JEPPIAAR INSTITUTE OF TECHNOLOGY

“Self-Belief | Self Discipline | Self Respect”

DEPARTMENT

OF

COMPUTER SCIENCE AND ENGINEERING

LECTURE NOTES

IT8076 – SOFTWARE TESTING

(Regulation 2017)

Year/Semester: III/VI CSE

2020 – 2021

Prepared by

Ms. R. Revathi

Assistant Professor/CSE

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.

Software Testing : Introduction

IEEE Definition :

A process of analyzing a s/w item to detect the difference between existing and required conditions (error /defect / bugs) and to evaluate the features of the s/w item .

Definition 1:

Process of evaluating a system or its components with the intent to find whether it satisfies the specified requirements or not.

Definition 2:

an investigation conducted to provide stakeholders with information about the quality of the product or service.

Who is involved in Testing ?

Test Specialist

S/w Developer

Project leader Manager

End User

When to start the testing process?

Every Phase of SDLC

When to end the testing process?

When Testing Deadline comes

Completion of test case execution

Bug rate falls below certain level & no high priority bugs are identified

Management decision

Example : S/W Testing Automation Tools

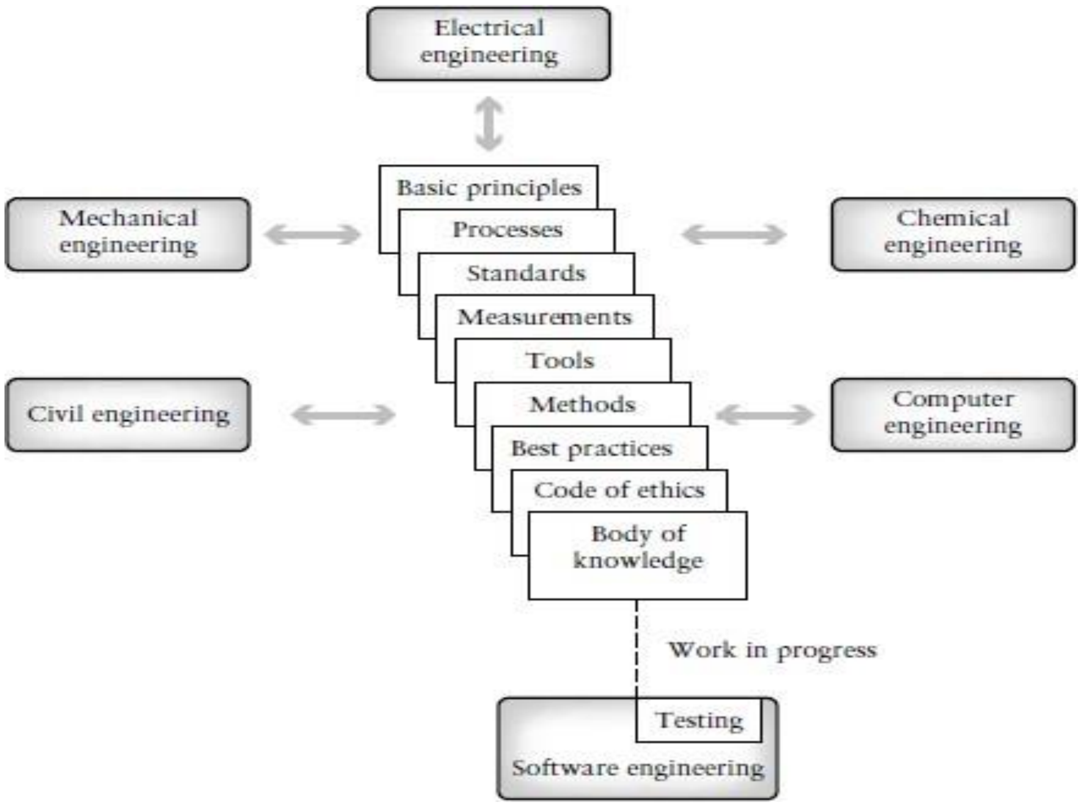
Telerik Test Studio , Selenium , TestComplete , HP Quick Test Professional , Silk Test, Win Runner ,Load Runner

Testing as an Engineering Activity:-

Poor quality of software that can cause loss of life or property is no longer acceptable to society .Failures can result in catastrophic losses. Conditions demand software development staffs with interest and training in the areas of software products and process quality. Highly qualified staffs ensures that software products are built on time, within budget and are of the highest quality with respect to attributes such as reliability, correctness, usability and ability to meet all user requirements.

The demand for high quality software's and the need for well educated software professionals there is a movement to change the way software is developed and maintained.

The profession of software engineering is slowly emerging as a formal engineering discipline .The movement towards this new profession is the focus of the entire November /December 1999 issues of *IEEE Software*.



The education and training of engineers in each engineering discipline is based on the teaching of related scientific principles, engineering processes, standards, methods, tools, measurements. The goal and task force team is to define a body of knowledge that covers the software engineering discipline, to discuss the nature of education for this new profession and to define a code of ethics for the software engineers

Using Engineering Approach to software development implies that:-

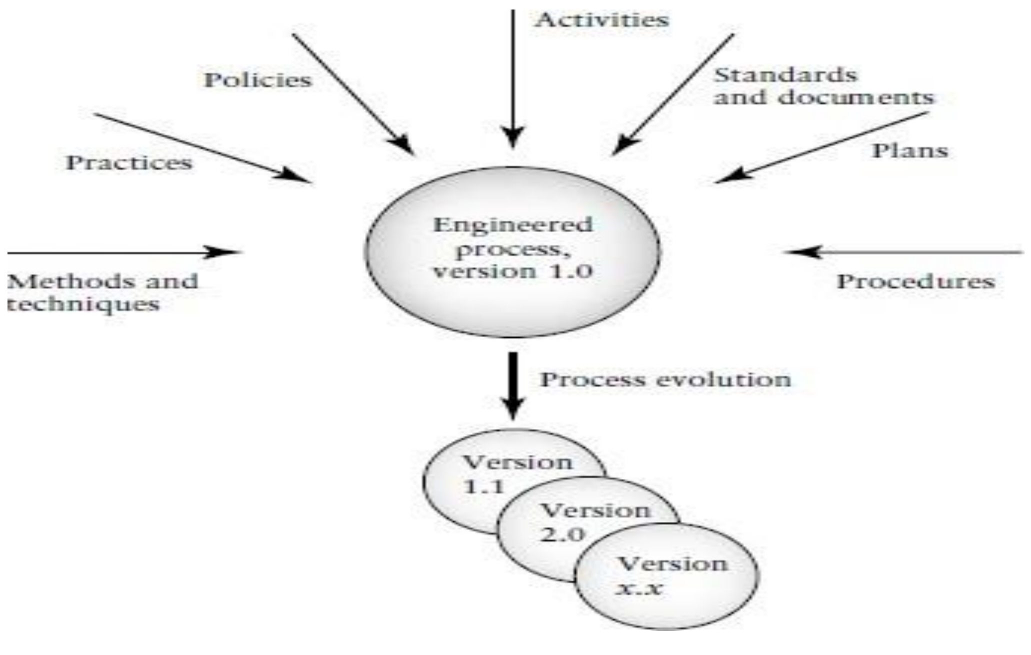
- ❖ The development process is well understood
- ❖ project are planned
- ❖ life cycle models are defined and adhered to
- ❖ standards are in place for products and process
- ❖ measurements are employed to evaluate products and process quality
- ❖ Components are reused.
- ❖ Validation and verification processes play a key role in quality determination

Engineers have proper education, training and certification

A test specialist is one whose education is based on the principles , practices and processes that constitute the software engineering discipline, and whose specific focus is on one area of that discipline – software testing . A test specialist who is trained as an engineer should have knowledge of test related principles, processes, measurements, standards, plans, tools and methods, and should learn how to apply them to the testing tasks to be performed.

Role of Process in Software Quality:-

Process, in the software engineering domain , is the set of methods, practices, standards , documents , activities , policies , and procedures that 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



The software development process like most engineering artifacts must be engineered . Software process improvement models that have had wide acceptance in industry are high level models, in the sense that they focus on the software process as a whole and do not offer adequate support to evaluate and import specific software development sub processes such as design and testing.

Testing as a Process:-

Process has been described as a series of phases, procedures, and steps that result in the production of a software product. Embedded within this are several processes.

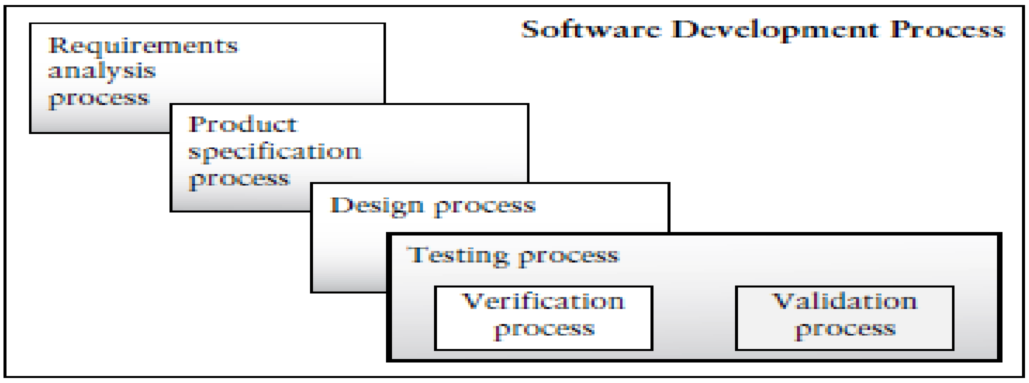
Validation is the process of evaluating a software system or components during or at the end of, the development cycle in order to determine whether it satisfies specified requirements

Validation is usually associated with traditional execution based testing that is exercising the code with test cases

Verification is the process of evaluating a software system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. Verification is usually associated with activities such as inspections and reviews of software deliverables.

Difference Between Verification & Validation

Verification	Validation
1. Verification is a static practice of verifying documents, design, code and program.	1. Validation is a dynamic mechanism of validating and testing the actual product.
2. It does not involve executing the code.	2. It always involves executing the code.
3. It is human based checking of documents and files.	3. It is computer based execution of program.
4. Verification uses methods like inspections, reviews, walkthroughs, and Desk-checking etc.	4. Validation uses methods like black box (functional) testing, gray box testing, and white box (structural) testing etc.
5. Verification is to check whether the software conforms to specifications.	5. Validation is to check whether software meets the customer expectations and requirements.
6. It can catch errors that validation cannot catch. It is low level exercise.	6. It can catch errors that verification cannot catch. It is High Level Exercise.
7. Target is requirements specification, application and software architecture, high level, complete design, and database design etc.	7. Target is actual product-a unit, a module, a bent of integrated modules, and effective final product.
8. Verification is done by QA team to ensure that the software is as per the specifications in the SRS document.	8. Validation is carried out with the involvement of testing team.
9. It generally comes first-done before validation.	9. It generally follows after verification.


Software Development Process
Requirements analysis process
Product specification process
Design process
Testing process
Verification process
Validation process

Testing is generally described as a process as a group of procedures carried out to evaluate some aspects of a piece of software.

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

Testing and Debugging are two different activities.

Debugging process begins after localization has been carried out and the tester has noted that the software is nit behaving as specified

Debugging or Fault Localization is the process of

(1) Locating the fault or defect

(2) Repairing the code , and

(3) Retesting the code

Testing as a process has economic, technical and managerial aspects.

- **Economic aspects** are related to the reality that resources and time are available to the testing group on a limited basis
- **Technical Aspect** of testing are relate to the techniques, methods, measurements, and tools used to insure that the software under test is a defect free and reliable as possible for the conditions and constraints under which it must operate
- **Managerial Aspect –** Minimally that means that an organizational policy for testing must be defined and documented. Testing must be planned, testers should be trained, the process should have associated quantifiable goals that can be measured and monitored.

<u>Testing Maturity Model - Introduction</u>

To know about Testing , one must find answer for following queries

- Where do we begin to learn more about testing ?
- What areas of testing are important?
- Which topics need to be addressed first?

TMM gives answer for all the questions listed.

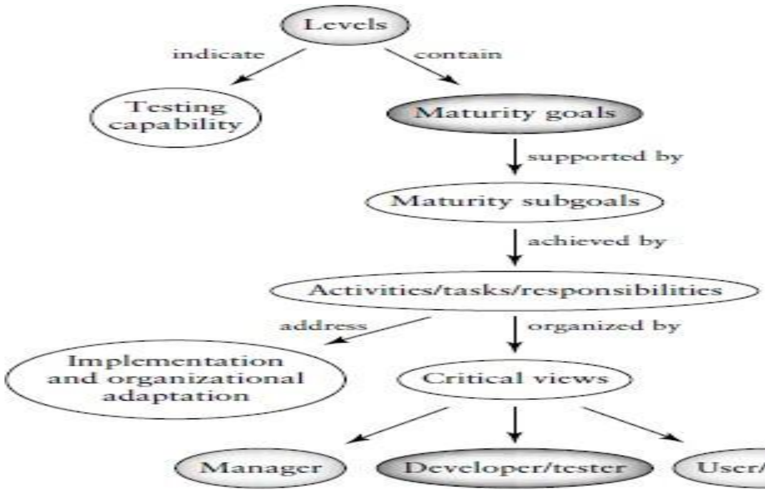
<u>Testing Maturity Model - Definition</u>

TMM is a learning tools ,or framework to learn about testing. It introduces both the technical and managerial aspects of testing. It evolve testing process both in the

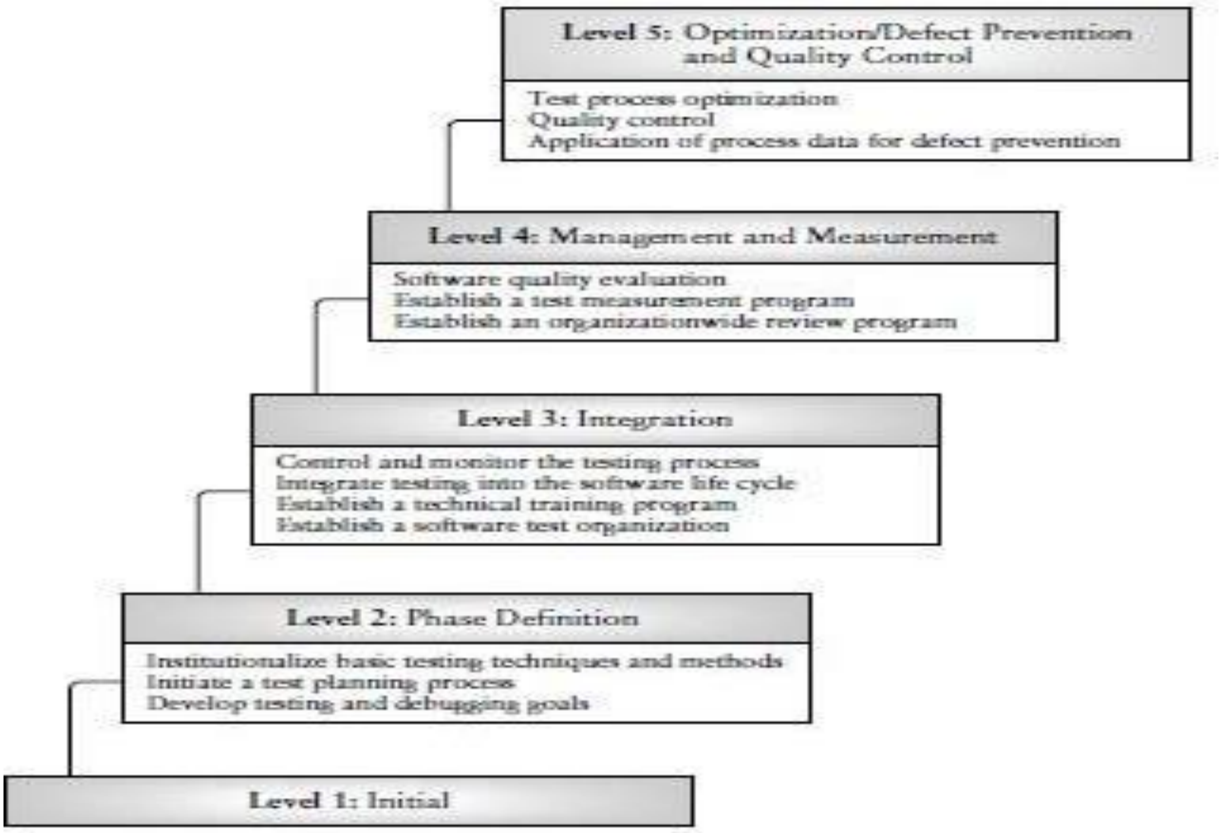
personal and organizational levels. It follows staged architecture for process improvement models.
It has five levels that prescribe a maturity hierarchy and an evolutionary path to test process improvement. Each level has (Except Level 1)

- ***A set of maturity goals*** - The maturity goals identify testing improvement goals that must be addressed in order to achieve maturity at that level.
- ***Supporting maturity subgoals -*** They define the scope, boundaries and needed accomplishments for a particular level.
- ***Activities, tasks and responsibilities (ATR) -*** address implementation and organizational adaptation issues at each TMM level. Supporting activities and tasks are identified, and responsibilities are assigned to appropriate groups.

Internal Structure of TMM maturity model



<u>Testing Maturity Model -</u> *5-level structure*


Level 5: Optimization/Defect Prevention and Quality Control
Test process optimization
Quality control
Application of process data for defect prevention
Level 4: Management and Measurement
Software quality evaluation
Establish a test measurement program
Establish an organizationwide review program
Level 3: Integration
Control and monitor the testing process
Integrate testing into the software life cycle
Establish a technical training program
Establish a software test organization
Level 2: Phase Definition
Institutionalize basic testing techniques and methods
Initiate a test planning process
Develop testing and debugging goals
Level 1: Initial

Level 1—Initial: (No maturity goals)

testing is a chaotic process; it is ill-defined
Not distinguished from debugging.
The objective of testing is to show the software works
Software products are often released without quality assurance.
lack of resources, tools and properly trained staff.

Level 2—Phase Definition:

Goal 1: Develop testing and debugging goals;
Goal 2: Initiate a testing planning process;
Goal 3: Institutionalize basic testing techniques and methods
testing is separated from debugging and is defined as a phase that follows coding.
It is a planned activity; however, test planning at level 2 may occur after coding for reasons related to the immaturity of the testing process.
use of black box and white box testing strategies, and a validation cross-reference matrix
Testing is multileveled - unit, integration, system, and acceptance levels.

Level 3—Integration

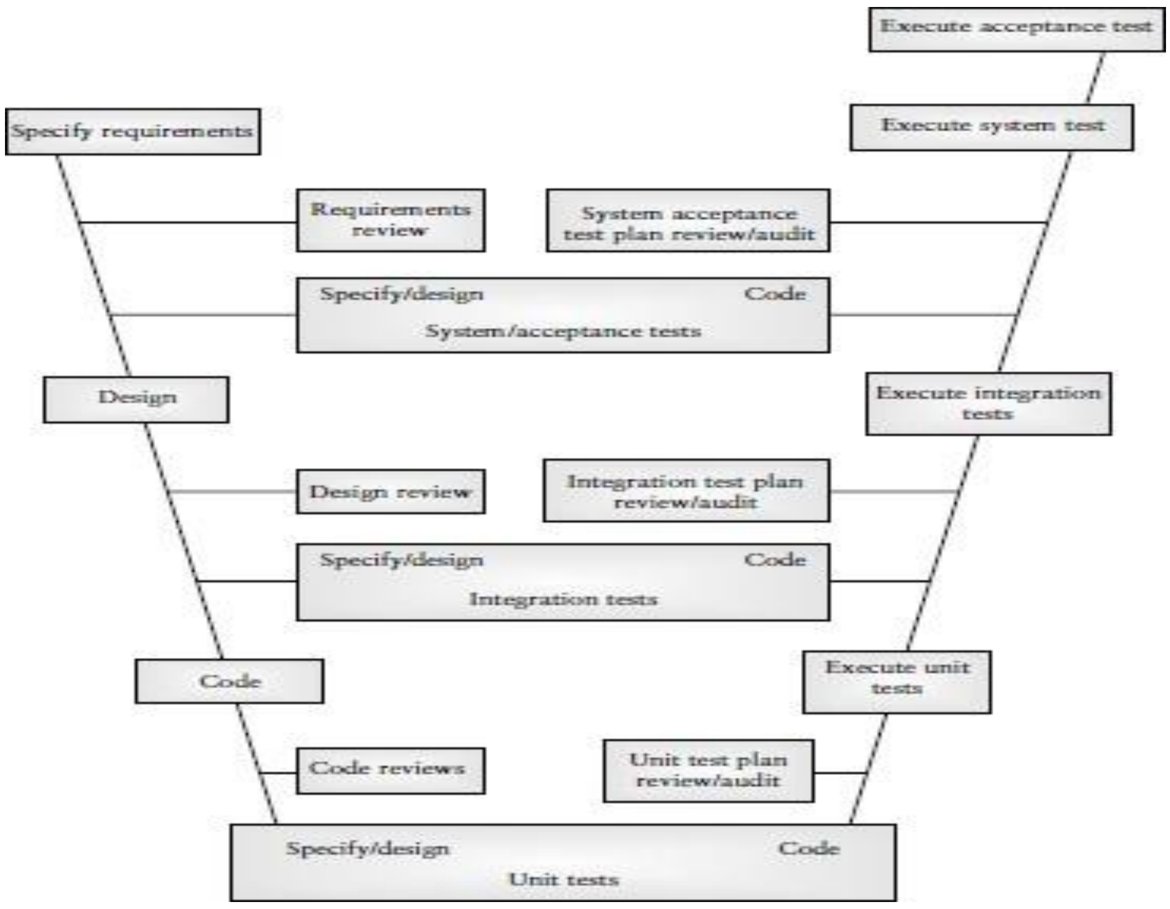
Goal 1: Establish a software test organization;
Goal 2: Establish a technical training program;
Goal 3: Integrate testing into the software life cycle;
Goal 4: Control and monitor testing
testing is integrated into the entire software life cycle
There is a test organization, and testing is recognized as a professional activity.
There is a technical training organization with a testing focus
Testing is monitored to ensure it is going according to plan and actions can be taken if deviations occur

Level 4—Management and Measurement

Goal 1: Establish an organization wide review program;
Goal 2: Establish a test measurement program;
Goal 3: Software quality evaluation
process that is measured and quantified. Reviews at all phases of the development process are now recognized as testing/quality control activities.
Software products are tested for quality attributes such as reliability, usability, and maintainability.
Test cases from all projects are collected and recorded in a test case database for the purpose of test case reuse and regression testing. Defects are logged and given a severity level.
Some of the deficiencies occurring in the test process are due to the lack of a defect prevention philosophy. An extension of the V-model as shown in Figure can be used to support the implementation of this goal

Level 5—Optimization/Defect Prevention/Quality Control

Goal 1: Defect prevention;
Goal 2: Quality control;
Goal 3: Test process optimization
the testing process is now said to be defined and managed; its cost and effectiveness can be monitored. Defect prevention and quality control are practiced. Automated tools totally support the running and rerunning of test cases



Extension of the V-model

Testing Axioms

Axioms: "rules of the road" or the "facts of life" for software testing and software development.

1. **It's Impossible to Test a Program Completely**
2. **Software Testing Is a Risk-Based Exercise**
3. **Testing Can't Show That Bugs Don't Exist**
4. **The More Bugs You Find, the More Bugs There Are**
5. **The Pesticide Paradox**
6. **Not All the Bugs You Find Will Be Fixed**
7. **When a Bug's a Bug Is Difficult to Say**
8. **Product Specifications Are Never Final**
9. **Software Testers Aren't the Most Popular Members of a Project Team**
10. **Software Testing Is a Disciplined Technical Profession**

1. It's Impossible to Test a Program Completely

due to four key reasons:

• The number of possible inputs is very large.

- The number of possible outputs is very large.
- The number of paths through the software is very large.
- The software specification is subjective

Ex: Microsoft Windows Calculator

Assume that you are assigned to test the Windows Calculator. You decide to start with addition. You try 1+0=. You get an answer of 1. That's correct. Then you try 1+1=. You get 2. How far do you go? The calculator accepts a 32-digit number, so you must try all the possibilities up to

1+99999999999999999999999999999999=

Once you complete that series, you can move on to 2+0=, 2+1=, 2+2=, and so on. Eventually you'll get to

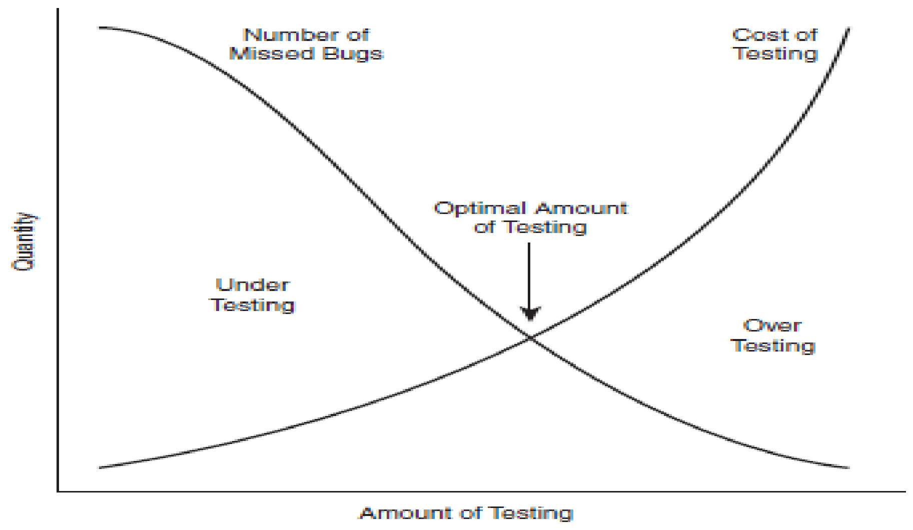
99999999999999999999999999999999+99999999999999999999999999999999=

Next you should try all the decimal values: 1.0+0.1, 1.0+0.2, and so on.

it's impossible to completely test a program, even software as simple as a calculator. If you decide to eliminate any of the test conditions because you feel they're redundant or unnecessary, or just to save time, you've decided not to test the program completely.

2. Software Testing Is a Risk-Based Exercise

One key concept that software testers need to learn is how to reduce the huge domain of possible tests into a manageable set, and how to make wise risk-based decisions on what's important to test and what's not.



Above graph shows the relationship between the amount of testing performed and the number of bugs found. If you attempt to test everything, the costs go up dramatically and the number of missed bugs declines to the point that it's no longer cost effective to continue.

If you cut the testing short or make poor decisions of what to test, the costs are low but you'll miss a lot of bugs. The goal is to hit that optimal amount of testing so that you don't test too much or too little.

3. Testing Can't Show That Bugs Don't Exist

You're an exterminator charged with examining a house for bugs. You inspect the house and find evidence of bugs

House1 :

Findings :—maybe live bugs, dead bugs, or nests.

Conclusion :- You can safely say that the house has bugs.

House2 :

Findings :- no evidence of bugs. e no signs of an infestation.

Maybe you find a few dead bugs or old nests but you see nothing that tells you that live bugs exist.

Conclusion : your search you didn't find any live bugs. Unless you completely dismantled the house down to the foundation, you can't be sure that you didn't simply just miss them.

Software testing works exactly as the exterminator does. It can show that bugs exist, but it can't show that bugs don't exist. You can perform your tests, find and report bugs, but at no point can you guarantee that there are no longer any bugs to find.

4. The More Bugs You Find, the More Bugs There Are

Reasons

Programmers have bad days. Like all of us, programmers can have off days. Code written one day may be perfect; code written another may be sloppy.

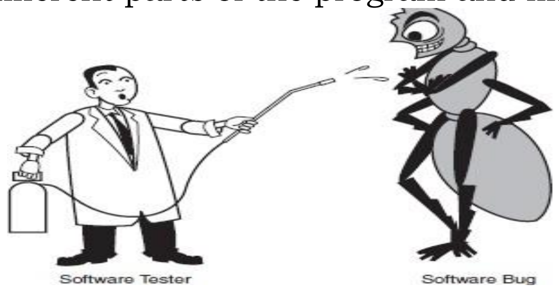
Programmers often make the same mistake. Everyone has habits. A programmer who is prone to a certain error will often repeat it.

Some bugs are really just the tip of the iceberg. Very often the software's design or architecture has a fundamental problem. A tester will find several bugs that at first may seem unrelated but eventually are discovered to have one primary serious cause.

5. The Pesticide Paradox

The test process repeats each time around the loop. With each iteration, the software testers receive the software for testing and run their tests. Eventually, after several passes, all the bugs that those tests would find are exposed. Continuing to run them won't reveal anything new.

To overcome the pesticide paradox, software testers must continually write new and different tests to exercise different parts of the program and find more bugs.



6. Not All the Bugs You Find Will Be Fixed

reasons why you might choose not to fix a bug:

• There's not enough time. In every project there are always too many software features, too few people to code and test them, and not enough room left in the schedule to finish. If you're working on a tax preparation program, April 15 isn't going to move—you must have your software ready in time.

• It's really not a bug. Maybe you've heard the phrase, "It's not a bug, it's a feature!" It's not uncommon for misunderstandings, test errors, or spec changes to result in would-be bugs being dismissed as features.

• It’s too risky to fix. You might make a bug fix that causes other bugs to appear. Under the pressure to release a product under a tight schedule, it might be too risky to change the software. It may be better to leave in the known bug to avoid the risk of creating new, unknown ones.
• It’s just not worth it. This may sound harsh, but it’s reality. Bugs that would occur infrequently or bugs that appear in little-used features may be dismissed.

7. When a Bug’s a Bug Is Difficult to Say

rules to define a bug

1. The software doesn’t do something that the product specification says it should do.
2. The software does something that the product specification says it shouldn’t do.
3. The software does something that the product specification doesn’t mention.
4. The software doesn’t do something that the product specification doesn’t mention but should.
5. The software is difficult to understand, hard to use, slow, or—in the software tester’s eyes—will be viewed by the end user as just plain not right.

8. Product Specifications Are Never Final

You’re halfway through the planned two year development cycle, and your main competitor releases a product very similar to yours but with several desirable features that your product doesn’t have.

- Do you continue with your spec as is and release an inferior product in another year?
- Or, does your team regroup, rethink the product’s features, rewrite the product spec, and work on a revised product?

9. Software Testers Aren’t the Most Popular Members of a Project Team

The goal of a software tester is to find bugs, find them as early as possible, and make sure they get fixed.

- Find bugs early.
- Temper your enthusiasm
- Don’t always report bad news

10. Software Testing Is a Disciplined Technical Profession

- If software testers were used, they were frequently untrained and brought into the project late to do some “ad-hoc banging on the code to see what they might find.” Times have changed.
- The software industry has progressed to the point where professional software testers are mandatory. It’s now too costly to build bad software.

Basic Definitions:-

1. Error :-

A Error is a mistake, misconception, or misunderstanding on the part of a software developers. Developers we include software Engineers, programming analysts and testers. It is the terminology of the developer. For Eg, a developer may understand a design notation, or a programmer might type a variable name incorrectly.

2. Faults / Defect :-

A fault(Defects) is introduced into the software as the result of an error. It is an anomaly in the software that may cause it to behave incorrectly , and not according to its specification , Faults or defects are sometimes called as "**bugs".** It is the terminology of the tester.

3. Failures

A failure is the inability of a software or component to perform its required functions within specified performance requirements. It is the terminology of the customer. Error leads to Defect , Defect leads to Failure.

A Fault in the code does not always produce a failure. In fact, faulty software may operate over a long period of time without exhibiting any in correct behavior. When the proper conditions occur the fault will manifest itself as a failure.

1.. The input to the software must cause the faulty statements to be executed
2. The faulty statements must produce different results than the correct statements.
   This event produces an incorrect internal state for the software.
3. The incorrect internal state must propagate to the output, so that the result of the fault is observable

4. Test Cases:-

Detecting defects in a piece of software is for the tester to select a set of input data and then execute the software with the input data under a particular set of conditions. To check the software is success or failed the tester need to know the output of the software, input of the software and soon

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

❖ *A set of test inputs :-* These are data items received from an external source by the code under test. The external source can be hardware , software or human.
❖ *Execution Condition:-* These are conditions required for running the test, for example , a certain state of database, or a configuration of hardware devices
❖ *Expected Outputs:-* These are the specified results to be produced by the code under test

Ex: biggest of 3 Numbers

Test Case Id	**Test I/p**	**Expected O/P**	**Actual O/p**	**Result :Pass/Fail**
TC1	**A=10 B=20 C=50**	**C IS BIG**		
TC2	**A=110 B=20 C=50**	**A IS BIG**		
TC3	**A=10 B=120 C=50**	**B IS BIG**		

5. Test :-

A test is a group of related test cases, or a group of related test cases and test procedures(steps needed to carry out a test)

A group of related tests is sometimes referred to as a test set. A group of related tests that are associated with the database, and are usually run together is sometimes referred to as test suite

6. 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.

A program, or a document that produces or specifies the expected outcome of a test, can serve as an oracle. Example include a specification, design documents, set of requirements.

Other sources are regression test suites. The suite usually contains components with correct results for previous versions of software.

7. Test Bed:-

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

This include testing environment, eg :- simulators, emulators memory checkers, hardware probes, software tools etc

8. Software Quality :-

Two concise definition for quality are found in the IEEE standards Glossary of Software Engg Terminology

a. Quality relates to the degree to which a system, system component, or process meets specified requirements

b. Quality relates to the degree to which a system, system components , or process meets customer or user needs, or expectations

Software artifacts we can measure the degree to which they posses a given quality attribute with quality metrics.

9. Metrics:-

A metrics is a quantitative measure of the degree to which a system, system component or process possesses a given attribute. There are product and process metrics. A very commonly used example software products metrics is software size, usually measured in Lines Of Code(LOC)

10. Quality Metrics:-

Is a quantitative measurement of the degree to which an Item possesses a given quality attribute. Quality attributes with brief explanation are the following :-

i. **Correctness:-** the degree to which the system performs its intended function
ii. **Reliability :-** the degree to which the software is expected to perform its required function under stated conditions for a stated period of time
iii. **Usability :-** related to the degree of effort needed to learn , operate, prepare input, and interpret output of the software.
iv. **Integrity:-** Relates to the system's ability to withstand both intentional and accidental attacks
v. **Portability:-** Relates to the ability of the software to be transferred from one environment to another.
vi. **Maintainability:-** the effort needed to make change in the software
vii. **Interoperability: -** the effort needed to link or couple one system to another.

Testability attribute is of more interest to developers/testers than to clients. It can be expressed in the following two ways:-

a) The amount of effort needed to test the software to ensure it performs according to specified requirements (relates the number of test cases needed)

b) The ability of the software to reveal defects under testing conditions (some s/w is designed in such a way that defects are well hidden during ordinary testing conditions)

11. Software Quality Assurance Group:-

SQA group in an organization has ties to quality issues. The Software Quality Assurance group SQA is a team of people with necessary training and skills to ensure that all necessary actions are taken during the development process so that the resulting software conforms to established technical requirements

12. Reviews:-

Dynamic execution based testing techniques that can be used to detect defects and evaluate software quality, review are types of static testing techniques that can be used to evaluate the quality of the software artifacts such as requirement documents , a test plan, a design document ,a code component.

A review is a group meeting whose purpose is to evaluate a software artifact or a set of software artifacts.

Software Testing Principles:-

Principle: Fundamental to the objective of testing, namely , to provide quality products to customers. The Fundamental principles of testing are as follows

1. **The Goal of testing is to find defects before customers find them out.**

   Ex: The Incomplete Car
2. **Exhaustive testing is not possible; program testing can only show the presence of defects , never their absence. Ex: Dijkstra's Doctrine**
3. **Testing applies all through the software life cycle and is not end of cycle activity. Ex: Test in time:-**
4. **Understand the reason behind the test. Ex: The Cat and the Saint**
5. **Test the tests first. Ex: Patient & Doctor**
6. **Test Develop immunity and have to be revised constantly.**

   Ex: The Pesticide Paradox
7. **Defect occurs in convoys or clusters and testing should focus on these convoys Ex: The Convoy and the Rags**
8. **Testing encompasses defect prevention. Ex: The Policeman on the Bridge**
9. **Testing is a fine balance of defect prevention and a defect detection.**

   Ex: The Ends of the pendulum
10. **Intelligent and well planned automation is key to realizing the benefits of testing. Ex: Automation Syndrome**
11. **Testing requires talented, committed people who believe in themselves and work in teams Ex: Men in Black**

1. The Incomplete Car:-

Car Salesman:- " the car is complete – you just need to paint it "

Sales Representative/ Engineer:- "This Car has the best possible transmission and brake, and accelerate from 0 to 80mph in under 20 seconds!".
Customer: "Well that may be true, but unfortunately it accelerates (even faster) when I press the break pedal"

Above conversation concludes that the car is not tested properly , so customer finding the fault. "*Testing Should Focus on Finding Defects before Customers Find Them"*

2. <u>Dijkstra's Doctrine:-</u>

Consider a program (<u>Dijkstra's Doctrine</u>) that is supposed to accept a six character code and ensure that
the first character is numeric and rests of the characters are alphanumeric. How many combinations of input data should we test.

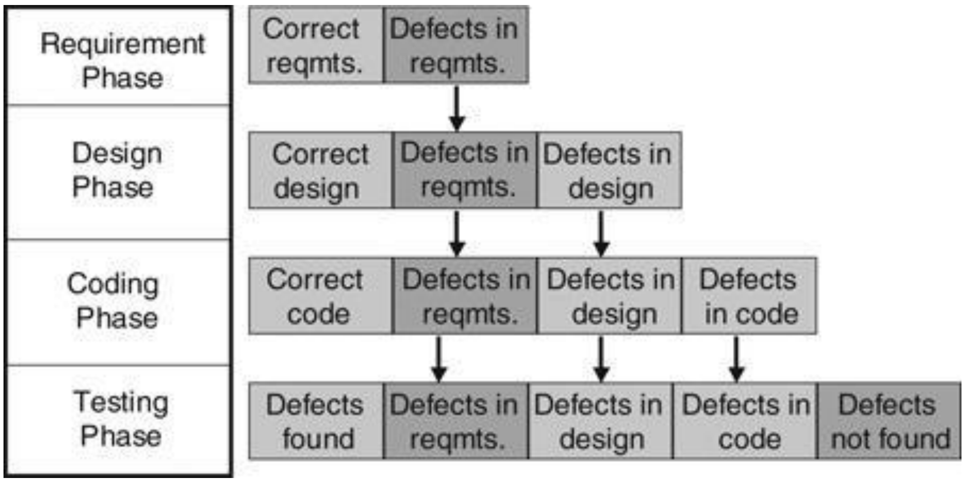
The first character can be filled up in one of 10 ways (the digits (0-9) . → 10
The Second through sixth characters can each be filled up in 62 ways(digits 0-9 (10) , lower case letters a-z (26) and capital letters A-Z (26)) → $(10+26+26)^5 \rightarrow 62^5$
This means that we have a total of 10 * (62^5) or 9,16,328,320 valid combinations of values to test.
Assuming that each combinations takes 10 seconds to test, testing all these valid combinations will take approximately <u>2905 years</u>.

"Testing can only prove the presence of defects, never their absence"

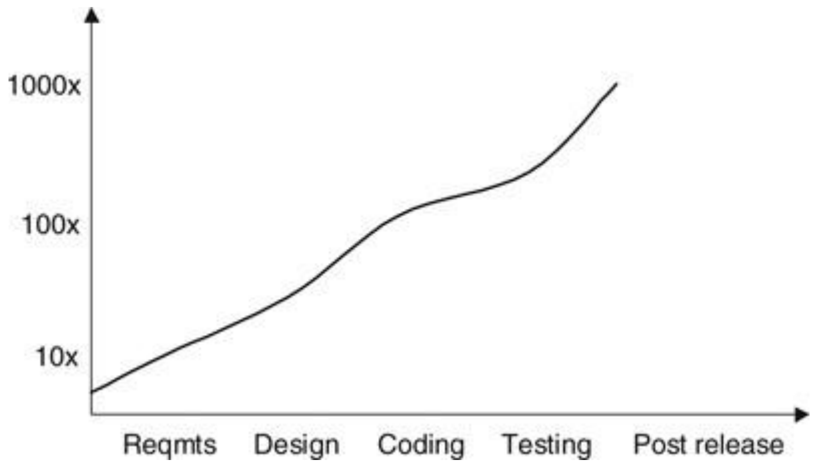
3. <u>A Test in time:-</u>

Defect in a product can come from any phase. There could have been errors while gathering initial requirements .If a wrong or incomplete requirement forms the basis for the design and development of a product, then that functionality can never be realized correctly in the eventual product. Similarly , when a product design – which forms the basis for the product development - is faulty, then the code that realizes the faulty design will also not meet the requirements. An Essential Conditions should be that every phase of the software development (requirements, design, coding)should catch and correct defects at that phase, without letting the defects seep to the next stage. Organization incurs extra expenses for

- ☞ Performing a wrong design based on the wrong requirements;
- ☞ Transforming the wrong design into wrong code during the coding phase
- ☞ Testing to make sure the product complies with the (wrong requirements
- ☞ Releasing the product with the wrong functionality



How defects from early phases add to the costs.



Compounding effect of defects on software costs.

The cost of building a product and the number of defects in it increases steeply with the number of defects allowed to seep into the layer phases.

4. <u>The Cat and the Saint</u>

A saint sat meditating. A cat that was prowling around was disturbing his concentration. Hence he asked his disciples to tie the cat to a pillar while he meditated . This sequence of events became a daily routine. The tradition continued over the years with the saint's descendents and the cat's descendents. One day, there were no cats on the hermitage. The disciples got panicky and searched for a cat, saying , " We need a car. Only when we get a cat, can we tie it to a pillar and only after that saint can start meditation !"

"Why one tests" it as important as " What to test " and "How to test ".

From the story , If we carry out tests without understanding why we are running them, we will end up in running inappropriate test that do not address what product should do.!

5. <u>Test the Tests First :-</u>

An audiologist was testing a patient , telling ,"I want to test the range within which you can hear . I will ask you from various distances to tell me your name, and you should tell me your name. Please turn back and answer." The patient understood what needs to be done

Doctor(from 30 feet): What is your name ?

……………..

Doctor (from 20 feet): What is your name ?

…………………………

Doctor (From 10 feet) What is your name ?

Patient : For the third time , let me repeat my name is Pushpa!.

From the above example it is important to make sure that the test themselves are not faulty before we start using them. One way of making sure that tests are tested is to document the inputs and expected outputs for a given test and have this description validated by an expert or get it counter checked by some means outside the tests themselves. " *Test the test first--- a defective test is more dangerous than defective products!"*

6. <u>The Pesticide Paradox:-</u>

Pest gets used to new pesticides, develop immunity, and render the new pesticides ineffective. In subsequent years, the old pesticides have to be used to kill the pests which have not yet developed this immunity and new improved formulae that can combat these tougher variants of pests have to be introduced. This combination of new and old pesticides could sometimes even hinder the effectiveness of the (working)

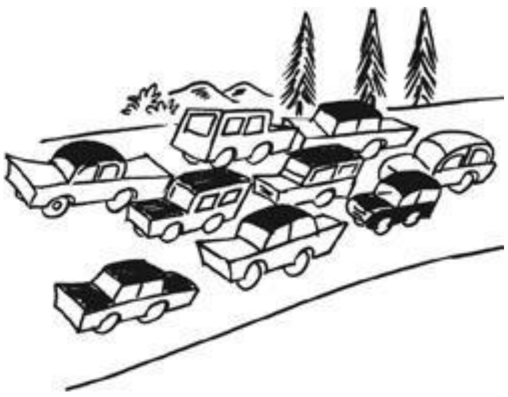
Defects are like pests, testing is like designing the right pesticides to catch and kill the pests, and the test cases that are written are like pesticides .

Tests are like pesticides- you have to constantly revise their composition to tackle new pests(Defects) .

There are two possible ways to explain how products develop this “immunity” against test cases. The initial tests go a certain distance into the code and are stopped from proceeding, further because of the defects they encounter. Once these defects are fixed, the tests proceed further, encounter newer parts of the code that have been dealt with before, and uncover new defects.

A next way is that when the tester starts exercising a product, initial defects prevents them from using the full external functionality. As tests are run, defects are uncovered, and problems are fixed, users get to explore new functionality that has not been used before and this cause newer defect to exposed.

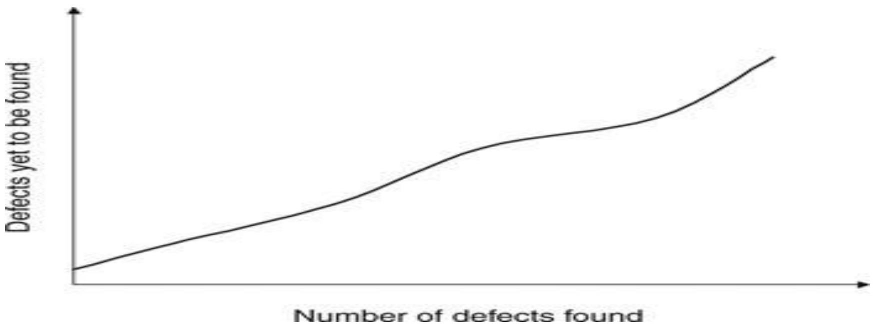
7. <u>The Convoy and the Rags</u>



All of us experience traffic congestions. Typically, during these congestions, we will see a convoy effect. There will be stretches of roads with very heavy congestions, with vehicles looking like they are going in a convoy. This will be followed by a stretch of smooth sailing (rather, driving) until we encounter the next convoy.

Testing can only find a part of defects that exist in a cluster; fixing a defect may introduce another defect to the cluster.

A fix for a defect is made around certain lines of code. This fix can produce side-effects around the same piece of code. A fix for one defect generally introduces some instability and necessitates another fix. All these cause the convoy of defects in certain parts of the product. whenever a product undergoes any change, these error-prone areas need to be tested as they may get affected.



The number of defects yet to be found increases with the number of defects uncovered.

8. <u>The Policeman on the Bridge:-</u>

There was a bridge in a city. Whenever people walked over it, they would fall down. To take care of this problem, the city appointed a strong policeman to stand under the bridge to save people who fall down. While this helped the problem to some extent, people continued to fall down the bridge when the policeman was not around, or when he could not catch them properly.

When the policeman retired, a new policeman was appointed to the job. During the first few days, instead of standing at the bottom of the bridge and catching the falling people, the new policeman worked with an engineer and fixed the hole on the bridge, which had not been noticed by the earlier policeman. People then stopped falling down the bridge and the new policeman did not have anyone to catch.

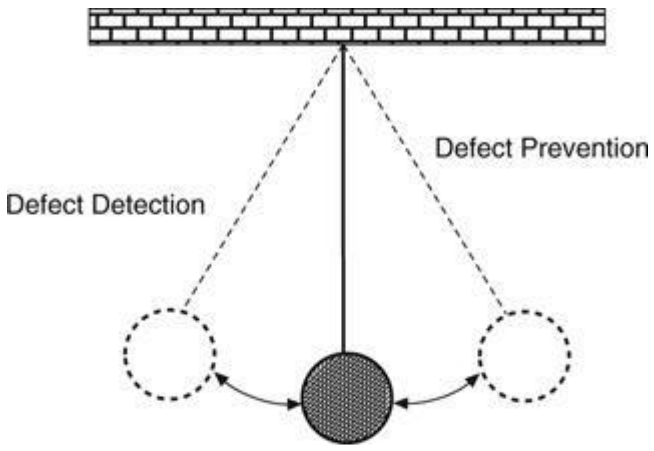
Like the second police officer in the above story, they know people fall and they know why people fall. Rather than simply catch people who fall (and thereby be exposed to the risk of a missed catch), they should also look at the root cause for falling and advise preventive action. It may not be possible for testers themselves to carry out preventive action. ***"Prevention is better than cure—you may be able to expand your horizon much farther."*** Defect prevention is a part of a tester's job.

9. <u>The Ends of the Pendulum :</u>-

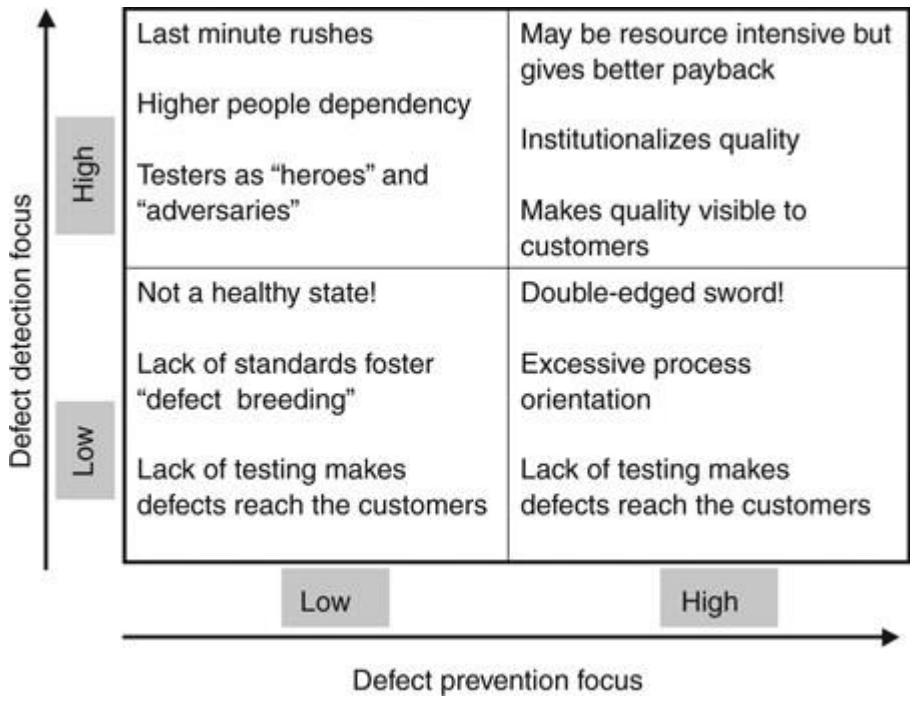
The eventual goal of any software organization is to ensure that the customers get products that are reasonably free of defects. There are two approaches to achieving this goal.

1) focus on defect detection and correction
2) focus on defect prevention.

These are also called quality control focus and quality assurance focus. Testing is traditionally considered as a quality control activity, with an emphasis on defect detection and correction. Quality assurance is normally associated with process models such as CMM, CMMI, ISO 9001, and so on. Organizations view these two functions as mutually exclusive, “either-or” choices.



Quality control and quality assurance as two methods to achieve quality.



Defect detection focus	Low (Defect prevention focus)	High (Defect prevention focus)
High	Last minute rushes Higher people dependency Testers as “heroes” and “adversaries”	May be resource intensive but gives better payback Institutionalizes quality Makes quality visible to customers
Low	Not a healthy state! Lack of standards foster “defect breeding” Lack of testing makes defects reach the customers	Double-edged sword! Excessive process orientation Lack of testing makes defects reach the customers

10.<u>Automation Syndrome:-</u>

A farmer had to use water from a well which was located more than a mile away.

Crop Cycle	Farmer’s Approach	Outcome
1st	• 100 people to draw water from the well • pot of water a day	crops failed

2nd	• thought about automation to increase productivity and to avoid failures • bought 50 motorcycles, laid off 50 of his workers • asked each rider to get two pots of water. • process of learning to balance the motorcycles, the number of pots of water they could fetch fell.	crops failed again.
3rd	• all workers were laid off except one • bought a truck to fetch water • realized the need for training and got his worker to learn driving. • Road was narrow, truck did not help in bringing in the water	crops failed again.
After these experiences the farmer said, “My life was better without automation!”		

The moral of the above story as it applies to testing is that automation requires careful planning, evaluation, and training. Automation may not produce immediate returns.

Some of the points that should be kept in mind while harping on automation are as follows.

- Know first why you want to automate and what you want to automate, before recommending automation for automation’s sake.
- Evaluate multiple tools before choosing one as being most appropriate for your need.
- Try to choose tools to match your needs, rather than changing your needs to match the tool’s capabilities.
- Train people first before expecting them to be productive.
- Do not expect overnight returns from automation.

11.<u>Men in Black:-</u>

The testing team was seeded with motivated people who were “free from cognitive conflict that hampers developers when testing their own programs.” The team was given an identity (by a black dress, amidst the traditionally dressed remainder of the organization) and tremendous importance. All this increased their pride in work and made their performance grow by leaps and bounds, “almost like magic.” Long after the individual founding members left and were replaced by new people, the “Black Team” continued its existence and reputation. The biggest bottleneck in taking up testing as a profession is the lack of self-belief. This lack of self-belief and apparent distrust of the existence of career options in testing makes people view the profession as a launching pad to do other software functions. As a result, testers do not necessarily seek a career path in testing and develop skepticism towards the profession.

The Tester's Role in a Software Development Organization:-

- Testing is sometimes erroneously viewed as a destructive activity.
- The testers job is to reveal defects, fins weak points, inconsistent behavior, and circumstances where the software does not work as expected.
- Given the nature of the testers task , it is difficult for developers to effectively test their own code
- **Effective Tester:** To be most effective as a tester requires extensive programming experience in order to understand how code is constructed, and where and what kind of, defects are likely to occur.
- **Goal as a tester** is to work with the developers to produce high quality software that meets customer's requirements
- Projects should have an appropriate developer /tester ratio. The ratio will vary depending on available resources, type of projects and TMM level (Testing Maturity Model)

  Example: - An embedded real time system needs to have a lower developer /tester ratio. **(2/1)** than a simple data base application (4/1) .At higher TMM levels where there is a well defined testing group, the developer/tester ratio would tend to be on the lower end(2/1 versus 4/1) because of the availability of tester resources.
- Cooperation with the code developers, tester also need to work along side with requirement engineers to ensure that requirement are testable, ant to plan for system and acceptance test(client are also involved in the latter part).
- Testers also need to work with designers to plan for integration and unit test.
- Test managers will need to cooperate with project managers in order to develop reasonable test plans, and with upper management to provide input for the development and maintenance of organizational testing standards , policies and goals.
- Testers also need to corporate software quality assurance staff and software engineering process group members.
- In view of these requirements for multiple working relationships, communication and extreme working skills are necessary for a successful career as a tester.

If you are employed by an organization that is assessed at **TMM levels 1 or 2** you may find that there is no independent software test function in the organization, so the testers in this case may be a part of the development group, but with special assignment to testing, or they may be apart of the software quality assurance group.

TMM level 3 and higher of the TMM

the testers may not necessarily belong to a independent organizational entity, testers should always have managerial independence from developers in the TMM level 3.

Testers are specialist their main function is

- to plan
- execute,
- record
- analyze tests. They do not debug software.

When defects are detected using testing, software should be returned to the developers who locate the defect and repair the code

Testers need the support of management. Developers , analysts and marketing staff need to realize that tester add value to a software products in that they detect defects and evaluate quality as early as possible in the software life cycle. Tester need to have a positive view of their work. Management must support them in their effort and recognize their contribution to the organization.

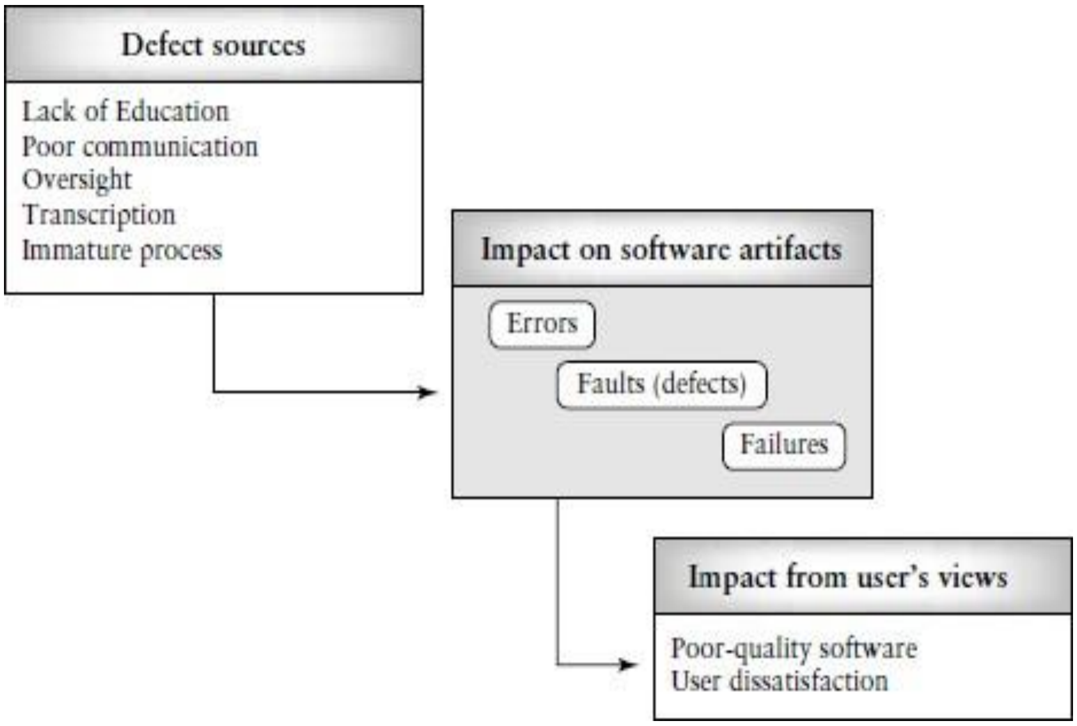
Origins of Defects:-

Defects have harmful effects on software users, and software engineers work very hard to produce high quality software with a low numbers of defects.

Reason for Defects are shown below

1. **Education :-** The software engineer did not have the proper educational background to prepare the software artifacts. They did not understand how to do something. For example, a software engineer who did not understand the precedence order of operators in a particular programming language could inject a defect in an equation that uses the operators for calculation.
2. **Communication:-** The software engineer was not informed abut something by a colleague. For example , if engineer 1 and engineer2 working on interfacing modules , and engineer 1 doesn't inform engineer2 that no error checking code will appear in the interfacing module he is developing , engineer 2 might have an incorrect assumption relating to the presence /absence of an error check and defects will result.
3. **Oversight :-** The software engineer omitted to do something .For example a software engineer might omit an initialization statement
4. **Transcription:-** The software engineer knows what to do, but makes a mistake in doing it. A simple example is a variable name being misspelled when entering the code.

5. **Process:**- The process used by the software engineer misdirected the action. For example, a development process that did not allow sufficient time for a detailed specification to be developed and reviewed could lead to specification defects.



- Goal as tester is to discover defects preferably before the software is in operation. One of the way we do this is by designing test cases that have a high probability of revealing defects.
- In the experimental scenario
  - a tester develops hypotheses about possible defects (Principle 2 and Principle 9)
  - Test cases are then designed based on the hypotheses.
  - The tests are run and results analyzed to prove, or disprove the hypotheses.
- Myers has a similar approach to testing. He describes the successful test as one that reveals the presence of Hypotheses defect.
- He compares the role of a tester as a doctor who is in the process of constructing a diagnosis for an ill patient. The doctor develops hypotheses about possible illness using her knowledge of possible diseases, and the patients symptoms. Test are made in order to make the correct diagnosis.
- A successful test will reveal the problem and the doctor can being the treatment. Completing the analogy of doctor and ill patient, one could view defective software as the ill patient. Testers as doctors need to have knowledge about possible defects (illness) in order tom develop defect hypotheses. They use the hypotheses to:-
  - Design Test cases;
  - Design Test procedures;

- Assemble test sets;
- Select the testing levels(unit, integration,etc) appropriate for the tests;
- Evaluate the result of the tests;

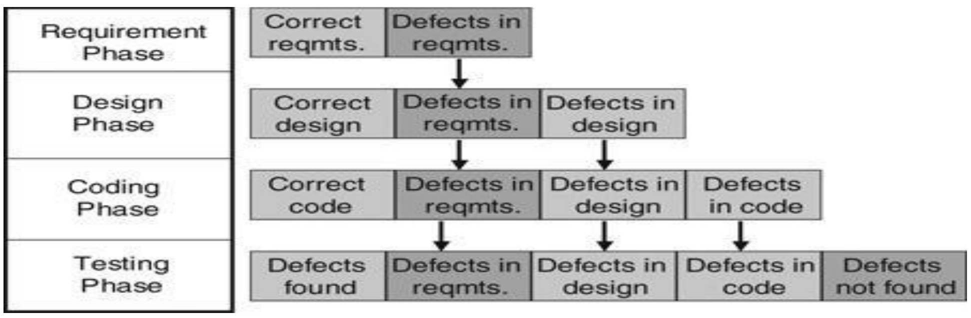
Very useful concept related to defects, testing, and diagnosis is that of the fault model.

- A fault (defect) model can be described as a link between the error made(eg., a missing requirement, a misunderstood design elements, a typographical error) and the fault/ defect in the software.

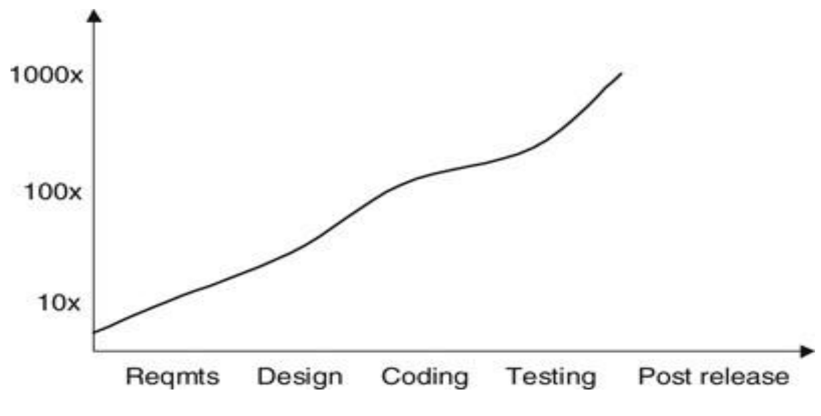
Example of fault model a software engineer might have in memory is “an incorrect value for a variable was observed because the precedence order for the arithmetic operators used to calculate its value was incorrect”→ this could be called as “incorrect operator precedence operator”.

Cost of Defect
Organization incurs extra expenses for

☞ Performing a wrong design based on the wrong requirements;
☞ Transforming the wrong design into wrong code during the coding phase
☞ Testing to make sure the product complies with the (wrong requirements
☞ Releasing the product with the wrong functionality



How defects from early phases add to the costs.



Compounding effect of defects on software costs.

The cost of building a product and the number of defects in it increases steeply with the number of defects allowed to seep into the layer phases.

DEFECT CLASSES:-

Defect can be classified in many ways. It is important for an organization to adapt a single classification scheme and apply it to all projects. Developers , testers and SQA staff should try to be as consistent as possible when recording defect data

Defects are assigned to four major classes reflecting their point of origin in the software life cycle- the development phases in which they were injected. These classes are:-

- **Requirements\ Specifications**
- **Design**
- **Code**
- **Testing**

Requirements and Specification Defects:-

The beginning of software life cycle is critical for ensuring high quality in the software being developed. Defects injects in early phases can persist and be very difficult to remove in later phases. Since many requirements are written unnatural language representation, there are very often occurrences of ambiguous, contradictory, unclear, redundant, and imprecise requirements.

1. **Functional Description Defects:-**

The overall description of what the product does, and how it should behave (Input/Output), is incorrect, ambiguous, and /or incomplete.

2. **Feature Defects**

Features may be described as distinguishing characteristics of a software component or system .Features refers to functional aspects of software that map to functional requirement described by the user and the client, it also maps quality such as performance and reliability. Feature defects are mainly due to features description that are missing, incorrect, incomplete or superfluous.,

3. **Feature Interaction Defects:-**

Mainly due to incorrect description of how the features should interact. For ex:- suppose one features of a software system supports adding a new customer to a customer database. This feature interacts with another feature that categorizes the new customer. Classification feature impact on where the storage algorithm places the new customer in the database, and also affects another feature that periodically support sending advertising information to customers in a specific category.

4. **Interface Description Defects:-**

Description of how the target software is to interface with external software, hardware and users. For detecting many functional description defects, black box testing techniques, which are based on functional specification of the software, offer the best approach. Black Box testing techniques such as equivalence class partitioning, boundary value analysis, state transition testing, and cause and effect graphing, which are useful defecting functional type of defects.

Black Box based tests can be planned at the unit, integration, system and acceptance levels to detect requirements/specification defects.

DESIGN DEFECTS:-

Design defects occur when system components, interactions between system components, interaction between the components and outside software /hardware, or users are incorrectly designed. Design of algorithm, control, logic, data elements, module interface description, and external software/ hardware/ user interface description. When describing these defects we assume that the detailed design description for the software modules is the pseudo code level with processing steps

1. **Algorithmic and Processing Defects:-**

These occur when the processing steps in the algorithm as described by the pseudo code are incorrect. Eg:- the pseudo code may contain a calculation that is incorrectly specified, or the processing steps in the algorithm written in pseudo code language may not be in the correct order.

Letter case a step may be missing or steps may be duplicated. Example of a defect in this sub class is the omission of error condition checks such as division by zero.

2. **Control, Logic and Sequence Defects:-**

Control defect occur when logic flow in the pseudo code is not correct. For example , branching to soon, branching to late, or use of an incorrect branching, condition. Other examples in this subclasses are unreachable pseudo code elements, improper nesting, improper procedure or function calls. Logic defects usually relate to incorrect use of logic operators, such as <,>

3. **Data Defects:-**

These are associated with incorrect design of data structures. For example a record may be lacking a field, an incorrect type is assigned to a variable or field in a record, an array may not have the proper number of elements assigned , or storage space may be allocated incorrectly.

4. **Module Interface Description Defects :-**

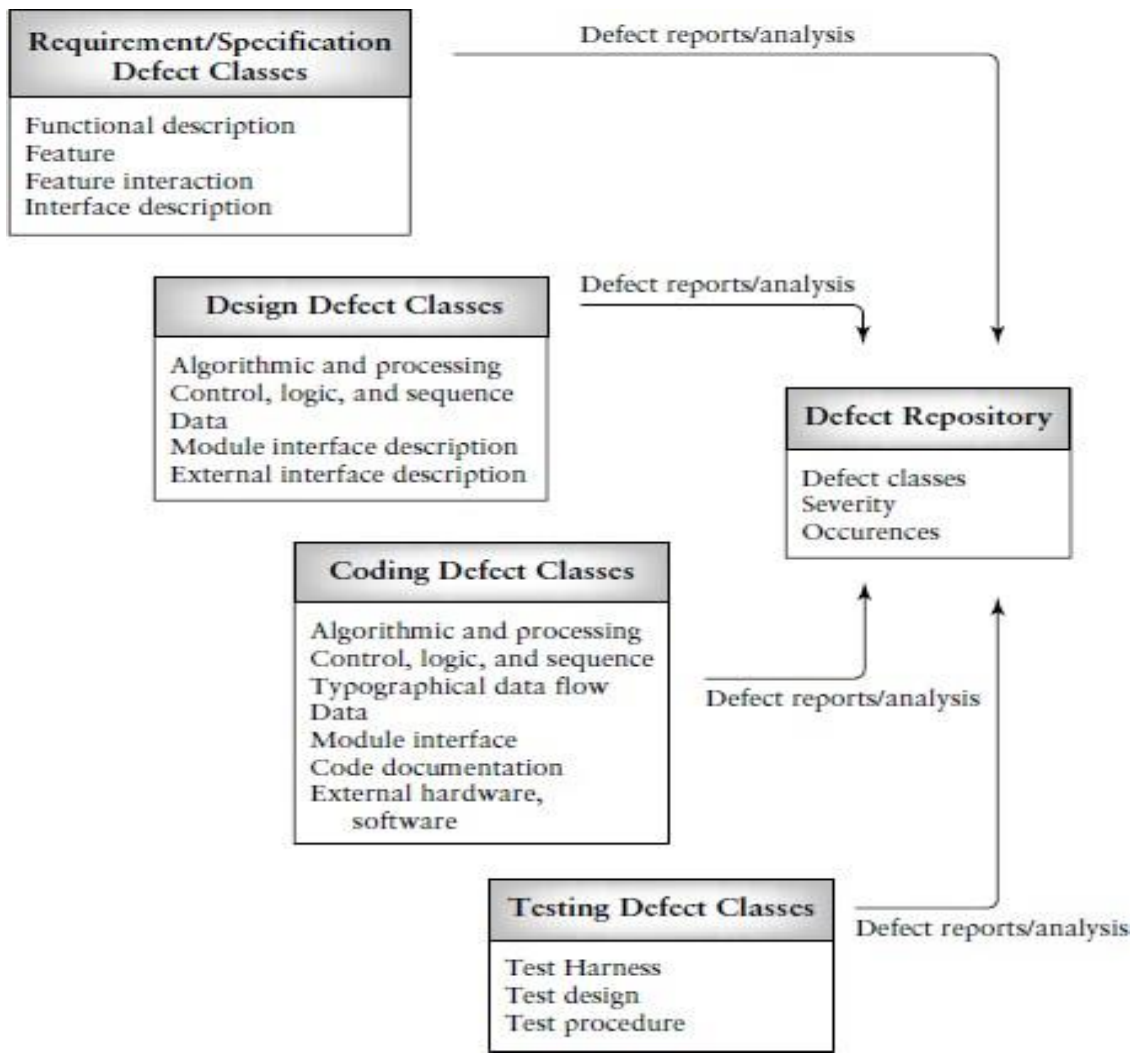
These are defects derived from , for example , using incorrect, and/or inconsistent parameter type, an incorrect number of parameters, or an incorrect ordering of parameters

5. **Functional Description Defects:-**

The defect in this category include incorrect, missing, and/or unclear design element . Eg the design may not properly describe the correct functionality of a module.

6. **External Interface Description Defects:-**

These are derive from in correct design description for interfaces with COTS components, external software systems, databases and hardware devices(eg:-I/O devices).Other example are user interface description defects where there are missing or improper commands, improper sequence of commands, lack of proper message, and/or lack of feedback message for the users.



<u>CODING DEFECTS:-</u>

Coding Defects are derived from error in implementing the code, coding defects are closely related to design classes especially if pseudo code has been used for detailed design. Some coding defects come from a failure to understand programming language constructs, and miscommunication with the designer.

1. **Algorithm and Processing Defects:-**

Adding levels of programming detail to design , code related algorithmic and processing defect would now include unchecked overflow and underflow conditions, comparing inappropriate data types, converting one data type to an other , in correct ordering of arithmetic operators, misuse or omission of parenthesis, precision loss and incorrect use of signs.

2. **Control, Logic and Sequence Defects :-**

On the coding level these would include incorrect expression of case statements, incorrect iteration of loops and missing paths

3. **Typographical Defects:-**

These are principally syntax errors, for example incorrect spelling of variable name, that are usually detected by compiler, self reviews, or peer reviews

4. **Initialization Defects:-**

These occur when initialization statements are omitted or are incorrect. This may occur because of misunderstanding or lack of communication between programmers, and /or programmers and designers, carelessness, or misunderstanding of programming environment.

5. **Data Flow Defects:-**

There are certain reasonable operational sequences that data should flow through. For example a variable should be initialized, before it is used in a calculation or a condition. It should not be initialized twice before there is an intermediate use. A variable should not be disregarded before it is used.

6. **Data Defects :-**

These are indicated by incorrect implementation of data structures. For example , the programmer may omit a filed in a record, an incorrect type or access is assigned to a file, an array may not be allocated the proper number of elements.

7. **Module Interface Defects:-**

As in the case of module design elements, interface defects in the code may be due to using in correct or inconsistent parameter types, an incorrect number of parameters, or improper ordering of the parameters, improper design, programmer may implement an incorrect sequence of calls or calls to nonexistent modules

8. **External Hardware, Software Interface Defects:-**

These defects arise form problems related to system calls, links to databases, input/output sequences, memory usage, resource usage, interrupts and exception handling , data exchange with hardware, protocols formats, interface with build files, and timing sequences

TESTING DEFECTS:-

Defects are not confined to code and its related artifacts. Test plans, test cases, test harnesses, and test procedures can also contain defects. Defects in test plans are best detected using review techniques.

1. **Test Harness Defect:-**

In order to test software, especially at the unit and integration levels, auxiliary code must be developed. This is called the test harness or **scaffolding** code. Test harness code should be carefully designed, implements and testes since it a work product and much of this code can be reused when the new release of the software are developed.

2. **Test Case Design and Test Procedure Defects:-**

These would encompass incorrect, incomplete, missing , inappropriate test cases, and test procedures. These defects are again best detected in test plan reviews. Defects are revealed during the testing process itself by means of a careful analysis of test conditions and test results. Repairs will then have to be made.

DEFECT EXAMPLES:- The Coin Problem Requirement Specification

Specification for program calculate_coin_value
This program calculates the total dollars and cents value for a set of coins. the user inputs the amount of pennies, nickels , dimes , quarters, half-dollars, and dollar coins held. There are six different denominations of coins. The program outputs the total dollar and cent values of the coins to the users

Input:number_of_coins is an integer
Outputs:- number_of_dollars is an integer
Number_of_cents is an integer

A spec above shows the sample informal specification for a simple program that calculates the total money value of a set of coins. The program could be a component of an incentive cash register system to support retail store clerks.

Coin Problem in Detail : (100 cent = 1 dollar)

if suppose input for coin values given as 1 for all then the calculation as shown below.

No of Coins (input)		**Coin Value**		
1(pennies)	**X**	**1**	=	**1**
1 (nickels)	**X**	**5**	=	**5**
1(dimes)	**X**	**10**	=	**10**
1(quarters)	**X**	**25**	=	**25**
1(half-dollars)	**X**	**50**	=	**50**
1(dollar)	**X**	**100**	=	**100**
				191

Output

No of Dollars : 1

No of Cents : 91

The given specification does not specify the above details clearly.

Requirements/ specifications defects,

1) Functional description defects

   a) No of coins, dollars , cents > 0

      pre & post conditions are helpful

   b) In each denomination – largest number allowed is missing

      upper limit for cents & dollars - not given

2) Interface description defect

   how user interact to provide i/p , o/p

Explanation

1) **Functional Description defects** arise because the functional description is ambiguous and incomplete .It does not state that the input, number_of_coins, and the output, number_of_dollars and number_of_cents, should all have values of zero or greater. The number_of_coins cannot be negative and the values in dollars and cents cannot be negative in the real world domain .

Formally stated set of [preconditions and post conditions would be helpful here, and would address some of the problem with the specification. These are also useful for designing black box tests.

A precondition is a condition that must be true in order for a software component to operate properly.

In this case a useful precondition would be one that states for example ,
Number_of_coins >=0

A Post condition is a condition that must be true when a software component completes its operation properly.

A useful post condition would be :-

Number_of _dollars, number_of_cents >=0

2) Interface Description defects

It is not clear from the specification how the user interacts with the program to provide input and how the output is to be reported. Because of ambiguities in the user interaction description the software may be difficult to use.

Design of Coin Problem

```
Design Description for Program Calculate_coin_values

Program calculate_coin_values
number_of_coins is integer
total _coin_value is integer
number_of_dollars is integer
number_of_cents is integer
coin_value is array of six integers representing each coins value in cents
initialized to 1,5,10,25,25,100
begin

initialize total_coin_value to zero
initialize loop_counter to one
wjile loop_counter is less than six
begin
    output "enter number of coins"
    read(number_of_coins)
total_coin_value=total_coin_value+ number_of_coins * coin_value[loop_counter]
increment loop_counter

end
number_dollars=total_coin_value/100
number_of_cents=total_coin_value-100 * number_of_dollars
output (number_of_dollars, number_of_cents)
end
```

Design Defect

1) Control Logic & Sequencing defect → while counter
2) Algorithmic & Processing Defect → invalid i/p value not checked
3) Data Defect → array value 25 - 2 times
4) External Interface Description defect → Order of i/p , when to stop , help msg , feedback not given

Explanation

1) **Control, Logic and Sequencing Defects:-**

The defect in this subclass arises from an incorrect "while" loop condition(should be less than or equal to six)

2) **Algorithmic and Processing Defects:-**

These arise from the lack of error checks for incorrect and /or invalid inputs, lack of path where users can correct erroneous inputs, lack of a path for recovery from input errors. The lack of an error check could be counted as functional design defects since the design does not adequately describe the proper functionality for the program .

3) Data Defects:-

This defect relates to an incorrect value for one of the elements of the integer array, coin_values, which should read 1,5,10,25,50,100

4) External Interface Description Defects:-

These are defects arising from the absence of input messages or prompts that introduced the program to the user and request inputs. The user has no way of knowing in which order the number of coins for each denomination must be input, and when to stop inputting values. There is absence of help message and feedback for user if he wishes to change an input or learn the correct format and order for inputting the number of coins.

The control and logic design defects are best addressed by white box based tests,(condition/Branch testing, loop testing).

The program below is a C like programming language. Without effectives reviews the specification and design defects could propagate to the code. Here additional defects have introduced in the coding phase.

Coin Problem Coding :

```
/*********** Program calculate_coin_values caluates the dollar and cents value
of a set of coins of different dominations input by the user denominations are
pennies, nickels, dimes ,quarters, half dollars and dollars **************/

main()
int total_coin_value;
int number_of_coin=0;
int number_of_dollar=0;
int number_of_cents=0;
int coin_value={1,5,10,15,25,25,30};
{
      int i=1;
      while (i < 6 )
            {
                  printf("input number of coins\n");
                  scanf("%d", number_of_coins);
      total_coin_value=total_coin_value+(number_of_coins *      coin_value{i]);
}
      i=i+1;
      number_of_dollars=total_coin_value/100;
      number_of_cents=total_coin_value-(100*number _of_dollars);
      printf("%d\n", number _of_dollars);
      printf("%d\n",number_of_cents);
}
/******************************/
```

Coin Problem Coding Defect :

1) Data flow defect → total_coin _value not initialized
2) Data Defect → Array   value 25 - 2 times
3) Control , logic and sequence Defect → While(i<6)
4) External Interface Description defect → scanf without &
5) Algorithmic and Processing Defects
6) Code Documentation Defect

1) **Data Flow Defects:-**

The variable total_coin_value is not initialized. It is used before it is defined.

2) **Data Defects:-**

The error in initializing the array coin_values is carried over form design and should be counted as design defect.

3) **Control , Logic and Sequence Defects:-**

These include the loop variable increment step which is out of the scope of the loop. Note that incorrect loop condition(i<6) is carried over from design and should be counted as a design defects

4) **External Hardware , Software Interface Defects:-**

The call to the external function "scanf" is incorrect. The address of the variable must be provided (&number_of_coins)

5) **Algorithmic and Processing Defects:-**

The division operator may cause problems if negative values are divided, although this problem could be eliminated with an input check.

6) **Code Documentation Defects:-**

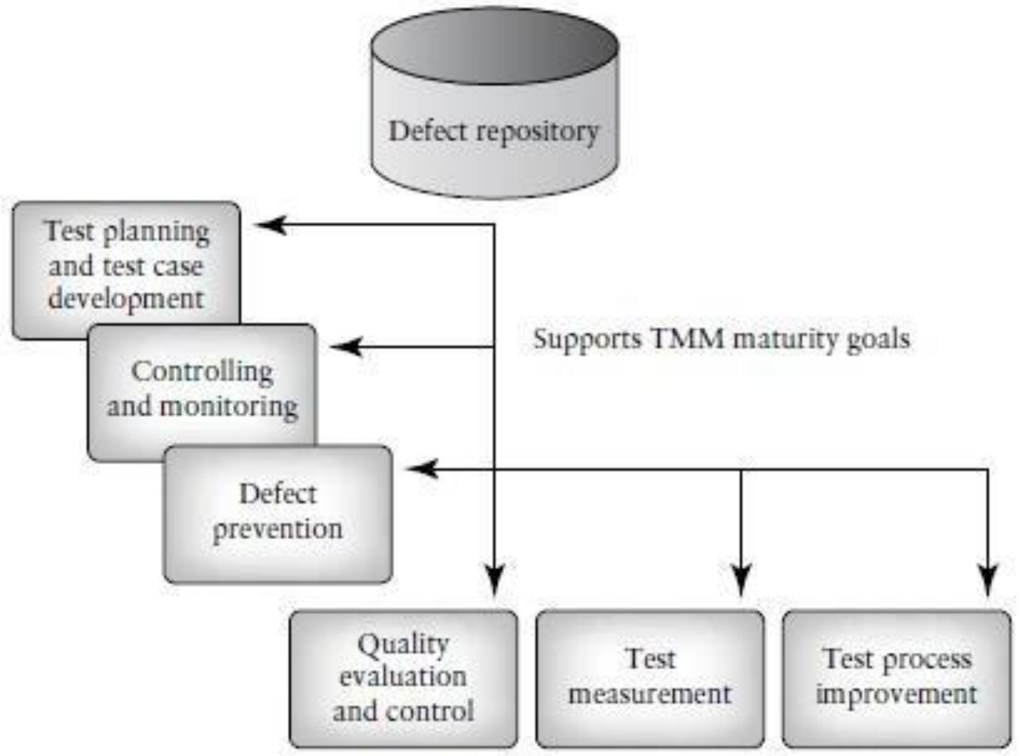
The documentation that accompanies this code is incomplete and ambiguous. It reflects the deficiencies in the external interface description and other defects that occurred during the specification and design. Vital information is missing for anyone who will need to repair, maintain or reuse this code.

The control, logic and sequence, data flow defects found in this sample could be detected by combination of white and black box testing techniques.

Black Box tests may work well to reveal the algorithmic and data defects. the code documentation defects require a code review for detection. The external software interface defects would probably be caught by a good complier.

Poor quality of this small program is due to defects injected during several of the life cycle phases with probable causes ranging from lack of education, a poor process, to oversight on the part of the designers and developers.

DEVELOPER/ TESTER SUPPORT FOR DEVELOPING A DEFECT REPOSITORY :-


Defect repository
Test planning and test case development
Controlling and monitoring
Defect prevention
Supports TMM maturity goals
Quality evaluation and control
Test measurement
Test process improvement

It is important if you are a member of a test organization to illustrate to management and colleagues the benefit of developing a defect repository to store defect information. Software Engineers and Test Specialists we should follow the example of engineers in other disciplines who realized the usefulness of defect data. Defect monitoring should continue for each ongoing projects. The distribution of defects will change as you make changes in your processes. The defect data is useful for test planning, a TMM level 2 maturity goals. It helps you to select applicable testing techniques, design and the test cases you need and allocate the amount of resources you will need to devote to detecting and removing these defects.

A defect repository can help to support achievements and continuous implementation of several TMM maturity goals including controlling and monitoring of test, software quality evaluation and control ,test measurements, and test process improvement.