step 2 - Understanding the criteria to choose the test cases

Generally two methods are followed in order to select the test cases.

Approach 1: A constant collection of regression tests are employed to run for each build.

Approach 2: Test cases are dynamically selected by making judicious choices of the test cases.

Effective regression tests can be done by selecting following test cases,

1. Test cases which have frequent defects
2. Functionalities which are more visible to the users.
3. All integration test cases
4. All complex test cases
5. Boundary value test cases

Step 3: Categorizing the test cases:

* Test cases are classified into many types

Eg: Functionality test cases, UI test cases, DB test cases

The test cases are basically divided into 2 types

1. Importance 2. Customer usage.

* Three level priority categorization scheme is used for regression testing.

Priority -0: Allocated to all tests that must be executed in any case. (Very high priority value)

Priority -1: Allocated to be tests which can be executed, only when time permits.

Priority -2: Allocated to tests, which even if not executed, will not cause big upsets (moderate project values)

Step 4: Methods to choose test cases:

* once the test case are prioritized, test cases can be selected.

* Several approaches to Regression testing which need to be decided on a case by case basis.

Case 1:

* If Criticality and impact of the defects are low, then its enough to select few test cases from Test case DataBase (TCDB) and execute them.

* These can fall under any priority (0, 1 or 2)

Case 2:

* If the criticality and the impact of the bug fixes are medium, then we need to execute all priority-0, priority-1, test cases.

* Selecting priority-2 test cases in this code desirable but not a must.

Case 3:

* If the criticality and impact of the bug fixes are high, then we need to execute an priority-0, 1 and carefully selected priority-2 test cases.

Step 5: Reselting the test cases for execution:

* Reselting of the test cases need to be done with the following considerations,

1. when there is a major change in the product
2. Different results generated compare to previous stage/cycles.
3. when there is a change in the build procedure which affects the products.

Step 6:- Confirming the Results of regression testing:-

* Regression testing uses only are build for testing.
* It is expected that all 100% test cases pass using the same build.
* In Situation where the pass % is not 100, the test manager check the previous result and conclude the results.

1. If the result of a particular test cases was pass using the previous builds and fail in the current build then regression is failed.
2. If the result of a particular test cases was a fail. Using the previous builds and a pass in the current build, them it is easy to assume the bug fixes worked.

3. If the result of a particular test cases was a fail using the previous build and a fail in the current build and if there are no bug fixes for this particular test cases.

Internationalization Testing (I18N) Testing:

* It is a non-functional testing.
* It is a process of designing a software application, so that it can be adopted to various languages & regions without any changes.
* It is also called "Globalization" (or) "G11N" testing.

Purpose:

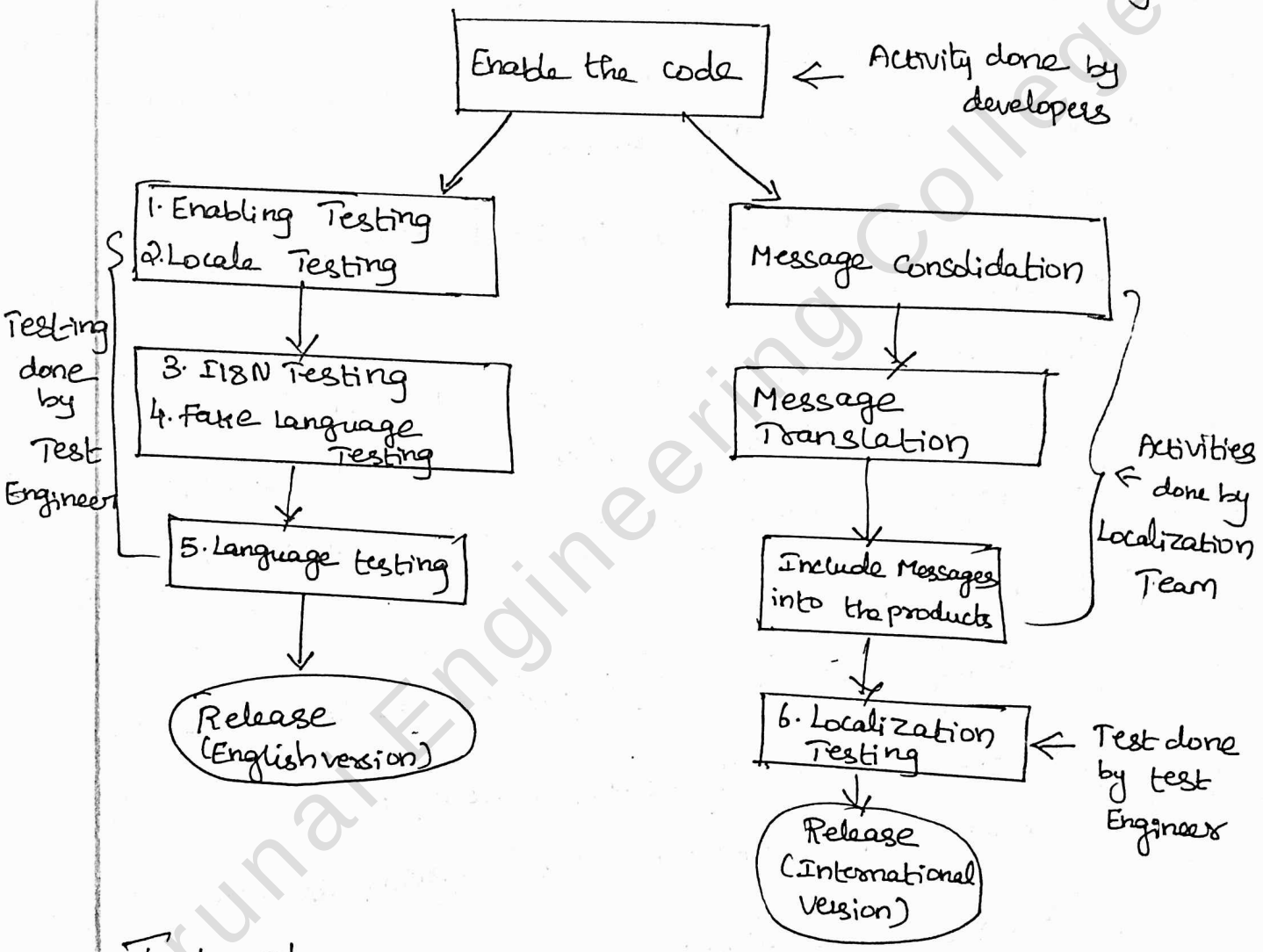
* It is to check if the code can handle all international support without breaking functionality that might cause data loss or data integrity issues.

Relevant terms:

1. Language - It is a tool employed for communication.
2. character set - It holds, Unicode, ASCII
3. Locale - It represent local Parameters/Conventions.
4. Internationalization - It is used to mean all activities that are required to make the software available for international market
5. Globalization - It is used to mean internationalization and localization.

6. Localization - It is used to mean the translation work of all software resources such as messages to the target language and conventions.

Test Phases for Internationalization (I18N) Testing



Test Phases:

* Some important aspects of internationalization testing are

1. Testing the code for how it handles input, strings and sorting items.
2. Display of messages for various language.

3. Processing of messages for various languages and conventions.

1. Enabling Testing:

* It is a white Box Testing methodology. It ensure that the source code used in the software allows I18N.

* An activity of code review or code inspection mixed with some test cases for unit testing, which an objective to catch I18N defects is called "Enabling testing".

* Enabling testing uses a check lists. It follows

1. Check the code for API/function calls.
2. Check the code for hard coded date/Currency format/ ASCII code / character constant.
3. Check the code to see that there are no computations done on data variables.
4. Check the dialog boxes and screens.
5. Ensure region cultural based messages.
6. Ensure that no String operations are performed in the code
7. Ensure that adequate size provided for buffer, variables.
8. Ensure all the messages.
9. Ensure that all the resources.

2. Locale Testing :-

* Changing the different locales using the system setting or environment variables, and testing the Software functionality, number, date, time and currency format is called "Locale testing".

* Locale testing uses check list, It follows,

1. Hot keys, functions keys and help screens are tested with many application locales.
2. Date/Time format are in line with defined locale of language.
3. Time zone information and day light saving time calculations.
4. Number format is in line with selected locale and languages.
5. Currency is in line with the locale selected language.

3. Internationalization validation:

Internationalization validation is performed with the following objectives.

1) The software is tested for functionality with ASCII, DBCS and European Characters.

2) The software handles string operations, sorting sequencing operations as per the language and Character selected.

3) The software display is consistent with characters which are non-ASCII in GUI and menus.

4) The software messages are handled properly.

A Checklist for the ISBN validation includes.

1. Functionality in all languages and locales are the same.

2. Sorting/sequencing the items to be as per the convention of languages and locale.

3. The input to the software can be in non ASCII or special characters.

4. The cut/copy and paste of non-ASCII characters retain their style after pasting.

Fake language Testing:

* It helps to software translators to catch the translation and localization issues.

* It helps to identifying the issues proactively before the product is Localized.

* It helps to English testers to find the defects that many otherwise be found only by language experts during localization testing.

* Fake language testing checklists contains,

1. Ensure software functionality is tested for single byte or double byte.

2. Ensure all strings are displayed properly on the screen.

3. Ensure the screen width, size, Pop-ups and dialog boxes etc.

Language Testing:-

* It is also called "Language Compatibility Testing".

* It ensure that the functionality of the software is not broken on other languages settings and its still compatible.

* Checklists must contains.

1. Check the functionality on English, one non English and are double byte language platform combination.
2. Check the Performance of key functionality on different language platforms.

Localization Testing:

* It performed to verify the quality of a software's localization for a particular target culture/locale and is executed only on the localized version of the Product.

* It helps to check for linguistic errors and resources attributes.

* To find typographical errors.

Tools used for I18N:

Microsoft OS- 1. IME 2. MS localization studio

Linux OS- GNU gettext, 2. Pylatin 3 Unicode IME

AD Hoc Testing:-

Definition:

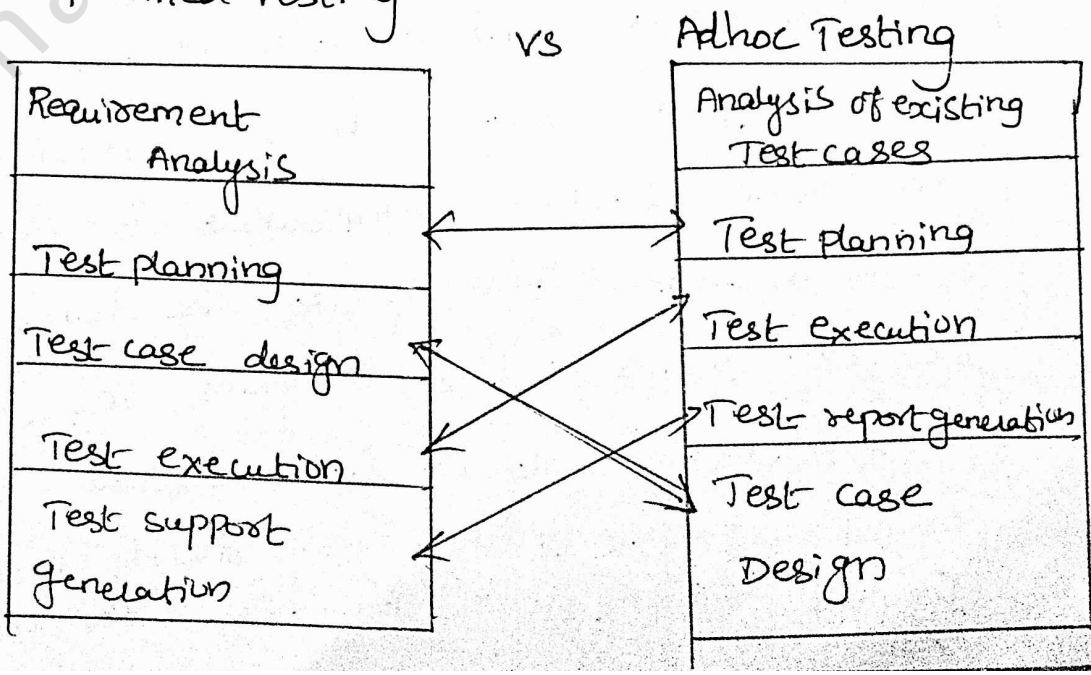
* It Performed without planning and documentation The tester tries to "break" the system by randomly trying the systems functionality.

* It is generally done to uncover defects that are not covered by planned testing.

Ad hoc testing can be planned in one of two ways.

1) After a certain number of planned test cases are executed.

2) Prior to planned testing.



When to execute Ad hoc testing?

* It can be performed when there is limited time to do exhaustive testing.

* Its done by after the formal test execution.

* It will be effective only if the tester has in depth understanding about the system under test

Forms of Ad hoc testing:

1. Buddy Testing 2. Pair testing 3. Exploratory testing 4. Iterative testing 5. Agile/Extreme testing 6. Monkey testing 7. Defect seeds

1. Buddy Testing:

* A developer and tester working as buddies to help each other on testing and in understanding the specification is called "Buddy Testing".

* A developer and a tester usually becomes buddies.

* Buddying people with good working relationships yet having diverse backgrounds is a kind of a safety measure that improves the chance of detecting errors in the program very clearly.

* Buddies should not feel mutually threatened or get a feeling of insecurity during buddy testing. They are trained on the philosophy & objective of buddy training.

* Buddy testing uses both white-box and black box testing approaches.

* The buddy, after testing, generates specific review comments and points out specific defects.

* The buddy may suggests ideas to fix the code when pointing out an error in the work product.

* The developer reviews the comments and if the buddies agree, the appropriate changes are implemented or else both of them discuss the comments and come to a conclusion.

* It is also called "Quick testing"

2. Pair testing:

* Two testers are assigned the same modules and they share ideas and work on the same systems to find defects

* one tester executes the tests while another tester records the notes on their findings.

* Its done by two testers working simultaneously on the same machine to find defects in the product

* Its done in any phase in testing

Situations when pair testing becomes ineffective.

1. Individual high performers may lead to problems.
2. pairing up juniors/experienced members may result in the former doing tasks that the senior may not want to do.

3. Exploratory testing:

* Another technique to find defects in ad hoc testing is to keep exploring the product, covering more depth and breadth.
* The coverage of all activities in level by level during testing is called "Exploratory testing".
* Due to lack of knowledge, test engineers follows Exploratory testing.
* Its done in any phase of testing.
* Testers may execute tests based on their Past experience, similar product, similar domain or product in a technology area.
* It can be used to test software that is untested unknown or unstable.
* It not only testing functionality of system and also perform different environments, configurations, parameters test data

Exploratory Test techniques:

1. Guesses
2. Architecture diagrams, use cases
3. Past defects
4. Error handling
5. Discussions
6. Questions & checklists

4. Iterative testing:

* The iterative model is where the requirements keeps coming and the product is developed iteratively for each environment. The testing associated for this process is called "Iterative testing".

* It requires repetitive testing.
* Majority of these tests are executed manually.
* Iterative testing aims at testing the product for all requirements, irrespective of the phase they belong to in the spiral model.
* Each iterative unit test cases are added, edited or deleted to keep up with the revised requirement for the current phase.

5. Agile / Extreme testing:-

* Agile / Extreme models take the processes to the extreme to ensure that customer requirements are met in a timely manner.

* A software testing practice that follows the principles of the agile manifesto, emphasizing testing from the perspective of customers who will utilize the system.

* Its done by QA team members.

* A typical Agile and extreme project day starts with a meeting called "Stand up meeting".

* At the start of each day, the team meets to decide on the plan of action for the day.

* During the meeting, the team up any clarification or concerns, They are discussed & resolved.

1. Develop and understand User story
2. Prepare acceptance testing
3. Test plan and estimation
4. code
5. Test
6. Refractor
7. Automate
8. Accepted and delivered.

6. Monkey Testing:

* It is also called "Chimpanjee testing"

* It cover/check the main activities of the build during testing

7 Defect seeding:

* Error Seeding is also known as "Debugging" It acts as a reliability measure for the release of the product.

* Usually one group members injects the defects and another group tests to remove them.

* The purpose of this exercise is while finding the known seeded defers, the unseeded defects may also be uncovered. Defects that are seeded are similar to read defects.

* Defects that can be seeded may vary from severe or critical defects to cosmetic errors.

* It act as guide to check the efficiency of the inspection or testing process.

* when a group knows that there are seeded defects in the system, it acts as a challenge for them to find as many of them as possible. it adds new energy into their testing.

* In case of manual testing, defects before the start of the testing process.

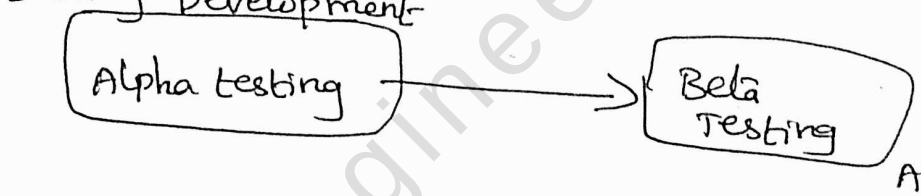
* when the tests are automated, defects can be seeded any time.

Alpha - Beta Test:

* If the Software has been developed for the mass market, then testing it for individual clients / users is not practical or even possible in most cases.

* This type of software undergoes two stages of acceptance test.

1) Alpha test 2) Beta test



Alpha Testing	Beta Testing
1) It always performed by the developers at the software developers site.	1) It always performed by the customer at their own site
2) It is done by testing team	2) It is done by testing team and customers
3) It is not open to the market and public	3) It is always open to the market and the public
4) It comes with white box and black box testing	4) It is comes only the black box testing.

Testing OO Systems:

Basic concepts of OO system that are relevant for testing are

1. Classes - A class is a representation of a real-time object
2. Objects - Objects are the dynamic instantiation of a class.
3. Constructor - Specific instantiation are done using Constructor.
4. Encapsulation - The implementation of the methods is hidden from the use. This enables the person writing the methods to optimize the implementation without changing the external behavior. This is called Encapsulation.
5. Inheritance - Defining a new classes from already existing classes.
6. Polymorphism - Same method name but performing different functions.

Object oriented Testing:

* Testing an object oriented system should tightly integrate data and algorithm.

Testing on system has

1. Unit testing a class
2. Putting classes to work together (integration testing of a classes)
3. System Testing
4. Regressing Testing
5. Tools for testing object oriented systems.

1. Unit testing a Class:

* Classes are the building blocks for an entire OO system.

Unit test the classes for the following reasons.

1) A class is intended for heavy reuse
2) Any defects get introduced at the time of class defined
3) A class may have different features.
4) A class is a combination of data and methods.

The Alpha-omega method:

* It takes the object under test from the alpha State to the omega state by sending the message to every method at least once.

* It shows that all methods in a class are minimally operable. It support Extensive testing.

* An alpha-omega method has six basic steps

1. New or Constructor method
2. Accessor (get) method
3. Boolean (Predicate) method
4. Modifier (set) method
5. Iterator method
6. Delete (Destructor) method

2. Putting Classes to work together (Integration testing of a class)

* It focuses on groups of classes that collaborate of communicate in some manner.

* Integration of operations once at a time into classes is often meaningless.

* Regression testing is important as each thread, cluster, subsystem is added to the system.

* Integration applied 3 different incremental Strategies.

1. Thread based testing - Integrates classes required to respond to one input or event.

2. Use based testing - Integrates classes required by one use case.

3. Cluster testing - Integrates classes required to demonstrate one collaboration.

3. System Testing and Interoperability testing:

* Different classes may be combined together by a client and this combination may load to new defects.

* The complexity of interactions among classes can be substantial.

* It is important to ensure that proper Unit & Component testing is done before attempting system Testing.

* proper entry/exit conditions should be set for the system Testing.

Design error handling paths that test external information.

* Conduct a series of tests that simulate bad data
* Record the results of tests to use as evidence.

Types: Recovery, stress, load, performance etc.

4. Regression Testing of OO Systems:

* Changes to any one component could have potentially unintended side effects on the clients that use the components
* Frequent integration and regression runs become very essential for testing OO systems.
* Because of the cascaded effects of changes resulting from properties like inheritance, it make sense to catch the defects as early as possible.

Tools for testing of OO system:

There are several tools that aid in testing OO systems.

1) Use cases
2) Class diagrams
3) Sequence diagrams
4) Activity diagrams
5) State diagrams.

1) Use cases:

* It represents the various tasks that a user will perform when interacting with the system.

* Uses cases go into the details of the specific steps that the user will go through in accomplishing each task and the system responses for each steps

2) Class diagrams:

* It represent the different entities and the relationship that exists among the entities

* class diagrams elements are

i) Boxes - Classes

ii) Association - Relationship between two classes by a line

iii) Generalization - Child class derived from parent class

A class diagram is useful for testing in following ways

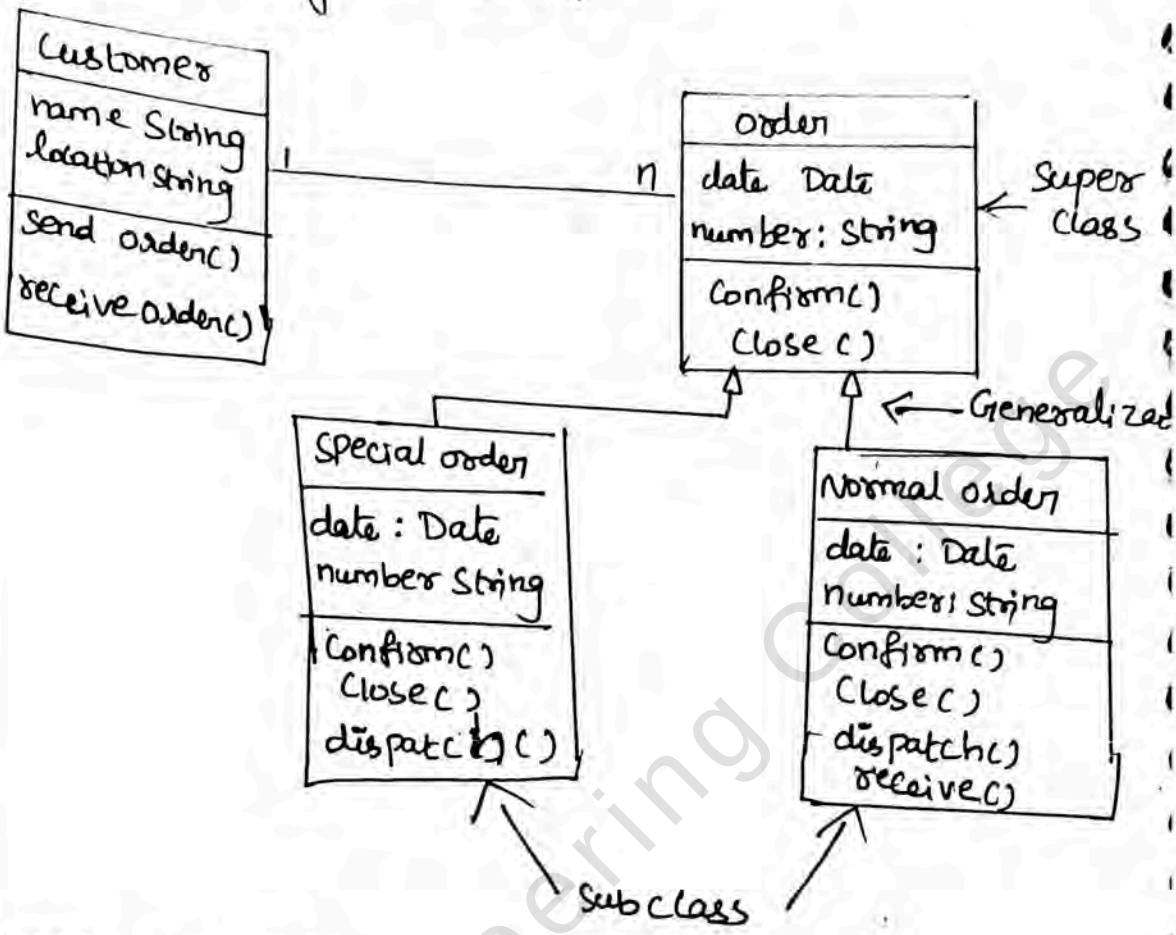
1. It identifies the element of a class

Eg ECP, BVA etc

2. The associations help in identifying tests for referential integrity constraints across classes

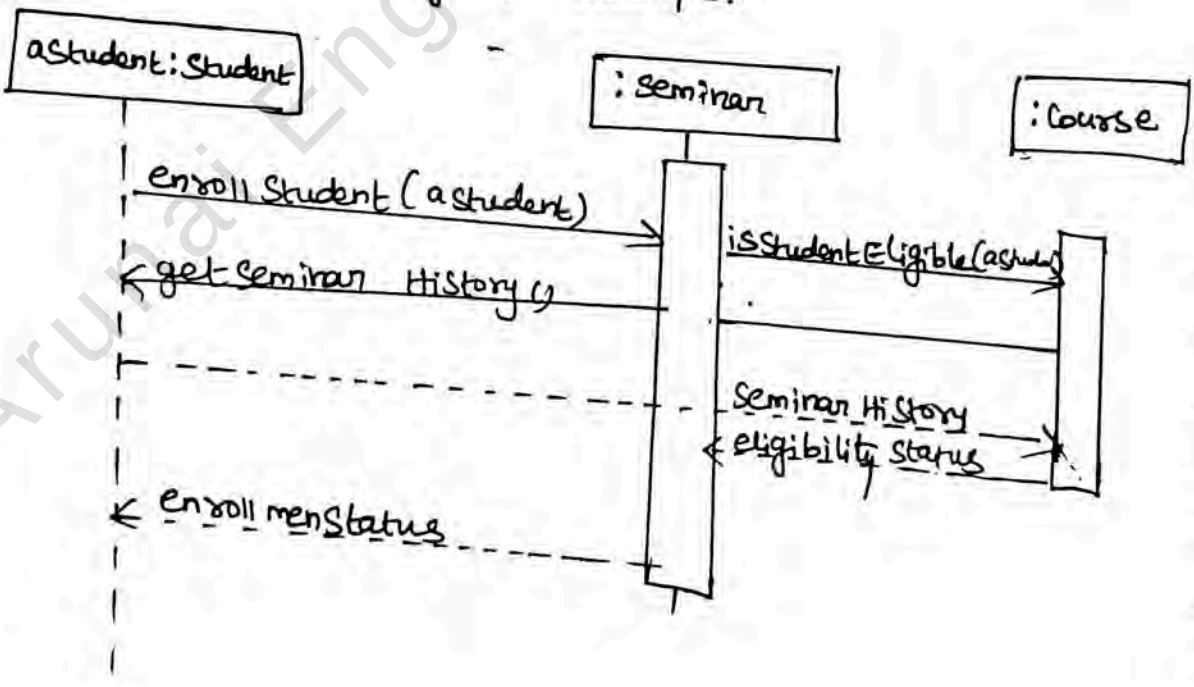
3. Generalizations helps in identifying class hierarchies

Class diagram - Example



3. Sequence Diagrams:

Sequence Diagram Example:



* Sequence diagrams represents a sequence of messages passed among to accomplish in a given application scenario or use case.

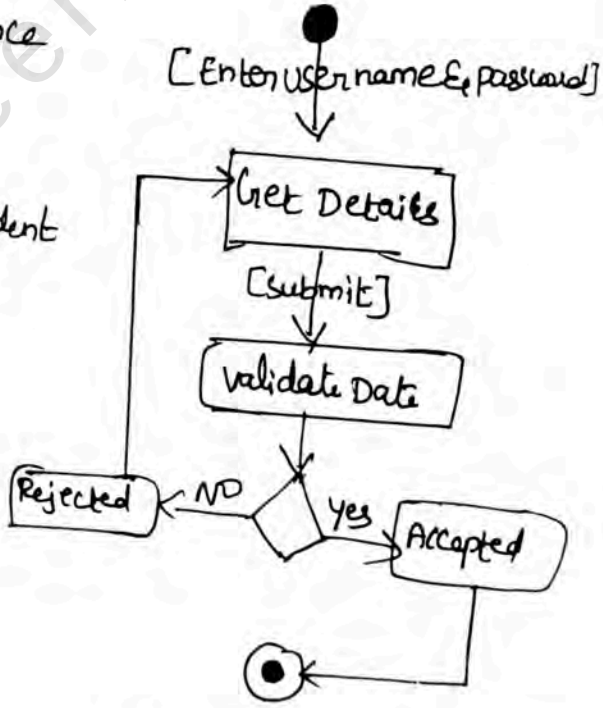
* A sequence diagram helps in testing by

1. Identifying temporal end-to-end messages
2. Tracing the intermediate points in end-to-end transaction.
3. providing for several typical message calling
   Eg: Blocking Call, Non blocking call
4. Activity Diagram:

* It represents the sequence of activities.

* Its simply represent independent Program Paths whenever code complexity is arrive.

* To identify the possible message flows between Object and classes.

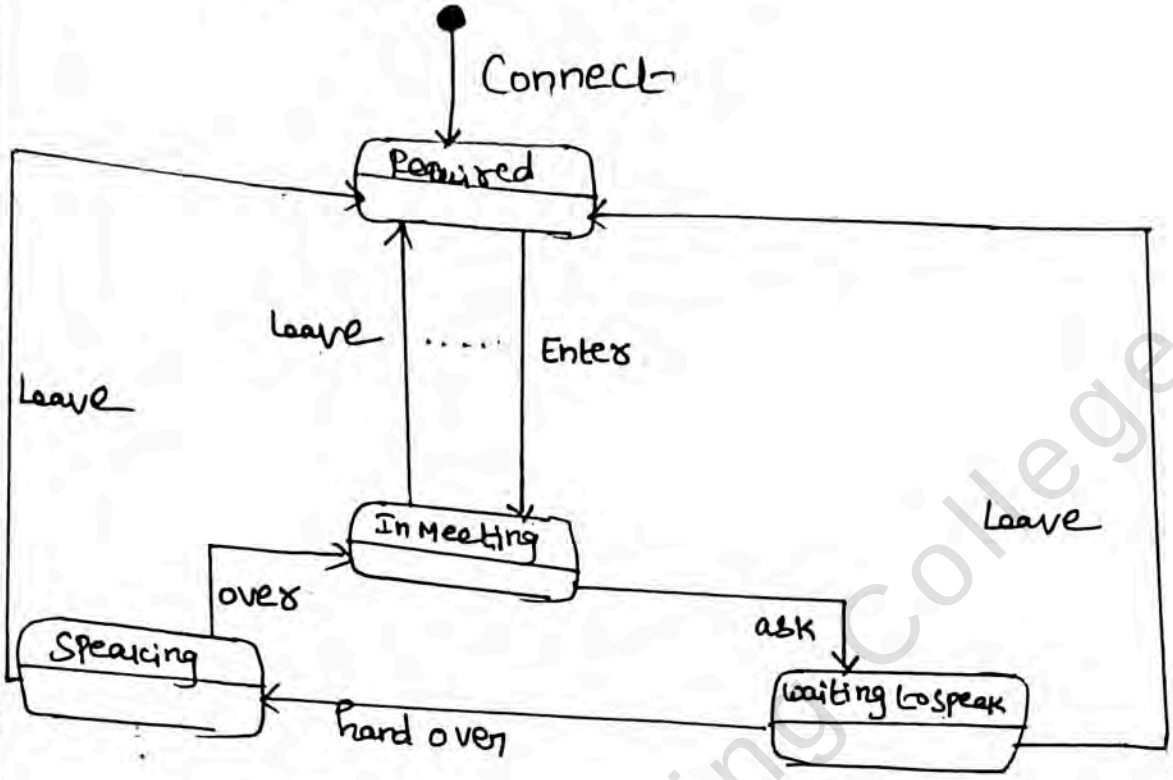


Activity diagram-Example

5. State Diagram:

* when an object can be modeled as a state machine, then the techniques of state based testing can be directly applied.

State diagram - Example



Configuration testing:

* It is also called "Hardware compatibility Testing".
* During this test, test engineer validates that whether our application build run an different technology hardware devices or not?
* It is the process of checking the operation of the softwares, you are testing with all these various types of hardware.

Eg Different technology printers, Different LAN topologies etc.

* The different configuration possibilities are

1. The PC - well known Computer manufacturers, such as DELL, HP, etc
2. Components - Various system boards, components, device drivers (CD-ROM, HDD, FDD) video, sound, fax modem
3. Peripherals - printers, scanners, keyboard
4. Interfaces - RJ-11, RJ-45, Fire wire
5. Options and memory - Various amount of memory
6. Device Drivers - It is a low level software

Isolating Configuration bugs:

* It is usually dynamic while box testing and Programmer debugging effort.

* A configuration problem can occur for several reasons all requiring someone to carefully Examine the Code, while running the software under different configurations to find the by

1. Your software may have a bug that appears under a broad Class of configurations.

Eg Greeting card program work fine with laser Printers but not with inkjet printers.

2. The software may have a bug specific only to one Particular configuration.

3. The hardware device or its device drivers may have a bug that only one the uses a Unique display setting.

Eg software is run with a specific video card, the PC Crashes

4. The hardware device or its device drivers may have a bug that can be seen with lots of other software

Sizing up the job:

* There are huge number of display cards, bund cards, modem available in network.

* These combinations are not possible to test.

These sizing up problem is solved by

1. Equivalence Partitions

2. Boundary Value analysis.

Approaching the tasks:

* When planning the configuration testing to keep the following steps carefully.

1. Decide the types of hardware will need

2. Decide what hardware brands, models and device drivers are available

3. Decide which hardware features, modes & options are possible.

4. Identify software unique features that works with the hardware configuration.

Obtaining the hardware:

* It required dozens of hardware set up for configuating testing. It is a expensive one.

Few ideas for overcoming this problem:

1. A great plan every tester on the team to have different hardware. It helps different configuration set up in user concern.

2. Create and maintain good relationship between hardware manufacturers. It help to solve bugs easily.

3. Collect all the required hardware in our team and then, Purchase all the cheaper hardware.

4. If the user have budget, work with your project manager to contract out our test work to a professional configuration and compatibility test lab.

Identifying hardware Standards:

* Hardware specifications, standards to be tested by relevant concern illustration.

↳ These information collected from relevant concern websites.

Eg Apple hardware - developer - apple.com/hardware

windows logo/Software - msdn.Microsoft.com/software

Configuration testing other hardware:

1. Create equivalence partitions of the hardware based on input from the people who work with the equivalent, user project manager or user sales people.

2. Develop test cases and collect the selected hardware and run the tests

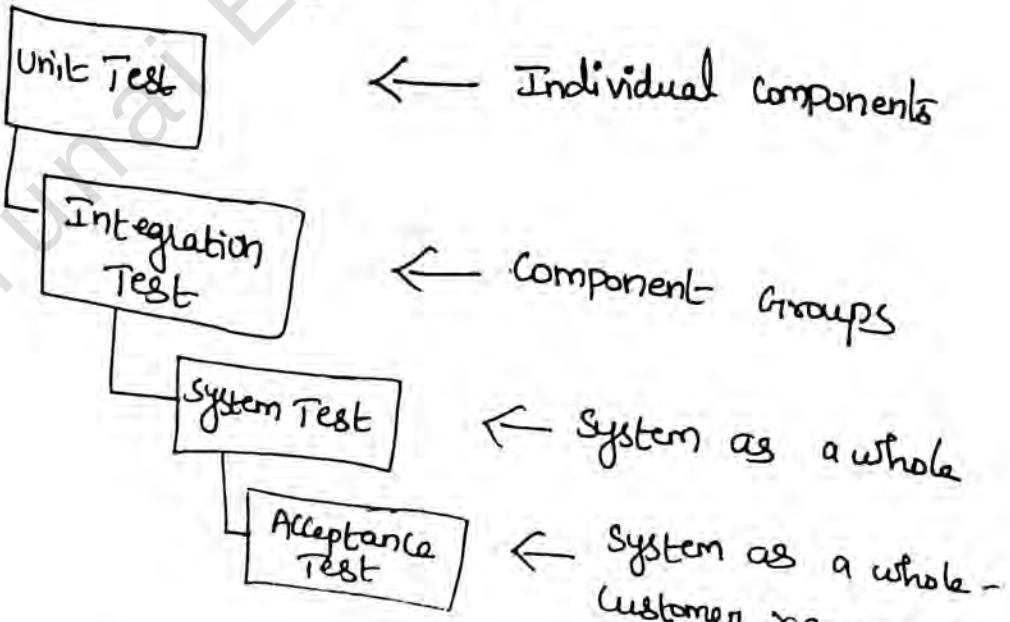
3. Follow configuration testing approaches.

the levels of testing

* It is defined by a given environment. Environment is a collection of People, hardware, software, interfaces,

Need for levels of testing:

* It helps to enhance the quality of software Testing.
* To produce a more Unified testing methodology applicable across several projects.
* Testing process to be abstracted easily.
* To identify missing Areas.
* To prevent overlap and repetition between the development life cycle phases.



← Individual components
← Component Groups
← System as a whole
← System as a whole - Customer requirements

Levels of Testing

The levels of Testing are

1. Unit Testing 2. Integration Testing
3. System Testing 4. Acceptance Testing

1. Unit Testing:

* A Unit test is smallest testable piece of Software. A principle goal is to detect functional and structural defects in the unit.
* It can be compiled, linked & loaded.

Eg: Functions/procedures, Classes, Interfaces.

* Its done by programmers.
* Individual components are to be tested, and assemble together.
* It is a better use a Buddy Testing

Types:

1. Execution Testing
   a) Statement Coverage b) Branch Coverage
   c) Path Coverage.
2. operations Testing
3. Mutation Testing.

Buddy Testing:

* Team approach to coding and testing
* one programmer codes the other tests and vice versa

2 Integration Testing:

* At the integration level several components are tested as a group and the tester investigates component interactions.

* Test for correct interaction between system units.

* Defects are find between integrated modules

* Each modules are coded by different people.

System/modules built by Merging existing libraries.

* It helps to tests the interfaces among units.

* Integration testing is done by developer / tester

* It is done on programmers work branch.

* Integration testing is done when test cases written when detailed specification is ready.

* It helps to improve continuous throughput of system.

* It discovers inconsistencies in the combination of units.

Types:

1. TOP down approach
   ↳ Use of stubs
2. Bottom up approach
   ↳ Use of Drivers
3. Hybrid Approach

3. System Testing:

* At the system level the system as a whole is tested and a principle goal is to evaluate attributes such as usability, reliability and performance.

* The major testing levels for both object-oriented and procedural-based types of system are similar.

* The nature of the code that results from each developmental approach demands different testing strategies.

Eg To identify the individual components and to assemble them into subsystems.

* System test begins when all of the components have been integrated successfully. It usually requires the bulk of testing resources.

* Eg Lab hardware, special hardware, special software,

* At the system level the tester looks for defects, but the focus is on evaluating Performance, Usability, reliability and other quality-related requirements.

* System Testing is done by the testing team.

Types:

1. Functionality Testing
2. Recoverability Testing
3. Interoperability Testing
4. Performance Testing.
5. Scalability Testing
6. Reliability Testing.
7. Regression Testing
8. Security Testing.

Testing approaches are used two major types of Programming language.

1) procedure-oriented Programming language.
2) object-oriented programming language.

The written code needs testers to use the different strategies to

i) TO find the test components
ii) TO find component groups

System development with procedural languages are represented as a compassed things of,

* passive data * ACTIVE procedures.

* When test case are developed then it must to Generate the input data to pass to the procedures inorder to reveal defects.

* The object oriented systems are taken into a composition of active data along with allowed operations on that data.

*

Procedural System:

* In a procedural system, lower level of abstraction is described as a function or a procedure.

* The Higher level of abstraction is described by group of producers (or) functions.

* Both level to be combined and finally produces the system as whole, which is the highest level of abstraction.

4. Acceptance Testing.

* During acceptance Testing the development Organization must show that the software meets all of the client's requirements.

* A successful acceptance test provides a good opportunity for developers to request recommendation letters from the client.

* Software developed for the mass market often goes through a series of tests called alpha and beta tests.

Alpha tests - Alpha tests bring potential users to the developer's site to use the software. Developers note any problems

Beta tests - Beta tests send the software out to Potential users who use it under real-world conditions and report defects to the developing organization.

Levels of Testing and Software Development Paradigms.

* When an approach is used to design and develop a software system, it is based on

1. Tester's plan 2. Design of tests

Two approaches to system development are

1. Bottom up approach 2. Top-down approach.

2. Object oriented system:

* In a object objected system, lower level of abstraction is described as the method or member function.

* Next highest level is described by the class that encapsulates data and model.

* Next highest level is cluster. It helps to combine more than classes.

* Finally, the system is described clusters and auxiliary code

* Testing is straight forward one,

Eg Encapsulation - Hide details from testers.

* Object oriented code is characterized by
  - ↳ use of messages
  - ↳ Dynamic binding state changes etc.

Usability and Accessibility Testing:-

Usability Testing:-

Definition:

* It is a technique used in user centered interaction design to evaluate a product by testing it on users.

* Testing team follow two testing techniques, They are
1. UI Testing 2. Manual support Testing.

* Usability testing attempts to characterize the "look and feel" and usage aspects of a product, from the point of view of users.

* Some of characteristics of usability testing are. 1) Ease of use 2) Speed 3) pleasantness and aesthetics.

* Usability testing addresses these aspects from the point of view of a user.

Approach to Usability:

* When doing usability testing, certain human factors can be represented in a quantifiable way and can be tested objectively.

* The number of mouse clicks, number of sub-menus of navigate, number of keystrokes, number of commands to perform a task can all be measured and checked as a part of usability testing.

* The People suited to perform usability testing are

1) Representatives of the actual user Segments

2) people who are new to the product, so that they can start without any bias and be able to identify usability problems.

* Another aspect of usability is with respect to messages that a system gives to its users.

* Messages are classified into three types

1) Informal message

2) Warning message

3) Error message

1) Informal message:

* Informal message is verified to find out whether an end-user can understand that message and associate it with the operation done and the context

2) Warning Message:

* Warning message is checked for why it happened and what to do to avoid the warning

3) Error message:

* Whenever there is an error message, three things are looked for what is the error, why that error happened, and what to do to avoid or work around that error.

* Usability should also consider command Line Interface (CLI) and other interfaces that are used by the users.

When to do usability testing?

* Usability testing done in 2 phases They are

1. Design validation Phase
2. Integration testing phase.

* Usability testing requirements are collected/ Planned from beginning of the test planning.

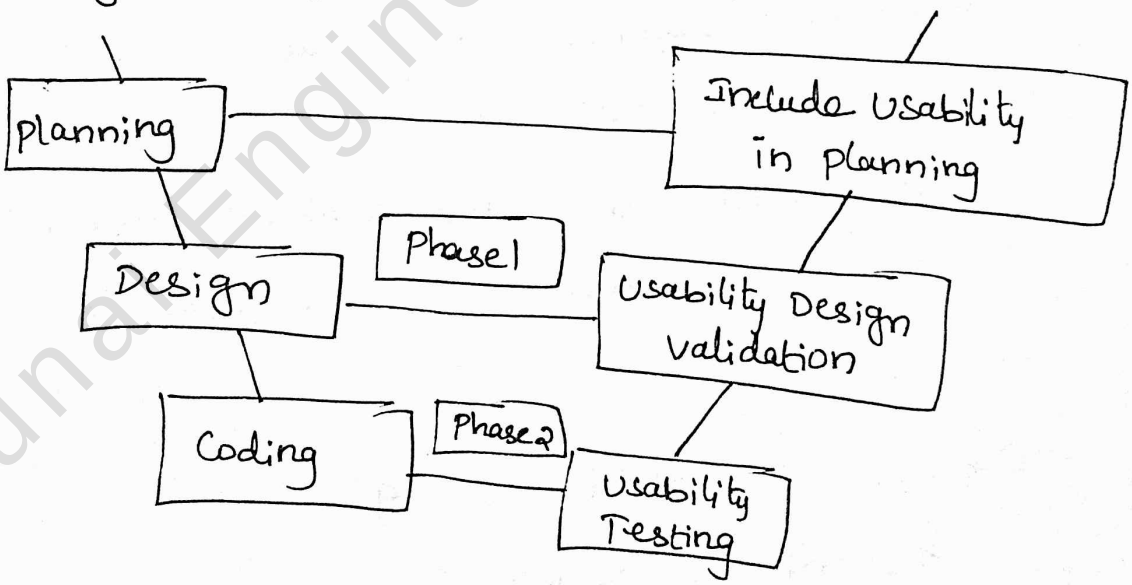
* Usability defects is a higher priority compare to other defects.

* In the first phase of usability testing design in validated.

* In the second phase, tests are run to test the Product for usability

* Usability design is verified (first phase) through several forms, They are

1. Style Sheets 2. Screen layouts 3. paper Design 4. layout Design



Phases and Activities of Usability Testing

Steps for development and testing of client applications:

1. Design for functionality
2. Perform coding for functionality
3. Design for UI
4. Perform coding for UI
5. Integrate UI with functionality
6. Test the User interface (UI) with functionality (Phase 1 and Phase 2)

Steps for development and testing of web applications

1. Design the UI
2. Perform coding for UI
3. Test UI (Phase 1)
4. Design for functionality
5. Perform coding for functionality
6. Integrate UI with functionality
7. Test UI along with functionality (Phase 2)

How to achieve usability?

* Customers to give feedback on all the UI requirements upfront or during the design phase is essential.

* Different categories of users are

1. Experts 2. Beginners 3. Novice users.

Usability is achieved by

1. Recording of Sequence, Screens, user reactions frequent.
2. Observation of product
3. Closely monitored the activities
4. Recorded the defects
5. Report the problem.

Quality factors for usability:

Some of the Quality factors for usability are

1. Comprehensibility
2. Consistency
3. Navigation
4. Responsiveness

Aesthetics testing:

* It helps to improve usability testing
* It ensure the product is beautiful.
* Beauty sets the first impression of any product
* Its not in the external look done, It help to test messages, screens, colors and images.
* A Pleasant look for menus, colors, nice icons
* It must be done in design phase.
* Its Performed by everyone who appreciates beauty that means every one.

Accessibility Testing:

* There are a large number of people who are challenged with vision, hearing and mobility related problems- partial or complete.

* product usability that does not look into their requirements would result in lack of acceptance.

* Accessibility tools are available to help them with alternatives.

* Accessibility testing involves testing these alternative methods of using the product and testing the product along with accessibility tools.

* Accessibility is a subset of usability and should be included as part of usability test-planning.

* Accessibility to the product can be provided by two means.

1) Basic accessibility 2) product accessibility

1) Basic accessibility:

* Basic accessibility is provided by the hardware and operating system.

* All the input and output devices of the computer

and their accessibility options are categorized under basic accessibility

a) keyboard accessibility b) screen accessibility.

a) keyboard Accessibility:

* It support vision impaired users to get a feel and align their fingers for typing

* keyboard improvements helps to vision impaired users for understanding key size, shortcut and functions etc.

* The operating system vendors came up with some more improvements in the keyboard.

* some of the improvements are

↳ Sticky keys :- one of the most complex sequences for vision-impaired and mobility impaired users is <CTRL><ALT><DEL>

* This keyboard sequence is used for various purpose such as log in, log out, locking and unlocking machines, shutdown and brining up task manager.

↳ Toggle keys:

* when toggle keys are enabled the information typed may be different from what the user desires

Eg INSERT key, NUM Lock key - vision-impaired users find it difficult to know the status of the toggle keys.

- TO solve this problem sound is enabled and the different tones are played when enable/disabling the toggle keys

↳ Arrow keys:

* Mobility-impaired users have problems moving the mouse so keyboard arrow button is necessary for such users.

* Two buttons of the mouse and their operations too can be directed from the keyboard.

↳ Filter keys.

* When keys are pressed for more than a particular duration, they are assumed to be repeated.

* It helps (Filter keys) in either stopping the repetition completed or slowing down the repetition.

↳ Sound keys:

* To help vision-impaired users, there is one more mechanism that pronounces each character as and when they are hit on the keyboard.

↳ Narrator:

* Narrator is a utility which provides auditory feed back.

Eg: Read out the characters typed
Notify the system events by different sounds.

b) Screen Accessibility:

* Hearing-impaired users require extra visual feedback on the screen.

* Some accessibility features that enhance usability using the screen are as follows.

1. Visual Sound - It is a "waveform" or "graph form" of the sound.
2. Enabling captions for multimedia - speech it can be converted into equivalent texts
3. Soft key board :- To displaying the keyboard on the screen.
4. Easy reading high contrast.

High contrast mode helps to users for
1. uses pleasing colors and
2. Font size for all the menus in the screen

2) Product Accessibility:

* A Product Accessibility do everything possible to ensure that the basic accessibility features are utilized by it.

Eg Text equivalent of multimedia files.

* product accessibility provides accessibility in the Product through standards and guidelines.

Accessibility sample requirements are

1. Text equivalents have to be provided for audio, video and picture images.

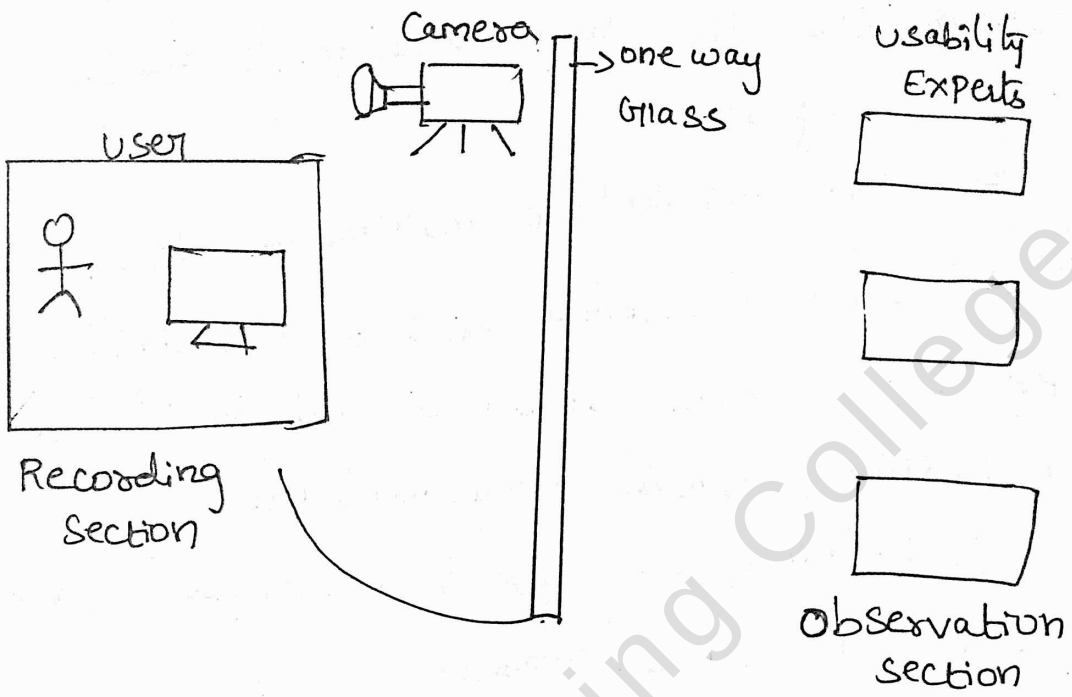
2. UI should be designed, so that all information conveyed with color is also available without color.

Tools for usability:

Name of the tool	Purpose
1. JAWS	- For testing accessibility of the product
2. HTML validate	- To validate the HTML source file and Standards
3. Style sheet validator	- To validate style sheet standards set by W3C

Usability lab setup:

Key elements of a Usability Lab



* Usability lab contains 2 sections, They are,
  1. Recording section
  2. observation section

Test Roles for Usability:

* Various methods adopted by companies for usability testing are as follows.

1. Performing usability testing as a separate cycle of testing.
2. Hiring external consultants to do usability validation.
3. Setting up a separate group for usability to institutionalize the practices across various product development teams and to set-up organization-wide standards for usability.

TESTING THE DOCUMENTATION

Documentation testing is a non-functional type of software testing.

It is a type of non-functional testing.

- Any written or pictorial information describing, defining, specifying, reporting, or certifying activities, requirements, procedures, or results'. Documentation is as important to a product's success as the product itself. If the documentation is poor, non-existent, or wrong, it reflects on the quality of the product and the vendor.
- As per the IEEE Documentation describing plans for, or results of, the testing of a system or component, Types include test case specification, test incident report, test log, test plan, test procedure, test report. Hence the testing of all the above mentioned documents is known as documentation testing.
- This is one of the most cost effective approaches to testing. If the documentation is not right: there will be major and costly problems. The documentation can be tested in a number of different ways to many different degrees of complexity. These range from running the documents through a spelling and grammar checking device, to manually reviewing the documentation to remove any ambiguity or inconsistency.
- Documentation testing can start at the very beginning of the software process and hence save large amounts of money, since the earlier a defect is found the less it will cost to be fixed.

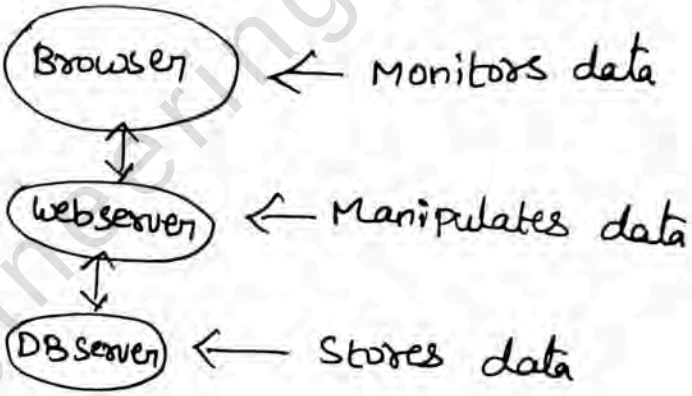
Website testing:

Before performing a testing on developed/web sites or web applications, it is necessary to determine the following 3 things.

1. whether web based risk should be included in the test plan.
2. which types of web based testing is used?
3. selecting the appropriate web based test tools for the test execution phase.

*web sites testing done for 3 tier applications

web page/websites



Objective:

* The objective of this test program is to assess the adequacy of the web components of the software application.

Concerns:

1. Browser compatibility 2. Functional correctness 3. Integration 4 usability 5. security 6. performance 7 Verification of code.

Web Page Fundamentals

Internet web pages contains

1. Text of different sizes, 2. Fonts and colors
3. Graphics and photos 4. Hyperlinked text, images, graphics
5. Rotating ads 6. Drop down selection boxes
7. Text fields, buttons etc

Website creation is also complex, because.

1. Dynamic changes of text 2. Dynamic layout
3. Customizable layouts 4. Customizable contents

Different testing types to performed for website testing:

1. Black Box Testing (BBT)
   * Its functional testing. Its used to test
   1. Text, 2. Hyperlinks 3. Forms 4. Graphics
   Eg HTML validation, Link checking, Functions Testing
2. Gray Box Testing (GBT)
   * It is a mixture of white box testing (WBT) and black box testing (BBT)
   * It help to test, HTML code (ie HTML validation)
4. White Box Testing (WBT)
   * Its used for testing the internal Part of the Source code
   * It must require Coding knowledge
   * MUST hared knowledge about
   1. Dynamic content 2. Database driven web pages
   3. programmatically Created web pages 4. Security.

4. Configuration and compatibility Testing:

* For web site checking you must keep or follow possible configuration hardware and software

1. Hardware Platform 2. Browser plug in 3. Text size
4 Modem speeds 5. Browser options

5 Usability Testing:

* It gives the better appearance, look and feel for websites.

TOP mistakes in web design.

1. Gratuitous use of bleeding edge technology
2. Scrolling text, marquees and constantly running animations.
3. Long scrolling pages
4. Non-Standard link colors
5. out dated information

Automation tools for website Testing:

Performance tools - Stress tested, load runner

Java testing tools - java NCSS, J cover, J Unit

Link checking tools - link tiger, Link Scan, link Storm

Regressing checking tools - webcat, Cubic test

IT8076-SOFTWARE TESTING

UNIT-4
TEST MANAGEMENT

People and organization Issues in Testing:

Perceptions and misconceptions about Testing

Perception-1

"Testing is not technically challenging"

* Testing is simple and repetitive job
* It does not require specilized Skills
* It is a manual Task
* If product is simple, easy to test
* "There is testing in all development and development in all testing."

Misconception:

Testing needs following things

1. It needs a holistic understanding of the entire project
2. Test engineer must be a "Domain expert" (or) "Domain Specialists"
3. Testing requires thorough understanding of multiple domains.
4. Testers must specilization in programming and test ~~Script languages~~.

5. For better design and integration tester/ developer must understand the usage of tools.
6. opportunities for conceptualization and out-of-the box thinking.
7. Significant investments are made in testing today sometimes a lot more than in development

Perception - 2

"Testing does not provide Me a Career path or Growth"

* Career path & rules defined for the each Organization People differently.

* It helps to improve development, Standard, and different software engineering functions.

* "Testing is not a devil and development is not an angle", "Opportunities abound equally in testing & development".

Misconceptions:

1. No much Career path opportunities in testing field.

2. Define separate rules for development engineer, senior development engineer, Business Analyst, domain expert.

Perception - 3

"I am put in Testing - what is wrong with me?"

* If a person is not suitable for development, for the same or similar reason he or she may not be suitable for testing either
* People are sometimes made to feel that they are in testing because they could not fit in anywhere else

Consider some of the causes and effects of such messages.

1) Filling up positions for testing should not be treated as a second string function.
2) If a person having capability in the field of testing then the person is appointed for testing.
3) Then compensation schemes should not discreminate against any specific function.
4) Appropriate recognition should be given to the engineer who participate in activities such as testing, maintenance and documentation

Perception - 4

"These folks are my adversaries"

* Testing & Development teams should reinforce each other and not be at logger heads.
* The main function of testing is to find errors in the ~~software. It is easy to the adversary~~

attitude to develop between testing & development team

Perception-5

"Testing is what I can do in the end if I get time"

* Testing is not what happens in the end of the project - it happens through out and continues even beyond the release.

* Testing is a planned one. It is not constructed end of the cycle.

* Adequate time, resources, software, knowledge are essential for testing.

* otherwise it creates risk (ie Loss)

These problems (risks) are to be avoided in testing are,

1. Stipulate completion/acceptance criteria for testing team to accept a products from the development team.
2. Giving the testing team to work freely to mandate a minimum quality in the product before it can be released.

Perception-6

"There is no sense of ownership in Testing"

* Testing has deliverables just as development has and hence testers should have the same sense of ownership

* The ownership is sometimes not created for testing functions.

* A possible contributor to this feeling of lack of ownership is the apparent lack of "deliverables" for testing functions.

Perception-7:

"Testing is only Destructive"

* Testing is destructive as much it is constructive, like the two sides of a coin.

* Tester must perform/destructive

1. The part which does not work
2. what work's going on the product
3. Express the work in the product
4. Analyze the risk of the product before release.
5. Giving the mitigation plan for perfect perspective.
6. Save company money & image.

* Say the ways to solve the problem.

Challenges and Issues in Testing Services organizations:

* All testing organizations face certain common challenges.

* In the case of a testing services organization to which testing is outsourced, some of these challenges are exacerbated, primarily because of the arm's length distance from the development team. Main challenges are

1. The outsider effect and estimation of resources
2. Domain expertise
3. Privacy and customer isdation issues
4. Apportioning hardware and software resources and costs
5. Maintaining a "Bench"

1. The outsider effect and estimation of resources

* The testing services organization is an "outsider" to the development organization

Some of the implication are as follows:

1) They are not holding the product internal or code

2) It is not possible to know the product history for them

3) They are not maintaining the same level of rapport with the development teams

4) Internal development and testing teams do not necessarily to have the information about the software and hardware resources required.

2. Domain Expertise:

* A testing team in a product organization can develop domain expertise in a specific domain.

Eg:- Domain

1. ERP domain
2. Logistics domain
3. Banking domain etc

* Domain Expertise extract the information from customers. It is hard to retrieve the information.

* Software organization can develop specialized expertise in the ERP domain.

* The organization can hire a few domain specialists who can augment the testing team.

* Such specialists will find it interesting to join product organization because they can find a natural transition from their domain to the more attractive IT arena

3) Privacy and customer isolation issues:

* A Testing Services organization will have to work with multiple customers.

* AS an organization works with customers in a given domain (Eg pharmaceutical or financial services) it develops not only general expertise but also domain expertise in a particular domain

* Two factors contribute to this being a major challenge.

1. The testing service organization has a common infrastructure and hence physical isolation of the different teams may be difficult

2. The customers can go to one project from other project.

* At that time it must to confirm that specific knowledge required in one project cannot be taken to other project

4) Apportioning hardware & software resources and costs

* when a testing service organization bids for and works with multiple customers, it would have to use internal hardware and software resources.

* Some of these resources can be identified directly and specifically for a particular account.

* There are other resources that get multiplexed across multiple projects.

Eg Communication infrastructure such as satellite links, common infrastructure such as email servers and physical infrastructure costs.

5) Maintaining a "Bench"

* TO keep a" People on the bench", because customer suddenly give a new project to the testing service organization.

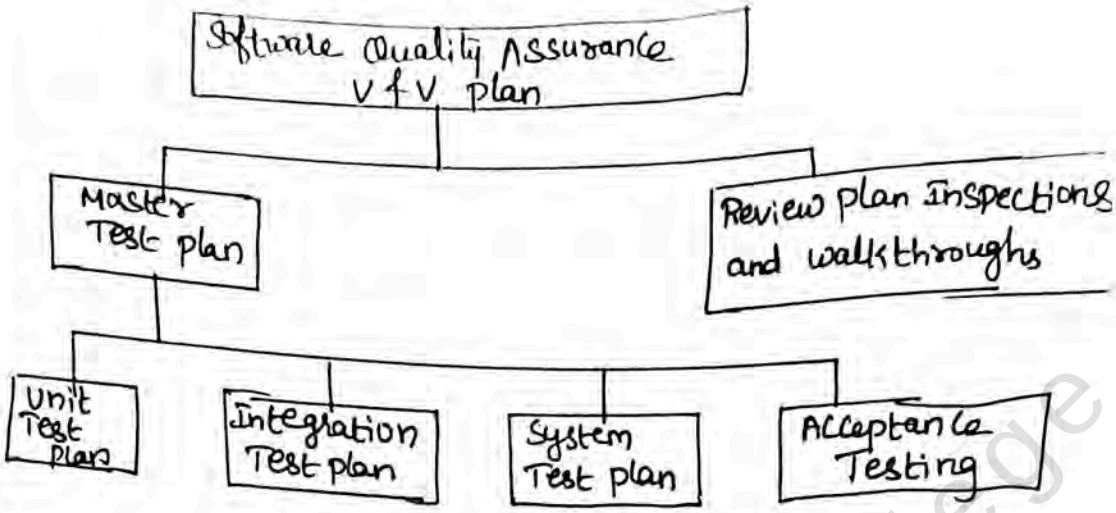
* These people not allocated to any other projects but ready in the wings to take on new projects or to convince customers.

Test plan Components:

* It is also called test plan format / Test plan Unit.

Test plan components are represented as

1. Test plan Identifier
2. Introduction
3. Items to be tested.
4. Features to be tested
5. Approach
6. Pass/fail Criteria
7. Suspension / resumption Criteria
8. Test deliverables
9. Testing tasks
10. Test Environment
11. Responsibilities
12. Staffing and training needs
13. Scheduling
14. Risks and Mitigations
15. Testing Costs
16. Approvals



1. Test Plan Identifier:

* Every test plan holds unique number or name.
* It help to used to identify a particular project
* Organizational standards Explain
  - ↳ Format of test plan identifier
  - ↳ Version representation.
* It is used to find whether it is a configuration item or not in the configuration management system

2. Introduction:

* The test planner gives an overall description of the project
* It includes
  1. Software system development
  2. software items to be checked
  3. High level description of testing goals
  4. Testing approaches 5. References

* Organizational policy, Standards, Quality plan, Configuration plan.

3. Test items

* This is list of the entities to be tested and should include names, identifiers and version numbers for each entry.

* The items listed could include procedures, classes, modules, libraries, subsystems, References Functions, services

4. Features to be tested:

* Features may be described as distinguising characteristics of a software component or system.

Eg: Performance, portability, Functionality.

* The test plan references to test design specifications for each feature and each combination of features are identified to establish the associations with actual test cases.

5. Approach:

* This section of the test plan provides broad coverage of the issues to be addressed when testing the target software.

* The level of descriptive detail should be sufficient so that the major testing task and task durations can be identified.

* More details will appear in the accompanying test design specification.

* The planner should also include for each feature or combination of features the approach that will be taken to ensure that each is adequately tested.

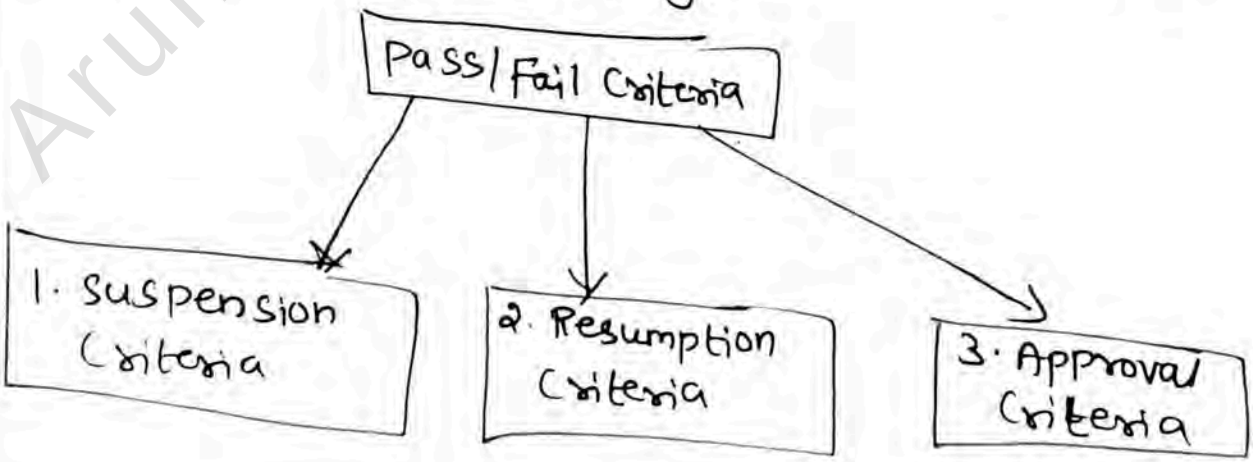
* Tools and techniques necessary for the tests should be included.

*↳ Expectations for test completeness

*↳ How the degree of completeness will be determine should be described

6. Item Pass/Fail Criteria

* It specify the criteria that will be used to determine whether each test item (software/product) has passed or failed testing.



7 Suspension Criteria:

* It specify the criteria used to suspend all or a portion of the testing activity on test items associated with the plan

Resumption Criteria:

* Specify the conditions that need to be met to resume testing activities after suspension.

* Specify the test items that must be repeated when testing is resumed

Approval Criteria:

* Specify the conditions that need to be met to approve test results.

* Define the formal testing approval process

8. Test Deliverables:

* Identify the deliverable document from the test process.

* Deliverable may also include other documents that results from testing such as test logs, test transmitted reports, test incident reports and test summary reports.

9. Testing tasks:

* The test planner should identify all testing-related tasks and their dependancies.

* Using a Work Breakdown Structure (WBS) is useful here.

* A work breakdown structure is a hierarchical or tree like representation of all the tasks that are required to complete a project

10. The testing Environment:

The test planner describes the software and hardware needs for the testing effort.

Eg Any special equipment or hardware needed such as emulators, telecommunication equipment or other devices should be noted.

11. Responsibilities:

* Identify the groups responsible for managing, designing preparing, executing, witnessing, checking, transmitting, developing, tracking & Monitoring. Interacting and resolving testing activities

* These groups may include the developers testers, operations staff, technical support staff data administration staff & the user staff

4 Cost of training

5 Costs of maintaining the DB

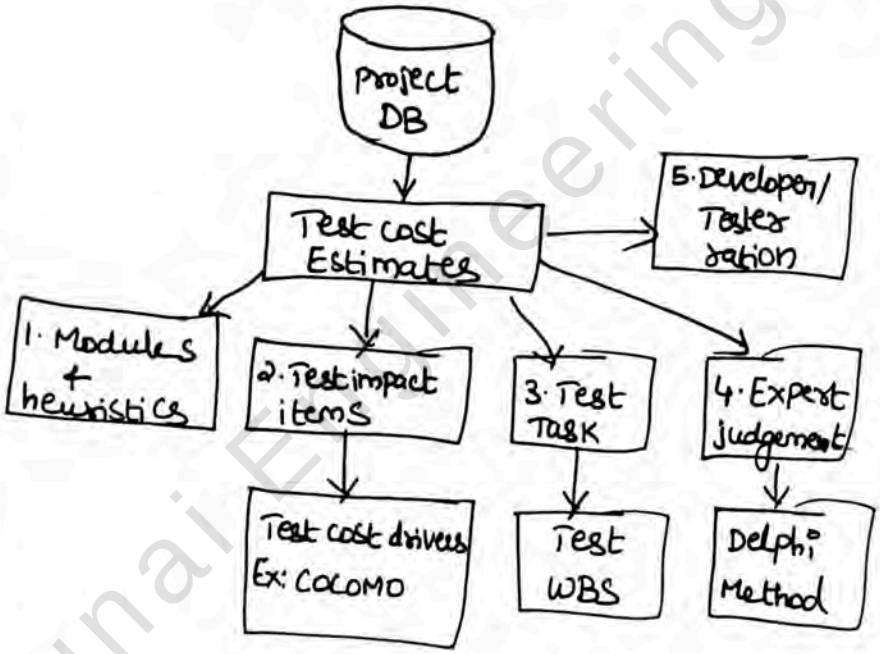
* Test cost impact item are

1. Maturity level of organization
2. Nature of Software product
3. Scope of test requirement
4. Tester ability

* Project Planners follow some cost estimation models

Eg COCOMO model

Test cost Estimation Techniques



16. Test Approvals:

* Identify the plan approvers

* Specify the list the name, Signature & data of plan approvals.

12 Staffing & Training needs:

* The test planner should describe the staff and the skill levels needed to carry out test-related responsibilities.

* Any special training needed to perform a task should be noted.

13. Scheduling:

* Identify the high level schedule for each testing task.

* Establish specific milestones for initiating & completing each type of test activity, for the receipt of each test input & for the delivery of test output.

14 Risk & Mitigations:

* Identify significant constraints on testing such as test item availability, test resource availability and time constraints

* Identify the risks & mitigations associated with testing tasks including schedule, resources, approach & documentation.

15 Testing Costs:

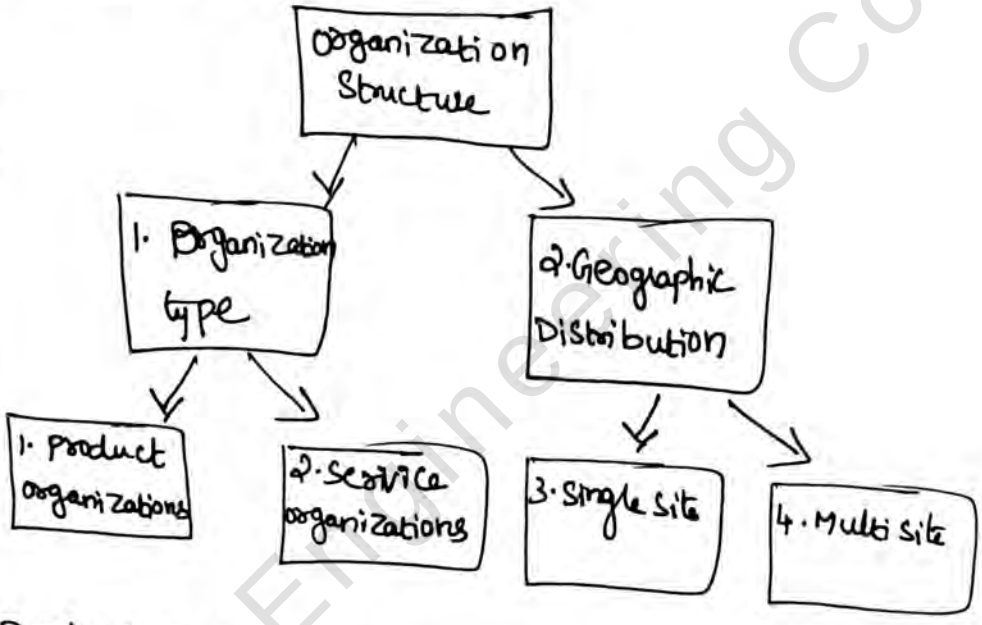
It includes in the plan

1. cost of planning
2. costs of designing the tests
3. costs of test environment

Organizational Structures for testing team:

Dimensions of organization structure:

* organization structures gives a road map for the team members to take over their career paths
* organization structures is viewing upon two dimensions.



Product organizations

* Produce software products and have a "womb to tomb" responsibility for the entire product.

Service organizations:

* Does not have the complete product responsibility.
* They provide testing services to other organizations
* These organizations are specilist for testing services only.

Geographic Distribution

Single site team

* All the team members are located & worked in one place

Multi-site team

* Entire team is spreaded and worked across many locations.

Structure for single-product companies.

Structure for multi-product companies.

* In multi-product companies, every product is treated as separate business unit

* The organization of test teams in multi product companies is dicated largelly by the following factors.

1. How tightly coupled the product are interms of technology
2. Dependence among various products
3. How synchronous are the release cycle of products.

To organize the testing teams for a multi product company, there are many chances. They are

1. A Central "test think-tank/brain trust" team, which formulates the test strategy for the organization.
2. one test team for all the products
3. Different test teams for different types of tests
4. Different test teams for each product.
5. A Hybrid of all the above methods.

Testing teams as part of "CTO's office:

* Development & Testing are the same level of importance in the concern.
* Testing team report directly to the CTO as a part to the design & Development teams.

Advantages:

1) Developing a Product architecture that is testable or Suitable for testing.
2) Testing team will have better product and technology skills.
3) The testing team can get a clear Understanding of what design & architecture are built for and plan their tests accordingly.
4) The technical road map for Product development and test.
5) The CTO's team can evolve a consistent, cost effective Strategy for test automation.

* CTO deals only the architecture & test teams in this model
* The reasons to report to the CTO is that the team is Cross divisional & Cross functional.

In order that such a team reporting to the CTO be effective

1. It should be small team size
2) It should be team of equals
3. It should have organization-wide representation.

Single Test Team for all products

* Single testing team is responsible for all type of Products.

* Single product team divided into multiple components

* Every component is developed by an independent team.

* The testing team can be made to report to the "CTO think-tanks".

Testing teams organized by product:

* It is based on accountability, decision making and scheduling

* Complete responsibility is assign to testing team

* Unit head organize the testing & development team.

Separate testing teams for different phases of testing

* Testing is a single & homogeneous activity

* Testing is done by several types of testing and different levels of skills.

* Types of Testing - Black box testing, white box Testing, Regression testing.

* Levels of skills - Test Script, Test case, Test report

* Each of these different types of tests may be carried out at different point of time.

Organization people to perform different types of testing

Type of Test	Reports into	Rationale
White box Testing	Development team	It is inherently close to code, developers, develop & also runs the code
Black box Testing	Testing team	It is first level of external testing which a product may hold
Integration Testing	organization wide testing team	Combine multiple components, multiple products
System Testing	Product Management	It takes place by doing the testing in real time scenarios
Performance Testing	A Central bench marking group	Interproduct dependencies
Acceptance Testing	product management	Proxy for customer acceptance
I18N Testing	Local teams (or) I18N teams	Knowledge of local languages and convention
Regression Testing	All test team	* Part of Smoke test * continue to report into the product testing teams

Test Planning:-

* Preparing a test plan
* Scope Management
* Deciding test approach/strategy
* Setting up criteria for testing
* Identifying responsibilities, staffing and training needs
* Identifying resource requirements
* Identifying test deliverables.
* Testing tasks: Size and effort estimation
* Activity breakdown and scheduling
* Communication management
* Risk management.

Preparing a Test plan:

"Failing to plan is planning to fail"

* Testing - like any project - should be driven by a plan.
* The test plan acts as the anchor for the execution, tracking and, reporting of the entire testing projects.

1) what need to be tested the scop of testing, including clear identification of what will be tested and what will not be tested.

2) How the testing is going to be perform breaking down the testing into small and manageable tasks and identifying the strategies to be used for carrying out the tasks.

3) The time lines by which the testing activities will be performed.

Scope Management

* Scope management pertains to specifying the scope of a project. For testing scope management
1) Understanding what constitutes a release of a product.

2) Breaking down the release into features.

3) Prioritizing the features for testing.

4) Deciding which features will be tested & which will not be.

5) Gathering details to prepare for estimation of resources for testing.

The following factors drive the choice and Prioritization of features to be tested.

* Features that are new and critical for the release.
* Features whose failures can be catastrophic.
* Features that are expected to be complex to test
* Features which are extensions to be complex to test

Deciding Test Approach / Strategy:

* once we have this prioritized feature list, the next step is to drill down into some more details of what needs to be tested, to enable estimation of size, effort and schedule. This includying identifying.

1) what type of testing would you use for testing the functionality?

2) what are the configurations or scenarios for testing the features?

3) what integration testing could you do to ensure these features work together?

4) what localization validations would be needed?

5) what "non-functional" tests would you need to do?

Setting Up Criteria For Testing:

* Suspension Criteria specify when a test cycle or a test activity can be suspended.

* Resumption Criteria specify when the suspended tests can be resumed.

Some of the typical suspension criteria include.

1) Encountering more than a certain number of defects, causing frequent stoppage of testing activity.

2) Hitting show stoppers that prevent further progress of testing.

3) Developers releasing a new version which they advise should be used in lieu of the product under test.

Identifying Responsibilities, Staffing and Training needs:

* As a part of planning for a testing project, the project manager should provide estimates for the various hardware and software resources required. Some of the following factors need to be considered.

1) Machine configuration (RAM, processor, disk) needed to run the product under Test.
2) Overheads required by the test automation tool, if any.
3) Supporting tools such as compilers, test data generators, configuration management tools and so on.

Identifying Test Deliverables:

* The test plan also identifies the deliverables that should come out of the test cycle / testing activity. The deliverables include the following all reviewed and approved by the appropriate people.

1) The test plan 2) Test case design specifications 3) Test cases 4) Test logs 5) Test summary reports.

Testing tasks: Size and Effort Estimation

* Necessary tasks to do before starts every features to be testing (it is called estimation).

Eg: Size estimation, Effort estimation, Schedule Estimation.

* Size estimate quantifies the actual amount of testing that needs to be done.

1. Size of the product under test.
2. Extent of automation required.
3. Number of platforms and inter-operability environments to be tested.
4. Productivity data
5. Reuse opportunities
6. Robustness of processes.

Activity Breakdown and Scheduling:-

* Activity Breakdown and Scheduling estimation entail translating the effort required into specific time frames.

The following steps make up this translation.

1. Identifying external and Internal dependencies among the activities.
2. Sequencing the activities
3. Allocation time for WBS activities
4. Monitoring the process
5. Rebalancing the schedules & resources.

Communication Management:

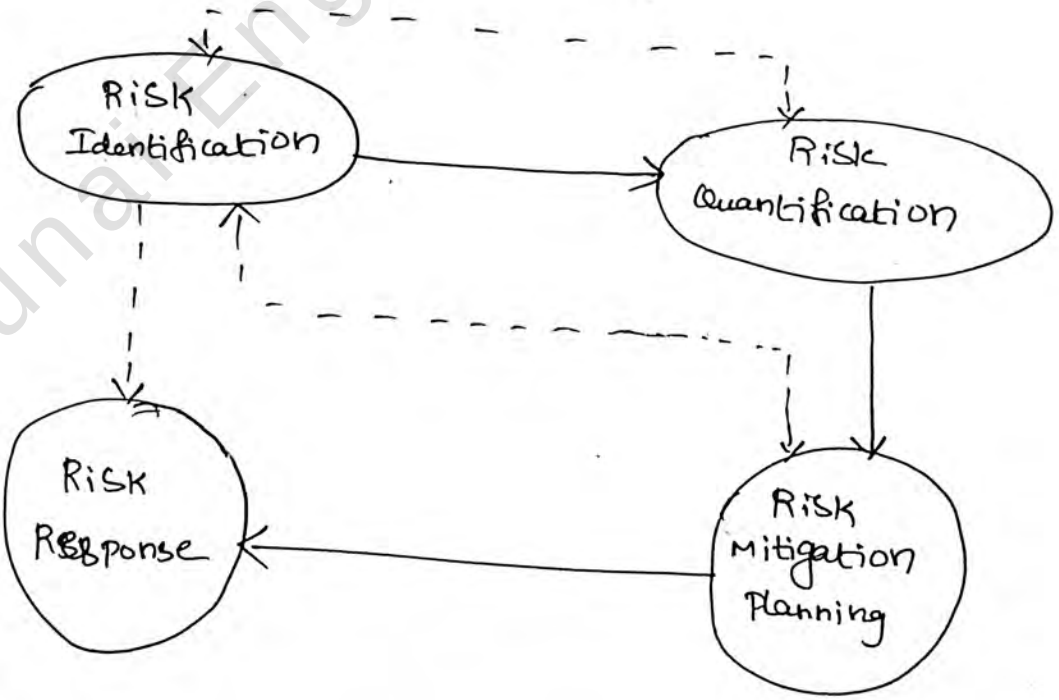
* Communication management consists of evolving and following procedures for communication that ensure that everyone is kept in synchronization with the right level.

Risk Management:

* Risks are events that could potentially affect a projects outcome. These events are normally beyond the control of the project manager.

1) Identifying the possible risks.
2) Quantifying the risks
3) Planning how to mitigate the risks and
4) Responding to risks when they become a reality.

Aspects of risk Management



Test Plan Attachments

Test Plan required list of documents for detailed information extraction.

1. Test design specification
2. Test case specification
3. Test procedure specification

1. Test design specification:

* The test design is the first stage in developing the tests for software testing projects.
* It records which features of the test item are to be tested, and how a successful test of these features would be recognized.
* It doesnot record the values to be entered for a test, but describes the requirements for defining those values.

Components of Test design specification:

1. Test design specification Identifier
2. Features to be tested
3. Approach refinements
4. Test identification.

1. Test design Specification identifier.

* Specify the unique identifier assigned to the test procedure.
* Supply a reference to the associated test procedure specification.
* It is unique identifier for every test design spec.

2. Features to be tested.

* The set of test objectives covered by this test design specification.
* It is the overall purpose of this document to group related test item together.

3. Approach refinements:

* Add the necessary level of detail & refinement based on the original approach defined in the test plan associated with this test design specification.

4. Test Identification:

* Identification of each case with a short description of the case, it's test level or any other appropriate information required to describe the test relationship.

II Test case specification :

* It describe the purpose of a specific test, identifies the required inputs and expected result.
* It provides step-by-step procedures for executing the test.
* It outlines the pass/fail criteria for determining acceptance.
* It is developed in the development phase by the organization responsible for the formal testing of the application.

IEEE 829 Test case specification Template
Test case specification Identifier
Test Items (It describe features and condition tested)
Input specification Data items Ordering Values, Status, Timing
Output specification Data items, Ordering, Values, Status, Timing.
Environmental needs Hardware, Software, Others
Special procedural Requirements
Inter-case Dependencies.

III Test Procedure specification.

* It is a collection of steps in sequence to do a particular test.
* A detail of how the tester will Physically run the test, the Physical set up required and the procedure steps that need to be followed.
* In this specification, planner gives the required steps to the test cases execution.

IEEE 829 Test Procedure Specification Template

Test Procedure specification Identifier

It is a unique identifier

Purpose.

* It describe the purpose of procedure
* Refer to the test cases being executed.

Special requirments.

* manual / Automated
* Stages in which test is to be used.
* Test environment
* Special skills or training required.

Procedure steps:

* What are the activities associated with the Procedure
* It includes.

1. Log
2. Set-up
3. Start
4. Proceed
5. Measure
6. Shut down
7. Restart
8. Stop
9. Wrap-up
10. Contingency.

Locating Test Items

* When a tester is ready to do test, the testers has to locate an item and should have knowledge of its current status.
* It is function of Test item transmittal report.

Test item transmitted Report

* A detail of when specific tested items have passed from one stage of testing to another.
* It's not a part of test plan.
* It is used to locate & track the items which required to test.

Reason for the document:

Information to the test team from the development team that a test team is in the test area & ready to test, or from test team that items is ready to move to next stage / back for network.

Audience:

* Development team
* Test team
* configuration / change management team

When to write?

* Have Standard form
* Complete as items are ready for test
* Complete as items are ready for retest.

How to use?

* Audit trail.
* Permission / start Criterion.

IEEE 829 Test item transmitted report Template

1. Identifier
   * Unique number
   * And reference to test plan & test design.
2. List of transmitted items.
   * Version number of items.
   * Person responsible for the item
   * Reference to item documentation is Test Plan
3. Known issues
4. Item Status
5. Application information
6. Approvals
   * Space for Signatures of Staffs who approves the transmittal.

Test Management.

What is test management?

* Test management is a process of managing the tests.

* A test management is also performed using tools to manage both types of test.

* Test management tools allow automatic generation of the requirement test matrix (RTM) which is an indication of functional coverage of the application under test (AUT).

Test Management Responsibilities.

* Test management has a clear set of roles responsibilities for improving the quality of the product.

* It enables developers to make sure that there are fewer design or coding faults.

Choice of Standards.

* It is a important part of planning in organization.

* Organizations standards to be classified into two types.

1. External Standards
2. Internal Standards

1. External Standards

These are the standards that a products should comply with and externally visible.

Ex. Quality Standards
working Standards.

2. Internal Standards

These standards are created by testing, testing organization to put.

Ex. Cosistency, Predictatability, robustness.

Test Infrastructure Management:

Following element's are necessary for to design.

1. Test case Data Base (TCDB)
2. Defect Repository
3. Configuration management Repository & It's tools.

1. Test case Data Base (TCDB)

* It maintains a test case information.
* It records all the static information about the texts.

eg. Test case id, Test case Name, Test case owner.

2. Defect Repository.

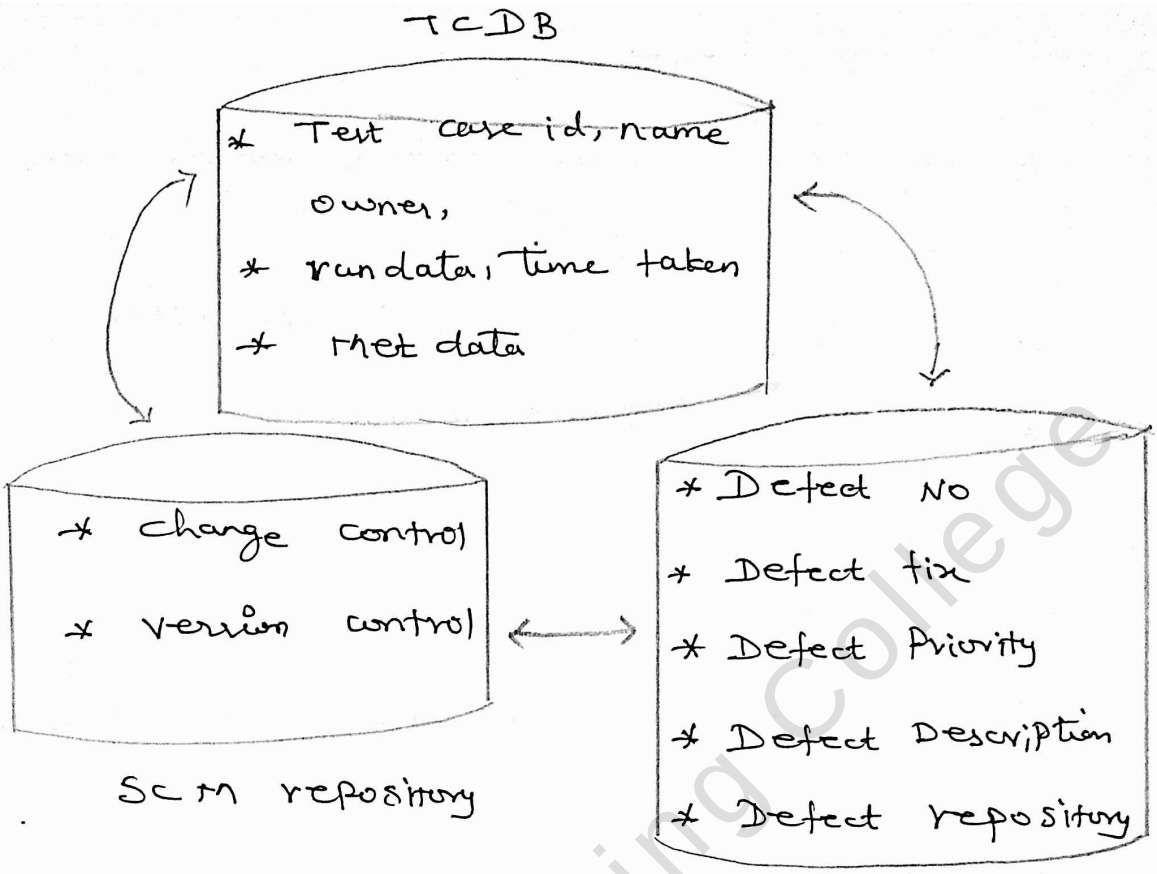
* It keeps all the defect and its detail for a Product.
* It gives defect fix details also.
* It is used for communication work flow for an organization.

3. Software Configuration management Repository (Scm)

* It track the change control and version controls for all files.
* change controls assures,
  - changes to test files are done in controlled way
  - changes done by one test engineer are not lost by other changes.
* Version Control assures.
  - Test scripts along with a Product are baselined

Base lining:

* It is a snap shot of set of related files of a version, assigning a unique identifier to this set.
* Test case Data Base (TCDB) defect repository SCM repository works together in an integrated manner.



Test people management

* It needs the ability to hire, motivate and retain the right people.

Integrating with Product release

* Product success is based on the effectiveness of integration of the development and testing activity.

1. Product support Group
2. Product Management Group

Test Management Phases

* It Contains 2 main Phases

1. Planning
2. Execution

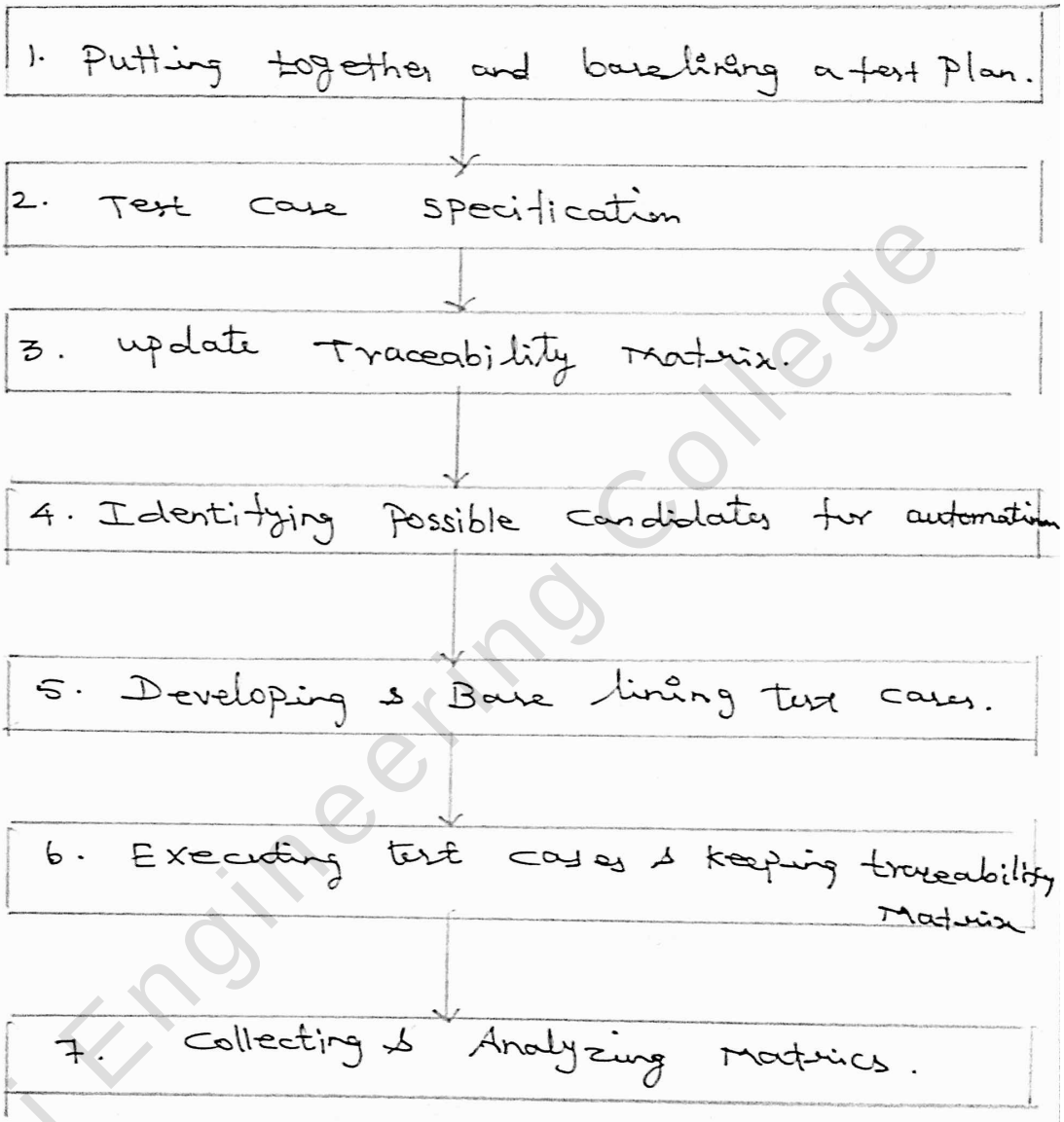
* Planning Phases Contains.

1. Risk analysis
2. Test estimation
3. Test Planning
4. Test organizatin

* Execution Phases includes..

1. Test monitoring
2. Issues management
3. Test Report and evolution.

Test Process.



1. Putting together & baselining a test plan.

* Test Plan is an anchor point for every Project
* Every test plan is designed based on templet.
* It is reviewed by designated people in an organization.
* A complete authority later approved it.

2. Test case specification

Test cases creation depends on,

1. Test purpose
2. Items to be tested
3. Environment to run the test cases
4. Input data to be given
5. Execution steps.

3. update traceability matrix.

* It helps to verify, every requirement is tested or not.
* It is developed in requirement gathering phase.

4. Identifying possible candidates for automate

It helps to decide

1. which test cases to be automated?
2. Before writing the test cases, which test to be automated?

5. Developing & Baselining testcases.

* The creation of test cases entails translating the test specification to a form from which test can be executed.
* when a test case need a manual, writing the detailed information of test scenario.

6. Executing test cases & keeping traceability matrix.

* The test case to be executed at correct time.

Ex. Done smoke / sanity test frequently.

7. Collecting & analyzing metrics.

* The basic measurements from running the tests are converted to meaningful metrics by the use of suitable transformations.

Reporting test Results.

What is test report?

* Document that records data obtained from an experiment of evalution in an organized manner, describes the environmental or operating conditions, and shows the comparison of test results with test objectives.

* Some of the test report are

1. Test Log report
2. Test incident report
3. Test Summary report
4. Test status report.

1. Test log report

* It is a document, which the all information about the test results, and it contains, the expected results also.

Test log Report - Template

1. Test log identifier

   It is a unique number.

2. Description

   * Items to be tested
   * Version number
   * Associated test item.

3. Activity & Events entries

   * Dates & names of test log authors
   * Execution description
   * Procedure results
   * Anamalous events.

2. Test incident report

   * It is also called Bug report
   * It is uniquely numbered and tracked to resolution.
   * Here defect tracking tools are Preferred.

3. Test Summary report.

* It is Prepared at the end of a testing Projects or rather after testing is completed.

* It contains.

1. Purpose of the document
2. Application overview
3. Testing Scope
4. Metrics
5. Types of Testing performed
6. Test environment & Tools
7. Lessons Learned
8. Recommendation
9. Best Practices
10. Exit criteria.
11. Conclusion / Sign off.

Test incident Report

1. Test incident report identifier

2. Summary
   * References
   * Test Procedure
   * Test case Specifications
   * Test Logs
   * Other supporting materials.

3. Incident Description
   * Inputs
   * Expected results
   * Actual results
   * Anomalies
   * Date & Time
   * Testers

4. Impact
   * Severity
     1. Mission Critical
     2. Major
     3. Minor.
   * Priority
     1. Immediate
     2. Delayed
     3. Deferred (duplicate)

The Role of various groups in Test planning and Policy Development.

* Each group views the testing process from a different perspective that is related to their particular goals, needs and requirements

3 critical groups in testing are

1. Managers
2. Testers/Developers
3. User/Clients

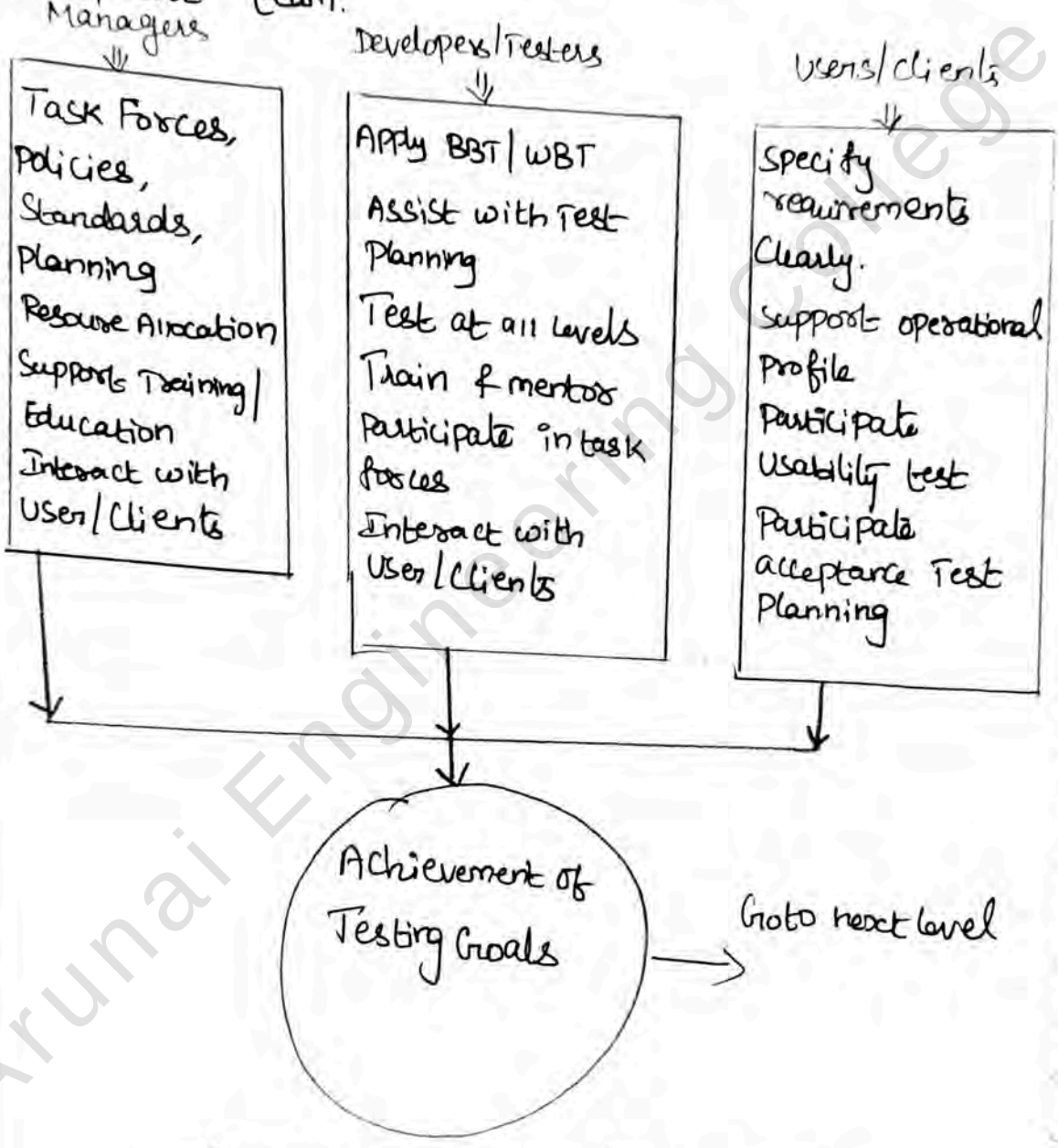
↳ The Manager's view involves commitment and support for those activities and tasks related to improving testing process quality.

↳ The Developer (Tester's view) encompasses the technical activities and tasks that when applied, constitute best testing practices

↳ The User/Clients view is defined as a cooperating or supporting view.

* Developers have an important in the development of testing goals & policies.

* They serve as members of the goal/policy development team.



Summary of critical Group roles

* The developers or testers work with client/user groups on quality-related activities and tasks that concern user-oriented needs.

* The developers/testers activities are,

1. working with management to develop testing & debugging policies & goals.
2. Participating in the teams that oversee policy compliance & change management.
3. Familiarizing themselves with the approved set of testing/debugging goals & policies
4. Keeping up-to-date with revisions & making suggestions for changes when appropriate
5. When Developing test plan, setting testing goals for each project at each level of test that reflect organizational testing goals and policies.
6. Carrying out testing activities that are in compliance with organizational policies.

↳ Users/Clients plan an indirect role in the formation of an organization's testing goals & policies.

* Since these goals & policies reflect the organizations effort to ensure customer/Client/user satisfaction

Management Support:

1. Establishing an organization wide test planning committee with funding.
2. Ensuring the testing policy statement & quality standards support test planning with commitment of resources, tools, templates and training.
3. Ensuring that all projects are in compliance with the test planning policy.
4. Ensuring that all developers/ testers complete all the necessary post test documents such as test log, test incident, test summary reports.

Project Manager-

* They support test planning maturity goals by preparing the test plans for each project with inputs & support from developers.

Developers-

* who are experienced in testing support this maturity goal by participating in test planning.

* They assist the project manager in determining test goals, selecting test methods, procedure & tools & developing the test case specification, test procedure specification.

* From the user/client point of view support for test planning is in the form of articulating their requirements clearly and supplying input to the acceptance test plan.

Skills needed for a test specialist:

* The nature of technical and managerial responsibilities assigned to the tester that are listed many managerial and personal skills are necessary for success in the area of work.

Personal and Managerial Skills

1) organizational and planning skills
2) The ability to keep track of and pay attention to, detail
3) The determination to discover and solve problems
4) The ability to work with others and resolve conflicts
5) Mentor and train others
6) The ability to work with users and clients
7) Strong written and oral communication skills
8) The ability to work in a variety of environments
9) The ability to think Creatively.

* The first three skills are necessary because testing III detail and problem oriented.

* In addition, testing involves policy making, a knowledge of different types of application areas

Planning and the ability to organize and monitor information, tasks and people

Technical Skills

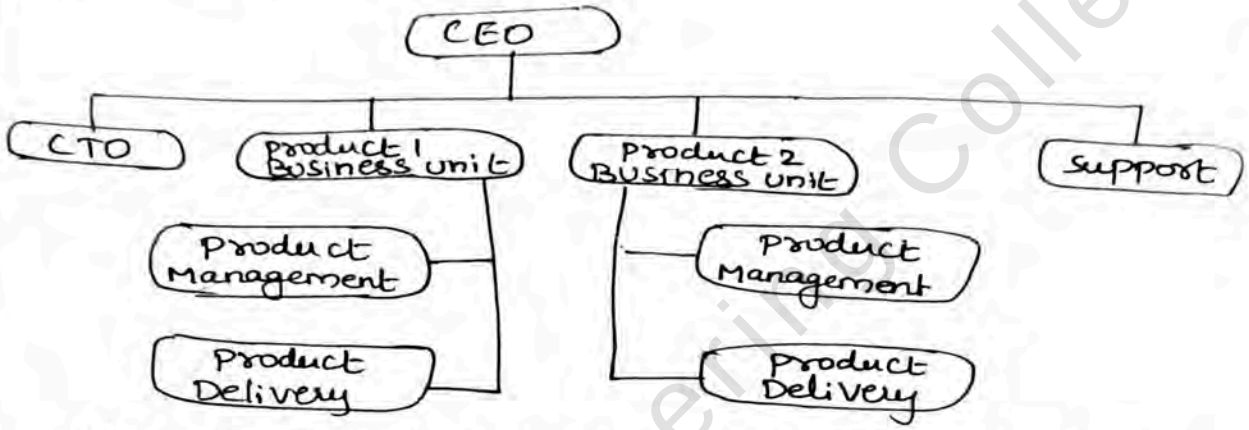
1. General software Engineering principles and practices
2. Understanding of testing principles and practices
3. Understanding of basic testing strategies and methods.
4. Ability to plan, design and execute test cases
5. Knowledge of process issues
6. knowledge of networks, databases and os
7. knowledge off configuration management
8. knowledge of test-related documents
9. Ability to define, Collect and analyze test-measurements
10. Ability, training and motivation to work with testing tool
11. knowledge of quality issues.

* A good understanding of testing principle and practices
* A good understanding of basic testing strategies, methods and techniques.
* A knowledge of process issues
* A knowledge of configuration management
* The ability to define, collect and analyze test-related measurements.

b) Organization Structures for Testing Teams

Structure - in Single product companies.

* Product companies in general have a high-level organization structure.



Organization Structure of a multi-product company.

* The CTO's office sets the high level technology directions for the company.

* A business unit is in charge of each product that the company produces.

* A product business unit is organized into a Product management group and a product delivery group

* The product management group has the responsibility of merging the CTO's direction with specific market needs to come out with a product road map.

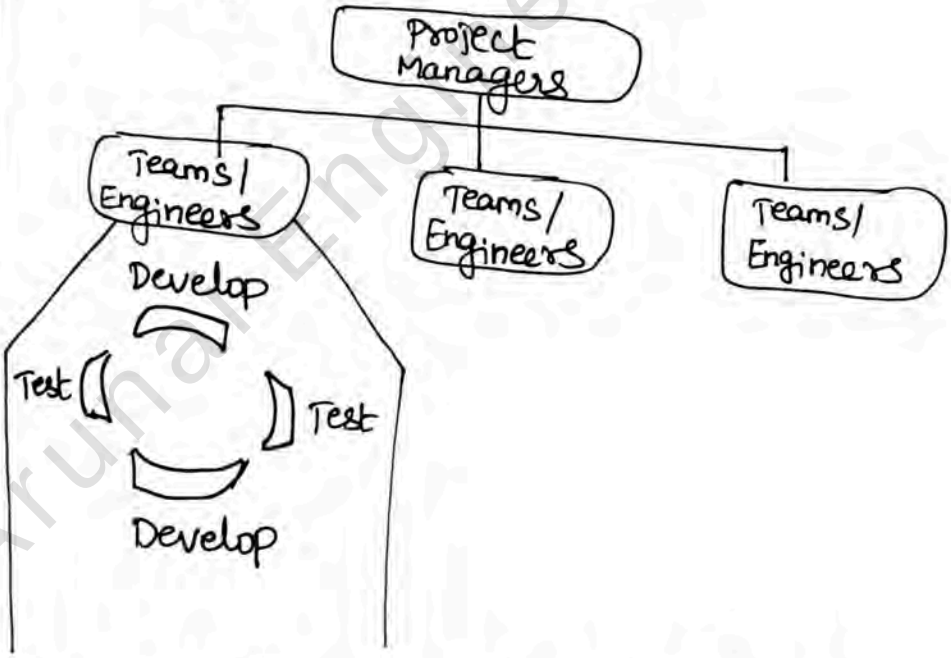
* The product delivery group is responsible for delivering the product and handles both the development and testing function

Testing team structures for single-product companies:

* Most product companies start with a single product

* The product delivery team members distribute their time among multiple tasks and often wear multiple hats.

* Very thin line separating the development team and "testing team"



Typical organization structures in early stages of a product

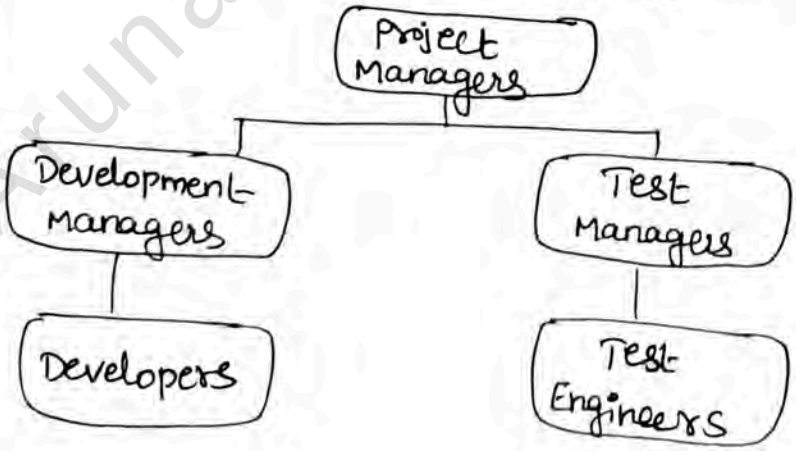
Advantages:

1. Exploits the rear-loading nature of testing activities
2. Enables engineers to gain experience in all aspects of life cycle.
3. Is amenable to the face that the organization mostly only has informal processes
4. Some defects may be detected early.

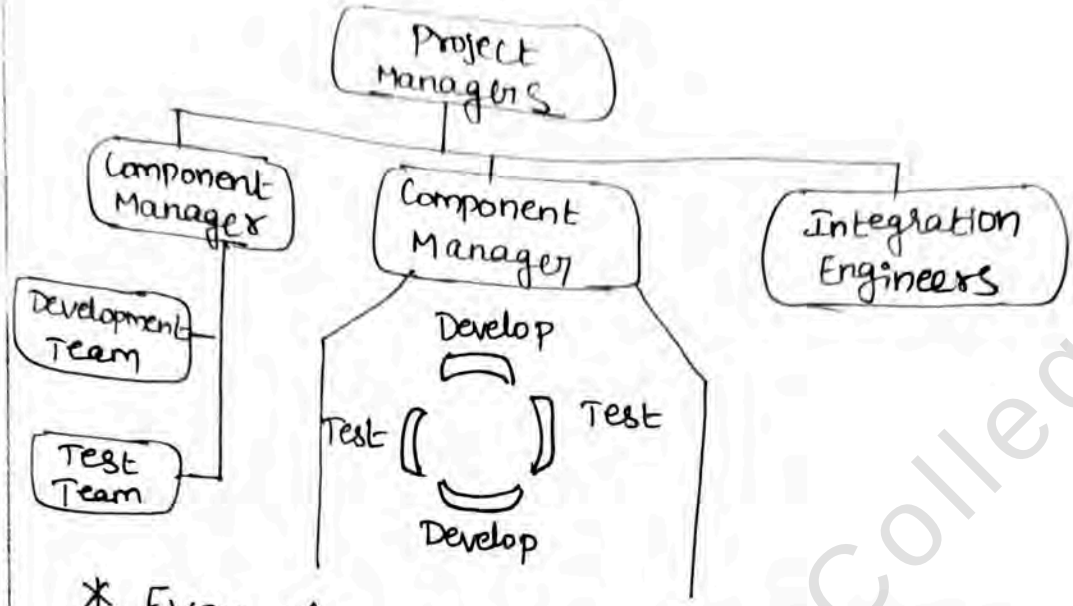
Disadvantages:

1. Accountability for testing & Quality reduces
2. Developers do not in general like testing and hence the effectiveness of testing suffers.
3. Schedule pressures generally compromise testing.
4. Developers may not be able carry out the different types of tests.

Separate groups for testing & Development-



Component wise Testing teams



* Even if a company produces only one product, the product is made up of a number of components that fit together as a whole

* Every components is developed & tested by separate team

* All these components are integrated by using a single integration test team

* This team report to the project Manager.

BUILDING A TESTING GROUP

Establishing a specialized testing group is a major decision for an organization. The steps in the process are summarized in the following Figure. To initiate the process, upper management must support the decision to establish a test group and commit resources to the group. Decisions must be made on how the testing group will be organized, what career paths are available, and how the group fits into the organizational structure (See: The Structure of the Test Group).

When hiring staff to fill test specialist positions, management should have a clear idea of the educational and skill levels required for each testing position and develop formal job descriptions to fill the test group slots.

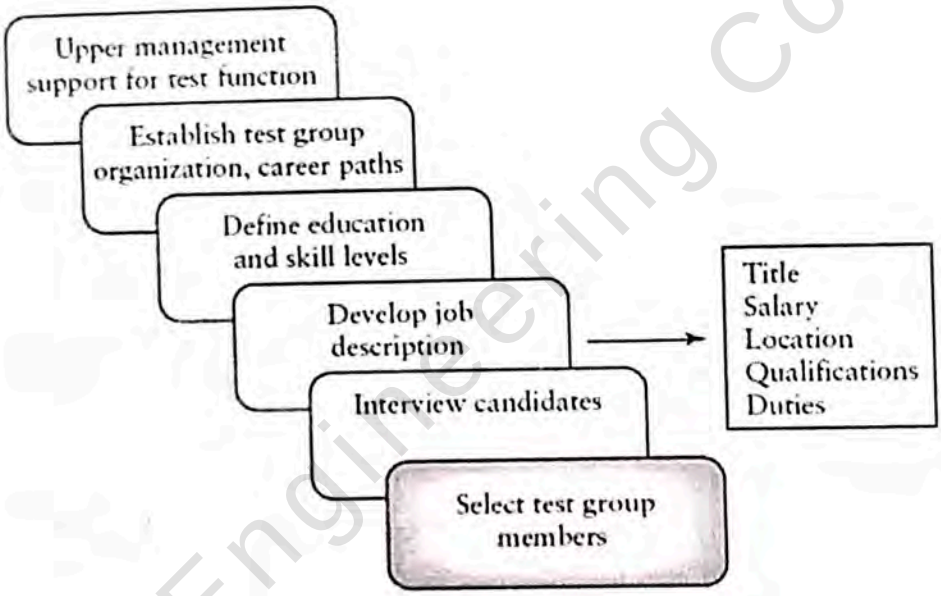


FIG. Steps in forming a test group

THE STRUCTURE OF THE TEST GROUP

It is important for a software organization to have an **independent testing group.** The group should have a formalized position in the organizational hierarchy. A reporting structure should be established and resources allocated to the group. The group should be staffed by people who have the skills and motivation. They should be dedicated to establishing awareness of, and achieving, existing software quality goals, and also to strengthening quality goals for the future software products. They are **quality leaders—the test and quality policy makers.** They measure quality, and have responsibilities for ensuring the software meets the customers' requirements.

A test organization is expensive, it is a strategic commitment. Given the complex nature of the software being built, and its impact on society, organizations must realize that a test organization is necessary and that it has many benefits. By investing in a test organization a company has access to **a group of specialists who have the responsibilities and motivation to:**

- maintain testing policy statements;
- plan the testing efforts;
- monitor and track testing efforts so that they are on time and within budget;
- measure process and product attributes;
- provide management with independent product and process quality information;
- design and execute tests with no duplication of effort;
- automate testing;
- participate in reviews to insure quality;
- work with analysts, designers, coders, and clients to ensure quality goals are meet;
- maintain a repository of test-related information;
- give greater visibility to quality issues organization wide;
- support process improvement efforts.

The duties of the team members

The duties of the team members may vary in individual organizations. The following gives a brief description of the duties for each tester that are common to most organizations:

- **The Test Manager**: In most organizations with a testing function, the test manager (or test director) is the central person concerned with all aspects of testing and quality issues.

  The test manager is usually responsible for test policy making, customer interaction, test planning, test documentation, controlling and monitoring of tests, training, test tool acquisition, participation in inspections and walkthroughs, reviewing test work, the test repository, and staffing issues such as hiring, firing, and evaluation of the test team members. He or she is also the liaison with upper management, project management, and the quality assurance and marketing staffs.

- **The Test Lead**: The test lead assists the test manager and works with a team of test engineers on individual projects. He or she may be responsible for duties such as test planning, staff supervision, and status reporting. The test lead also participates in test design, test execution and reporting, technical reviews, customer interaction, and tool training.

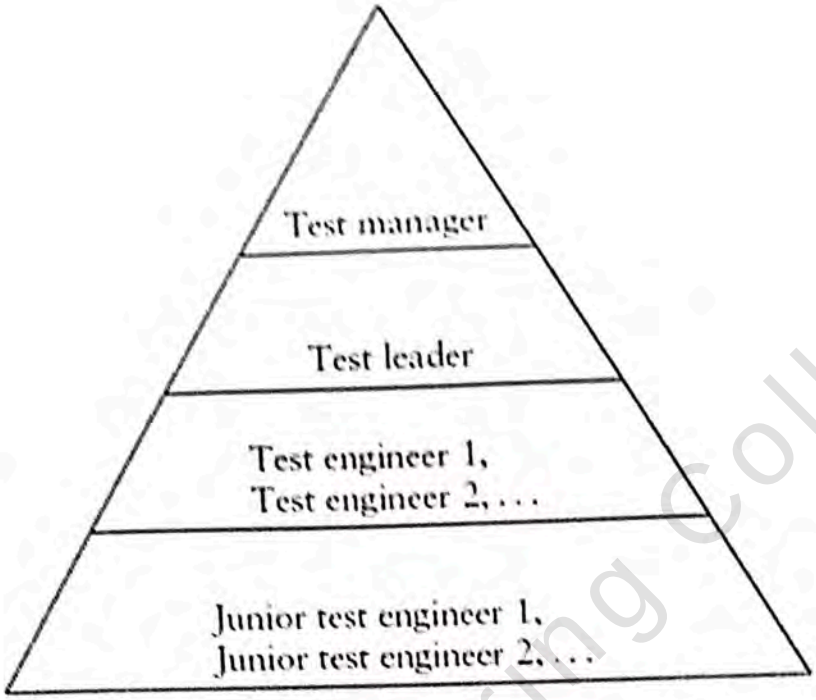


FIG. The test team hierarchy

- **The Test Engineer:** The test engineers design, develop, and execute tests, develop test harnesses, and set up test laboratories and environments. They also give input to test planning and support maintenance of the test and defect repositories.
- **The Junior Test Engineer:** The junior test engineers are usually new hires. They gain experience by participating in test design, test execution, and test harness development. They may also be asked to review user manuals and user help facilities defect and maintain the test and defect repositories.

Providing carrier paths for testing professionals:

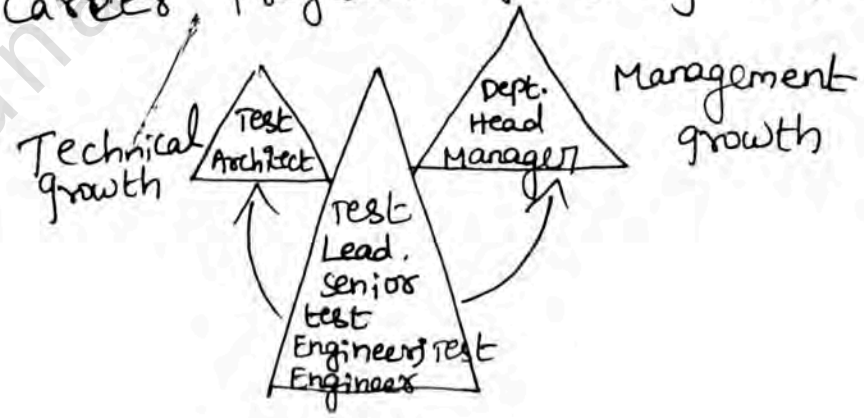
* When performing carrier path testing, must noted following areas.

1. Technical challenge
2. Learning opportunities
3. Increasing responsibility
4. Increasing authority
5. Increasing independence
6. Ability of an organization success
7. Rewards
8. Recognition

Three stages of individual in testing group goes through

1. Follow Stage. 2. Formulation stage
3. Test lead Stage.

Career progression for testing professionals



Comparison between Testing & development Team:

1. Testing is frequently said to be a crunch time function.

* Testing is treated carefully in the product release time.

* Because it throws some unique planning, management challenges, deliverables.

2. More "elasticity" is allowed in project's in early phases.

* planning testing projects affords less flexibility than development projects.

3. Testing functions are arguably the most difficult ones to staff

* Testing is not a smaller function. only minimum number of people chooses the testing as a job.

* It is hard to attract and retain top talent for testing functions and its also carried under time pressure.

4. Testing functions hold several External dependencies when compared with the development functions

* Testing is done at the end of the project life cycle.

The role of Ecosystem and a call for action
Role of education System.

* There are collective and much higher-level actions that need to be done.

* The entire eco system covering the eduction System, and the community as a whole

i) Role of Education system:

* Education system does not provide sufficient importance in the testing field

The reasons are

1. Many formal core courses on programming but only few universities offer core courses on software testing.
2. only few "lab courses" for common testing tools, compare to other programming lab courses.
3. No complete weightage for coding & testing.
4. There is no reward for test engineers and their skills.
5. Products are based on Quality, not an demands So demand reduces the ability/quality.

* To improve the awareness of testing is essential for software testing knowledge, skills, attitude towards testing etc.

ii) Role of Senior management: 4.11.

* The Senior management of organizations plays a vital role in a development of testing peoples

* It is simply not enough to use words like "Quality is our top screen"

(or)

"People are out top priority"

* Senior management people commitment has to be must translated into "visible action".

Some of the concrete works are

1. Ensuring a fair distribution of talented people in the testing arena.
2. Not allowing development engineers to look down upon Test engineers.
3. Dont treat test engineers as a low grade employees.
4. Encourage active & talented test peoples.
5. To confirm the perfect job rotation amoungd development, testing & support functions.

iii) Role of community:

* Important roles of testing community are

1. Give the freedom for tester, Tester should begin with a sense of pride in their job.
2. Test engineers need not be just followers of technology. They can take an active part in shaping & using new technology.
3. Sharing experience & knowledge to forum people.
4. Generally if learning the product and testing it is done at the same time then it gives inefficient testing.

Challenges of Global Teams

1) Cultural Challenges
2) Work allocation challenges
3) Parity across teams
4) Ability to track effectively
5) Dependence of communication infrastructure
6) Time difference

IT8076-SOFTWARE TESTING

UNIT-5
TEST AUTOMATION

Test Automation:

* Developing software to test the software is called test automation.

* Automation saves time as software can execute test cases faster than human do.

* The time thus saved can be used effectively for test engineers to

1) Develop additional test cases to achieve better coverage.

2) Perform some esoteric or specialized tests like ad hoc testing or

3) Perform some extra manual testing.

Skills needed for Automation:

* There are different "Generations of automation". The skills required for automation depends on what generation of automation the company is in or desires to be in the near future.

The automation of testing is broadly classified into three generations.

1) First generation - Record and playback
2) Second generation - Data - Driven
3) Third generation - Action - Driven

1) First generation - Record and playback

* Record and playback avoids the repetitive nature of executing tests.

* A test engineers records the sequence of actions by keyboard characters or mouse clicks and those recorded scripts are played back later, in the same order as they were recorded.

* It is simple to record and save the script This generation of tools has several disadvantages.

* The scripts may contain hard-coded values thereby making it difficult to Perform general types of tests.

Eg: when a report has to use the current date and time it becomes difficult to use a recorded script.

The handling error condition is left to the testers and the played back scripts may

require a lot of manual intervention to detect and correct error conditions.

* when the application changes, all the scripts have to be recorded. thereby increasing the test maintenance costs

2) Second Generation - Data-Driven:

* This method helps in developing test scripts that generates the set of input conditions and corresponding expected output.

* This enables the tests to be repeated for different input and output conditions

* The approach takes as much time and effort as the product.

* However changes to application does not require the automated test cases to be changed as long as the input conditions and expected output are still valid.

* This generation of automation focuses on input and output conditions using the black box testing approach.

3) Third generation Action-Driven

* This technique enables a layman to create automation tests. There are no input and expected output conditions required for running the tests

* All actions that appear on the application are automatically tested, based on a generic set of controls defined for automation.

* The set of actions are represented as objects and those objects are reused.

* The users needs to specify only the operations (such as log in, download & so on) and everything else that is needed for those actions are automatically generated.

* Automation in third generation involves two major aspects

i) Test case automation and
ii) Framework design

* The skills needed for automation are classified into four levels for three generations as the third generation of automation introduces two levels of skills for development of test cases and framework.

Classification of Skills for automation

Automation first generation	Automation second generation	Automation third generation	
Skills for test case automation	Skills for test case automation	Skills for test case automation	Skills for Framework
Scripting languages	Scripting languages	Scripting languages	Programming languages
Record-playback tools usage	programming languages	programming languages	Design and architecture Skills for framework Creation.
	knowledge of data generation techniques usage of the product under test	Design and architecture of the product undertest usage of the frame work	Generic test requirement for multiproducts

Scope of Automation:

* The automation requirements define what needs to be automated looking into various aspects.

* The specific requirements can vary from product to product from situation to situation, from time to time.

* Tips for identifying the scope for automation.

* Identifying the types of testing and amenable to automation
* Automating areas less prone to change.
* Automate tests that pertain to standards
* Management aspects in automation.

Identifying the types of Testing Amenable to Automation

Certain types of tests automatically lend themselves to automation.

* Stress, reliability, scalability and performance testing.
* Regression tests
* Functional tests

1. Stress, reliability, scalability and performance testing:

* These types of testing require the test cases to be run from a large number of different machines for an extended period of time, such as 24 hours, 48 hours and so on.
* It is just not possible to have hundreds of users trying out the product day in and day out
* They may neither be willing to perform the repetitive tasks, nor will it be possible to find that many people with the required skill set.

2. Regression tests.

Regression test are repetitive in nature These test cases are executed multiple times during the Product development phase.

* Given the repetitive nature of the test cases, automation will save significant time and effort in the long run.

3. Functional tests:

* These kinds of tests may require a complex set up and thus require specialized skill, which may not be available on an ongoing basis.

* Automating these once, using the expert skill sets, can enable using less-skilled people to run these tests on an ongoing basis.

Automating Areas less prone to change:

* In a product scenario, the changes in requirements are quite common.

* In such situation what to automate is easy to answer.

* Automation should consider those areas where requirements go through lesses or no changes.

* Normally change in requirements cause scenario and new features to be impacted, not the basic functionality of the product

Automate Tests that Pertain to standards

* one of the tests that products may have to undergo is compliance to standards.

Eg A product providing a JDBC interface should satisfy the standard IDBC tests.

* These tests undergo relatively less change. Even if they do change, they provide backend compatibility by which automated scripts will continue to run.

* Automating for standards provides a dual advantage. Test suites developed for standard are not only used for product testing but can also be sold as test tools for the market.

* A large number of tools available in the commercial market were internally developed for in-house usage.

* Hence, automating for standards creates new opportunities for them to be sold as commercial tools.

* Testing for Standards have certain legal and Organization Requirements.

* To Certify the software or hardware, a test suite is developed and handed over to different companies.

* The certification suites are executed every time by the supporting organization before the release of software and hardware.

* This is called Certification testing.

Management Aspects in Automation

* what to automate it takes into account the technical and management aspects as well as the long-term vision.

* Adequate effort has to be spent to obtain management commitment

* The automated test cases need to be maintained till the product reaches obsolescence.

* Automation involves effort over an extended period of time, management permissions are only given in phases and Part by part.

* Hence automation effort should focus on those areas for which management commitment exists already.

* Return on investment is another aspect to be considered seriously.

* Effort estimates for automation should give a clear indication to the management on the expected return on investment.

Design and Architecture for Automation:

* External Modules
* Scenario and configuration file modules
* Test cases and test framework modules
* TOOLS and results modules
* Report generator and reports/metrics modules

External Modules:

* There are two modules that are external modules to automation TCDB and defect DB, all the test cases, the steps to execute them and the history of their execution are stored in the TCDB.

* The test cases, in TCDB, can be manual or automated.

* The interface shown by thick arrows represents the interaction between TCDB and the automation framework only for automated test cases.

* Manual Test cases do not need any interaction between the framework TCDB.

* Defect DB on defect database or deffect repository contains details of all the defects that are found in various products that are tested in a particular organization.

* It contains defects and all the related information

(ie) when the defect was found, to whom it is assigned, what is the current status, the type of defects, its impact & so on.

* Design and architecture is an important aspect of automation.

* As in product development, the design has to represent all requirements in modules.

* Architecture for test automation involves two major heads

i) a test infrastructure that covers a test case database

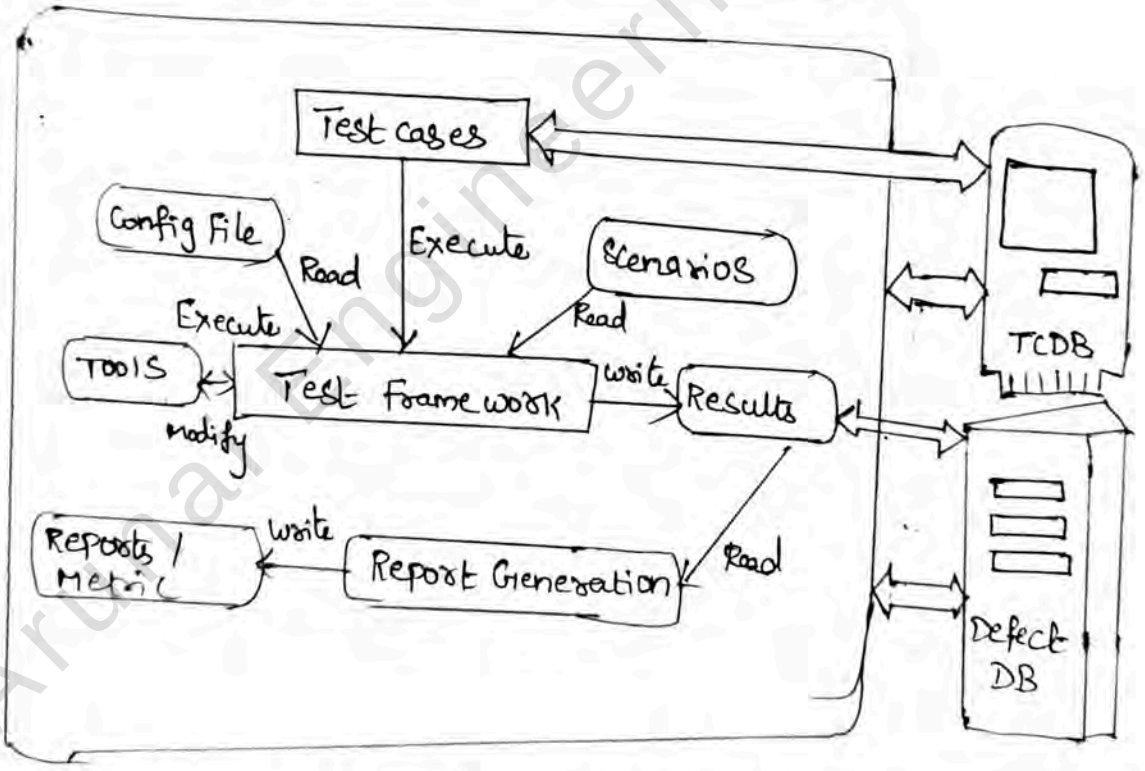
ii) a defect database or defect repository.

* Using this infastructure, the test framework provides a backbone that ties the selection and execution of test cases.

* For automated test cases, the framework can automatically submit the defects to the defect DB during execution.

* These external modules can be accessed by any module in automation framework not just one or more modules.

* The light shaded thick arrows (with lines) show specific interactions and dark shaded arrows (with lines) show multiple interactions.



Components of test Automation

* Thin arrows - internal interfaces & Direction of flow
* Thick arrows - External interfaces

Scenario and Configuration file modules

* Scenarios are nothing but information on "how to execute a particular test case".

* A configuration file contains a set of variables that are used in automation.

* The variables could be for the test framework or for other modules in automation such as tools and metrics or for the test suite or for a set of test cases or for a particular test case

* A configuration file is important for running the test cases for various execution conditions and for running the tests for various input and output conditions and states.

* The values of variables in this configuration file can be changed dynamically to achieve different execution, input, output and state conditions.

Requirements for a Test Tool:

Requirement-1:

No hard coding in the test suite

* Keep all variables separately in a file. Test suite variables are called "configuration variables."

* The file in which all variables names and their associated values are kept is called "configuration files."

Requirement-2:

Test case/suite Expandability

* In a product scenario involving several release and several test cycle and defect fixes

* The test cases go through large amount of changes and additionally then one situations for the new test cases to be added to the test suite

To Summarize:

1) Adding a test case should not affect other test cases.

2) Adding a test case should not result in retesting the complete test suite.

3) Adding a new test suite to the framework should not affect existing test suites.

Requirement 3:

* Reuse of code for different types of testing, test cases.

* The functionality of the product when subjected to different scenarios becomes test cases for different types of testing.

* The reuse of code applicable for modules with in automation.

* The test suite should only do what a test is expected to do. The test framework needs to take care of "how", and the test programs need to be modular to encourage reuse of code.

Requirement - 4 Automatic Setup & Clean up

* The test cases may expect some objects to be created or certain portions of the product to be configured in a particular way.

* The reuse of code

* Each test programs should have a "setup" program that will create the necessary setup before executing the test cases.

* Negative test cases are involved by "Undo" Setup.

* "CleanUP" program invokes after test execution for a test case is over.

Requirement-5 Independent test cases

* TO execute a particular test case, it should not expect any other test case to have been executed before not should it implicitly assume that certain other test case will be run after it.

* Each tests cases should be executed alone.

Test case -1 execution $\not\Longleftrightarrow$ Test case-2 Execution

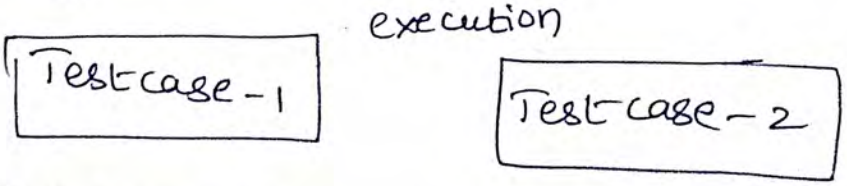
NOT depends on

Requirement - 6 : Test case Dependency

* Making a test case dependent on an other makes it necessary for a particular test case to be executed before or after a dependent test case is selected for execution.

* The test case depends on other test case dynamically.

Eg



Test case 2 is depends on test case1 execution

Requirement 7: Insulating test cases durin Execution

* some events or interrupts or signals in the system that may affect the execution.
* Test suite provide pop up screens during execution
* Unforeseen events to be block by the framework
* Framework specify what events can affect the test suite and what should not.

Requirement 8: coding standards and directory structure.

* coding standards and proper directory structure for a test suite may help the new engineers in understanding the test suite fast and help in maintaining the test suite

Requirement 9: Selective execution of test cases

* A framework may have multiple test suites; A test suite may have multiple test programs and a test program may have multiple test cases.

Requirement 10: Random execution of test cases:

The same test engineer may some time need to select a test case randomly from a list of test cases. Giving a set of test cases and expecting the test tool to select the test case is called random execution of test cases.

Requirement 11: Parallel execution of test cases:

* There are certain defects which can be unearthed if some of the test cases are run at the same time.

* In a multi-tasking and multiprocessing operating systems it is possible to make several instances of the tests and make them run in parallel.

Requirement 12: Looping the test cases.

* Reliability testing requires the test cases to be executed in loop.

↳ Two types of loops that are available.

i) Iteration loop - which gives the number of iterations of the particular test case to be executed.

ii) Timed loop - which keeps executing the test case to be executed in a loop till the specified time duration is reached.

Requirement 13: Grouping of Test Scenarios

* The group scenarios allow the selected test cases to be executed in order random, in a loop all at the same time.

* The grouping of scenarios allows several tests to be executed in a predetermined combination of scenarios.

Requirement 14: Test case execution based on Previous results

* Automation may not be of much help if the previous results of test execution are not considered for the choice of testing.

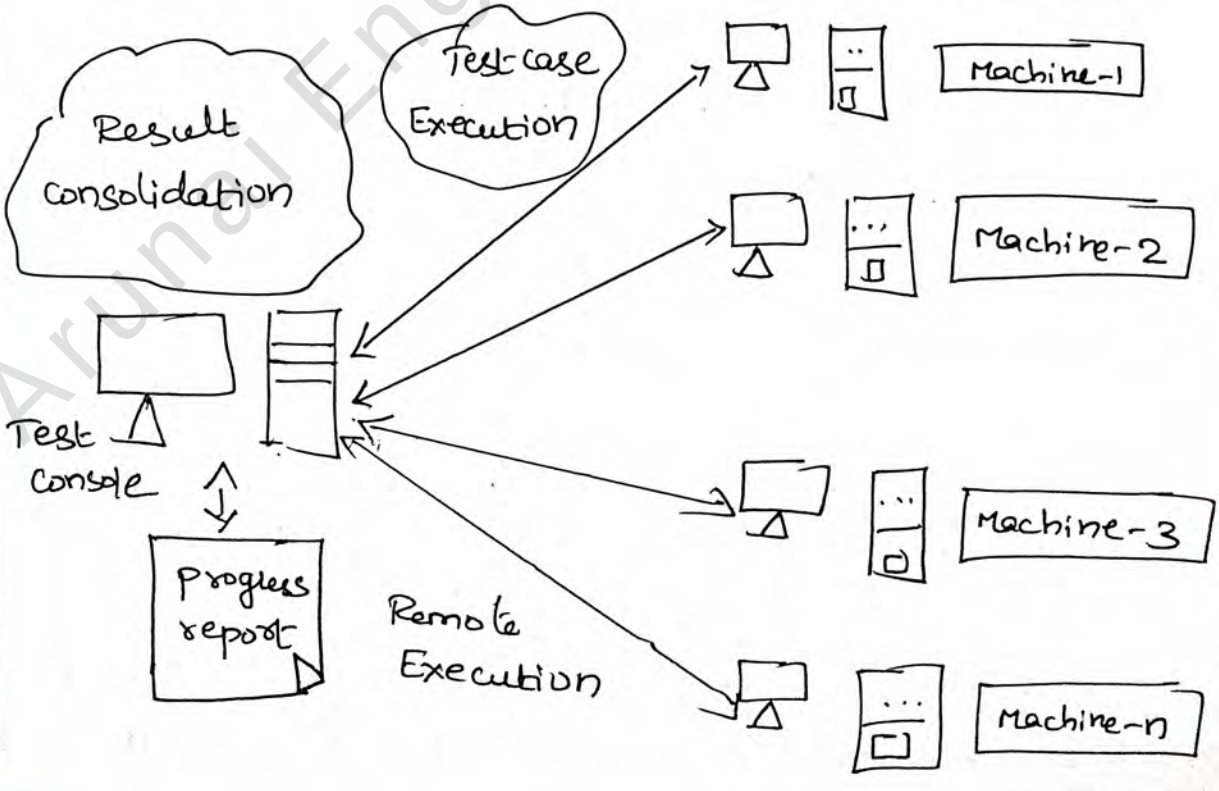
* Not only for regressing testing, it is a requirement for various other types of testing also.

1) Return all test cases which were executed previously.

2) Resume the test cases from where they were stopped the previous time.

3) Rerun only failed/not run test cases

Requirement 15: Remote execution of test cases.



* The Central machine that allocates tests to multiple machines and co-ordinates the execution and result is called "test console" or "test monitor".

1) It should be possible to execute/stop the test suite on any machine/set of machine from the test console.

2) The test results and logs can be collected from the test console.

3) The progress of testing can be found from the test console.

Requirement 16: Automatic archival of test data

* Archival of test data must include

1) what configuration variables were used.

2) what scenario was used and

3) what programs were executed and from what path.

Requirement 17: Reporting scheme

* Meaningful reports can be extracted every time

* Develop "dynamic" reports

* Store all information related to test cases in the result files

* "Audit logs" - store detailed information for each operations.

Reporting Scheme must include

1. Framework Scenario, test suite, test program starting/ completion.
2. Log messages
3. Category/ log of events
4. Audit reports

Requirements -18 Independent of Languages

* Framework (or) test tool provide choice of languages and script that are popular in the Software development area.
* Test scripts are written in Particular languages.

Requirement-19 Portability of Different platforms.

* The framework/ interface support on all platforms.
* The test tool / suite compulsory provide portability of different platforms.
* The language/ script to be selected carefully depends on platforms.
* Don't write platform specific calls.

Challenges in Automation:-

* Automation should not be viewed as a panacea for all problems nor should it be perceived as a quick-fix solution for all the quality problems in a product.

* Management commitment is an important challenge of test automation.

* It takes time/effort and pays off in the long run.

* It needs initial outlay of money as well as a steep learning curve for test engineers before It can start paying off.

* Management should hold patience and persist with the automation.

* The main challenge is because of the heavy front loading of costs of test automation and managements starts to look for an easy pay back.

Test cases and Test framework Modules

* A test case means the automated test cases that are taken from TCDN and executed by the framework.

* Test case is an object for execution for other modules in the architecture and does not represent any interaction by itself.

* A test framework is a module that combines "what to execute" and "how they have to executed".

* It picks up the specific test cases that are automated from TCDB and picks up the scenarios and executes them.

* A test framework is considered the core of automation design. It subjects the test cases to different scenarios.

Eg: If there is a scenario that requests a particular test case be executed for 48 hours in a loop

* The test framework executes those test cases in the loop and times out when the duration is met.

Reporting scheme must include

1. Framework scenario, test suite, test program starting/ completion.
2. Log messages
3. Category/ log of events
4. Audit reports

Requirements -18 Independent of Languages

* Framework (or) test tool provide choice of languages and script that are popular in the software development area.
* Test scripts are written in particular languages.

Requirement-19 Portability of Different platforms.

* The framework/ interface support on all platforms.
* The test tool / suite compulsory provide portability of different platforms.
* The language/ script to be selected carefully depends on platforms.
* Don't write platform specific calls.

* The test framework contains the main logic for interacting, initiating and controlling all modules.

* A test framework can be developed by the organization internally or can be bought from the vendor.

Tools and Results Modules

* when a test framework Performs its operation these are a set of tools that may be required.

Eg: when test cases are stored as source code file in TCDB they need to be extracted and compiled by build tools.

* In order to run the compiled code certain runtime tools and utilities may be required.

Eg: IP Packets Simulators or user login Simulators or machine simulators may be needed.

* when a test framework executes a set of test cases with a set of scenarios for the different values provided by the configuration file

* The results for each of the test case along with scenarios and variable values have to be stored for future analysis and action

Report Generator and Report/Metrics Modules

* once the results of a test run are available, the next step is to prepare the test reports and metrics

* Preparing reports is complex and time-consuming effort and hence it should be part of the automation design.

* There should be customized reports such as executive report, which gives very high level status technical reports, which gives a moderate level of detail of the tests run.

* Detailed or debug reports which are generated for developers to debug the failed tests cases and the product.

* The periodicity of the reports is different such as daily, weekly, monthly and milestone reports, having reports of different levels of detail and different periodicities can address the need of multiple constituents and provide significant returns.

* The module that takes the necessary inputs and prepares a formatted report is called a report generator.

* All the reports and metrics that are generated are stored in the reports/metrics module of automation for future use and analysis.

What are Metrics and Measurements:

* Metrics derive information from raw data with a view to help indecision making

* Some of the areas that such information would shed light on are

1) Relationship between the data points

2) Any cause and effect correlation between the observed data points

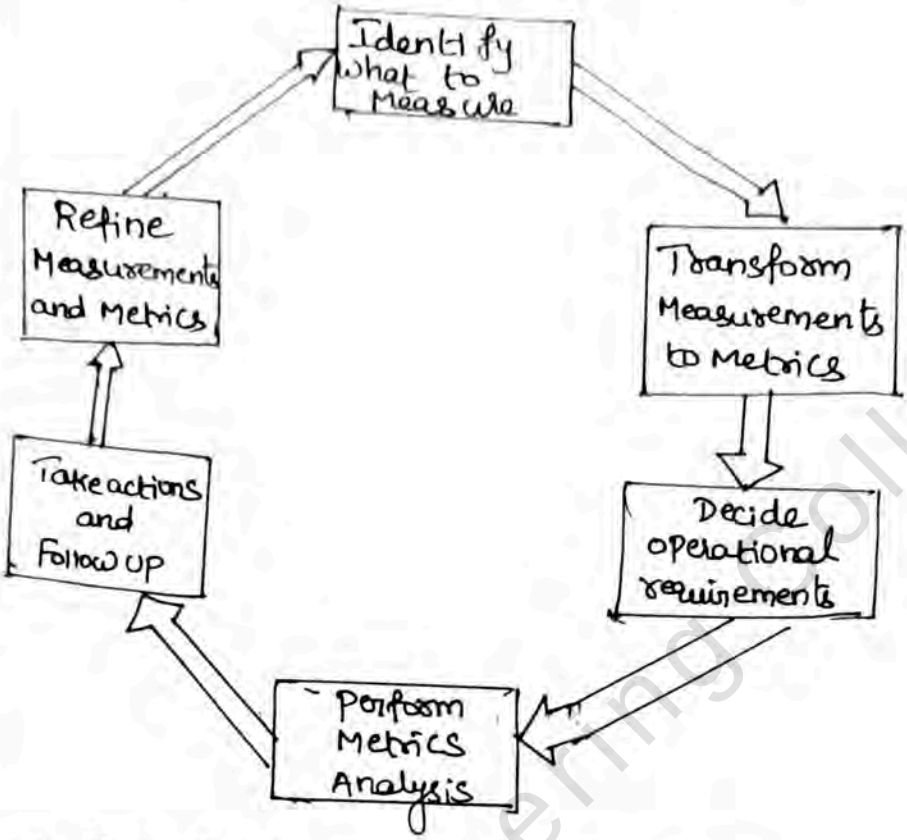
3) Any pointers to how the data can be used for future planning and continuous improvements.

* The metrics and analysis of metrics may convey the reason when data points are combined.

* Relating several data points and consolidating the result in terms of charts and pictures simplifies the analysis and facilities use of metrics for decision making.

* Collecting and analyzing metrics involves effort and several steps

Steps in a Metrics program



* The first step involved in a metrics program is to decide what measurements are important and collect data accordingly.

Eg of Measurements, the effort spent on testing, number of defects and number of test cases.

While deciding what to measure the following aspect need to be kept in mind.

1) what is measured should be of relevance to what we are trying to achieve.

For testing functions, user obviously be interested in effort spent on testing number of test cases, ~~number of defects reported from test cases.~~

2) The entities measured should be natural and should not involve too many overheads for measurements.

3) What is measured should be at the right level of granularity to satisfy the objective for which the measurement is being made.

Data Drilling:

* The level of granularity of data obtained depends on the level of detail required by a specific audience.

* The measurements and the metrics derived from them will have to be at different levels for different people.

* An approach involved in getting the granular detail is called data drilling.

Why Metrics in Testing:

* Metrics are needed to know test case execution productivity and to estimate test completion date.

* The defect trend collected over a period of time gives a rough estimate of the defects that will come through future test cycles

* The defect fixing trend collected over a period of time gives another estimate of the defect-fixing capability of the team.

* The defect fixes may arrive after the regular test cycles are completed.

* These defect fixes will have to be verified by regressing testing before the product can be released.

* The measurements collected during the development and test cycle are not only used for release but also used for post-release activities.

* Looking at the defect trend for a period help in arriving at approximate estimates for the number of defects that may get reported post release.

* Metrics and their analysis help in preventing the defects proactively, saving cost & effort

* Metrics are used in resource management to identify the right size of product development teams.

* Metrics in testing help in Identifying
↳ When to make the release ↳ What to release
↳ whether the product is being released with known quality

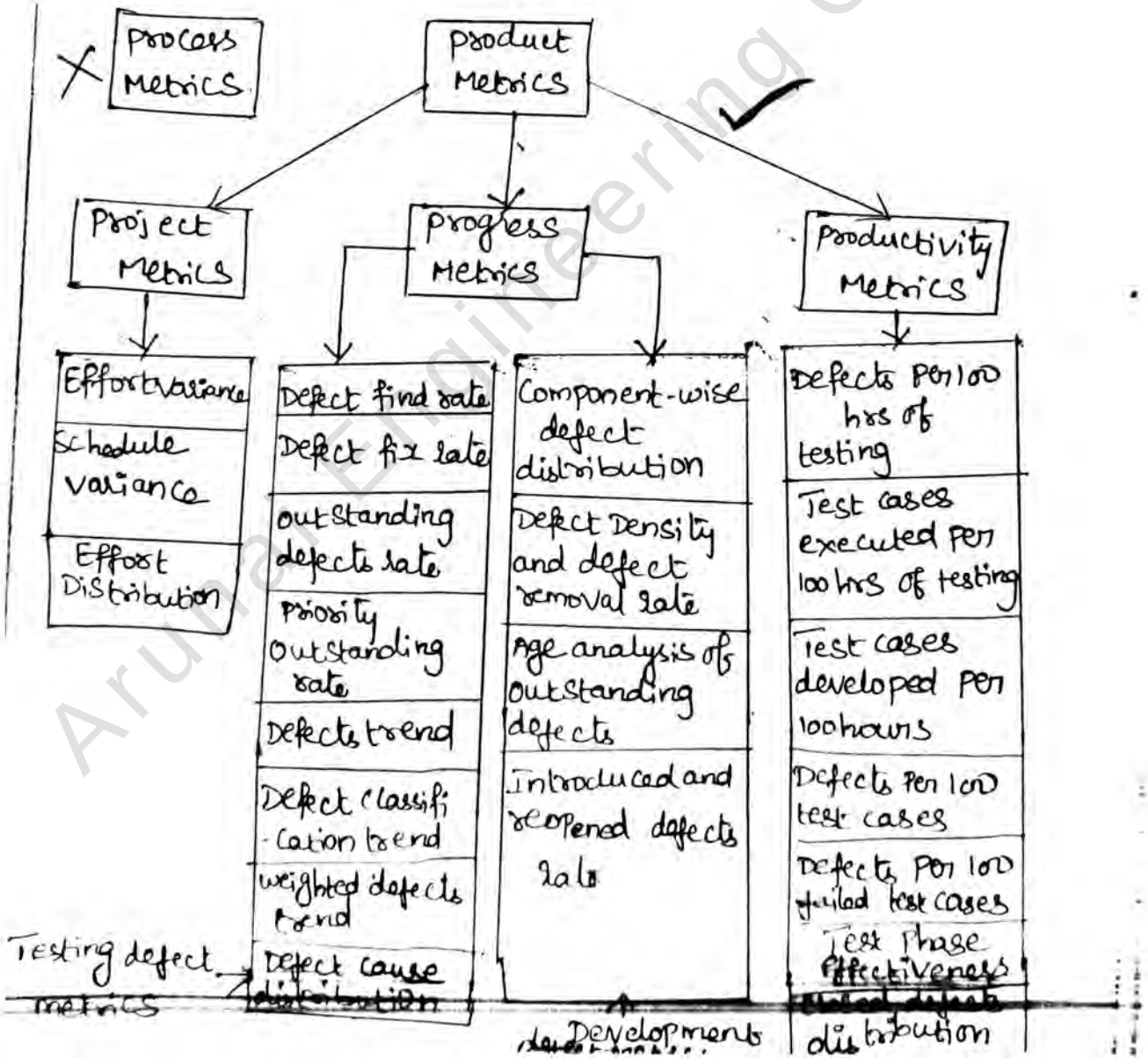
Types of Metrics:

*Metrics can be classified into different types based on what they measure and what area they focus on.

*Metrics can be classified as

i) product Metrics

ii) process Metrics



Product metrics can be further classified as

1. Project metrics:

* A set of metrics that indicates how the project is planned and executed.

2. Progress metrics:

* A set of metrics that tracks how the different activities of the project are progressing.

* The activities include both development activities and testing activities.

* progress metrics is monitored during testing phases.

* Progress metrics helps in finding out the status of test activities and they are also good indicators of product quality.

* The defects that emerge from testing provide a wealth of information that help both development team and test team to analyze and improve.

* Progress metrics, for convenience, is further classified into test defect metrics and development defect metrics.

3. Productivity metrics:

* A set of metrics that takes into account various productivity numbers that can be collected and used for planning and tracking testing activities.

* These metrics help in planning and estimating of testing activities.

Project Metrics:

* A typical project starts with requirements gathering and ends with product release.

* All the phases that fall in between these points need to be planned and tracked.

* In the planning cycle, the scope of the project is finalized.

* The project scope gets translated to effort estimate for each of the phases and activities by using the available productivity data available.

* This initial effort is called baselined effort

* As the Project progresses and if the scope of the project changes or if the available productivity numbers are not correct, then the effort estimates are re evaluated again and this re-evaluated effort estimate is called revised effort.

The basic measurements that are very natural, simple to capture and form the inputs to the metrics in this section are

1) The different activities and the initial baselined effort and schedule for each of the activities this is input the beginning of the project/phase.

2) Calculating effort variance for each of the phase (as calculated by the formula below) provides a quantitative measure of the relative difference between the revised and actual effort.

3) The actual effort and time taken for the various activities, this is entered as and when the activities take place.

4) The revised estimate of effort and schedule these are re-calculated at appropriate times in the project life.

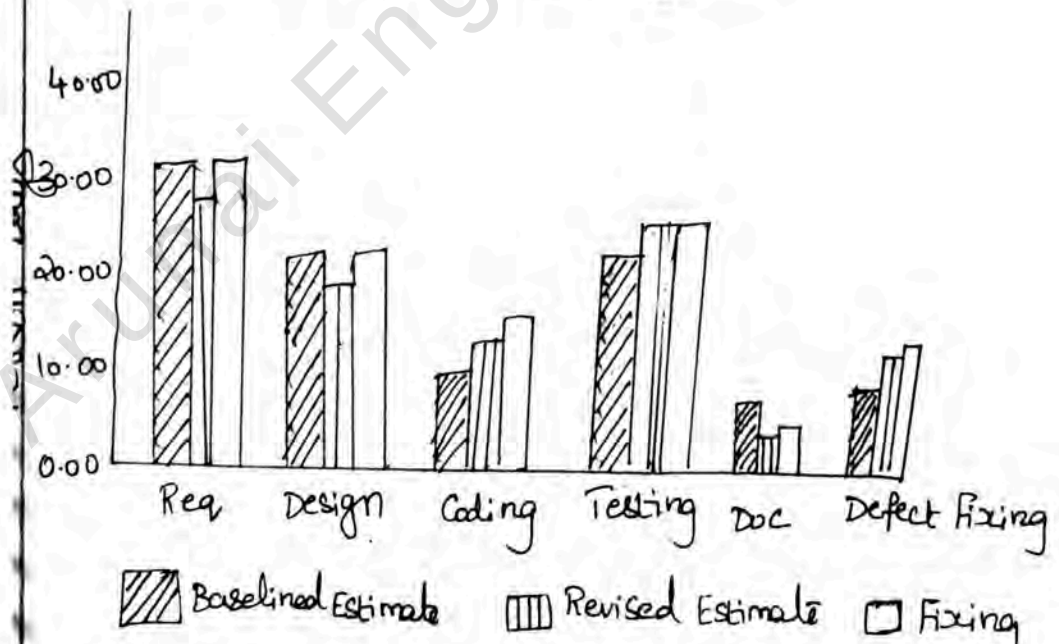
Effort Variance (Planned Vs Actual)

* When the baselined effort estimates, revised effort estimates and actual effort are plotted together for all the phases of SDLC, it provides many insights about the estimation process.

* As different set of people may get involved in different phases, it is a good idea to plot these effort numbers phase-wise.

* If there is a substantial difference between the baselined and revised effort, it points to incorrect initial estimation.

Sample variance percentage by phase



Phase-wise Effort Variation

Variance % = [(Actual effort - Revised Estimate)/Revised Estimate] * 100

Effort	Req	Design	Coding	Testing	DOC	Defect Fixing
Variance %	7.1	8.7	5	0	40	15

* If this is the case, the right parameter for variance calculation is the baselined estimate.

* In this case analysis should point out the problems in the revised estimation process.

Schedule Variance (Planned Vs Actual)

* Most software projects are not only concerned about the variance in effort, but are also concerned about meeting schedules.

* This leads us to the schedule variance metric.

* Schedule variance, life effort variance is the deviation of the actual schedule from the estimated schedule.

* There is one difference, depending on the SDLC model used by the project, several phases could be active at the same time.

* Further the different Phases in SDLC are interrelated and could share the same set of individuals

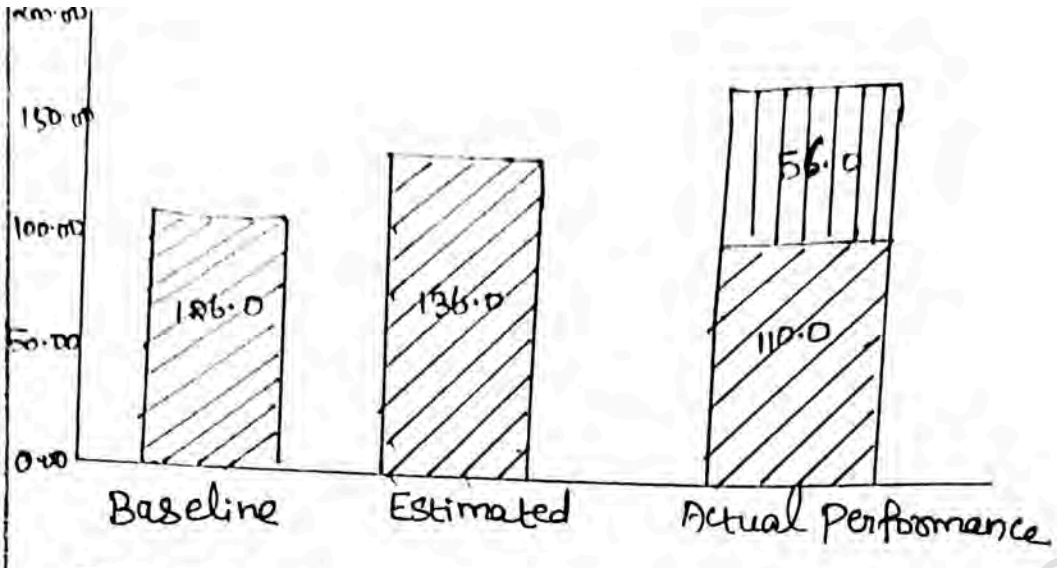
* Because of all these complexities involved, Schedule variance is calculated only at the overall project level, at specific milestones, not with respect to each of the SDLC phases

* Using the data in the chart, the Variance Percent can be calculated using a similar formula.

* Considering the estimated schedule and actual schedule.

* Schedule variance is calculated at the end of every milestone to find out how well the project is doing with respect to the schedule.

* To get a real picture on schedule in the middle of project execution, it is important to calculate "remaining days yet to be Spent" on the project and plot it along with the "actual schedule spent".



Schedule Variance

Interpretation of ranges of effort and schedule variation

Effort Variance	Schedule Variance	Probable cause/ result
Zero or acceptable variance	Zero variance	A well-executed project
Zero or acceptable variance	Acceptable variance	Need slight improvement in effort/ schedule estimation
unacceptable variance	zero or acceptable variance	Under estimation, needs further analysis
unacceptable variance	unacceptable variance	under estimation of both effort and schedule
Negative variance	Zero or acceptable variance	over estimation and schedule both effort and schedule estimation need improvement
Negative variance	Negative variance	over estimation and over schedule both effort and schedule estimation need improvement

* Remaining days yet to be spent can be calculated by adding up all remaining activities.

* If the remaining days yet to be spent on project is not calculated and plotted, it does not give any value to the chart in the middle of the project, because the deviation cannot be inferred visually from the chart.

* The remaining days in the schedule becomes zero when the release is met.

Effort Distribution Across Phases:

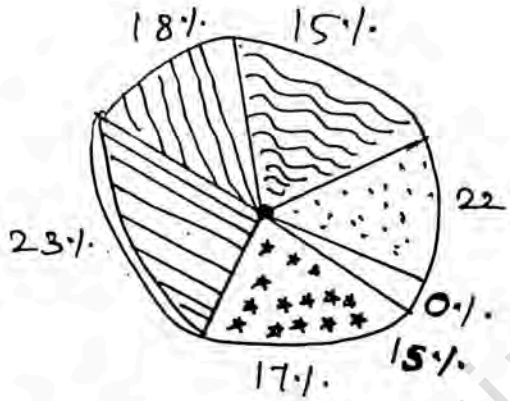
* Adequate and appropriate effort needs to be spent in each of the SDLC phase for a quality product release.

* Variance calculation helps in finding out whether commitments are met on time and whether the estimation method works well.

* In addition, some indications on product quality can be obtained if the effort distribution across the various phases are captured and analyzed.

Eg1: Spending very little effort on requirements may lead to frequent changes but one should also leave sufficient time for development and testing Phase.

a. Spending less effort in testing may cause defects to crop up in the customer's place but spending more time in testing than what is needed may make the product lose the market window.



* Mature organizations spent at least 10-15% of the total effort in requirements and approximately the same effort in the design phase.

* The effort percentage for testing depends on the type of release and amount of change to the existing code base and functionality.

* Typically, organizations spend about 20-50% of their total effort in testing.

Progress Metrics:

* Any project needs to be tracked from two angles

1. How well the project is doing with respect to effort and schedule.

2. Equally important angle is to find out how well the product is meeting the quality requirements for the release.

* Defects get detected by the testing team and get fixed by the development team.

* Defect metrics are further classified into test defect metrics - which help the testing team in analysis of product quality and testing.

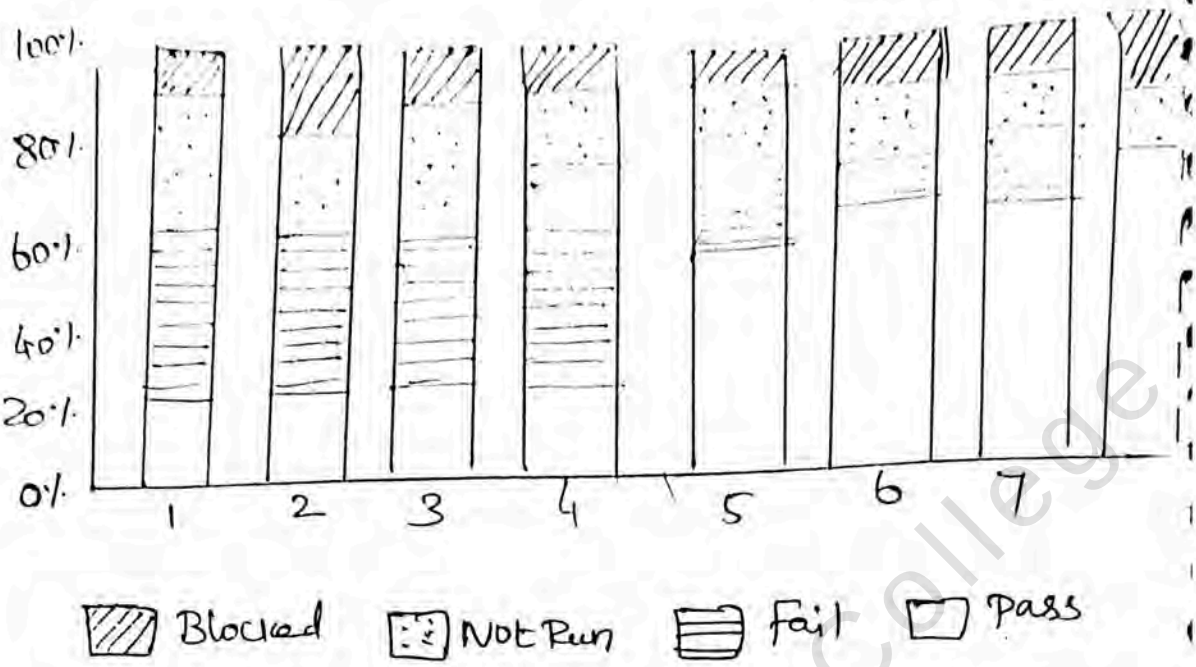
* Development defect metrics - which help the development team in analysis of development activities

Two parameters that determine product quality and its assessment

1. How many defects have already been found

2. How many more defects may get Unearthed.

* If only 50% of testing is complete and if 100 defects are found, assuming that the defects are Uniformly distributed over the product



Progress of Test Case Execution

* The progress chart gives the pass rate and fail rate of executed test cases, pending test cases and test cases that are waiting for defects to be fixed.

* Representing testing progress in this manner will make it is easy to understand the status and the further analysis.

* Another perspective from the chart is that the pass percentage increases and fail percentage decreases showing the positive progress of testing and product quality.

* A scenario represented by such a progress Chart shows that not only is testing progressing well, but also that the product quality is improving.

* If, on the other hand, the chart had shown a trend that as the weeks progress, the "not run" cases are not reducing in number or "blocked" cases are increasing in number or "pass" cases are not increasing

* It would clearly point to quality problems in the product that prevent the product from being ready for release.

Test Defect metrics:

* The next set of metrics help us understand how the defects that are found can be used to improve testing and product quality.

* Some organizations classify effects by assigning a defect priority Eg P_1, P_2, P_3 & so on.

* The priority of a defect provides a management Perspective for the order of defect fixes.

Eg A defect with priority P_1 indicates that it should be fixed before another defect with priority P_2.

* Some organizations use defect severity levels Eg S1, S2, S3 & soon.

* The Severity of defects provides the test team a perspective of the impact of the defect in product functionality.

Eg A Defect with severity level

S1 = Either the major functionality is not working or the software is crashing

S2 = Mean a failure or functionality not working

Defect Priority and Defect Severity - Sample Interpre-tation;

Priority	what it means
1	Fix the defect on highest priority Fix it before the next build
2	Fix the defect on high priority before next testcyc
3	Fix the defect on moderate priority when time permits, before release
4	Postpone this defect for next Release or live with this defect

Severity	what it means
1.	The basic Product functionality Failing or product crashes
2.	Unexpected error condition or a functionality not working
3.	A minor functionality is failing or behaves differently than expected.
4	Cosmetic issue and no impact on the issues

A common defect definition and classificatio

Defect classification	what it means
Extreme	Product crashes or unusable Needs to be fixed immediately
Critical	Basic functionality of the product not working. Needs to be fixed before next test cycle Starts
Important	Extended functionality of the product networking. Does not affect the progress of testing Fix it before the release.
Minor	Product behaves differently No impact on the test team or customers
Casmetic	Minor hesitant Need not be fixed for this release

Defect Find Rate:

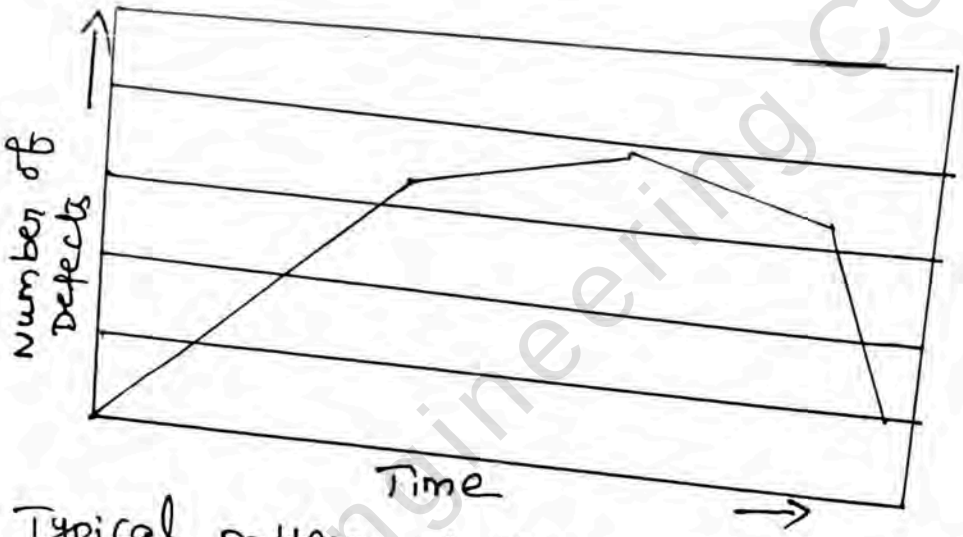
* The purpose of testing is to find defects early in the test cycle.

* when tracking and plotting the total number of defects found in the products at regular intervals (daily or weekly) from beginning, to end of a product development cycle, it may show a pattern for defect arrival

* The idea of testing is to find as many defects as possible early in the cycle.

* Once a majority of the modules become available and the defects that are blocking the tests are fixed.

* The defect arrival rate increases. After a certain period of defect fixing and testing, the arrival of defects tends to slow down and a continuation of that trend enables product release.



Typical pattern finding defects in a product

Defect Fix Rate:

* The purpose of development is to fix defects as soon as they arrive.

* If the goal of testing is to find defects as early as possible, it is natural to expect that the goal of development should be to fix defects as soon as they arrive.

* If the defect fixing curve is in line with defect arrival a "bell curve" as shown above figure.

Outstanding Defects Rate

* The number of defects outstanding in the product is calculated by subtracting the total defects fixed from the total defects found in the product

* In a well-executed project, the number of outstanding defects is very close to zero all the time during the test cycle.

* The defects need to be fixed as soon as they arrive and defects arrive in the pattern of bell curve.

* If the defect fixing pattern is constant like a straight line, the outstanding defects will result in a ball curve again

Priority Outstanding Rate:

* Having an eye on the find rate, fix rate and outstanding defects are no enough to give an idea of the sheer quantity of defects

* The modification to the outstanding defects rate curve by plotting only the high priority defects and filtering out the low-priority effects is called priority outstanding defects

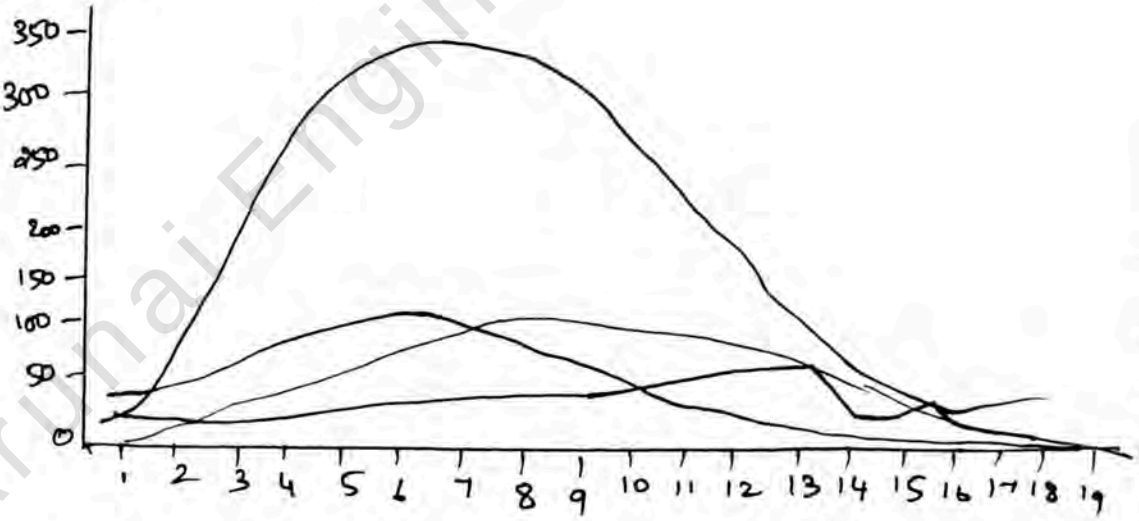
i) High priority defects - Require a change in design or architecture.

If they are found late in the cycle, the release may get delayed to address the defect.

ii) Low-priority defect - It is close to the release date and it requires a design change a likely decision of the management would be not to fix that defect.

Defect Trend:

* The effectiveness of analysis increases when several perspective of find rate, fix rate, outstanding, and prior outstanding effects are combined.



- Defect find rate
- Defect fix rate
- Outstanding defects
- Priority outstanding

Defect Trend

1) The find rate, fix rate, outstanding defects and Priority outstanding follow a bell curve pattern indicating readiness for release at the end of the 19th week.

2) A sudden downward movement as well as upward spike in defect fixes rate needs analysis.

3) There are close to 75 outstanding defects at the end of the 19th week.

* By looking at the priority outstanding which shows close to zero defects in the 19th week, it can be concluded that all outstanding defects need analysis before the release.

4) Defect fix rate is not in line with outstanding defect rate.

* If defect fix rate had been improved, it would have enabled a quicker release cycle (reduced the schedule by four to five weeks) as incoming defects from the 14 weeks were in control.

5) Defect fix rate was not at the same degree of defect find rate. Find rate was more than the fix rate till the 10th week.

* Making find rate and fix rate equal to each other would have avoided the outstanding defects peaking from the 4th to 10th week.

6) A Smooth priority outstanding rate suggest that priority defects were closely tracked and fixed.

Defect classification Trend:

* Providing the perspective of defect classification in the chart helps in finding out release readiness of the product.

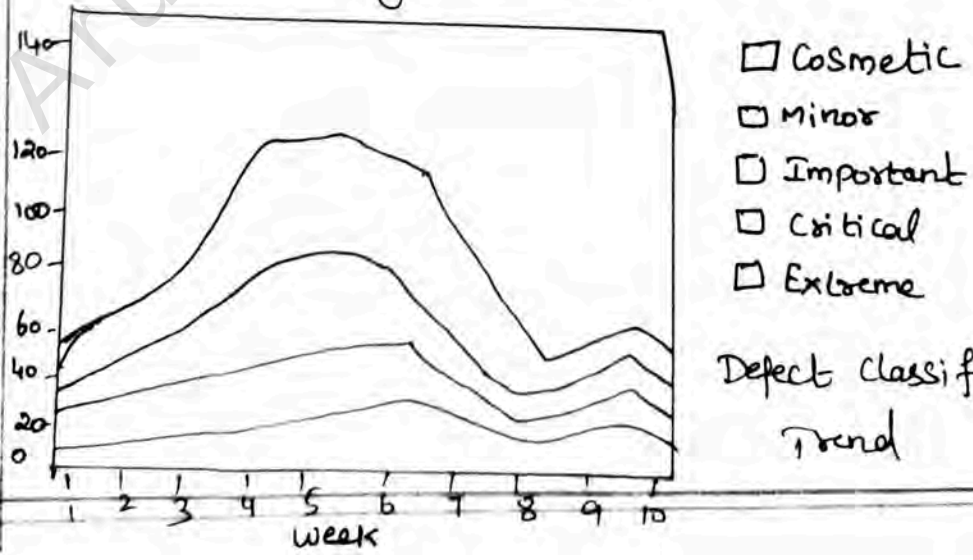
* Some of the data drilling or chart analysis needs further information on defects with respect to each classification of defects, extreme, critical, important, minor and cosmetic

* When talking about the total number of outstanding defects, some of the questions that can be asked are

✓ How many of them are extreme defects?

✓ How many are critical

✓ How many are important?



Defect classification Trend

Productivity Metrics:-

* Productivity metrics combine several measurements and parameters with effort spent on the product.

1) Estimating for the new release

2) Finding out how well the team is progressing, Understanding the reasons for (both positive and negative) variations in results.

3) Estimating the number of defects that can be found.

4) Estimating release date and quality

5) Estimating the cost involved in the release

Defects per 100 Hours of Testing:

* Program testing can only prove the presence of defects, never their absence.

* It is reasonable to conclude that there is no end to testing and more testing may reveal more new defects.

* If incoming defects in the product are reducing it may mean various things.

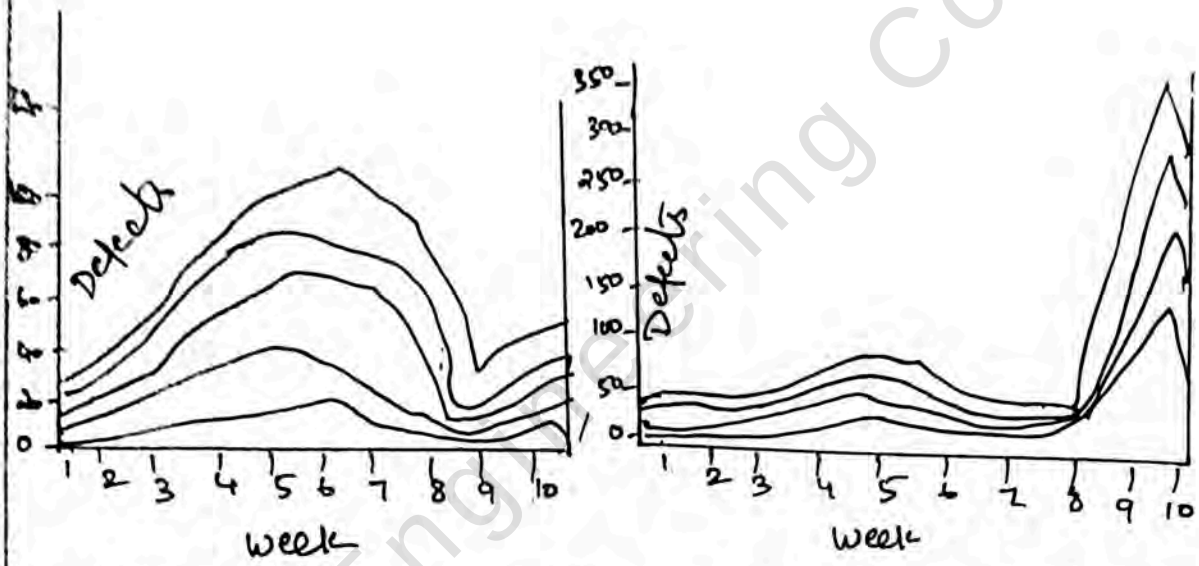
~~1. Testing is not effective~~

2) the quality of the product is improving
3) Effort spent in testing is falling.

Defects Per 100hours of testing

$$= \left(\frac{\text{total defects found in the product for a period}}{\text{total hours spent to get those defects}} \right) * 100$$

Defect Classification trend.



Defects per 100 hours of testing Defects per 100 hours of testin

* The chart produced a bell curve indicating readiness for the release.

Test cases executed Per 100hours of testing:

* The number of test cases executed by the test team for a particular duration depends on team productivity and quality of Products

Test cases executed per 100 hours of testing

$$= \left(\frac{\text{Total test cases executed for a period}}{\text{Total hours spent in test execution}}\right) * 100$$

Test cases developed per 100 hours of testing:

* Both manual execution of test cases and automating test cases require estimating and tracking of productivity numbers.

* The formula for test cases developed uses the count corresponding to added/modified and deleted test cases

Test cases developed per 100 hours of testing

$$= \left(\frac{\text{Total test cases developed for a period}}{\text{Total hours spent in test case development}}\right) * 100$$

Defects per 100 test cases:

* The goal of testing is to find out as many defects as possible, it is appropriate to measure the "defect yield" of tests (ie) how many defects get uncovered during testing.

* This is a function of two parameters

1) The effectiveness of the tests in uncovering defects

2) The effectiveness of choosing tests that are capable of uncovering defects

Defects Per 100 test cases

$$= \left(\frac{\text{Total defects found for a Period}}{\text{Total test cases executed for the same Period}} \right) * 100$$

Defects Per 100 failed test cases

* Defects per 100 failed test cases is a good measure to find out how granular the test cases are. It indicates

1) How many test cases need to be executed when a defect is fixed.

2) what defects need to be fixed so that an acceptable number of test cases reach the pass : State and

3) How the fail rate of test cases and defects affect each other for release readiness analysis

Defects Per 100 failed test cases

$$= \left(\frac{\text{Total defects found for a Period}}{\text{Total test cases failed due to those defects}} \right) * 100$$

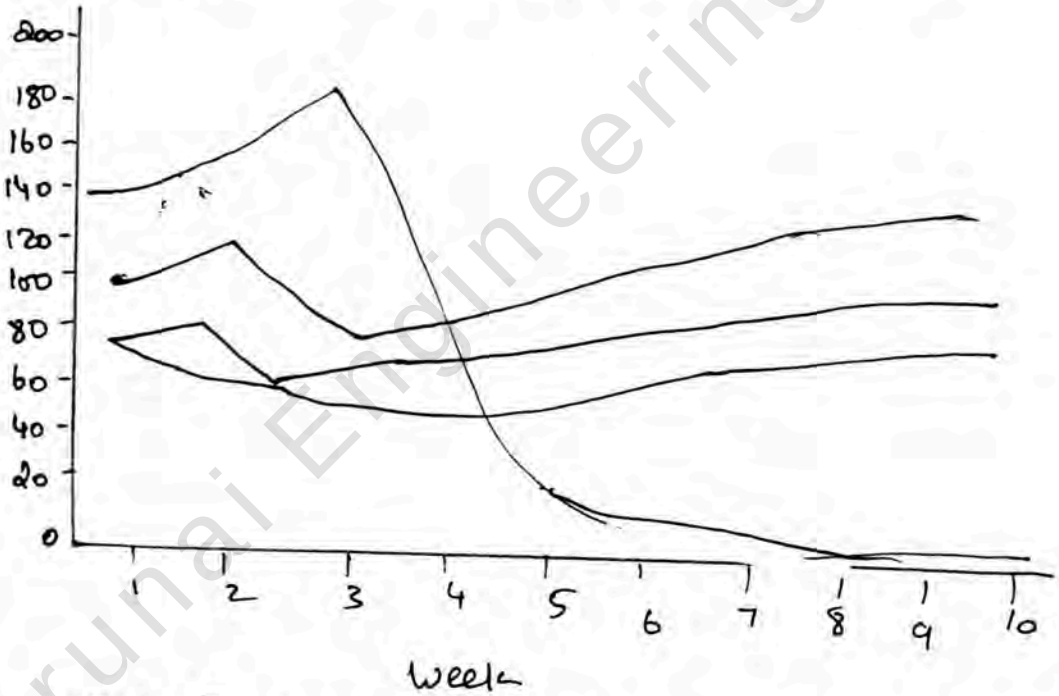
The following observations can be made by looking the chart

1) Defects Per 100 test cases showing a downward trend suggests product readiness or release

2) Test cases executed per 100 hours on upward trend suggests improved productivity and product quality (week 3 data needs analysis)

3) Test cases developed per 100 hours showing a slight upward movement suggests improved Productivity

4) Defects per 100 failed test cases in a band of 80-90 suggests equal number of test cases to be verified when defects are fixed.



- Defects per 100 test cases
- Test cases executed per 100 hours
- Defects per 100 failed test cases
- Test cases developed per 100 hours

Productivity Metrics

Types of Reviews:

* Reviews can be formal or informal. They can be technical or managerial.

* Managerial reviews usually focus on project management and project status

↳ Verify that Software artifact meets its specification

↳ To detect defects and

↳ check for compliance to standards

* The colleague requesting the review receives feedback about one or more attributes of the reviewed software artifact.

* Informal reviews are an important way for colleagues to communicate and get peer input with respect to their work.

Two major types of reviews

1) Inspections as a type of technical review

2) walkthrough

1). Inspections as a type of technical Review:

* Inspections are a type of review that is formal in nature and require preview preparation on the part of the review team.

* The responsibility for initating and carrying through the steps belongs to the inspection leader (or moderator) who is usually a member of the technica staff or the software quality assurance team.

* The inspection leader plans for the inspection, sets the date, invites the participants distribute the required documents, runs the inspection meeting, appoints a recorder to record results and monitors the follow up period after review.

* The inspection participants address each item on the checklist.

* The recorder records any discrepancies, misunderstanding, errors and ambiguities, in genera any problems associated with an item.

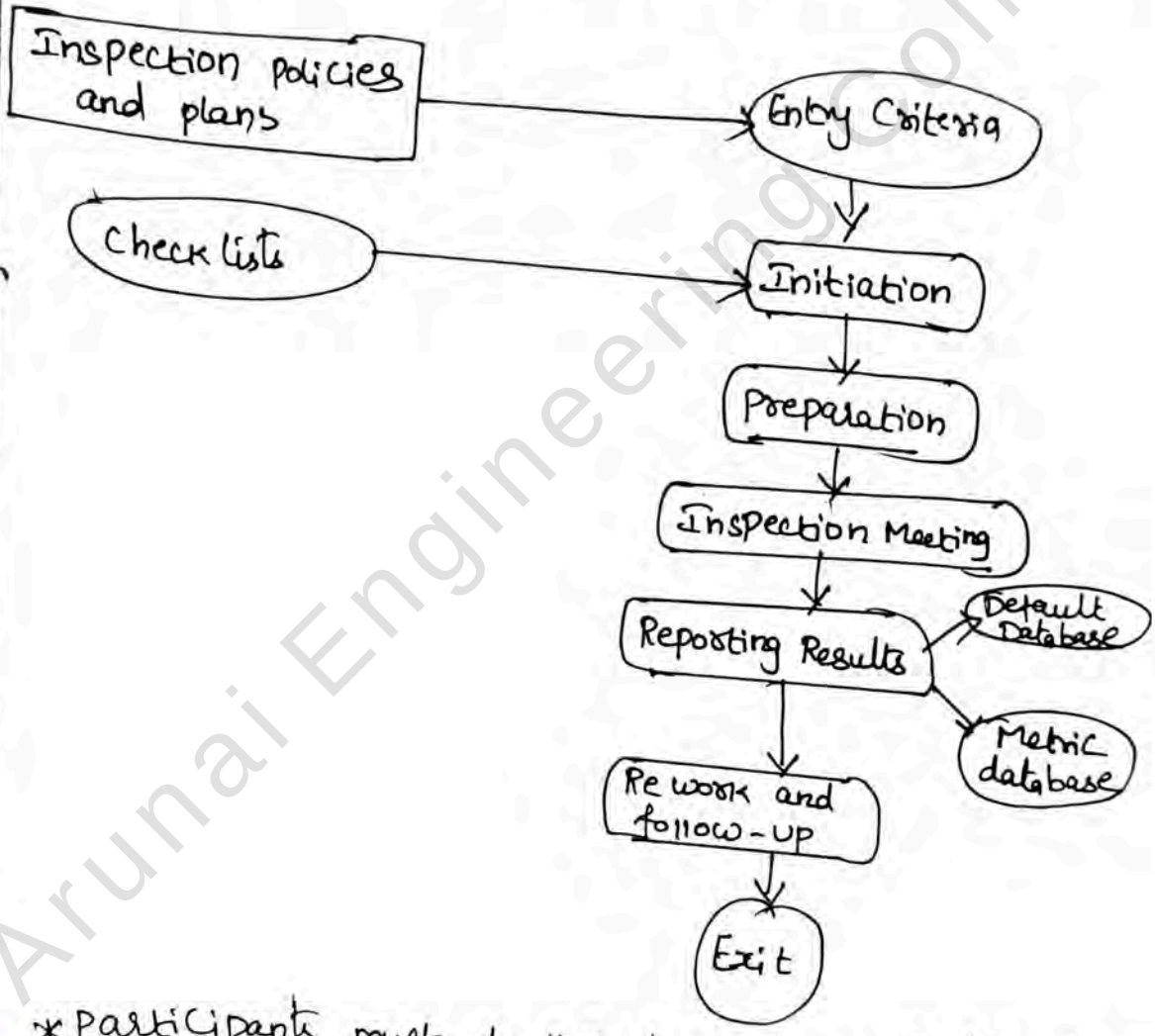
* The completed checklist is part of the review.

* The inspection process begins when inspection precondition are met as specified in the inspection policies, Procedures and plans.

* The inspection leader announces the inspection meeting and distributes the items to be inspected the checklist

* Any other auxiliary material to the participants Usually a day or two before the scheduled meeting

Steps in the Inspection process



* Participants must do their homework and study the items and the checklists

* Attention is paid to issue related to quality, adherence to standards, testability, traceability and satisfaction of the users/clients requirements

* The inspection process requires a formal follow-up process.

* Rework sessions should be scheduled as needed and monitored to ensure that all problems identified at the inspection meeting have been addressed and resolved.

* only when all problems have been resolved and the item is either re inspected by the group or the moderator is the inspection process completed.

Walkthrough as a Type of Technical Review:

* Walk through are a type of technical review where the producer of the reviewed material serves as the review leader and actually guides the progression of the review.

* walkthrough have traditionally been applied to design and code.

* The whole group "plays computer" to step through an execution lead by reader or presenter

* This is a good opportunity to "pretest" the design or code.

* If the presenter gives a skilled presentation of the material, the walkthrough participants are able to build a comprehensive mental model of the detailed design or code are able to both evaluate its quality and detect defects.

* walk through may be used for material other than code. Eg Data Descriptions, reference manuals or even specifications.