SOFTWARE TESTING

Basics of Software Testing-I

In this chapter, we discuss the following topics:

1. Introduction to Software Testing
2. Understanding Error, Fault and Failure
3. Software Quality Attributes
4. Requirements, Behavior and Correctness
5. Correctness Vs Reliability
6. Testing and Debugging
7. Test Metrics

Summary

1. Introduction to Software Testing

1.1 Software: Software is a set of instructions to perform some task.

Software is used in many applications of the real world. Some of the examples are

Application software, such as word processors Firmware in a embedded system
Middleware, which controls and co-ordinates distributed systems
System software such as operating systems
Video Games
Websites

All of these applications need to run without any error and provide a quality service to the user of the application. In this regard the software has to be tested for its accurate and correct working.

1.2 Software Testing:

Testing can be defined in simple words as "Performing Verification and Validation of the Software Product" for its correctness and accuracy of working.

Other definitions of Software Testing:

Software testing is an investigation conducted to provide stakeholders with information about the quality of the product or service under test.
Software Testing also ensures whether the software program/application/product:

Meets the business and technical requirements that guided its design and development;
Works as expected; and
Can be implemented with the same characteristics.

Testing is done manually or using automated tools. Testing is done by a separate group of Testers. Testing is done right from the beginning of the software development life cycle till the end; it is delivered to the customer.

1.3 Functional Vs non-functional testing

Functional testing refers to tests that verify a specific action or function of the code. These are usually found in the code requirements documentation, although some development methodologies work from use cases or user stories. Functional tests tend to answer the question of "can the user do this" or "does this particular feature work".

Non-functional testing refers to aspects of the software that may not be related to a specific function or user action, such as scalability or security. Non-functional testing tends to answer such questions as "how many people can log in at once", or "how easy is it to hack this software".

2. Error, Fault and Failure:

Humans make errors in their thoughts, actions, and in the products that might result from their actions. Errors occur in the process of writing a program.

A programmer makes an error (mistake), which results in a defect (fault, bug) in the software source code. If this defect is executed, in certain situations the system will produce wrong results, causing a failure. Not all defects will necessarily result in failures. For example, defects in dead code will never result in failures. A defect can turn into a failure when the environment is changed. Examples of these changes in environment include the software being run on a new hardware platform, alterations in source data or interacting with different software. A single defect may result in a wide range of failure symptoms.

Not all software defects are caused by coding errors. One common source of expensive defects is caused by requirement gaps, e.g., unrecognized requirements that result in errors of omission by the program designer. A common source of requirements gaps is non-functional requirements such as testability, scalability, maintainability, usability, performance, and security.

Errors – Examples

Incorrect usage of software by users
Bad architecture and design by architects and designers Bad programming by developers
Inadequate testing by testers
Wrong build using incorrect configuration items by Build Team Member

Fault - Examples

A fault is the manifestation of one or more errors

An incorrect statement
Wrong data type
Wrong mathematical formula in design document
Missing functionality in the system

Failure

A failure occurs when a faulty piece of code is executed leading to incorrect state that f propagates to the program's output.

The following figure tells us how Error made by human will result in failure of the software.

Figure (1) to understand Error, Fault and Failure

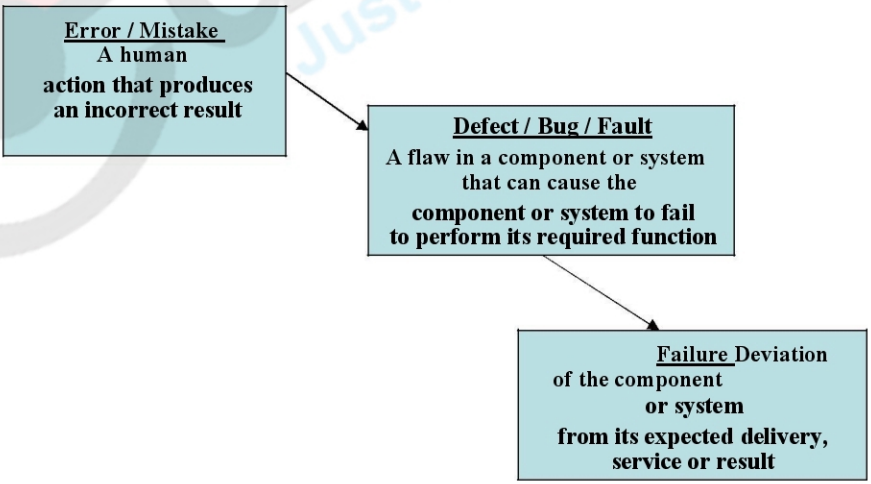
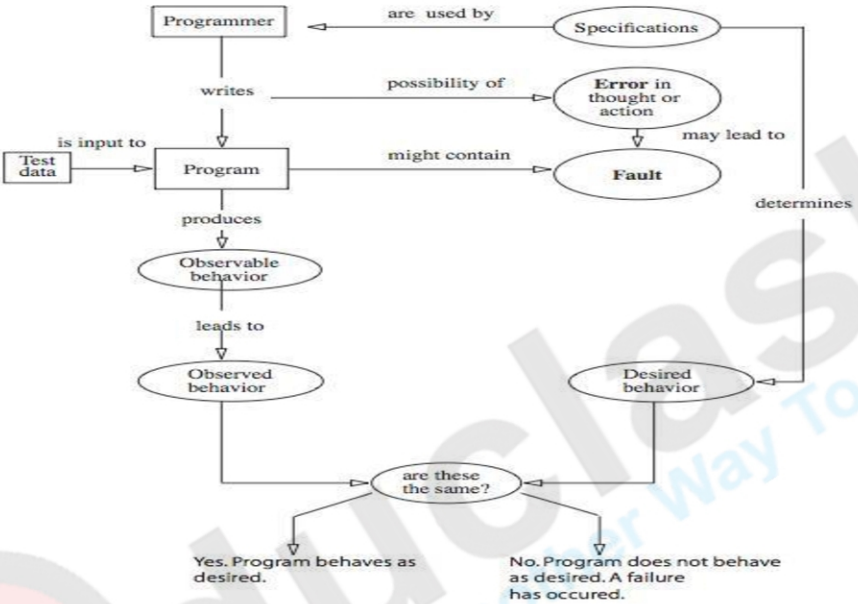


Figure (2) to understand Error, Fault and Failure:



Finding faults early

It is commonly believed that the earlier a defect is found the cheaper it is to fix it. The following table shows the cost of fixing the defect depending on the stage it was found. For example, if a problem in the requirements is found only post-release, then it would cost 10–100 times more to fix than if it had already been found by the requirements review.

		Time Detected				
		Requirements	Architecture	Construction	System Test	Post-Release
Time Introduced	Requirements	1×	3×	5–10×	10×	10–100×
	Architecture	-	1×	10×	15×	25–100×
	Construction	-	-	1×	10×	10–25×

Software Testing Objectives:

Testing is done to fulfill certain objectives

To discuss the distinctions between validation testing and defect testing
To describe the principles of system and component testing
To describe strategies for generating system test cases
To understand the essential characteristics of tool used for test automation
To find or prevent defects
To determine that software products satisfy specified requirements Ensuring that a system is ready for use
Gaining confidence that it works
Providing information about the level of quality
Determining user acceptability
Software quality measures how well software is designed (quality of design), and how well the software conforms to that design (quality of conformance).

3. Software quality:

1) **Conformance to specification**: Quality that is defined as a matter of products and services whose measurable characteristics satisfy a fixed specification – that is, conformance to an in beforehand defined specification.

2) **Meeting customer needs**: Quality that is identified independent of any measurable characteristics. That is, quality is defined as the products or services capability to meet customer expectations – explicit or not.

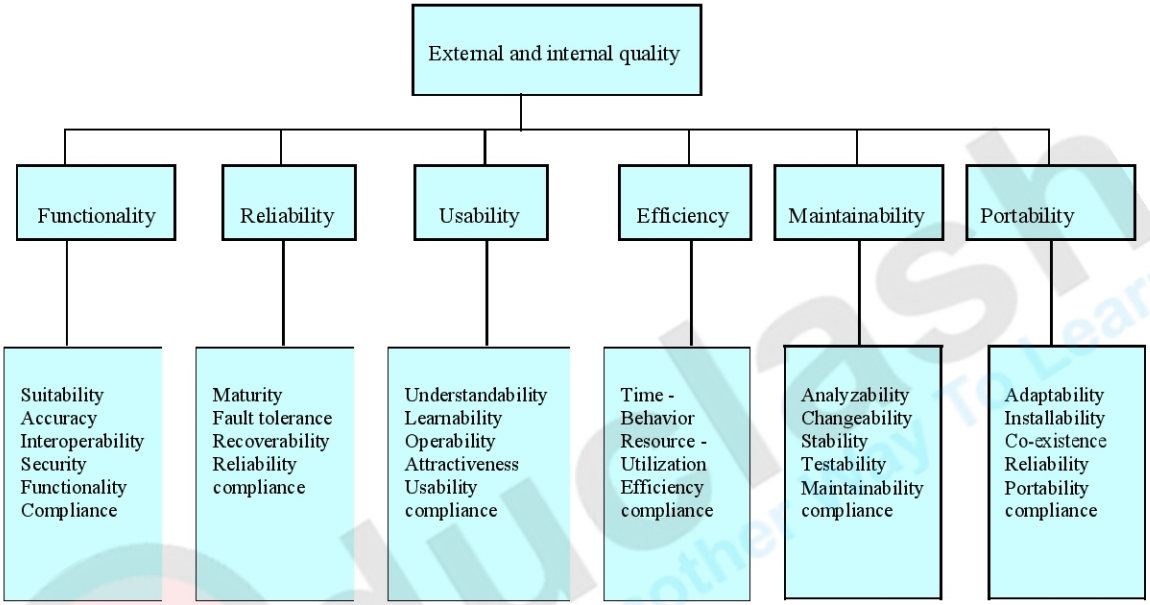
Software quality is a multidimensional quantity and is measurable.

To do this, we need to divide and measure software quality in terms of quality attributes:

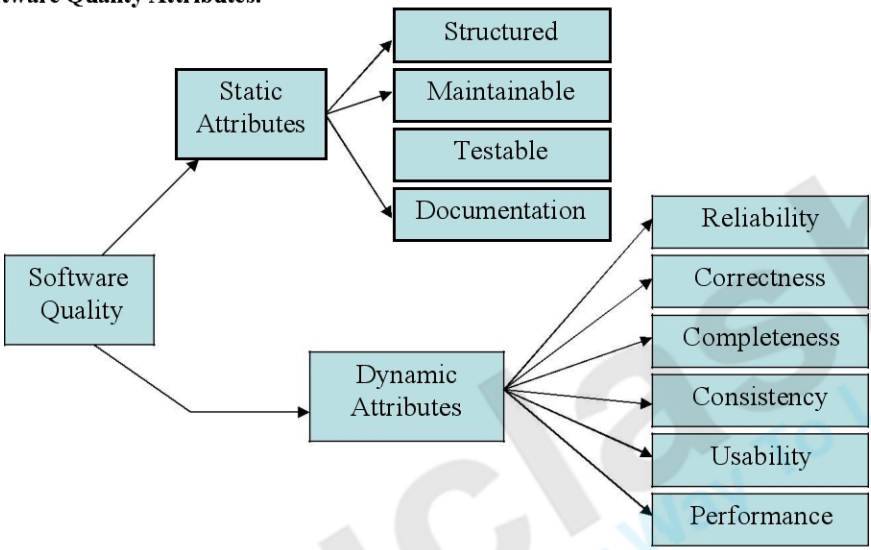
Static Quality Attributes

Dynamic Quality Attributes

The following figure shows the different quality attributes:



Software Quality Attributes:



Software Quality Attributes:
- **Static Attributes:**

1. Maintainability and its Sub-characteristics:

In software engineering, the ease with which a software product can be modified in order to:

correct defects
meet new requirements
make future maintenance easier, or
cope with a changed environment

These activities are known as **software maintenance.**

A set of attributes that bear on the effort needed to make specified modifications are:

1.1 Analyzability: Attributes of software that bear on the effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified

1.2 Changeability: Attributes of software that bear on the effort needed for modification, fault removal or for environmental change

1.3 Stability: Attributes of software that bear on the risk of unexpected effect of modifications

1.4Testability: Attributes of software that bear on the effort needed for validating the modified software. The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria are met

Static Testability Ex: Software Complexity
Dynamic Testability Ex: Test Coverage Criteria

Software Quality Attributes:
- **Dynamic Attributes:**

1. Completeness: The availability of all the features listed in the requirements or in the user manual.

2. Consistency: adherence to a common set of conventions and assumptions.

2.1 Compliance: Attributes of software that make the software adhere to application related standards or conventions or regulations in laws and similar prescriptions

2.2 Conformance : Attributes of software that make the software adhere to standards or conventions relating to portability.

3. Usability: The ease with which an application can be used. Usability testing also refers to testing of a product by its potential users

3.1Understandability: Attributes of software that bear on the users' effort for recognizing the logical concept and its applicability

3.2 Learnability: Attributes of software that bear on the users' effort for learning its application

3.3Operability : Attributes of software that bear on the users' effort for operation and operation control

4. Performance: The time the application takes to perform a requested task. Performance is considered as a nonfunctional requirement.

4.1 Time behavior: Attributes of software that bear on response and processing times and on throughput rates in performances its function

4.2 Resource behavior: Attributes of software that bear on the amount of resource used and the duration of such use in performing its function

5. Reliability: Software Reliability is the probability of failure-free operation of software over a given time interval and under given conditions

5.1 Maturity: Attributes of software that bear on the frequency of failure by faults in the software.

5.2 Fault tolerance: Attributes of software that bear on its ability to maintain a specified level of performance in case of software faults or of infringement of its specified interface.

5.3 Recoverability: Attributes of software that bear on the capability to re-establish its level of performance and recover the data directly affected in case of a failure and on the time and effort needed for it.

For example, the Transmission Control Protocol (TCP) is designed to allow reliable two-way communication in a packet-switched network, even in the presence of communications links which are imperfect or overloaded. It does this by requiring the endpoints of the communication to expect packet loss, duplication, reordering and corruption, so that these conditions do not damage data integrity, and only reduce throughput by a proportional amount.

Data formats may also be designed to degrade gracefully. HTML for example, is designed to be forward compatible, allowing new HTML entities to be ignored by Web browsers which do not understand them without causing the document to be unusable.

Recovery from errors in fault-tolerant systems can be characterized as either roll-forward or roll-back. When the system detects that it has made an error, roll-forward recovery takes the system state at that time and corrects it, to be able to move forward. Roll-back recovery reverts the system state back to some earlier, correct version, for example using check pointing, and moves forward from there.

6. Correctness: The correct operation of an application

6.1 Accurateness: Attributes of software that bear on the provision of right or agreed results or effects

6.2 Suitability: Attributes of software that bear on the presence and appropriateness of a set of functions for specified tasks

7. Correctness: It attempts to establish that the program is error- free, testing attempts to find if there are any errors in it.

4. Requirement, Behavior and Correctness:

Requirements specify the "function or characteristic of a system that is necessary for the quantifiable and verifiable behaviors that a system must possess and constraints that a system must work within to satisfy an organization's objectives and solve a set of problems".

The documented representation of requirement is requirement specification.
The review of requirement specification involves
- Arriving at Team understanding of the Requirements and
- Reviewing Requirements Specification document to determine Quality Attributes

Step 1# Arriving at Team understanding of the Requirements

- Delivering Quality Product means meeting Stake Holders expectations.

- Needs review of requirements and understanding by relevant stake holders on acceptance

Step2# Review Requirements Specification document to determine Quality Attributes

- Complete: Nothing is missing
- Consistent: Does not conflict with other requirements
- Correct: Accurately states a customer need
- Feasible: Can be implemented within known constraints
- Modifiable: Can be easily changed
- Necessary: Documents something customers really need
- Prioritized: Ranked as to importance of inclusion in product
- Testable: Tests can be devised to demonstrate correct implementation
- Traceable: Linked to system requirements, and to designs, code, and tests
- Unambiguous: Has only one possible meaning

Example
Requirements for two different programs:

Requirement 1: It is required to write a program that inputs two integers and outputs the maximum of these

Requirement 2: It is required to write a program that inputs a sequence of integers and outputs the sorted version of this sequence

Example of Requirements: Incompleteness

Suppose that program **max** is developed to satisfy Requirement 1. The expected output of **max** when the input integers are 13 and 19 can be easily determined to be 19.

Suppose now that the tester wants to know if the two integers are to be input to the program on one line followed by a carriage return, or on two separate lines with a carriage return typed in after each number. The requirement as stated above fails to provide an answer to this question.

Example of Requirements: Ambiguity

Requirement 2 is ambiguous. It is not clear whether the input sequence is to be sorted in ascending or in descending order. The behavior of **sort** program, written to satisfy this requirement, will depend on the decision taken by the programmer while writing **sort**.

Input domain (Input space)
The set of all possible inputs to a program **P** is known as the input domain or input space, of **P**.

Using Requirement 1 above we find the input domain of **max** to be the set of all pairs of integers where each element in the pair integers is in the range -32,768 till 32,767.

Using Requirement 2 it is not possible to find the input domain for the sort program.

Modified Requirement 2:

It is required to write a program that inputs a sequence of integers and outputs the integers in this sequence sorted in either ascending or descending order.

The order of the output sequence is determined by an input request character which should be ``A'' when an ascending sequence is desired, and ``D'' otherwise.

While providing input to the program, the request character is input first followed by the sequence of integers to be sorted; the sequence is terminated with a period.

Based on the above modified requirement, the input domain for **sort** is a set of pairs. The first element of the pair is a character. The second element of the pair is a sequence of zero or more integers ending with a period.

Valid/Invalid Inputs

The modified requirement for sort mentions that the request characters can be ``A'' and ``D'', but fails to answer the question ``What if the user types a different character ?''.

When using **sort** it is certainly possible for the user to type a character other than ``A'' and ``D''. Any character other than ``A'' and ``D'' is considered as invalid input to **sort**. The requirement for **sort** does not specify what action it should take when an invalid input is encountered.

5. Correctness Vs Reliability:

The correctness will be established via requirement specification and the program text to prove that software is behaving as expected.

The reliability is the probability of the successful execution of program on randomly selected elements from its input domain.

Though correctness of a program is desirable, it is almost never the objective of testing.

To establish correctness via testing would imply testing a program on all elements in the input domain. In most cases that are encountered in practice, this is impossible to accomplish.

Thus correctness is established via mathematical proofs of programs.

While correctness attempts to establish that the program is error free, testing attempts to find if there are any errors in it.

Thus completeness of testing does not necessarily demonstrate that a program is error free.

6. Testing Vs Debugging:

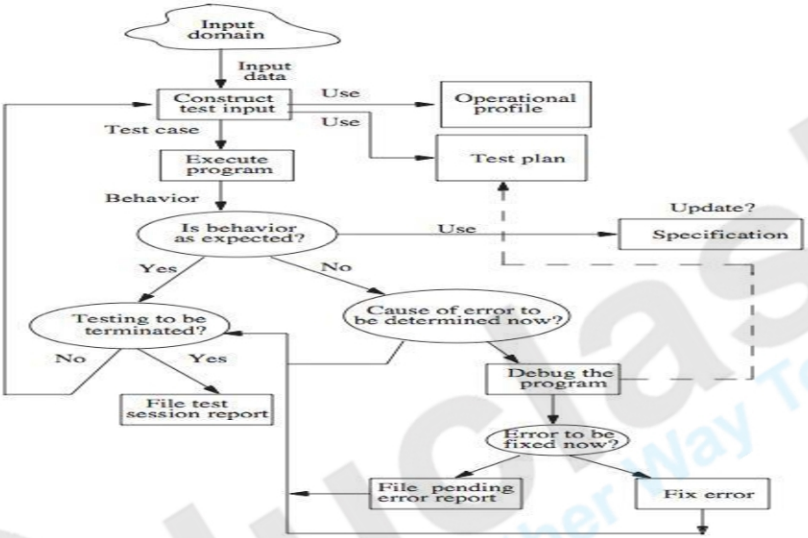
Testing is the process of determining if a program has any errors.

When testing reveals an error, the process used to determine the cause of this error and to remove it, is known as debugging.

- Testing catches and reports bugs.
- Testing reduces the probability of undiscovered bugs remaining in the software
- Testing is not a proof of correctness

- Testing can be planned with allocation of effort and schedule, resources, also, having criteria on when to stop testing.
- Testing starts with known conditions like what to test, test input, expected output and uses test procedures.
- Testing shows that bugs are present in a program, but cannot prove that there are no bugs
- There is no need to know design to carry-out testing
- Good testing is done by an outsider that is other than the team who develops the code
- Test automation in order to store and execute test cases can be done
- Debugging is the process of analyzing causes behind the bugs and removing them
- Debugging starts with a list of reported bugs with unknown initial conditions.
- In debugging it is not possible to plan and estimate schedule and effort for debugging
- Debugging is a problem solving involving deduction
- Detailed design knowledge is of great help in good debugging
- Debugging is done by the development team and hence is an insider's work
- Automation of debugging is not in place

The figure shows a test/debug cycle:



- Execute the program on an empty input sequence.

- Test the program for robustness against erroneous inputs such as "R'' typed in as the request character.

- All failures of the test program should be recorded in a suitable file using the Company Failure Report Form.

Software Testing Life Cycle:

Software testing life cycle identifies what test activities to carry out and when (what is the best time) to accomplish those test activities. Even though testing differs between organizations, there is a testing life cycle.

Software Testing Life Cycle consists of six (generic) phases:

Test Planning,
Test Analysis,
Test Design,
Construction and verification,

Testing Cycles,
Final Testing and Implementation
and Post Implementation.

Software testing has its own life cycle that intersects with every stage of the SDLC. The basic requirements in software testing life cycle is to control/deal with software testing – Manual, Automated and Performance.

Test Planning

This is the phase where Project Manager has to decide what things need to be tested, do I have the appropriate budget etc. Naturally proper planning at this stage would greatly reduce the risk of low quality software. This planning will be an ongoing process with no end point.

Activities at this stage would include preparation of high level test plan-(according to IEEE test plan template The Software Test Plan (STP) is designed to prescribe the scope, approach, resources, and schedule of all testing activities. The plan must identify the items to be tested, the features to be tested, the types of testing to be performed, the personnel responsible for testing, the resources and schedule required to complete testing, and the risks associated with the plan.). Almost all of the activities done during this stage are included in this software test plan and revolve around a test plan.

Test Analysis

Once test plan is made and decided upon, next step is to delve little more into the project and decide what types of testing should be carried out at different stages of SDLC, do we need or plan to automate, if yes then when the appropriate time to automate is, what type of specific documentation I need for testing.

Proper and regular meetings should be held between testing teams, project managers, and development teams, Business Analysts to check the progress of things which will give a fair idea of the movement of the project and ensure the completeness of the test plan created in the planning phase, which will further help in enhancing the right testing strategy created earlier. We will start creating test case formats and test cases itself. In this stage we need to develop Functional validation matrix based on Business Requirements to ensure that all system requirements are covered by one or more test cases, identify which test cases to automate, begin review of documentation, i.e. Functional Design, Business Requirements, Product Specifications, Product Externals etc. We also have to define areas for Stress and Performance testing.

Test Design

Test plans and cases which were developed in the analysis phase are revised. Functional validation matrix is also revised and finalized. In this stage risk assessment criteria is

developed. If you have thought of automation, then you have to select which test cases to automate and begin writing scripts for them. Test data is prepared. Standards for unit testing and pass / fail criteria are defined here. Schedule for testing is revised (if necessary) & finalized and test environment is prepared.

Construction and verification

In this phase we have to complete all the test plans, test cases, complete the scripting of the automated test cases, Stress and Performance testing plans needs to be completed. We have to support the development team in their unit testing phase. And obviously bug reporting would be done as when the bugs are found. Integration tests are performed and errors (if any) are reported.

Testing Cycles

In this phase we have to complete testing cycles until test cases are executed without errors or a predefined condition is reached. Run test cases --> Report Bugs --> revise test cases (if needed) --> add new test cases (if needed) --> bug fixing --> retesting (test cycle 2, test cycle 3….).

Final Testing and Implementation

In this we have to execute remaining stress and performance test cases, documentation for testing is completed / updated, provide and complete different matrices for testing. Acceptance, load and recovery testing will also be conducted and the application needs to be verified under production conditions.

Post Implementation

In this phase, the testing process is evaluated and lessons learnt from that testing process are documented. Line of attack to prevent similar problems in future projects is identified. Create plans to improve the processes. The recording of new errors and enhancements is an ongoing process. Cleaning up of test environment is done and test machines are restored to base lines in this stage.

Example for Test plan

A test cycle is often guided by a test plan.

Example: The sort program is to be tested to meet the requirements given earlier. Specifically, the following needs to be done.

1. Execute sort on at least two input sequences, one with ``A'' and the other with ``D'' as request characters.

2. Execute the program on an empty input sequence.

3. Test the program for robustness against erroneous inputs such as “R" typed in as the request character.

4. All failures of the test program should be recorded in a suitable file using the Company Failure Report Form..

Test case/data

A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

A test case is a pair consisting of test data to be input to the program and the expected output. The test data is a set of values, one for each input variable.

A test set is a collection of zero or more test cases.

A test case in software engineering is a set of conditions or variables under which a tester will determine whether an application or software system is working correctly or not. The mechanism for determining whether a software program or system has passed or failed such a test is known as a test oracle. In some settings, an oracle could be a requirement or use case, while in others it could be a heuristic. It may take many test cases to determine that a software program or system is functioning correctly. Test cases are often referred to as test scripts, particularly when written. Written test cases are usually collected into test suites.

Contents

Formal test cases

In order to fully test that all the requirements of an application are met, there must be at least two test cases for each requirement: one positive test and one negative test; unless a requirement has sub-requirements. In that situation, each sub-requirement must have at least two test cases. Keeping track of the link between the requirement and the test is frequently done using a traceability matrix. Written test cases should include a

description of the functionality to be tested, and the preparation required to ensure that the test can be conducted.

What characterizes a formal, written test case is that there is a known input and an expected output, which is worked out before the test is executed. The known input should test a precondition and the expected output should test a post condition.

Informal test cases

For applications or systems without formal requirements, test cases can be written based on the accepted normal operation of programs of a similar class. In some schools of testing, test cases are not written at all but the activities and results are reported after the tests have been run.

In scenario testing, hypothetical stories are used to help the tester think through a complex problem or system. These scenarios are usually not written down in any detail. They can be as simple as a diagram for a testing environment or they could be a description written in prose. The ideal scenario test is a story that is motivating, credible, complex, and easy to evaluate. They are usually different from test cases in that test cases are single steps while scenarios cover a number of steps.

Typical written test case format

A test case is usually a single step, or occasionally a sequence of steps, to test the correct behavior/functionalities, features of an application. An expected result or expected outcome is usually given.

Additional information that may be included:
test case ID
test case description
test step or order of execution number
related requirement(s)
depth
test category
author
check boxes for whether the test is automatable and has been automated.

Additional fields that may be included and completed when the tests are executed: pass/fail
remarks

Larger test cases may also contain prerequisite states or steps, and descriptions.

A written test case should also contain a place for the actual result.

These steps can be stored in a word processor document, spreadsheet, database or other common repository.

In a database system, you may also be able to see past test results and who generated the results and the system configuration used to generate those results. These past results would usually be stored in a separate table.

Test suites often also contain
Test summary
Configuration

Besides a description of the functionality to be tested, and the preparation required to ensure that the test can be conducted, the most time consuming part in the test case is creating the tests and modifying them when the system changes.

Under special circumstances, there could be a need to run the test, produce results, and then a team of experts would evaluate if the results can be considered as a pass. This happens often on new products' performance number determination. The first test is taken as the base line for subsequent test / product release cycles.

Acceptance tests, which use a variation of a written test case, are commonly performed by a group of end-users or clients of the system to ensure the developed system meets the requirements specified or the contract. User acceptance tests are differentiated by the inclusion of happy path or positive test cases to the almost complete exclusion of negative test cases. The Sample test cases are discussed below:

Sample test case for sort:

Test data: <"A" 12 -29 32 >
Expected output: -29 12 32

Test Case 2
Test data: <"D" 12 -29 32. >
Expected output: 32 12 -29

Test Case 3
Test data: <"A" .>
Expected output: No input to be sorted in ascending order

Test Case 4
Test data: <"D". >
Expected output: No input to be sorted in descending order

Test Case 5
Test data: <"R" 12 -29 32. >
Expected output: Invalid request character; valid characters 'A' and 'D'

Test Case 6
Test data: <"D" c,17,2 . >
Expected output: Invalid number

- Test Cases 1 and 2 corresponds to Test Plan 1
- Test Case 3 and 4 corresponds to Test Plan 2
- Test Case 5 corresponds to Test Plan 3

Executing the Program for testing it:

Often a Tester might construct a Test Harness to aid in Program execution

Test Harness

In software testing, a test harness or automated test framework is a collection of software and test data configured to test a program unit by running it under varying conditions and monitoring its behavior and outputs. It has two main parts: the Test execution engine and the Test script repository.

Test harnesses allow for the automation of tests. They can call functions with supplied parameters and print out and compare the results to the desired value. The test harness is a hook to the developed code, which can be tested using an automation framework.

A test harness should allow specific tests to run (this helps in optimizing), orchestrate a runtime environment, and provide a capability to analyze results.

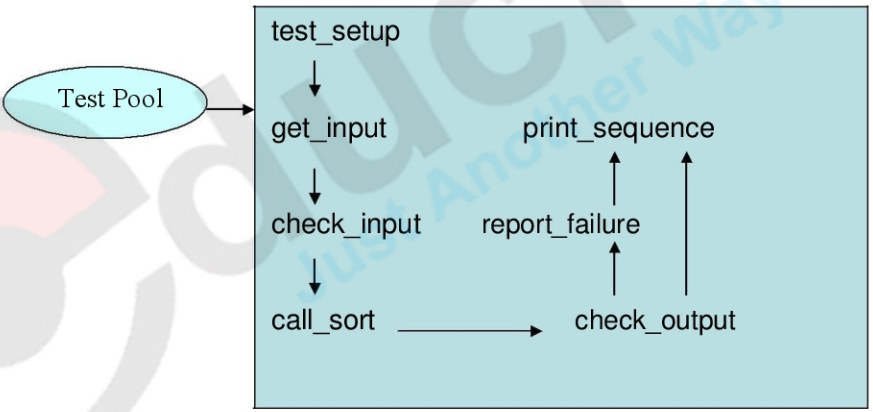
The typical objectives of a test harness are to:

Automate the testing process.
Execute test suites of test cases.
Generate associated test reports.

A test harness may provide some of the following benefits:

Increased productivity due to automation of the testing process.
Increased probability that regression testing will occur.
Increased quality of software components and application.

The figure gives an example of Test Harness:



The Test Harness Reads an input sequence, checks for its correctness, and then calls sort.

The sorted array sort_sequence returned by sort is printed using print_sequence

The Test Cases are assumed to be in the Test Pool

Assumptions: sort is code as a procedure; the get_input procedure reads the request character and the sequence to be sorted into variables request_char, num_items, and in_numbers; the input is checked prior to calling sort by the check_input.

The test_setup procedure is invoked first to setup the test data; identifying and opening the file containing tests;

The check_output procedure serves as the oracle that checks if the program under test behaves correctly.

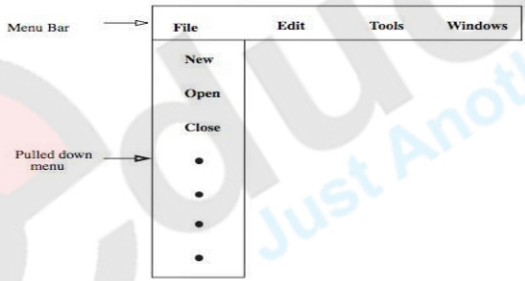
The report_failure procedure is invoked in case the output from sort is incorrect.

Program behavior

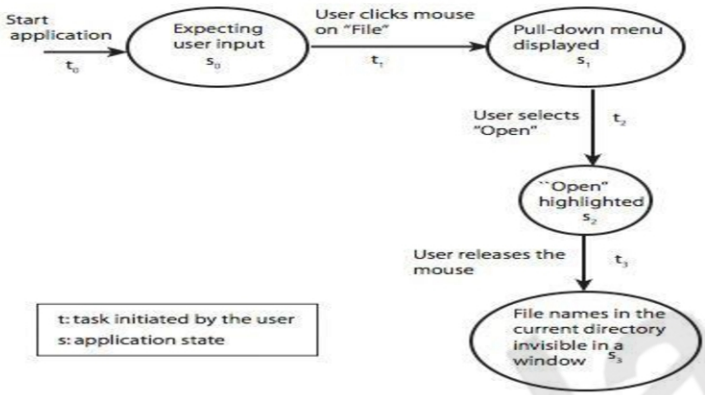
Can be specified in several ways: plain natural language, a state diagram, formal mathematical specification, etc.

A state diagram specifies program states and how the program changes its state on an input sequence.

Program behavior: Example
Consider a menu driven application.



Program behavior: Example



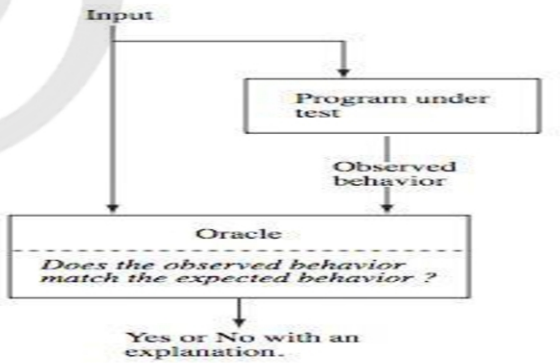
Behavior: observation and analysis

In the first step one observes the behavior.

In the second step one analyzes the observed behavior to check if it is correct or not. Both these steps could be quite complex for large commercial programs.

The entity that performs the task of checking the correctness of the observed behavior is known as an oracle.

Oracle: Example



Oracle: Programs

Oracles can also be programs designed to check the behavior of other programs.

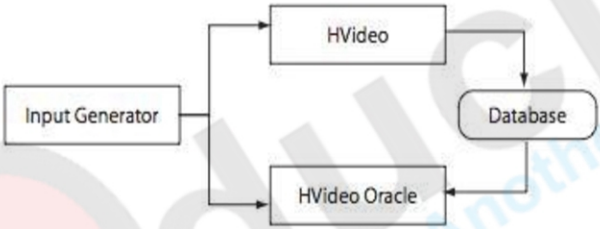
For example, one might use a matrix multiplication program to check if a matrix inversion program has produced the correct output. In this case, the matrix inversion program inverts a given matrix A and generates B as the output matrix.

Oracle: Construction

Construction of automated oracles, such as the one to check a matrix multiplication program or a sort program, requires the determination of input-output relationship.

In general, the construction of automated oracles is a complex undertaking.

Oracle construction: Example

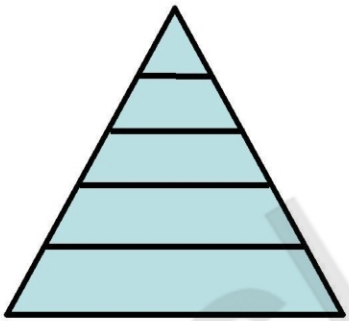


Test Metrics

Quantitative measurement determining the extent to which a software process, product or project possesses a certain
attribute (used for tracking purposes)

Goal for the metric is to quantify the progress of the product toward a specified quality objective

Process Metrics:
feedback to improve the process, productivity



Project Metrics
used to track project progress

Product Metrics
used to track quality of product (Static, Dynamic)

Organizational Metrics

Metrics at organizational level are useful in project planning and management.
Examples:
- Defects per thousand lines of code (defects/KLOC)
- Testing cost per KLOC
- Actual schedule of system testing
- Delivery schedule slippage

Project Metrics

Project Metrics relates to a specific project. Useful in the monitoring and control of the project

Examples:
- The ratio of actual to expected system test effort
- Ratio of number of successful tests to the total number of tests.

Process Metrics

Every project uses the process. The goal of Process Metrics to measure the goodness of the process

Examples:

- The number of errors found in different testing phases

Product Metrics

Product metrics are useful in making decisions related to the product. For example should the product be released to the client?

Examples:

- Cyclomatic complexity:

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

- V(G) is the complexity of the control flow graph. E is the edges, N nodes, p the connected procedures

Software Testing
Unit-1

- Introduction
- Importance of testing in SDLC
- Testing life cycle
- Test planning
- Types of testing
- Verification & Validation
- Quality Assurance & Control
- Bug reporting

Topics

- What is Software ?
- Types of Software
- What is Software testing ?
- Why Software Testing ?
- Who should Test ?

What is Software ?

- Software is a set of instruction to perform some task

Types of Software:-

- Application software, such as word processors
- Firmware in a embedded system(permanent software programmed into a read-only memory)
- Middleware, it acts as a bridge between an operating system or database and applications, especially on a network.
- System software such as operating systems
- Video Games
- Websites

Software Testing

Software testing is a process used to identify the correctness, completeness and quality of developed computer software.

It is the process of executing a program / application under positive and negative conditions by manual or automated means. It checks for the :-

- Specification
- Functionality
- Performance

Why Software Testing ?

- Disney had done a deal with Compaq Computers to ship the Lion King Game pre-installed on a million Compaq Computers destined as Christmas presents for children everywhere across the country(The Wall Street Journal published a feature article about Disney spoiling Christmas for children everywhere as the much anticipated Disney Lion King game blu-screened on computers across the country Christmas morning)
- Software Testing is important as it may cause mission failure, impact on operational performance and reliability if not done properly.
- Effective software testing delivers quality software products satisfying user's requirements, needs and expectations.

What ….????

…is an ”ERROR”??

….is a ”Bug”??

….is Fault, Failure ??

Bug, Fault & Failure

A person makes an **Error**
That creates a **fault** in software
That can cause a **failure** in operation

Error : An error is a human action that produces the incorrect result that results in a fault.

Bug : The presence of error at the time of execution of the software.

Fault : State of software caused by an error.

Failure : Deviation of the software from its expected result. It is an event.

Who is a Software Tester??..

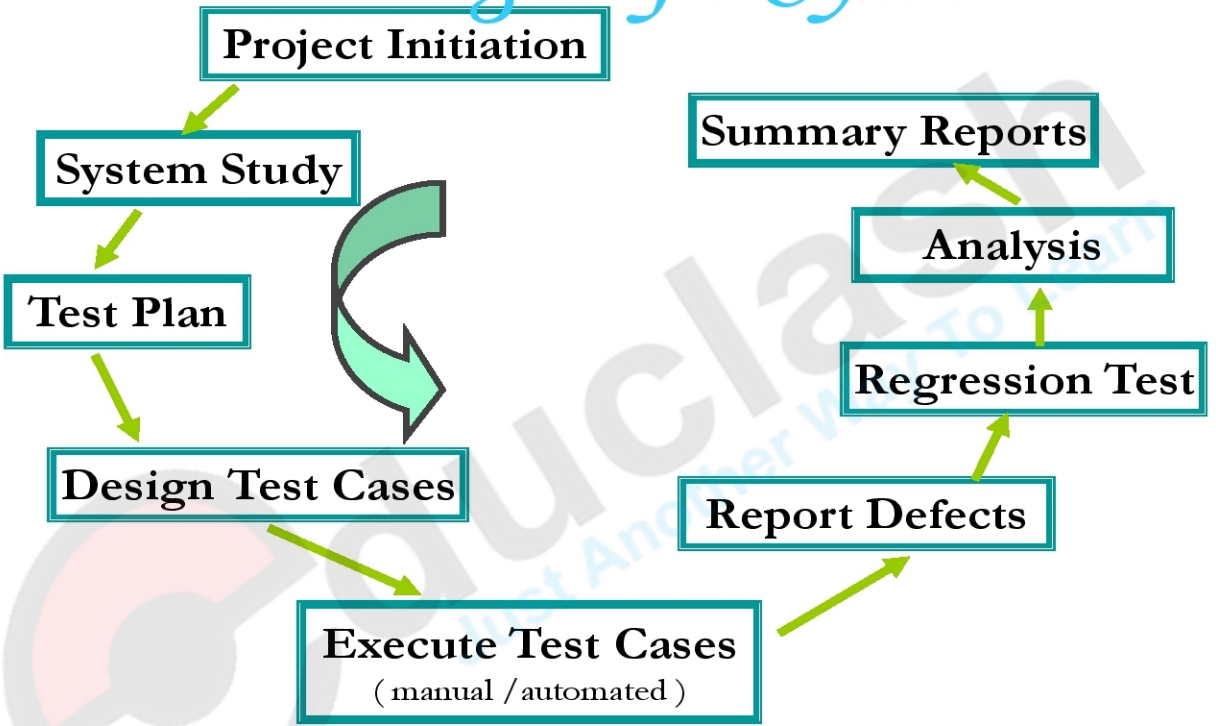
Software Tester is the one who performs testing and find bugs, if they exist in the tested application.

When to Start Testing in SDLC

- Requirement •
Analysis
- Design •
Coding •
Testing
- Implementation •
Maintenance

❖ ***Testing starts from Requirement Phase***

Testing Life Cycle



The Testing Team

✓ Program Manager-

- The planning and execution of the project to ensure the success of a project minimizing risk throughout the lifetime of the project.
- Responsible for writing the product specification, managing the schedule and making the critical decisions and trade-offs.

✓ QA Lead-

- Coach and mentor other team members to help improve QA effectiveness
- Work with other department representatives to collaborate on joint projects and initiatives
- Implement industry best practices related to testing automation and to streamline the QA Department.

Test Analyst\Lead-

- Responsible for planning, developing and executing automated test systems, manual test plans and regressions test plans.
- Identifying the Target Test Items to be evaluated by the test effort
- Defining the appropriate tests required and any associated Test Data
- Gathering and managing the Test Data
- Evaluating the outcome of each test cycle

Test Engineer-

- Writing and executing test cases and Reporting defects
- Test engineers are also responsible for determining the best way a test can be performed in order to achieve 100% test coverage of all components

Test Plan

A test plan is a systematic approach to testing a system i.e. software. The plan typically contains a detailed understanding of what the eventual testing workflow will be.

Test Case

A test case is a specific procedure of testing a particular requirement.

It will include:

- Identification of specific requirement tested
- Test case success/failure criteria
- Specific steps to execute test
- Test Data

Unit Testing

1. Test each module individually.
2. Follows a white box testing (Logic of the program)
3. Done by Developers

Integration Testing

After completing the unit testing and dependent modules development, programmers connect the modules with respect to HLD for Integration Testing through below approaches.

System Testing

After completing **Unit** and **Integration** testing through white box testing techniques development team release an .exe build (all integrated module) to perform black box testing.

- Usability Testing
- Functional Testing
- Performance Testing
- Security Testing

Usability Testing

During this test, testing team concentrates on the user friendliness of build interface. It consists of following sub tests.

- **User Interface Test:** Ease of use (screens should be understandable to operate by End User)
- **Look & Feel :-** attractive
- **Speed in interface :-** Less number of task to complete task
- **Manual Support Test :-** Context sensitiveness of user manual.

Functional Testing

- The process of checking the behavior of the application.
- It is geared to functional requirements of an application.
- To check the correctness of outputs.
- Data validation and Integration i.e. inputs are correct or not.

Performance Testing

- **LOAD TESTING –** Also Known as Scalability Testing. During this test, test engineers execute application build under customer expected configuration and load to estimate performance.

- **STRESS TESTING –** During this test, Test engineers estimates the peak load. To find out the maximum number of users for execution of out application user customer expected configuration to estimate peak load.

  PEAK LOAD > CUSTOMER LOAD (EXPECTED)

- **DATA VOLUME TESING --** Testing team conducts this test to find the maximum limit of data volume of your application.

Security Testing

Testing how well the system protects against unauthorized internal or external access, willful damage, etc, may require sophisticated testing techniques

Smoke testing

'**Smoke Testing**' is came from the hardware testing, in the hardware testing initial pass is done to check if it did not catch the fire or smoked in the initial switch on

Alpha Testing

1. The application is tested by the users who doesn't know about the application.
2. Done at developer's site under controlled conditions
3. Under the supervision of the developers.

Acceptance Testing

A formal test conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system.

It is the final test action before deploying the software. The goal of acceptance testing is to verify that the software is ready and can be used by the end user to perform the functions for which the software was built.

Beta Testing

1. This Testing is done before the final release of the software to end-users.
2. Before the final release of the software is released to users for testing where there will be no controlled conditions and the user here is free enough to do what ever he wants to do on the system to find errors.

Regression Testing

Testing with the intent of determining if bug fixes have been successful and have not created any new problems. Also, this type of testing is done to ensure that no degradation of baseline functionality has occurred.

Monkey Testing

Testing the application randomly like hitting keys irregularly and try to breakdown the system there is no specific test cases and scenarios for monkey testing.

Black Box v/s White Box

Black Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is not known to the tester. Tester is mainly concerned with the validation of output rather than how the output is produced(functionality of the item under test is not important from tester's pov).

White Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is known to the tester. tester validates the internal structure of the item under consideration along with the output.

Programming knowledge and implementation knowledge (internal structure and working) is required in **White Box testing**, which is not necessary in **Black Box testing**.

Black box testing is done by the professional testing team and can be done without knowledge of internal coding of the item. **White Box testing** is generally done by the programmers who have developed the item or the programmers who have an understanding of the item's internals

Correctness Vs Reliability

What is Correctness?

- Correctness can be defined as the support to the specifications that determine how users can interact with the software and how the software should behave when it is used correctly.
- If the software behaves incorrectly, it might take considerable amount of time to achieve the task or sometimes it is impossible to achieve it.

Important rules:

- Defining the problem completely.
- Develop the algorithm and then the program logic.
- Reuse the proved models as much as possible.
- Prove the correctness of algorithms during the design phase.
- Developers should pay attention to the clarity and simplicity of your program.
- Verifying each part of a program as soon as it is developed.

What is **Reliability**?

- **Reliability** refers to the consistency of a measure. A **test** is considered reliable if we get the same result repeatedly.
- **Software Reliability** is the probability of failure-free **software** operation for a specified period of time in a specified environment.
- Software reliability is a key part in software quality. The study of software reliability can be categorized into three parts:
- 1. Modeling
  2. Measurement
  3. Improvement

Why to do Reliability Testing

- Reliability testing is done to test the software performance under the given conditions.

1. To find the structure of repeating failures.
2. To find the number of failures occurring is the specified amount of time.
3. To discover the main cause of failure
4. To conduct performance testing of various modules of software application after fixing defect

- After the release of the product too, we can minimize the possibility of occurrence of defects and thereby improve the software reliability.

Correctness Vs Reliability

The correctness will be established via requirement specification and the program text to prove that software is behaving as expected.

The reliability is the probability of the successful execution of program on randomly selected elements from its input domain.

Though correctness of a program is desirable, it is almost never the objective of testing.

While correctness attempts to establish that the program is error free, testing attempts to find if there are any errors in it.

Thus completeness of testing does not necessarily demonstrate that a program is error free.

Testing Vs Debugging

Testing Vs Debugging

- Testing is the process of determining if a program has any errors.
- When testing reveals an error, the process used to determine the cause of this error and to remove it, is known as debugging.
- Testing catches and reports bugs.
- Testing reduces the probability of undiscovered bugs remaining in the software
- Testing is not a proof of correctness
- Testing can be planned with allocation of effort and schedule, resources, also, having criteria on when to stop testing.
- Testing starts with known conditions like what to test, test input, expected output and uses test procedures.
- Testing shows that bugs are present in a program, but cannot prove that there are no bugs
- There is no need to know design to carry-out testing

- Good testing is done by an outsider that is other than the team who develops the code
- Test automation in order to store and execute test cases can be done
- Debugging is the process of analyzing causes behind the bugs and removing them
- Debugging starts with a list of reported bugs with unknown initial conditions.
- In debugging it is not possible to plan and estimate schedule and effort for debugging
- Debugging is a problem solving involving deduction
- Detailed design knowledge is of great help in good debugging
- Debugging is done by the development team and hence is an insider's work

Testing Principles

(The Seven Key Principles)

1. Testing shows presence of Defects
2. Exhaustive Testing is Impossible!
3. Early Testing
4. Defect Clustering
5. The Pesticide Paradox
6. Testing is Context Dependent
7. Absence of Errors Fallacy

1.Testing shows the presence of Defects

We test to find Faults (as known as Defects)

- As we find more defects, the probability of undiscovered defects remaining in a system reduces (decreasing nature).
- However Testing cannot prove that there are no defects present

Why Testing is necessary

Testing Pearls of Wisdom

- ***"The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that program"***
- ***"Do not plan a test effort under the tacit assumption that no errors will be found"***
- ***"A good test is one that has a high probability of detecting an as yet undiscovered error"***
- ***"A successful test is one that detects an as-yet undiscovered error"***

Myers 2004

2.Exhaustive Testing is Impossible!

- We have learned that we cannot test everything (i.e. all combinations of inputs and pre-conditions).
- That is we must Prioritise our testing effort using a Risk Based Approach.

Why don't we test everything ?

System has 20 screens
Average 4 menus / screen
Average 3 options / menu
Average of 10 fields / screen
2 types of input per field
Around 100 possible values

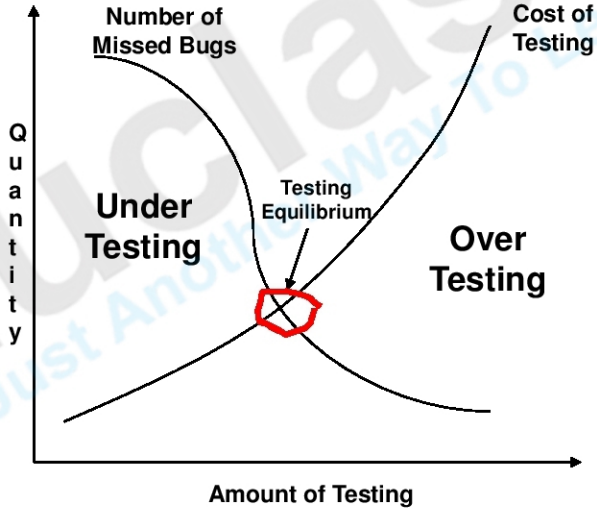
Approximate total for exhaustive testing
20 x 4 x 3 x 10 x 2 x 100 = 480,000 tests

Test length = 1 sec then test duration = 17.7 days
Test length = 10 sec then test duration = 34 weeks
Test length = 1 min then test duration = 4 years
Test length = 10 mins then test duration = 40 years!

It is not a matter of time. But,time is money (salary is taken by hour. So second is valuable for software houses)

Urgency of Equilibrium

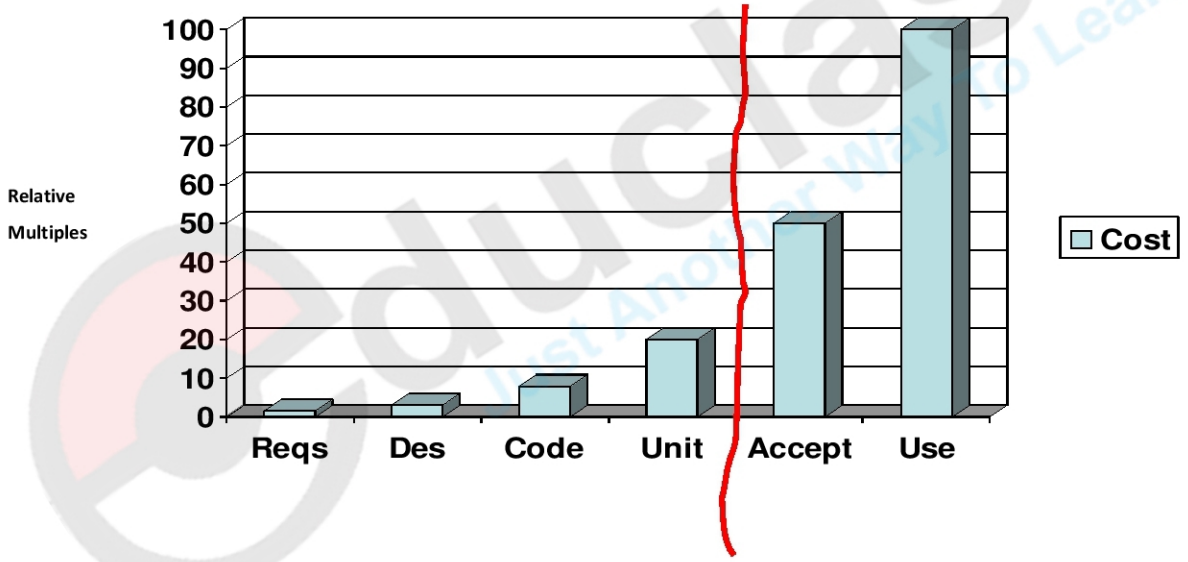
- If you test too little, the probability of software failure increases
- If you try to test too much, the development cost becomes unaffordable
- So, we need to conduct some sort of equilibrium



3.Early testing

- Testing activities should start as early as possible in the development life cycle
- These activities should be focused on defined objectives – outlined in the Test Strategy
- Remember from our Definition of Testing, that Testing doesn't start once the code has been written!

Cost of Fault Correction



4.Defect Clustering

- Defects are not evenly distributed in a system They are 'clustered'
- In other words, most defects found during testing are usually confined to a small number of modules (80% of uncovered errors focused in 20% modules of the whole application)
- Similarly, most operational failures of a system are usually confined to a small number of modules
- An important consideration in test prioritisation!

5.The Pesticide Paradox

- Testing identifies bugs, and programmers respond to fix them
- As bugs are eliminated by the programmers, the software improves
- As software improves the effectiveness of previous tests, gradually destroy
- Therefore we must learn, create and use new tests based on new techniques to catch new bugs (i.e. It is not a matter of repetition. It is a matter of learning and improving)

- *N.B It's called the "pesticide paradox" after the agricultural phenomenon, where bugs such as the boll weevil build up tolerance to pesticides, leaving you with the choice of ever-more powerful pesticides followed by ever-more powerful bugs or an altogether different approach.' – Beizer 1995*

6.Testing is Context (background)Dependent

- Testing is done differently in different contexts
- For example, safety-critical software is tested differently from an e-commerce site
- Testing can be 50% of development costs, in NASA's Apollo program (it was 80% testing)
- 3 to 10 failures per thousand lines of code (KLOC) typical for commercial software
- 1 to 3 failures per KLOC typical for industrial software
- 0.01 failures per KLOC for NASA Shuttle code!
- Also different industries impose different testing standards

7.Absence of Errors Fallacy

- If we build a system and, in doing so, find and fix defects ....

  It doesn't make it a good system

- Even after defects have been resolved, it may still be unusable and/or does not fulfil the users' needs and expectations

Test *Metrics*

Metrics can be defined as "STANDARDS OF MEASUREMENT".

- In other words, metrics helps estimating the progress, quality and health of a software testing effort. The ideal example to understand metrics would be a weekly mileage of a car compared to its ideal mileage recommended by the manufacturer.

Test metrics example:

- How many defects are existed within the module?
- How many test cases are executed per person?
- What is the Test coverage %?

"Software testing metrics - Improves the efficiency and effectiveness of a software testing process."

Why do Test Metrics?

"We cannot improve what we cannot measure" and Test Metrics helps us to do exactly the same.

- Take decision for next phase of activities
- Evidence of the claim or prediction
- Understand the type of improvement required
- Take decision or process or technology change

Types of Metrics

•**Process Metrics**: It can be used to improve the process efficiency of the SDLC
•**Product Metrics**: It deals with the quality of the software product
•**Project Metrics**: It can be used to measure the efficiency of a project team or any tools being used by the team members

Test Metrics Life Cycle

Analysis-

- Identification of the Metrics
- Define the identified Metrics

Communicate-

- Explain the need for metric to stakeholder and testing team
- Educate the testing team about the data points need to be captured for processing the metric

Evaluation-

- Capture and verify the data
- Calculating the metrics value using the data captured

Report-

- Develop the report with effective conclusion
- Distribute report to the stakeholder and respective representative
- Take feedback from stakeholder

Steps to test metrics

Steps to test metrics

- Identify the key software testing processes to be measured
- The number of test cases planned to be executed per day
- The actual test execution per day will be captured by the test manager at the end of the day
- The actual test cases executed per day
- The test case execution falls below the goal set, we need to investigate the reason and suggest the improvement measures

How to calculate test metric

- Percentage test cases executed= (No of test cases executed / Total no of test cases written) X 100
- Likewise, you can calculate for other parameters like test cases not executed, test cases passed, test cases failed, test cases blocked, etc.

Verification

- **Verification is the process confirming that -software meets its specification, done through inspections and walkthroughs**

***Use*– To identify defects in the product early in the life cycle**

Validation

- **Validation is the process confirming that it meets the user's requirements. It is the actual testing.**

Verification : Is the Product Right

Validation : Is it the Right Product

What is Quality ?

Quality is defined as meeting the customer's requirements and according to the standards

The best measure of Quality is given by FURPS

- **Functionality**
- **Usability**
- **Reliability**
- **Performance**
- **Scalability**

Why Quality ?

- **Quality is the important factor affecting an organization's long term performance.**
- **Quality improves productivity and competitiveness in any organization.**

Quality Assurance

Quality Assurance is a planned and systematic set of activities necessary to provide adequate confidence that products and services will conform to specified requirements and meets user needs.

- It is process oriented.
- Defect prevention based.
- Throughout the Life Cycle.
- It's a management process.

Quality Control

Quality control is the process by which product quality is compared with the applicable standards and the action taken when non conformance is detected.

- It is product oriented
- Defect detection based

QA vs. QC

• Quality Assurance makes sure that we are doing the right things, the right Way.	• Quality Control makes sure the results of what we've done and what we expected .
• QA focuses on building in quality and hence preventing defects.	• QC focuses on testing for quality and hence detecting defects.
• QA deals with process.	• QC deals with product.
• QA is for entire life cycle.	• QC is for testing part in SDLC.
• QA is preventive process.	• QC is corrective process.