SATHYABAMA UNIVERSITY

(Established under Section 3, UGC Act 1956)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

SCSX1038 SOFTWARE QUALITY ASSURANCE AND TESTING

UNIT I INTRODUCTION

Software Quality

Conformance to explicitly stated functional and performance requirements, explicitly documented development standards and implicit characteristics that are expected of all professionally develop software

Three (3) important points to remember on software quality

1. Software requirements are the foundation from which quality is measured. Lack of conformance to requirements is lack of quality.
2. Specified standards define a set of development criteria that guide the manner in which software is engineered. If the criteria are not followed, lack of quality will almost surely result.
3. There is a set of implicit requirements that often goes unmentioned (e.g., the desire for good maintainability). If software conforms to its explicit requirements, but fails to meet implicit requirements.

Quality Characteristics

Is any property or element that can be used to define the nature of a product. Each characteristic can be physical or chemical properties such as size, weight, volume, color or composition.

Software Quality

1. Is achieved through a disciplined approach - called software engineering SE
2. Can be defined, described, and measured
3. Can be assessed before any code has been written
4 Cannot be tested into a product

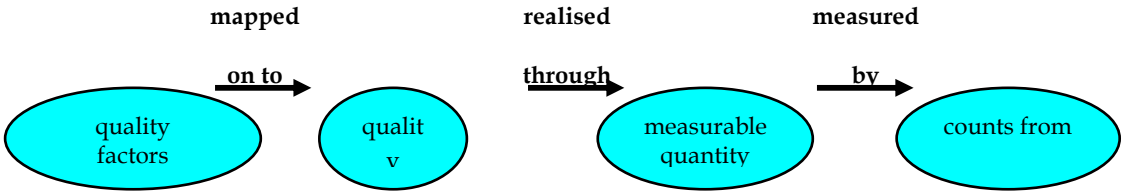
Software quality challenges

1. Defining it
2. Describing it (qualitatively)
3. Measuring it (quantitatively)
4. Achieving it (technically)

Designing software is a *creative* task, and like most such tasks, success is more likely if the designer follows what might be termed a set of *rules of form.*The rules of form also provide some way of assessing the *quality* of the eventual product, and possibly of the processes that led to it.

REALISING QUALITY

A set of abstract quality factors (‘the *ilities*’) has been defined. These cannot be measured directly but do relate to the ultimate goal.



Software Quality Factors (by McCall)

1) Product Revision (changing it)

- *Flexibility (can I change it?)*
  The effort required to modify an operational program. Change and enhancement of the system should be easily implementable.

- *Maintainability (can I fix it?)*
  The effort required to locate and fix an error in a program. The system should be easy to keep up for its intended use. Changes for improving operational efficiency should be easy to implement. Failed operations should be easy to restore to satisfactory condition.

- *Testability (can I test it?)*
  The effort required to test a program to ensure that it performs its intended function . The ability of the system to produce quality product units should be easily testable. Useful messages should be generated for testing and debugging purposes.

2) Product Transition (modifying it to work in a different environment)

Interoperability (Will I be able to interface it with another system?)

The effort required to couple one system to another.

- *Portability (Will I be able to use it on another machine?)*
  The effort required to transfer the program from one hardware and/or software system environment to another. The system should be portable

among people and among machines. Attainment of theother quality characteristics greatly facilitates portability.

- *Reusability (Will I be able to reuse some of the software?)*
  The extent to which a program (or part of a program) can be reused in other applications-related to the packaging and scope of the functions that the program performs.

3) Product Operations (using it)

- *Correctness (Does it do what I want?)*
  The extent to which a program satisfies its specification and fulfills the customer's mission objectives. The extent to which software is free from design defects and from coding defects.; that is fault-free.

- *Reliability (Does it do it accurately all of the time?)*
  The extent to which a program can be expected to perform its intended function with required precisions under stated conditions for a stated period of time.

- *Efficiency (Will it run on my hardware as well as it can?)*
  The extent to which a software performs its function.with a minimum consumption of computing resources. It should not use any hardware components or peripheral equipment unnecessarily.

- *Integrity (Is it secure?)*
  The extent to which access to software or data by unauthorized persons can be controlled.

- *Usability (Is it designed for the use?)*
  The effort required to learn, operate, prepare input, and interpret output of a program.

Quality Metrics

Provides an indication of how closely software conforms to implicit (essential) and explicit (specific) requirements.

- *Auditability*
  The ease with which conformance to standards can be checked.

- *Accuracy*
  The precision of computations and control.A qualitative assessment of freedom from error.A quantitative measure of the magnitude of error. The correct data values are recorded.

- *Communication commonality*
  The degree to which standards interfaces, protocols, and bandwidths are used.

- *Completeness*
  The degree to which full implementation of required function has been achieved. All data items are captured and stored for use. Data items are properly identified with time periods.

- *Conciseness*
  The compactness of the program in terms of lines code.

- *Consistency*
  The use of uniform design and documentation techniques throughout the software development project.

- *Data commonality*
  The use of standard data structures and types throughout the program.

- *Error tolerance*
  The damage that occurs when the programs encounters an error. Suitable error prevention and detection procedures are in place. There are procedures for reporting and correcting errors. Various audit procedures are applied.

- *Execution efficiency*
  The run-time performance of a program.

- *Expandability*
  The degree to which architectural, data, or procedural design can be extended.

- *Generality*
  The breadth of potential application of program components.

- *Hardware independence*

  The degree to which the software is decoupled from the hardware on which it operates.

- *Instrumentality*
  The degree to which the program monitors its own operation and identifies errors that do occur.

- *Modularity*
  The functional independence of program components.

- *Operability*

The ease of operation of a program.

- *Robustness*
  The extent to which software can continue to operate correctly despite the introduction of invalid inputs

- *Security*
  The availability of mechanism that control or protect programs and data. The system and its operations are protected from various environmental and operation risks. There are provisions for recovery in the event of failure or destruction of part or all system

- *Self-documentation*
  The degree to which the source code provides meaningful documentation.

- *Simplicity*
  The degree to which a program can be understood without difficulty.

- *Software system independence*
  The degree to which the program is independent of nonstandard programming language features, operating system characteristics, and other environmental constraints.

- *Traceability*
  The ability to trace a design representation or actual program component back to requirements.

- *Training*
  The degree to which the software in enabling new users to apply the system.

Role of Software Testing

The Role of Software Testing is often mis-understood across the different stake holders of the application development & this list includes testers too.

Testing is considered to be part of Quality Assurance activity since the initial days of Software Development and the same trend is happening as of now too. Even most of the titles like QA Engineer / QA Lead are associated with Testers even though they are not performing the role of QA.

It's good to capture the mission of Testing & align your test teams in that direction. Every one in the team must be clear on his / her role and see how the same is helping to achieve the mission of the team.

It's good to note that

- Testing is not about assuring quality into the systems because theTester is not a Quality Police.
- Testing is not about targeting for Bug Free Product. It's just impossible since you can't build human brain into systems (Of course humans do commit mistakes).
- Testing is not about fighting with the Development Teams. Don't act like the enemy of developers.
- Testing is not about just looking at the documents (so called BRS, SRS, FRS) and writing the test cases.
- Don't fight with Developers on the issues need to be fixed for the release instead write good report that reduce the time to reproduce and debug.

Testing is a process followed to make things better. It helps to take informed decision by providing the relevant information based on the context.

Testers need to identify the critical issues with the system as soon as possible and make sure that the information supplied is sufficient to reproduce the issue. We need to supply the information on the presence of bugs in the system to the stake holders. The information should help the stake holders to take informed decisions.

The following list helps

- Identify the different end users of the system and their interaction with the same.
- Capture the important scenarios for each end user. It's good to note that we need to capture the story around the scenario and not just steps.
- Talk to different stake holders including the customers (incase if you have access) on how the feature might be used & capture the scenarios.
- Plan towards uncovering the critical issues of the system as early as possible.

Software Testing Fundamentals

Software Testing Fundamentals is a platform to gain (or refresh) basic knowledge in the field of Software Testing. If we are to 'cliche' it, the site is of the testers, by the testers, and for the testers. Our goal is to build a resourceful repository of *Quality content* on *Quality*.

VERIFICATION vs VALIDATION

The terms 'Verification' and 'Validation' are frequently used in the software testing world but the meaning of these terms are mostly vague and debatable. You will encounter (or have encountered) all kinds of usage and interpretations of those terms, and it is our humble attempt here to distinguish between them as clearly as possible.

Criteria	Verification	Validation
Definition	The process of evaluating work-products (not the actual final product) of a development phase to determine whether they meet the specified requirements for that phase.	The process of evaluating software during or at the end of the development process to determine whether it satisfies specified business requirements.
Objective	To ensure that the product is being built according to the requirements and design specifications. In other words, to ensure that work products meet their specified requirements.	To ensure that the product actually meets the user's needs, and that the specifications were correct in the first place. In other words, to demonstrate that the product fulfills its intended use when placed in its intended environment.
Question	Are we building the product *right*?	Are we building the *right* product?
Evaluation Items	Plans, Requirement Specs, Design Specs, Code, Test Cases	The actual product/software.
Activities	Reviews Walkthroughs Inspections	Testing

Testing principles

1.Testing shows presence of errors

In general, testing proves the presence of errors. Sufficient testing reduces the likelihood of existing, not discovered error conditions within the test object. It does not verify that no more bugs exist, even if no more errors can be found. Testing is not a prove that the system is free of errors.

2.Exhaustive testing is not possible

An exhaustive test which considers all possible input parameters, their combinations and different pre-conditions can not be accomplished (except for trivial test objects). Test are always spot tests. Therefore, the effort must be managed by risk, priorities and thoughtful selection.

3.Test early and regularly

Testing activities should begin as early as possible within the software life cycle. They should be repeated regularly and have its' own agenda. Early testing helps detecting errors at an early stage of the development process which simplifies error correction (and reduces the costs for this work).

4.Accumulation of errors

There is no equal distribution of errors within one test object. The place where one error occurs, it's likely to find some more. The testing process must be flexible and respond to this behavior.

5.Fading effectiveness

The effectiveness of tests fades over time. If test-cases are only repeated, they do not expose new errors. Errors, remaining within untested functions may not be discovered. In order to prevent this effect, test-cases must be altered and reworked time by time.

6.Testing depends on context

No two systems are the same and therefore can not be tested the same way. Testing intensity, the definition of end criteria etc. must be defined individually for each system depending on its testing context. E-commerce websites require a different approach than online-banking applications.

7.False conclusion: no errors equals usable system

Error detection and error fixing does not guarantee a usable system matching the users expectations. Early integration of users and rapid prototyping prevents unhappy clients and discussions.

Objectives and issues of testing

Software Testing has different goals and objectives.

The major objectives of Software testing are as follows:

- Finding defects which may get created by the programmer while developing the software.
- Gaining confidence in and providing information about the level of quality.
- To prevent defects.
- To make sure that the end result meets the business and user requirements.
- To ensure that it satisfies the BRS that is Business Requirement Specification and SRS that is System Requirement Specifications.
- To gain the confidence of the customers by providing them a quality product.

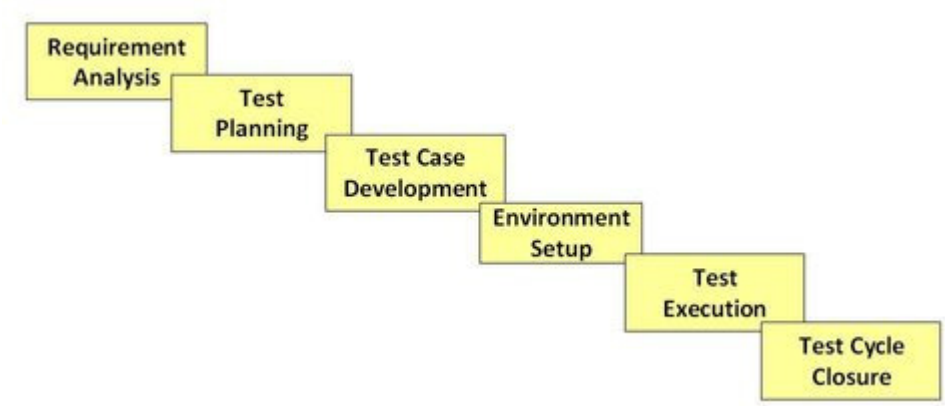
Issues in Software Testing

1 The organization doesn't know why it's testing.
2. The organization hasn't agreed on what kind of problems you're looking for.
3. Ttesting the wrong things.
4. Test team doesn't know how to test.
5. Test team needs more training
6. Development doesn't understand or can't reproduce your problem reports
7. Tested system isn't testable enough.

Testing lifecycle.

SOFTWARE TESTING LIFE CYCLE (STLC)

The different stages in Software Test Life Cycle -



Software Testing Life Cycle (STLC) defines the steps/ stages/ phases in testing of software.

Phase	Activity	Deliverables	Necessity
Requirements/ Design Review	You review the software requirements/ design (Well, if they exist.)	Review Defect' Reports	Curiosity
Test Planning	Once you have gathered a general idea of what needs to be tested, you 'plan' for the tests.	Test Plan Test Estimation Test Schedule	Farsightedness

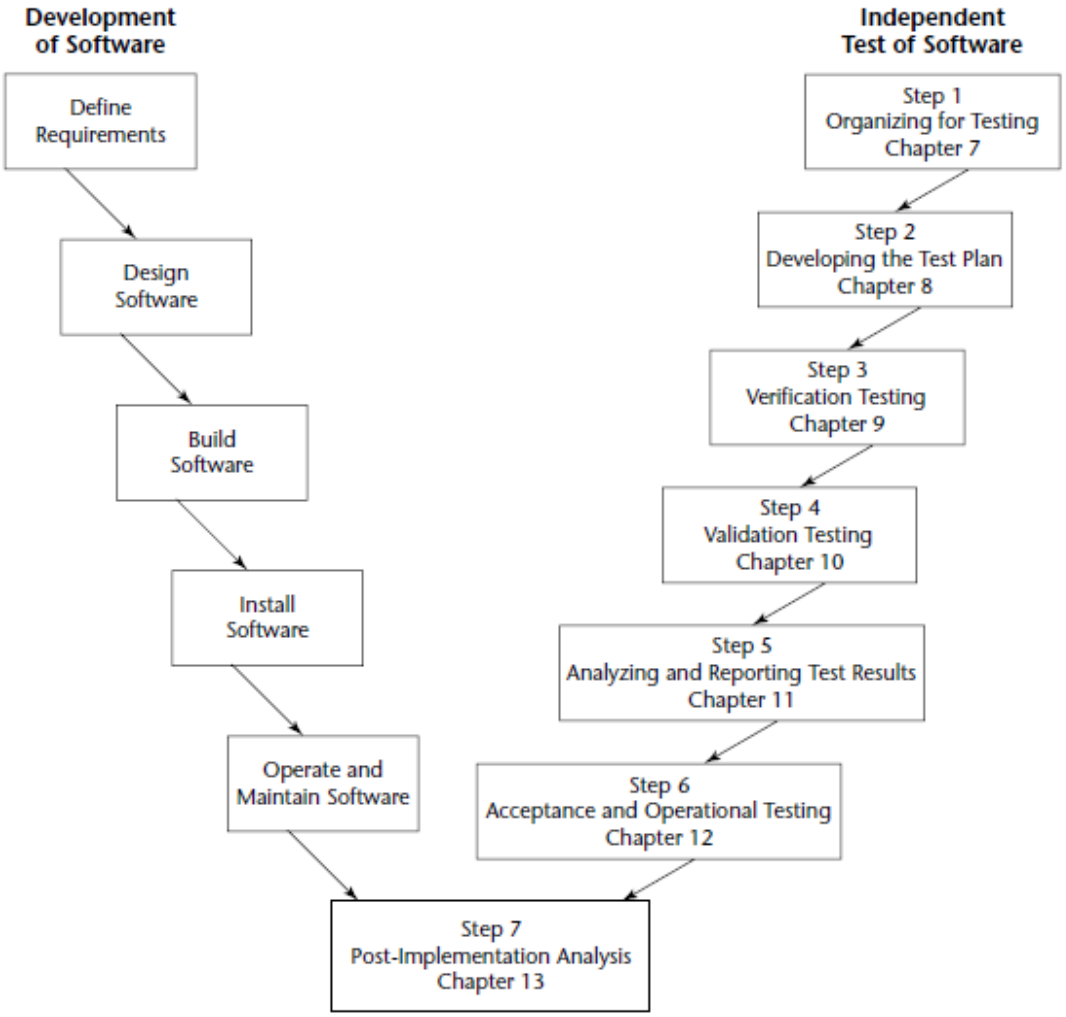
Test Designing	You design/ detail your tests on the basis of detailed requirements/design of the software (sometimes, on the basis of your imagination).	Test Cases / Test Scripts/Test Data Requirements Traceability Matrix	Creativity
Test Environment Setup	You setup the test environment (server/ client/ network, etc) with the goal of replicating the end-users' environment.	Test Environment	Rich company
Test Execution	You execute your Test Cases/ Scripts in the Test Environment to see whether they pass.	Test Results (Incremental) Defect Reports	Patience
Test Reporting	You prepare various reports for various stakeholders.	Test Results (Final) Test/ Defect Metrics Test Closure Report	Diplomacy

Test process

Testing is a process rather than a single activity. This process starts from test planning then designing test cases, preparing for execution and evaluating status till the test closure. So, we can divide the activities within the fundamental test process into the following basic steps:

1) Planning and Control
2) Analysis and Design

3) Implementation and Execution
4) Evaluating exit criteria and Reporting
5) Test Closure activities



Test Information Flow

Two classes of input are provided to the test process:

- a software configuration:
  Software Requirements Specification,
  Design Specification,
  and Source code

- a test configuration:
  Test Plan,
  Test Procedure and
  Testing tools

Test Cases

DEFINITION

A test case is a set of conditions or variables under which a tester will determine whether a system under test satisfies requirements or works correctly.

The process of developing test cases can also help find problems in the requirements or design of an application.

TEST CASE TEMPLATE

A test case can have the following elements. Note, however, that normally a test management tool is used by companies and the format is determined by the tool used.

Test Suite ID	The ID of the test suite to which this test case belongs.
Test Case ID	The ID of the test case.
Test Case Summary	The summary / objective of the test case.
Related Requirement	The ID of the requirement this test case relates/traces to.
Prerequisites	Any prerequisites or preconditions that must be fulfilled prior to executing the test.
Test Procedure	Step-by-step procedure to execute the test.
Test Data	The test data, or links to the test data, that are to be used while

	conducting the test.
Expected Result	The expected result of the test.
Actual Result	The actual result of the test; to be filled after executing the test.
Status	Pass or Fail. Other statuses can be 'Not Executed' if testing is not performed and 'Blocked' if testing is blocked.
Remarks	Any comments on the test case or test execution.
Created By	The name of the author of the test case.
Date of Creation	The date of creation of the test case.
Executed By	The name of the person who executed the test.
Date of Execution	The date of execution of the test.
Test Environment	The environment (Hardware/Software/Network) in which the test was executed.

TEST CASE EXAMPLE / TEST CASE SAMPLE

Test Suite ID	TS001
Test Case ID	TC001
Test Case Summary	To verify that clicking the Generate Coin button generates coins.
Related Requirement	RS001
Prerequisites	1. User is authorized. 2. Coin balance is available.
Test Procedure	1. Select the coin denomination in the Denomination field. 2. Enter the number of coins in the Quantity field. 3. Click Generate Coin.
Test Data	1. Denominations: 0.05, 0.10, 0.25, 0.50, 1, 2, 5 2. Quantities: 0, 1, 5, 10, 20
Expected Result	1. Coin of the specified denomination should be produced if the specified Quantity is valid (1, 5) 2. A message 'Please enter a valid quantity between 1 and 10' should be displayed if the specified quantity is invalid.

Actual Result	1. If the specified quantity is valid, the result is as expected. 2. If the specified quantity is invalid, nothing happens; the expected message is not displayed
Status	Fail
Remarks	This is a sample test case.
Created By	Sam
Date of Creation	01/14/2020
Executed By	Sam
Date of Execution	02/16/2020
Test Environment	OS: Windows YBrowser: Chrome N

TEST PLAN DEFINITION

A Software Test Plan is a document describing the testing scope and activities. It is the basis for formally testing any software/product in a project.

Definition

Test plan: A document describing the scope, approach, resources and schedule of intended test activities. It identifies amongst others test items, the features to be tested, the testing tasks, who will do each task, degree of tester independence, the test environment, the test design techniques and entry and exit criteria to be used, and the rationale for their choice,and any risks requiring contingency planning. It is a record of the test planning process.

Master Test Plan: A test plan that typically addresses multiple test levels.

Phase Test Plan: A test plan that typically addresses one test phase.

Monitoring and measuring test execution

Test monitoring can serve various purposes during the project, including the following:

1.Give the test team and the test manager feedback on how the testing work is going, allowing opportunities to guide and improve the testing and the project.

2..Provide the project team with visibility about the test results.

3. Measure the status of the testing, test coverage and test items against the exit criteria to determine whether the test work is done.

4. Gather data for use in estimating future test efforts.

For small projects, the test leader or a delegated person can gather test progress monitoring information manually using documents, spreadsheets and simple databases. But, when working with large teams, distributed projects and long-term test efforts, we find that the efficiency and consistency of data collection is done by the use of automated tools.

Tools for Software Test Automation

These tools provide automated functional and performance testing for environments including Java, SOAP, CORBA, HTML, WAP, client/server, UNIX, and Windows.

- SilkCentral - Test Management
- SilkTest - Automated functional and regression testing
- SilkPerformer - Automated load and performance testing
- SilkMonitor - 24x7 monitoring and reporting of Web, application and database servers
- SilkPilot - Unit testing of CORBA objects
- SilkObserver - End-to-end transaction management and monitoring for CORBA applications
- SilkMeter - Access control and usage metering
- SilkRealizer - Scenario testing and system monitoring
- SilkRadar - Automated defect tracking

HP / Mercury Quality Center

Hewlett Packard offers a suite of solutions that automate testing and quality assurance for client/server software and systems, e-business applications, and enterprise resource planning applications.

- Quality Center - management for tests and defect tracking
- QuickTest Professional™ - E-business functional testing
- LoadRunner® - Enterprise load testing
- WinRunner® - Test automation for the enterprise

_____End of Unit 1_____

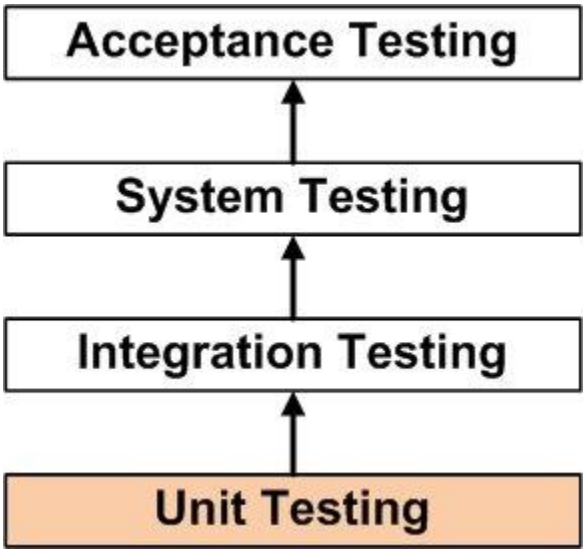
SCSX1038	SOFTWARE QUALITY ASSURANCE AND TESTING (Common to CSE & IT)	L	T	P	Credits	Total Marks
		3	0	0	3	100

UNIT - 2 Levels and Types of Testing

Unit testing

DEFINITION

Unit Testing is a level of software testing where individual units/ components of a software are tested. The purpose is to validate that each unit of the software performs as designed.



A unit is the smallest testable part of software. It usually has one or a few inputs and usually a single output. In procedural programming a unit may be an individual program, function, procedure, etc. In object-oriented programming, the smallest unit is a method, which may belong to a base/ super class, abstract class or derived/ child class.Unit testing frameworks, drivers, stubs, and mock/ fake objects are used to assist in unit testing.

METHOD

Unit Testing is performed by using the White Box Testing method.

When is it performed?

Unit Testing is the first level of testing and is performed prior to Integration Testing.

Who performs it?

Unit Testing is normally performed by software developers themselves or their peers. In rare cases it may also be performed by independent software testers.

TASKS

- Unit Test Plan
  - Prepare
  - Review
  - Rework
  - Baseline
- Unit Test Cases/Scripts
  - Prepare
  - Review
  - Rework
  - Baseline
- Unit Test
  - Perform

BENEFITS

- Unit testing increases confidence in changing/ maintaining code. If good unit tests are written and if they are run every time any code is changed, we will be able to promptly catch any defects introduced due to the change. Also, if codes are already made less interdependent to make unit testing possible, the unintended impact of changes to any code is less.
- Codes are more reusable. In order to make unit testing possible, codes need to be modular. This means that codes are easier to reuse.
- Development is faster. How? If you do not have unit testing in place, you write your code and perform that fuzzy ‘developer test’ (You set some breakpoints, fire up the GUI, provide a few inputs that hopefully hit your code and hope that you are all set.) If you have unit testing in place, you write the test, write the code and run the test. Writing tests takes time but the time is compensated by the less amount of time it takes to run the tests; You need not fire up the GUI and provide all those inputs. And, of course, unit tests are more reliable than ‘developer tests’. Development is faster in the long run too. How? The effort required to find and fix defects found during unit testing is very less in comparison to the effort required to fix defects found during system testing or acceptance testing.

- The cost of fixing a defect detected during unit testing is lesser in comparison to that of defects detected at higher levels. Compare the cost (time, effort, destruction, humiliation) of a defect detected during acceptance testing or when the software is live.
- Debugging is easy. When a test fails, only the latest changes need to be debugged. With testing at higher levels, changes made over the span of several days/ weeks/ months need to be scanned.
- Codes are more reliable. Why? I think there is no need to explain this to a sane person.

Integration Testing

- integration testing: Testing performed to expose defects in the interfaces and in the
  interactions between integrated components or systems. See also c*omponent integration*
  testing, system integration testing.
- component integration testing: Testing performed to expose defects in the interfaces and
  interaction between integrated components.
- system integration testing: Testing the integration of systems and packages; testing
  interfaces to external organizations (e.g. Electronic Data Interchange, Internet).

ANALOGY

During the process of manufacturing a ballpoint pen, the cap, the body, the tail and clip, the ink cartridge and the ballpoint are produced separately and unit tested separately. When two or more units are ready, they are assembled and Integration Testing is performed. For example, whether the cap fits into the body or not.

METHOD

Any of Black Box Testing, White Box Testing, and Gray Box Testing methods can be used. Normally, the method depends on your definition of ‘unit’.

TASKS

- Integration Test Plan
  - Prepare
  - Review
  - Rework
  - Baseline
- Integration Test Cases/Scripts
  - Prepare
  - Review
  - Rework
  - Baseline
- Integration Test
  - Perform

When is Integration Testing performed?

Integration Testing is performed after Unit Testing and before System Testing.

Who performs Integration Testing?

Either Developers themselves or independent Testers perform Integration Testing.

APPROACHES

- *Big Bang* is an approach to Integration Testing where all or most of the units are combined together and tested at one go. This approach is taken when the testing team receives the entire software in a bundle. So what is the difference between Big Bang Integration Testing and System Testing? Well, the former tests only the interactions between the units while the latter tests the entire system.
- *Top Down* is an approach to Integration Testing where top level units are tested first and lower level units are tested step by step after that. This approach is taken when top down development approach is followed. Test Stubs are needed to simulate lower level units which may not be available during the initial phases.
- *Bottom Up* is an approach to Integration Testing where bottom level units are tested first and upper level units step by step after that. This approach is taken when bottom up development approach is followed. Test Drivers are needed to simulate higher level units which may not be available during the initial phases.

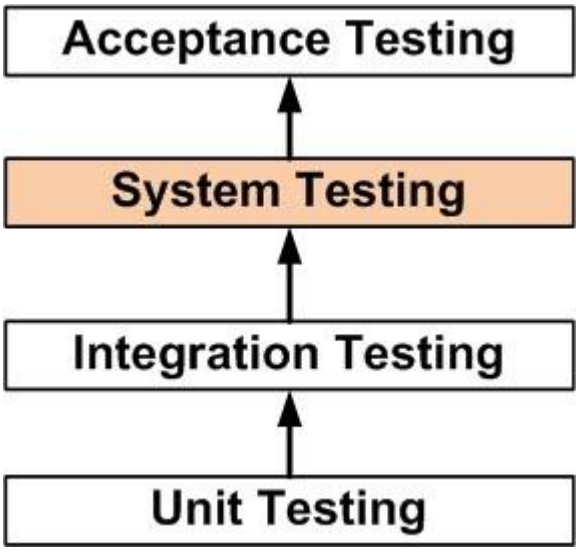
- *Sandwich/Hybrid* is an approach to Integration Testing which is a combination of Top Down and Bottom Up approaches.

System Testing

DEFINITION

System Testing is a level of the software testing where a complete and integrated software is tested.

The purpose of this test is to evaluate the system's compliance with the specified requirements.



- system testing: The process of testing an integrated system to verify that it meets specified
  requirements.

ANALOGY

During the process of manufacturing a ballpoint pen, the cap, the body, the tail, the ink cartridge and the ballpoint are produced separately and unit tested separately. When two or more units are ready, they are assembled and Integration Testing is performed. When the complete pen is integrated, System Testing is performed.

METHOD

Usually, Black Box Testing method is used.

TASKS

- System Test Plan
  - Prepare
  - Review
  - Rework
  - Baseline
- System Test Cases
  - Prepare
  - Review
  - Rework
  - Baseline
  - System Test
  - Perform

When is it performed?

System Testing is performed after Integration Testing and before Acceptance Testing.

Who performs it?

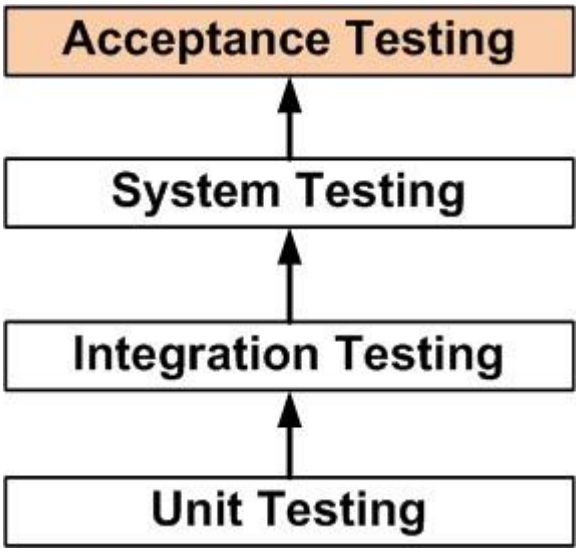
Normally, independent Testers perform System Testing.

Acceptance testing

DEFINITION

Acceptance Testing is a level of the software testing where a system is tested for acceptability.

The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.



- Acceptance Testing: Formal testing with respect to user needs, requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system.

ANALOGY

During the process of manufacturing a ballpoint pen, the cap, the body, the tail and clip, the ink cartridge and the ballpoint are produced separately and unit tested separately. When two or more units are ready, they are assembled and Integration Testing is performed. When the complete pen is integrated, System Testing is performed. Once System Testing is complete, Acceptance Testing is performed so as to confirm that the ballpoint pen is ready to be made available to the end-users.

METHOD

Usually, Black Box Testing method is used in Acceptance Testing. Testing does not normally follow a strict procedure and is not scripted but is rather ad-hoc.

TASKS

- Acceptance Test Plan
    - Prepare
    - Review
    - Rework
    - Baseline

- Acceptance Test Cases/Checklist
    - Prepare
    - Review
    - Rework
    - Baseline
- Acceptance Test
    - Perform

When is it performed?

Acceptance Testing is performed after System Testing and before making the system available for actual use.

Who performs it?

- *Internal Acceptance Testing* (Also known as Alpha Testing) is performed by members of the organization that developed the software but who are not directly involved in the project (Development or Testing). Usually, it is the members of Product Management, Sales and/or Customer Support.
- *External Acceptance Testing* is performed by people who are not employees of the organization that developed the software.
    - *Customer Acceptance Testing* is performed by the customers of the organization that developed the software. They are the ones who asked the organization to develop the software. [This is in the case of the software not being owned by the organization that developed it.]
    - *User Acceptance Testing* (Also known as Beta Testing) is performed by the end users of the software. They can be the customers themselves or the customers' customers

Alpha Testing

Alpha testing is one of the most common software testing strategyused in software development. Its specially used by product development organizations.

This test takes place at the developer's site. Developers observe the users and note problems.Alpha testing is testing of an application when development is about to complete. Minor design changes can still be made as a result of alpha testing.

Alpha testing is typically performed by a group that is independent of the design team, but still within the company, e.g. in-house software test engineers, or software QA engineers. Alpha testing is final testing before the software is released to the general public. It has two phases:

In the first phase of alpha testing, the software is tested by in-house developers. They use either debugger software, or hardware-assisted debuggers. The goal is to catch bugs quickly.

In the second phase of alpha testing, the software is handed over to the software QA staff, for additional testing in an environment that is similar to the intended use. Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site. Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing, before the software goes to beta testing.

Beta Testing

It is also known as field testing. It takes place at customer's site. It sends the system to users who install it and use it under real-world working conditions.A beta test is the second phase of software testing in which a sampling of the intended audience tries the product out. (Beta is the second letter of the Greek alphabet.) Originally, the term*alpha test* meant the first phase of testing in a software development process. The first phase includes unit testing, component testing, and system testing. Beta testing can be considered "pre-release testing. The goal of beta testing is to place your application in the hands of real users outside of your own engineering team to discover any flaws or issues from the user's perspective that you would not want to have in your final,released version of the application .Open and closed beta developers release either a closed beta or an open beta. Closed beta versions are released to a select group of individuals for a user test and are invitation only, while Open betas are from a larger group to the general public and anyone interested. The testers report any bugs that they find, and sometimes suggest additional features they think should be available in the final version.

Advantages of beta testing:

- You have the opportunity to get your application into the hands of users prior to releasing it to the general public.
- Users can install, test your application, and send feedback to you during this beta testing period.
- Your beta testers can discover issues with your application that you may have not noticed, such as confusing application flow, and even crashes.
- Using the feedback you get from these users, you can fix problems before it is released to the general public.
- The more issues you fix that solve real user problems, the higher the quality of your application when you release it to the general public.
- Having a higher-quality application when you release to the general public will increase customer satisfaction.
- These users, who are early adopters of your application, will generate excitement about your application.

Difference between manual testing and automation testing.

Manual Testing	Automation Testing
1. Time consuming and tedious: Since test cases are executed by human resources so it is very slow and tedious.	1. Fast Automation runs test cases significantly faster than human resources.
2. Huge investment in human resources: As test cases need to be executed manually so more testers are required in manual testing.	2. Less investment in human resources:Test cases are executed by using automation tool so less tester are required in automation testing.
3. Less reliable: Manual testing is less reliable as tests may not be performed with precision each time because of human errors.	3. More reliable: Automation tests perform precisely same operation each time they are run.
4. Non-programmable: No programming can be done to write sophisticated tests which fetch hidden information.	4. Programmable: Testers can program sophisticated tests to bring out hidden information.

Types of Testing

1. Smoke Testing

2. REGRESSION TESTING Fundamentals

3. Functional Testing

4. Usability Testing

5. Security Testing

6. Performance Testing

Smoke Testing

DEFINITION

Smoke Testing, also known as “Build Verification Testing”, is a type of software testing that comprises of a non-exhaustive set of tests that aim at ensuring that the most important functions work. The results of this testing is used to decide if a build is stable enough to proceed with further testing.

The term ‘smoke testing’, it is said, came to software testing from a similar type of hardware testing, in which the device passed the test if it did not catch fire (or smoked) the first time it was turned on.

ELABORATION

Smoke testing covers most of the major functions of the software but none of them in depth. The result of this test is used to decide whether to proceed with further testing. If the smoke test passes, go ahead with further testing. If it fails, halt further tests and ask for a new build with the required fixes. If an application is badly broken, detailed testing might be a waste of time and effort.

Smoke test helps in exposing integration and major problems early in the cycle. It can be conducted on both newly created software and enhanced software. Smoke test is performed manually or with the help of automation tools/scripts. If builds are prepared frequently, it is best to automate smoke testing.

As and when an application becomes mature, with addition of more functionalities etc, the smoke test needs to be made more expansive. Sometimes, it takes just one incorrect character in the code to render an entire application useless.

ADVANTAGES

- It exposes integration issues.
- It uncovers problems early.
- It provides some level of confidence that changes to the software have not adversely affected major

LEVELS APPLICABLE TO

Smoke testing is normally used in Integration Testing, System Testing and Acceptance Testing levels.

REGRESSION TESTING

DEFINITION

Regression testing is a type of software testing that intends to ensure that changes (enhancements or defect fixes) to the software have not adversely affected it.

ELABORATION

The likelihood of any code change impacting functionalities that are not directly associated with the code is always there and it is essential that regression testing is conducted to make sure that fixing one thing has not broken another thing. During regression testing, new test cases are not created but previously created test cases are re-executed.

LEVELS APPLICABLE TO

Regression testing can be performed during any level of testing (Unit, Integration, System, or Acceptance) but it is mostly relevant during System Testing.

EXTENT

In an ideal case, a full regression test is desirable but oftentimes there are time/resource constraints. In such cases, it is essential to do an impact analysis of the changes to identify areas of the software that have the highest probability of being

affected by the change and that have the highest impact to users in case of malfunction and focus testing around those areas.

Due to the scale and importance of regression testing, more and more companies and projects are adopting regression test automation tools.

LITERAL MEANING OF REGRESSION

Regression [noun]: the act of going back to a previous place or state; return or reversion.

Functional Testing*:*

DEFINITION

Functional Testing is a type of software testing whereby the system is tested against the functional requirements/specifications.

ELABORATION

Functions (or features) are tested by feeding them input and examining the output. Functional testing ensures that the requirements are properly satisfied by the application. This type of testing is not concerned with how processing occurs, but rather, with the results of processing.

During functional testing, Black Box Testing technique is used in which the internal logic of the system being tested is not known to the tester.

Functional testing is normally performed during the levels of System Testing and Acceptance Testing.

Typically, functional testing involves the following steps:

- Identify functions that the software is expected to perform.
- Create input data based on the function's specifications.
- Determine the output based on the function's specifications.
- Execute the test case.

- Compare the actual and expected outputs.

ADVANTAGES

- It simulates actual system usage.
- It does not make any system structure assumptions.

DISADVANTAGES

- It has a potential of missing logical errors in software.
- It has a high possibility of redundant testing.

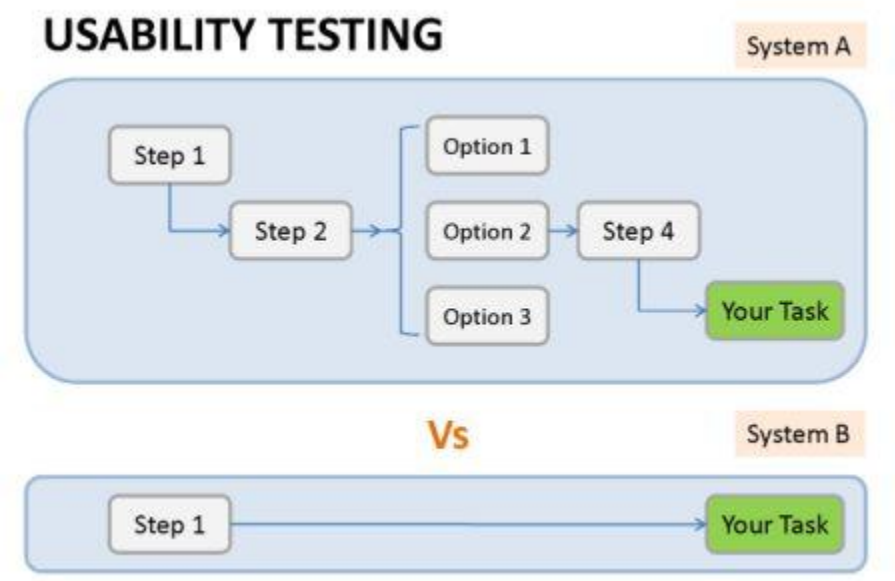
NOTE

Functional testing is more effective when the test conditions are created directly from user/business requirements. When test conditions are created from the system documentation (system requirements/ design documents), the defects in that documentation will not be detected through testing and this may be the cause of end-users' wrath when they finally use the software.

USABILITY TESTING

DEFINITION

Usability Testing is a type of testing done from an end-user's perspective to determine if the system is easily usable.



usability testing: Testing to determine the extent to which the software product is understood, easy to learn, easy to operate and attractive to the users under specified conditions.

Usability Test: The purpose of this event is to review the application user interface and other human factors of the application with the people who will be using the application. This is to ensure that the design (layout and sequence, etc.) enables the business functions to be executed as easily and intuitively as possible.

ELABORATION

Systems may be built 100% in accordance with the specifications. Yet, they may be 'unusable' when it lands in the hands of the end-users. For instance, let's say a user needs to print a Financial Update Report, every 30 minutes, and he/she has to go through the following steps:

1. Login to the system
2. Click Reports
3. From the groups of reports, select Financial Reports
4. From the list of financial reports, select Financial Update Report
5. Specify the following parameters
    1. Date Range
    2. Time Zone
    3. Departments
    4. Units
6. Click Generate Report
7. Click Print
8. Select an option
    1. Print as PDF
    2. Print for Real

If that's the case, the system is probably practically unusable (though it functions perfectly fine). If the report is to be printed frequently, wouldn't it be convenient if the user could get the job done in a couple of clicks, rather than having to go through numerous steps like listed above? What if there was a feature to save frequently generated reports as a template and if the saved reports were readily available for printing from the homepage?

LEVELS APPLICABLE TO

Usability Testing is normally performed during System Testing and Acceptance Testing levels.

USABILITY TESTING TIPS

- Understand who the users of the system are.
- Understand what their business needs are.
- Try to mimic their behavior.
- Are you good at role-playing? If not, practice. Hodor!

Usability Testing is NOT to be confused with User Acceptance Testing or User Interface / Look and Feel Testing.

Security Testing

DEFINITION

Security Testing is a type of software testing that intends to uncover vulnerabilities of the system and determine that its data and resources are protected from possible intruders.

FOCUS AREAS

There are four main focus areas to be considered in security testing (Especially for web sites/applications):

- Network security: This involves looking for vulnerabilities in the network infrastructure (resources and policies).

- System software security: This involves assessing weaknesses in the various software (operating system, database system, and other software) the application depends on.
- Client-side application security: This deals with ensuring that the client (browser or any such tool) cannot be manipulated.
- Server-side application security: This involves making sure that the server code and its technologies are robust enough to fend off any intrusion.

EXAMPLE OF A BASIC SECURITY TEST

This is an example of a very basic security test which anyone can perform on a web site/application:

- Log into the web application.
- Log out of the web application.
- Click the BACK button of the browser (Check if you are asked to log in again or if you are provided the logged-in application.)

Most types of security testing involve complex steps and out-of-the-box thinking but, sometimes, it is simple tests like the one above that help expose the most severe security risks.

OWASP

The Open Web Application Security Project (OWASP) is a great resource for software security professionals. Be sure to check out the Testing Guide:https://www.owasp.org/index.php/Category:OWASP_Testing_Project

OWASP Top 10 security threats for 2013 are:

- Injection
- Broken Authentication and Session Management
- Cross-Site Scripting (XSS)
- Insecure Direct Object References
- Security Misconfiguration
- Sensitive Data Exposure
- Missing Function Level Access Control
- Cross-Site Request Forgery (CSRF)
- Using Known Vulnerable Components

- Unvalidated Redirects and Forwards

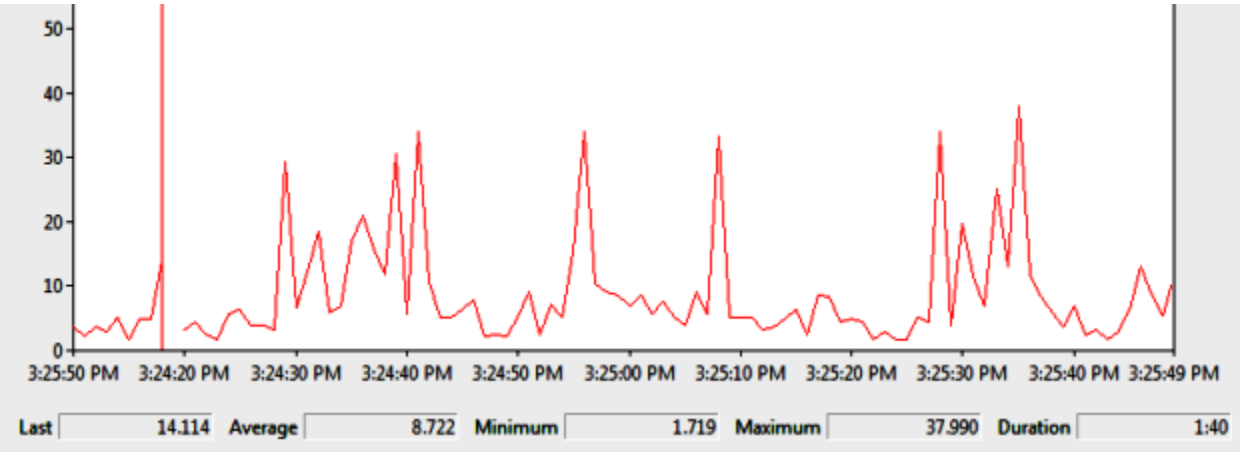
BUILDING TRUST

There is an infinite number of ways to break an application. And, security testing, by itself, is not the only (or the best) measure of how secure an application is. But, it is highly recommended that security testing is included as part of the standard software development process. After all, the world is teeming with hackers/pranksters and everyone wishes to be able to trust the system/software one produces or uses.

Performance Testing

DEFINITION

Performance Testing is a type of software testing that intends to determine how a system performs in terms of responsiveness and stability under a certain load.



There are basically four kinds of performance testing:

TYPES

- Load Testing is a type of performance testing conducted to evaluate the behavior of a system at increasing workload.
- Stress Testing a type of performance testing conducted to evaluate the behavior of a system at or beyond the limits of its anticipated workload.
- Endurance Testing is a type of performance testing conducted to evaluate the behavior of a system when a significant workload is given continuously.
- Spike Testing is a type of performance testing conducted to evaluate the behavior of a system when the load is suddenly and substantially increased.

TIPS

- Establish a test environment as close to the production environment as possible.
- Isolate the test environment even from the QA or UAT environment.
- Though there's no perfect tool for performance testing, research and decide on the tool that best fits your purpose.
- Do not rely on the results of one test. Conduct multiple tests to arrive at an average number. Be wary of any changes to the test environment from one test to the other.

Testing object oriented software

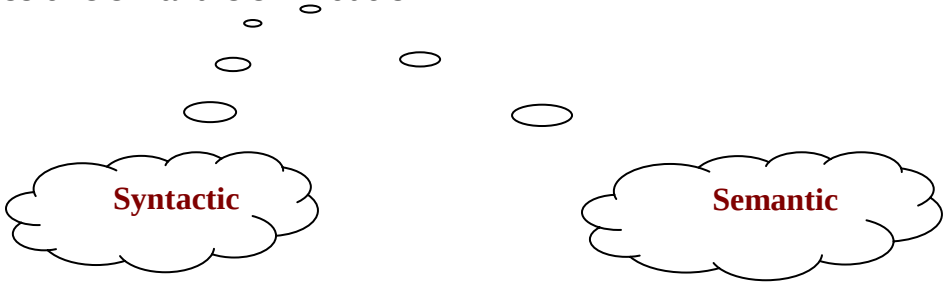
- The process of testing object-oriented systems begins with a review of the object-oriented analysis and design models. Once the code is written object-oriented testing (OOT) begins by testing "in the small" with class testing (class operations and collaborations). As classes are integrated to become subsystems class collaboration problems are investigated. Finally, use-cases from the OOA model are used to uncover software validation errors.
- OOT similar to testing conventional software in that test cases are developed to exercise the classes, their collaborations, and behavior.
- OOT differs from conventional software testing in that more emphasis is placed assessing the completeness and consistency of the OOA and OOD models as they are built.
- OOT tends to focus more on integration problems than on unit testing.

Object-Oriented Testing Activities

- Review OOA and OOD models
- Class testing after code is written
- Integration testing within subsystems
- Integration testing as subsystems are added to the system
- Validation testing based on OOA use-cases

Testing OOA and OOD Models

- OOA and OOD cannot be tested but can review the correctness and consistency.
- Correctness of OOA and OOD models



- Consistency of OOA and OOD Models

  - Assess the class-responsibility-collaborator (CRC) model and object-relationship diagram
  - Review system design (examine the object-behavior model to check mapping of system behavior to subsystems, review concurrency and task allocation, use use-case scenarios to exercise user interface design)
  - Test object model against the object relationship network to ensure that all design object contain necessary attributes and operations needed to implement the collaborations defined for each CRC card
  - Review detailed specifications of algorithms used to implement operations using conventional inspection techniques

- Object-Oriented Testing Strategies

  - Unit testing in the OO context

  - Smallest testable unit is the encapsulated class or object
  - Similar to system testing of conventional software
  - Do not test operations in isolation from one another
  - Driven by class operations and state behavior, not algorithmic detail and data flow across module interface
  - Complete test coverage of a class involves

    - Testing all operations associated with an object
    - Setting and interrogating all object attributes
    - Exercising the object in all possible states

  - Design of test for a class uses a verity of methods:

    - fault-based testing
    - random testing
    - partition testing

  - each of these methods exercises the operations encapsulated by the class
  - test sequences are designed to ensure that relevant operations are exercised
  - state of the class (the values of its attributes) is examined to determine if errors exist

  - Integration testing in the OO context

  - focuses on groups of classes that collaborate or communicate in some manner
  - integration of operations one at a time into classes is often meaningless
  - thread-based testing (testing all classes required to respond to one system input or event)

- use-based testing (begins by testing independent classes first and the dependent classes that make use of them)
- cluster testing (groups of collaborating classes are tested for interaction errors)
- regression testing is important as each thread, cluster, or subsystem is added to the system
- Levels of integration are less distinct in object-oriented systems

- Validation testing in the OO context

- focuses on visible user actions and user recognizable outputs from the system
- validation tests are based on the use-case scenarios, the object-behavior model, and the event flow diagram created in the OOA model
- conventional black-box testing methods can be used to drive the validation tests

- Test Case Design for OO Software

  - Each test case should be uniquely identified and be explicitly associated with a class to be tested
  - State the purpose of each test
  - List the testing steps for each test including:

    - list of states to test for each object involved in the test
    - list of messages and operations to exercised as a consequence of the test
    - list of exceptions that may occur as the object is tested
    - list of external conditions needed to be changed for the test
    - supplementary information required to understand or implement the test

  - Testing Surface Structure and Deep Structure

    - Testing surface structure (exercising the structure observable by end-user, this often involves observing and interviewing users as they manipulate system objects)
    - Testing deep structure (exercising internal program structure - the dependencies, behaviors, and communications mechanisms established as part of the system and object design)

- Testing Methods Applicable at The Class Level

- Random testing - requires large numbers data permutations and combinations, and can be inefficient

    - Identify operations applicable to a class
    - Define constraints on their use
    - Identify a minimum test sequence
    - Generate a variety of random test sequences.

- Partition testing - reduces the number of test cases required to test a class
    - state-based partitioning - tests designed in way so that operations that cause state changes are tested separately from those that do not.
    - attribute-based partitioning - for each class attribute, operations are classified according to those that use the attribute, those that modify the attribute, and those that do not use or modify the attribute
    - category-based partitioning - operations are categorized according to the function they perform: initialization, computation, query, termination

- Fault-based testing
    - best reserved for operations and the class level
    - uses the inheritance structure
    - tester examines the OOA model and hypothesizes a set of plausible defects that may be encountered in operation calls and message connections and builds appropriate test cases
    - misses incorrect specification and errors in subsystem interactions

- Inter-Class Test Case Design
    - Test case design becomes more complicated as integration of the OO system begins – testing of collaboration between classes
    - Multiple class testing
        - for each client class use the list of class operators to generate random test sequences that send messages to other server classes
        - for each message generated determine the collaborator class and the corresponding server object operator
        - for each server class operator (invoked by a client object message) determine the message it transmits
        - for each message, determine the next level of operators that are invoked and incorporate them into the test sequence
    - Tests derived from behavior models
        - Use the state transition diagram (STD) as a model that represent the dynamic behavior of a class.
        - test cases must cover all states in the STD
        - breadth first traversal of the state model can be used (test one transition at a time and only make use of previously tested transitions when testing a new transition)
        - test cases can also be derived to ensure that all behaviors for the class have been adequately exercised

- Testing Methods Applicable at Inter-Class Level

- Cluster Testing

  - Is concerned with integrating and testing clusters of cooperating objects
  - Identify clusters using knowledge of the operation of objects and the system features that are implemented by these clusters

  - Approaches to Cluster Testing
    - Use-case or scenario testing
      - Testing is based on a user interactions with the system
      - Has the advantage that it tests system features as experienced by users
    - Thread testing – tests the systems response to events as processing threads through the system
    - Object interaction testing – tests sequences of object interactions that stop when an object operation does not call on services from another object

- Use Case/Scenario-based Testing

  - Based on
    - use cases
    - corresponding sequence diagrams
  - Identify scenarios from use-cases and supplement these with interaction diagrams that show the objects involved in the scenario
  - Concentrates on (functional) requirements
    - Every use case
    - Every fully expanded extension (<<extend>>) combination
    - Every fully expanded uses (<<uses>>) combination
    - Tests normal as well as exceptional behavior

  - A scenario is a path through sequence diagram
  - Many different scenarios may be associated with a sequence diagram
  - using the user tasks described in the use-cases and building the test cases from the tasks and their variants
  - uncovers errors that occur when any actor interacts with the OO software
  - concentrates on what the use does, not what the product does
  - you can get a higher return on your effort by spending more time on reviewing the use-cases as they are created, than spending more time on use-case testing

  OO Test Design Issues

  White-box testing methods can be applied to testing the code used to implement class operations, but not much else

  Black-box testing methods are appropriate for testing OO systems

  Object-oriented programming brings additional testing concerns

  - classes may contain operations that are inherited from super classes
  - subclasses may contain operations that were redefined rather than inherited

- all classes derived from an previously tested base class need to be thoroughly tested.

SCSX1038 Software Quality Assurance and Testing
UNIT-3

Difference between Static and Dynamic Testing

	Static Testing	Dynamic Testing
1	In this type of testing we do not execute the code	In this type of Testing we always execute the code.
2	It means examining and reviewing the software.	It means testing, running, and using the software.
3	In this form of Testing methods like code review, inspection, walkthrough, reviews are used.	In this Testing methods like testing and validations are used.
4	It is done in the phase of verification.	It is done in the phase of validation.
5	This testing means "How we prevent" means it always talks about prevention.	This testing means "How we cure" means it always talks about cure.
6	It is not a time consuming job because its purpose is to examine the software or code.	It is always a time consuming job because its purpose is to execute the software or code and it may also involve running more test cases.
7	As it can always start early in the life cycle it definitely reduces the cost of product or you can say project.	As it not starting early in the life cycle hence it definitely increases the cost of product/project.
8	It is always considered as less cost effective job/task.	It is always considered as more cost effective job/task.
9	It can find errors that dynamic testing cannot find and it is a low level exercise.	It can find errors that static testing cannot find and it is a high level exercise.
10	It is not considered as a time consuming job or task.	It is always considered as a time consuming job or task because it requires several test cases to execute.
11	Techniques/methods of static testing are inspections, reviews, and walkthroughs etc.	Technique/method of dynamic testing is always software testing means testing.
12	Static testing is also known by the name Dry Run Testing.	Dynamic Testing is not known by any other name.
13	It definitely comes before	It definitely follows after static

	dynamic testing.	testing.

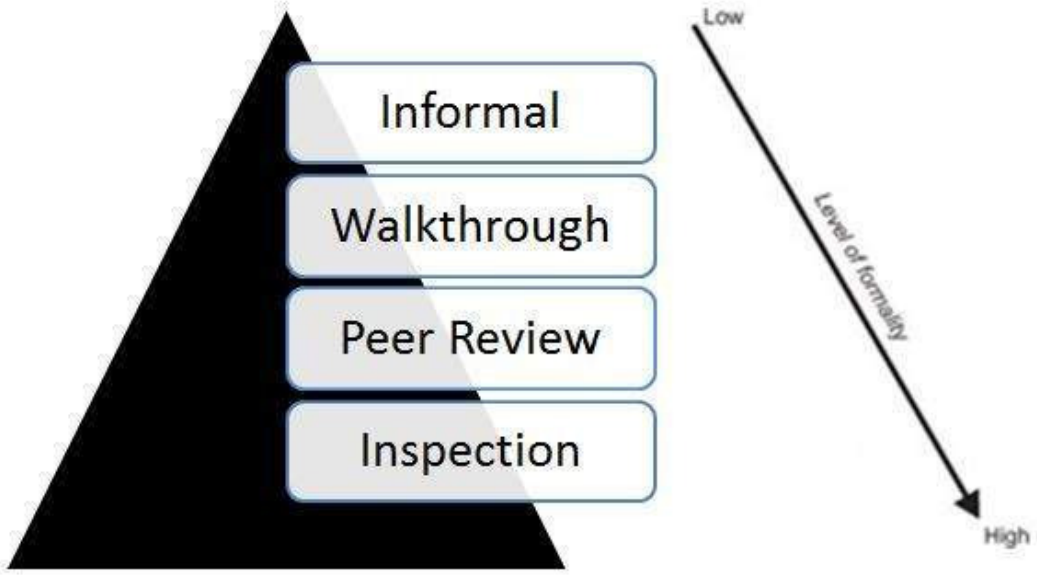
What is Static Testing?

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

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

Types of Reviews:

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



Static Analysis - By Tools:

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

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

- Security vulnerabilities
- Syntax violations
-

1. Informal reviews:

Informal reviews are applied many times during the early stages of the life cycle of the document. A two person team can conduct an informal review. In later stages these reviews often involve more people and a meeting. The goal is to keep the author and to improve the quality of the document. The most important thing to keep in mind about the informal reviews is that they are **not documented.**

The main review types that come under the **static testing** are mentioned below:

2. Walkthrough:

- It is not a formal process
- It is led by the authors
- Author guide the participants through the document according to his or her thought process to achieve a common understanding and to gather feedback.
- Useful for the people if they are not from the software discipline, who are not used to or cannot easily understand software development process.
- Is especially useful for higher level documents like requirement specification, etc.

The goals of a walkthrough:

- To present the documents both within and outside the software discipline in order to gather the information regarding the topic under documentation.
- To explain or do the knowledge transfer and evaluate the contents of the document
- To achieve a common understanding and to gather feedback.
- To examine and discuss the validity of the proposed solutions

3. Inspection:

- It is the most formal review type
- It is led by the trained moderators
- During inspection the documents are prepared and checked thoroughly by the reviewers before the meeting
- It involves peers to examine the product
- A separate preparation is carried out during which the product is examined and the defects are found
- The defects found are documented in a logging list or issue log
- A formal follow-up is carried out by the moderator applying exit criteria

The goals of inspection are:

- It helps the author to improve the quality of the document under inspection
- It removes defects efficiently and as early as possible
- It improve product quality
- It create common understanding by exchanging information
- It learn from defects found and prevent the occurrence of similar defects

static analysis (static code analysis)

Static analysis, also called static code analysis, is a method of computer program debugging that is done by examining the code without executing the program. The process provides an understanding of the code structure, and can help to ensure that the code adheres to industry standards. Automated tools can assist programmers and developers in carrying out static analysis. The process of scrutinizing code by visual inspection alone (by looking at a printout, for example), without the assistance of automated tools, is sometimes called program understanding or program comprehension.

The principal advantage of static analysis is the fact that it can reveal errors that do not manifest themselves until a disaster occurs weeks, months or years after release. Nevertheless, static analysis is only a first step in a comprehensive software quality-control regime. After static analysis has been done, dynamic

analysis is often performed in an effort to uncover subtle defects or vulnerabilities. In computer terminology, static means fixed, while dynamic means capable of action and/or change. Dynamic analysis involves the testing and evaluation of a program based on execution. Static and dynamic analysis, considered together, are sometimes referred to as glass-box testing.

What is Dynamic Testing?

Dynamic Testing is a kind of software testing technique using which the dynamic behaviour of the code is analysed.

For Performing dynamic, testing the software should be compiled and executed and parameters such as memory usage, CPU usage, response time and overall performance of the software are analyzed.

Dynamic testing involves testing the software for the input values and output values are analyzed. Dynamic testing is the Validation part of Verification and Validation.

Dynamic Testing Techniques

The Dynamic Testing Techniques are broadly classified into two categories. They are:

- Functional Testing
- Non-Functional Testing

Levels of Dynamic Testing

There are various levels of Dynamic Testing Techniques. They are:

- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing

What is Dynamic testing technique?

- This testing technique needs computer for testing.
- It is done during Validation process.

- The software is tested by executing it on computer.
- Example of this **Dynamic Testing Technique**: **Unit testing**,**integration testing**, **system testing**.

What is White Box Testing?

White box testing is a testing technique, that examines the program structure and derives test data from the program logic/code. The other names of glass box testing are clear box testing, open box testing, logic driven testing or path driven testing or structural testing.

White Box Testing Techniques:

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

Advantages of White Box Testing:

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

Disadvantages of White Box Testing:

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

Cyclomatic complexity and its Applications

To understand CyclomaticComplexity , lets first understand -

What is Software Metric?

Measurement is nothing but quantitative indication of size / dimension / capacity of an attribute of a product / process.

Software metric is defined as a quantitative measure of an attribute a software system possesses with respect to Cost, Quality, Size and Schedule.

Example-

Measure - No. of Errors

Metrics - No. of Errors found per person

What is Cyclomatic Complexity?

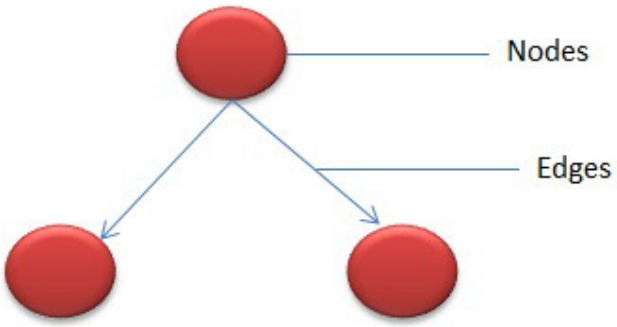
Cyclomatic complexity is a software metric used to measure the complexity of a program. These metric, measures independent paths through program source code.

Independent path is defined as a path that has atleast one edge which has not been traversed before in any other paths.

Cyclomatic complexity can be calculated with respect to functions, modules, methods or classes within a program.

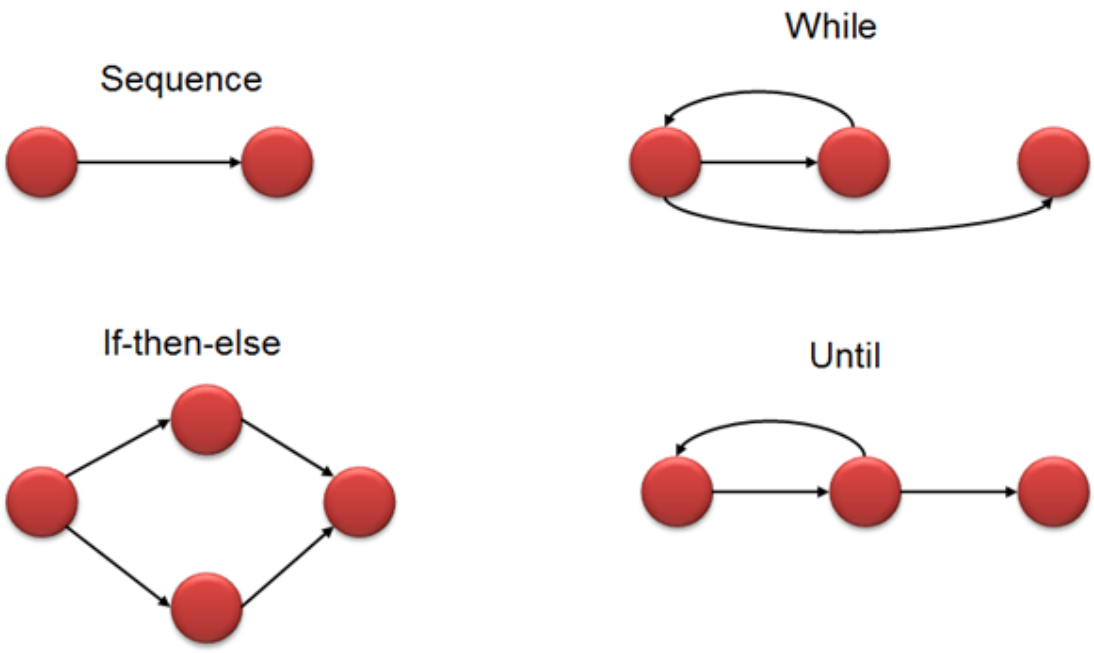
This metric was developed by Thomas J. McCabe in 1976 and it is based on a control flow representation of the program. Control flow depicts a program as a graph which consists of Nodes and Edges.

In the graph, Nodes represent processing tasks while edges represent control flow between the nodes.



Flow graph notation for a program:

Flow Graph notation for a program is defines .several nodes connected through the edges. Below are Flow diagrams for statements like if-else, While, until and normal sequence of flow.



Mathematical representation:

Mathematically, it is set of independent paths through the graph diagram. The complexity of the program can be defined as -

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

Where,

E - Number of edges

N - Number of Nodes

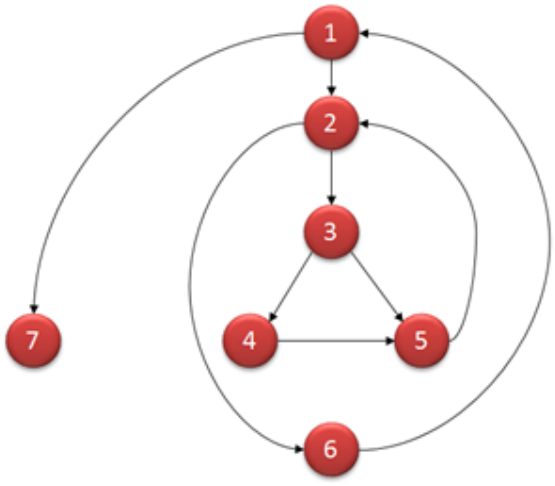
V (G) = P + 1

Where P = Number of predicate nodes (node that contains condition)

Example -

```
1       i = 0;
2
3       n=4; //N-Number of nodes present in the graph
4
5       while (i<n-1) do
6
7       j = i + 1;
8
9       while (j<n) do
10
11      if A[i]<A[j] then
12
13      swap(A[i], A[j]);
14
15      end do;
16
17      i=i+1;
18
19      end do;
```

Flow graph for this program will be



Computing mathematically,

- V(G) = 9 - 7 + 2 = 4
- V(G) = 3 + 1 = 4 (Condition nodes are 1,2 and 3 nodes)
- Basis Set - A set of possible execution path of a program
- 1, 7
- 1, 2, 6, 1, 7
- 1, 2, 3, 4, 5, 2, 6, 1, 7
- 1, 2, 3, 5, 2, 6, 1, 7

Properties of Cyclomatic complexity:

Following are the properties of Cyclomatic complexity:

1. V (G) is the maximum number of independent paths in the graph
2. V (G) >=1
3. G will have one path if V (G) = 1
4. Minimize complexity to 10

How this metric is useful for software testing?

Basis Path testing is one of White box technique and it guarantees to execute atleast one statement during testing. It checks each linearly independent path through the program, which **means number test cases, will be equivalent to the cyclomatic complexity of the program.**

This metric is useful because of properties of Cyclomatic complexity (M) -

1. M can be number of test cases to achieve branch coverage(Upper Bound)
2. M can be number of paths through the graphs.(Lower Bound)

Consider this example -

```
1       If (Condition 1)
2
3       Statement 1
4
5       Else
6
7       Statement 2
8
9       If (Condition 2)
10
11      Statement 3
12
```

13 Else
14
15 Statement 4

Cyclomatic Complexity for this program will be 9-7+2=4.

As complexity has calculated as 4, four test cases are necessary to the complete path coverage for the above example.

Steps to be followed:

The following steps should be followed for computing Cyclomatic complexity and test cases design.

Step 1 - Construction of graph with nodes and edges from the code

Step 2 - Identification of independent paths

Step 3 - Cyclomatic Complexity Calculation

Step 4 - Design of Test Cases

Once the basic set is formed, TEST CASES should be written to execute all the paths.

More on V (G):

Cyclomatic complexity can be calculated manually if the program is small. Automated tools need to be used if the program is very complex as this involves more flow graphs. Based on complexity number, team can conclude on the actions that need to be taken for measure.

Following table gives overview on the complexity number and corresponding meaning of v (G):

Complexity Number	Meaning
1-10	Structured and well written code High Testability Cost and Effort is less

10-20	Complex Code Medium Testability Cost and effort is Medium
20-40	Very complex Code Low Testability Cost and Effort are high
>40	Not at all testable Very high Cost and Effort

Tools for Cyclomatic Complexity calculation:

Many tools are available for determining the complexity of the application. Some complexity calculation tools are used for specific technologies. Complexity can be found by the number of decision points in a program. The decision points are if, for, for-each, while, do, catch, case statements in a source code.

Examples of tools are

- OCLint - Static code analyzer for C and Related Languages
- devMetrics - Analyzing metrics for C# projects
- Reflector Add In - Code metrics for .NET assemblies
- GMetrics - Find metrics in Java related applications
- NDepends - Metrics in Java applications

Uses of Cyclomatic Complexity:

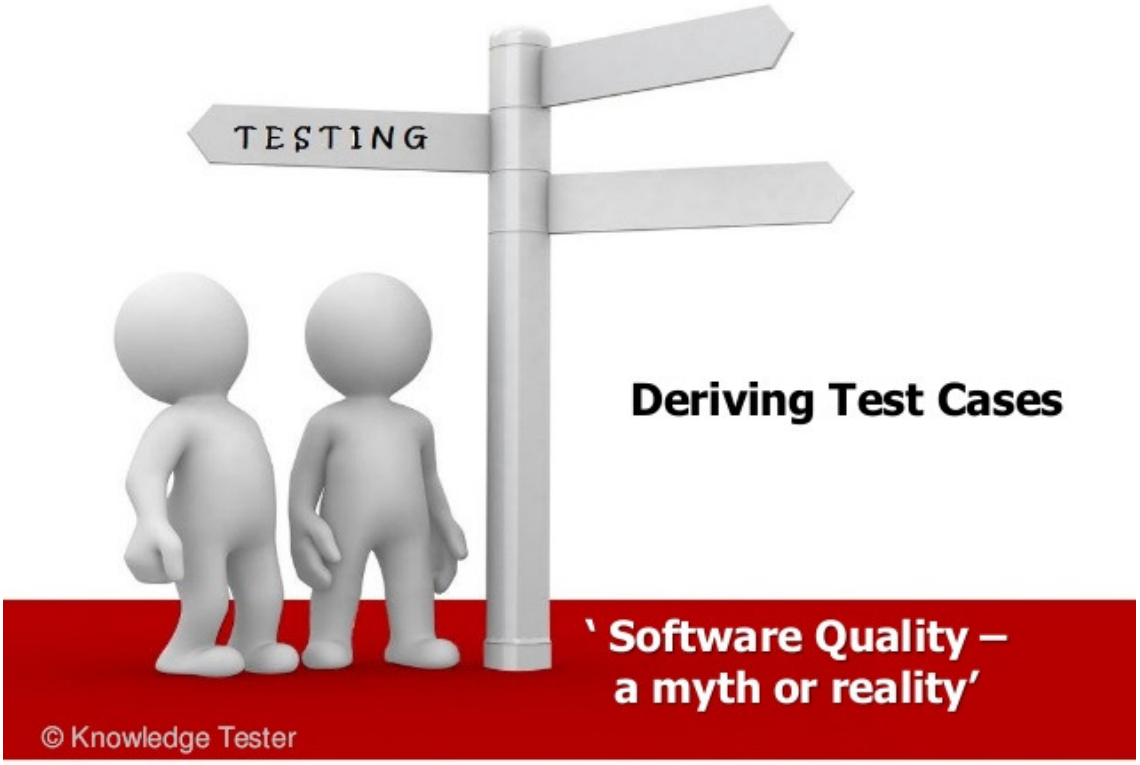
Cyclomatic Complexity can prove to be very helpful in

- Helps developers and testers to determine independent path executions
- Developers can assure that all the paths have been tested atleast once
- Helps us to focus more on the uncovered paths
- Improve code coverage
- Evaluate the risk associated with the application or program

- Using these metrics early in the cycle reduces more risk of the program

Conclusion:

Cyclomatic Complexity is software metric useful for structured or white box testing .It is mainly used to evaluate complexity of a program. If the decision points are more, then complexity of the program is more. If program has high complexity number, then probability of error is high with increased time for maintenance and trouble shoot.



Deriving test cases

- Software Testing looks interesting; let's learn it : One skill that every Tester needs is 'derive test cases for a specific feature'
- What is a Test case?
- Exercise 1: Adder application
- Equivalence class partition
- Exercise 2: Doc to Pdf publishing
- Combinatorial Testing
- Bonus Exercise: Looking at the code

What is a Test case?

- 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.

  http://en.wikipedia.org/wiki/Test_case
- Test case does have a corresponding with Use case.
- Normally a test case has:
  - –ID
  - –Description
  - –Input
  - –Expected output.
- A Test case either 'passes' or 'fails'.
- A Feature Test Plan is usually a set of Test cases.

Exercise 1: Adder application

Application: Adder application which takes two 2-digit no.s and displays the sum.

```
? 2
? 3
  5
? _
```

Reference: 'Testing Computer Software' book by Cem Kaner, Jack Falk and Hung Q. Nguyen.

Adder test cases 1

Test Case Inputs	Result	Notes
2 + 3	5	The basic test should pass first
99 + 99	198	Addition of 2 largest numbers
-99 + -99	-198	Addition of 2 largest negative numbers
99 + 14	113	First number is largest
-38 + 55	17	Mixing +ve and –ve numbers
56 + 99	155	Second number is largest
9 + 9	18	9 is the largest single digit number
0 + 0	0	0 is a special case
0 + 23	23	First number as 0
78 + 0	78	Second number as 0

Adder test cases 2

Test Case Inputs	Notes
100 + 100	Beyond maximum number
<Enter> + <Enter>	What if we provide nothing?
12345678 + 0	Trying a very large number
1.2 + 5	Trying a decimal number
A + b	Trying not a number
<Ctrl + A> + <F10>	Trying special keys
…	

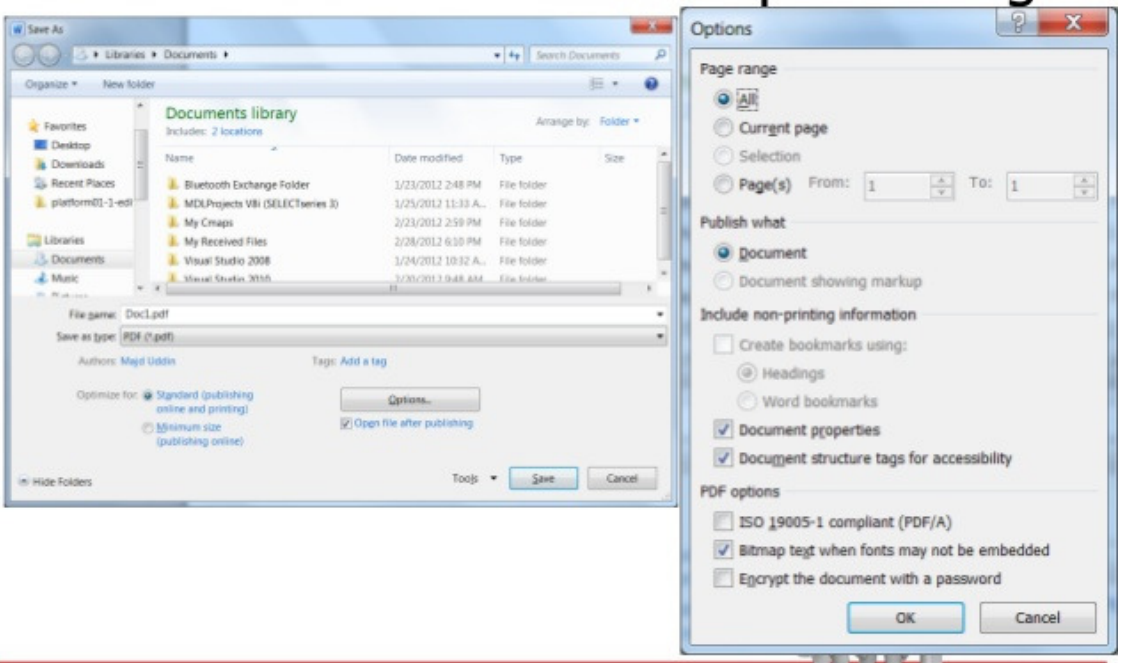
Equivalence Class Partition

The concept comes from Mathematics where we break Set into sub-sets based upon some criteria. Now assuming that one element from subset of a Set is equivalent to other member of same subset, we have one Test case represent one subset.

Revisiting test cases

- If we try 11+12, do we need to try 55+67?
- Is 0 +9 and 9+0 belong to same subset or different subsets?
- Are we missing some equivalence class?
- When we move this concept to other data types, it becomes interesting. Say we want to try special characters in a string: is 'test$' same as '$test' or 'te$st'?

Exercise 2: Word to PDF publishing

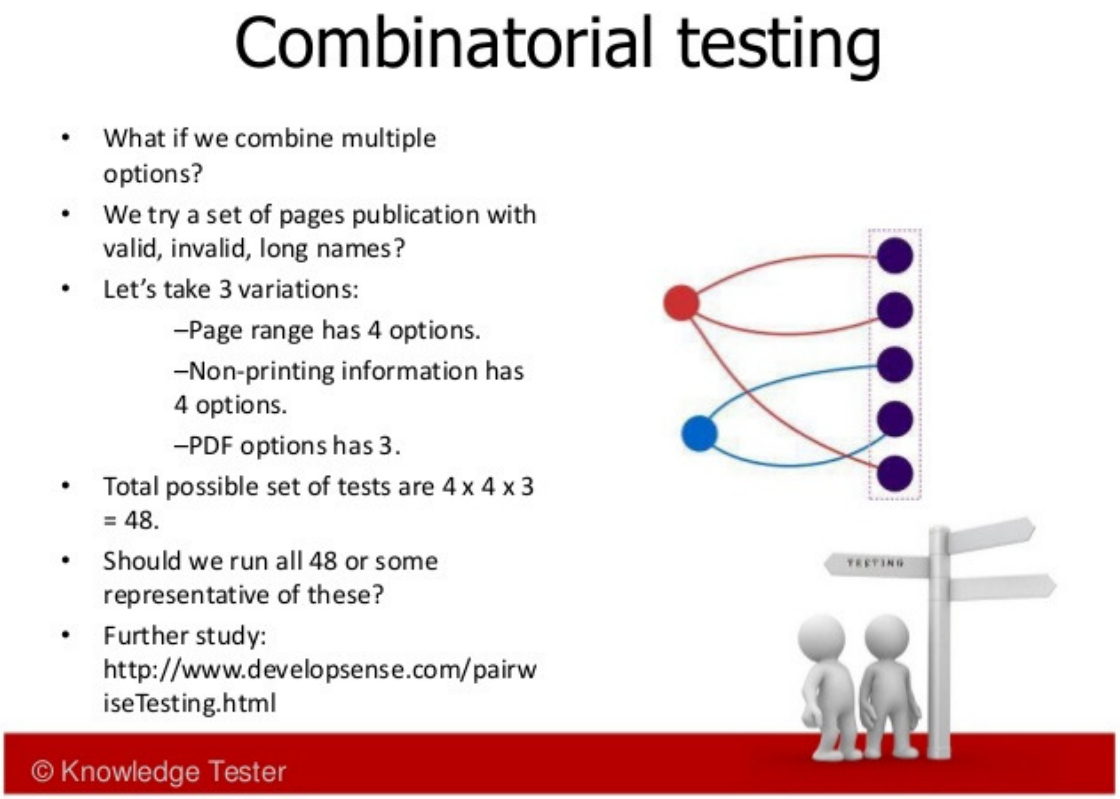


PDF Publishing test cases 1

Test case	Notes
PDF can be generated with default options	Basic test
All pages can be published	Page range options
Current page can be published	Page range options
Pages range can be published	Page range options
Document with Markups can be published	Different Mark ups / bookmarks
Document Properties are available after publishing	Properties like Author name, Organization name etc.
PDF options are applicable	What is ISO 19005-1?
PDF is opened after publishing	What if Adobe Reader is not installed?
PDF can be published at any path	Different type of directory selections

PDF Publishing test cases 2

Test case	Notes
PDF optimization works	Check for different file sizes
Any File names can be provided	
Publishing to locations over network	LAN, WAN
Publishing older word documents	Compatibility testing Office 2007/2003
Publishing huge word documents	Performance testing
Word drawings are published correctly	Word specific data
Embedded pictures are published correctly	Word specific data
Links in the word file are published correctly	Word specific data
...	



Control Structure testing.

Control structure testing is a group of white-box testing methods.

- 1.0 Branch Testing
- 1.1 Condition Testing
- 1.2 Data Flow Testing
- 1.3 Loop Testing

1.0 Branch Testing

- also called Decision Testing
- definition: "For every decision, each branch needs to be executed at least once."
- shortcoming - ignores implicit paths that result from compound conditionals.
- Treats a compound conditional as a single statement. (We count each branch taken out of the decision, regardless which condition lead to the branch.)
- This example has two branches to be executed:
  IF (a equals b) THEN

```
    statement 1
ELSE
    statement 2
END IF
```

- This examples also has just two branches to be executed, despite the compound conditional:

```
IF ( a equals b AND c less than d ) THEN
    statement 1
ELSE
    statement 2
END IF
```

- This example has four branches to be executed:

```
IF ( a equals b) THEN
    statement 1
ELSE
    IF ( c equals d) THEN
        statement 2
    ELSE
        statement 3
    END IF
END IF
```

- Obvious decision statements are if, for, while, switch.
- Subtle decisions are return *boolean expression*, ternary expressions, try-catch.
- For this course you don't need to write test cases for IOException and OutOfMemory exception.

- See this problem and solution.

1.1 Condition Testing

Condition testing is a test construction method that focuses on exercising the logical conditions in a program module.

Errors in conditions can be due to:

- Boolean operator error
- Boolean variable error
- Boolean parenthesis error
- Relational operator error
- Arithmetic expression error

definition: "For a compound condition C, the true and false branches of C *and every simple condition in C* need to be executed at least once."

Multiple-condition testing requires that all true-false *combinations* of simple conditions be exercised at least once. Therefore, all statements, branches, and conditions are necessarily covered.

1.2 Data Flow Testing

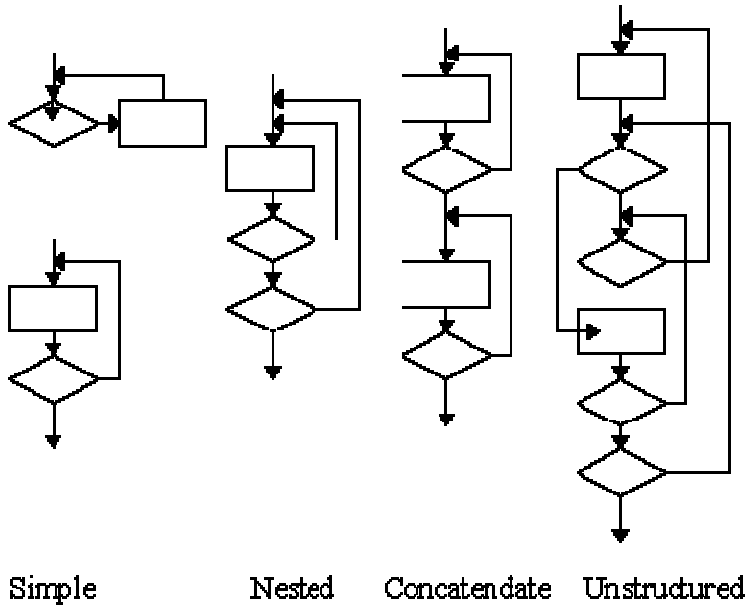
Selects test paths according to the location of definitions and use of variables. This is a somewhat sophisticated technique and is not practical for extensive use. Its use should be targeted to modules with nested if and loop statements.

1.3 Loop Testing

Loops are fundamental to many algorithms and need thorough testing.

There are four different classes of loops: simple, concatenated, nested, and unstructured.

Examples:



Create a set of tests that force the following situations:

- **Simple Loops,** where *n* is the maximum number of allowable passes through the loop.

- Skip loop entirely
- Only one pass through loop
- Two passes through loop
- m passes through loop where m<n.
- (n-1), n, and (n+1) passes through the loop.

- **Nested Loops**
  - Start with inner loop. Set all other loops to minimum values.
  - Conduct simple loop testing on inner loop.
  - Work outwards
  - Continue until all loops tested.
- **Concatenated Loops**
  - If independent loops, use simple loop testing.
  - If dependent, treat as nested loops.
- **Unstructured loops**
  - Don't test - redesign.

```
public class loopdemo
{
   private int[] numbers = {5,-3,8,-12,4,1,-20,6,2,10};

   /** Compute total of numItems positive numbers in the array
*@paramnumItems how many items to total, maximum of10.
*/
   public intfindTotal(intnumItems)
   {
int total = 0;
      if (numItems<= 10)
      {
      for (int count=0; count <numItems; count = count + 1)
         {
                  if (numbers[count] > 0)
                  {
            total = total + numbers[count];
                  }
         }
      }
      return total;
   }
}

        public void testOne()
        {
                loopdemo app = new loopdemo();
```

```
        assertEquals(0, app.findTotal(0));
        assertEquals(5, app.findTotal(1));
        assertEquals(5, app.findTotal(2));
        assertEquals(17, app.findTotal(5));
        assertEquals(26, app.findTotal(9));
        assertEquals(36, app.findTotal(10));
        assertEquals(0, app.findTotal(11));
    }
```

What is Black box Testing?

Black-box testing is a method of software testing that examines the functionality of an application based on the specifications. It is also known as Specifications based testing. Independent Testing Team usually performs this type of testing during the software testing life cycle.

This method of test can be applied to each and every level of software testing such as unit, integration, system and acceptance testing.

Behavioural Testing Techniques:

There are different techniques involved in Black Box testing.

- Equivalence Class
- Boundary Value Analysis
- Domain Tests
- Orthogonal Arrays
- Decision Tables
- State Models
- Exploratory Testing
- All-pairs testing

Equivalence Partitioning

Equivalence partitioning (EP) is a blackbox testing technique. This technique is very common and mostly used by all the testers informally. Equivalence partitions are also known as equivalence classes.

As the name suggests Equivalence partitioning is to divide or to partition a set of test conditions into sets or groups that can be considered same by the software system.

As you all know that exhaustive testing of the software is not feasible task for complex software's so by using equivalence partitioning technique we need to test only one condition from each partition because it is assumed that all the conditions in one partition will be treated in the same way by the software. If one condition works fine then all the conditions within that partition will work the same way and tester does not need to test other conditions or in other way if one condition fails in that partition then all other conditions will fail in that partition.

These conditions may not always be true however testers can use better partitions and also test some more conditions within those partitions to confirm that the selection of that partition is fine.

Lets take some examples:
A store in city offers different discounts depending on the purchases made by the individual. In order to test the software that calculates the discounts, we can identify the ranges of purchase values that earn the different discounts. For example, if a purchase is in the range of $1 up to $50 has no discounts, a purchase over $50 and up to $200 has a 5% discount, and purchases of $201 and up to $500 have a 10% discounts, and purchases of $501 and above have a 15% discounts.

Now we can identify 4 valid equivalence partitions and 1 invalid partition as shown below.

Invalid Partition	Valid Partition(No Discounts)	Valid Partition(5%)	Valid Partition(10%)	Valid Partition(15%)
$0.01	$1-$50	$51-$200	$201-$500	$501-Above

Boundary Value Analysis

What is a Boundary Value

A boundary value is any input or output value on the edge of an equivalence partition.

Let us take an example to explain this:
Suppose you have a software which accepts values between 1-1000, so the valid partition will be (1-1000), equivalence partitions will be like:

Invalid Partition	Valid Partition	Invalid Partition
0	1-1000	1001 and above

And the boundary values will be 1, 1000 from valid partition and 0,1001 from invalid partitions.

Boundary Value Analysis is a black box test design technique where test case are designed by using boundary values, BVA is used in range checking.
Example:2
A store in city offers different discounts depending on the purchases made by the individual. In order to test the software that calculates the discounts, we can identify the ranges of purchase values that earn the different discounts. For example, if a purchase is in the range of $1 up to $50 has no discounts, a purchase over $50 and up to $200 has a 5% discount, and purchases of $201 and up to $500 have a 10% discounts, and purchases of $501 and above have a 15% discounts.

We can identify 4 valid equivalence partitions and 1 invalid partition as shown below.

Invalid Partition	Valid Partition(No Discounts)	Valid Partition(5%)	Valid Partition(10%)	Valid Partition(15%)
$0.01	$1-$50	$51-$200	$201-$500	$501-Above

From this table we can identify the boundary values of each partition. We assume that two decimal digits are allowed.

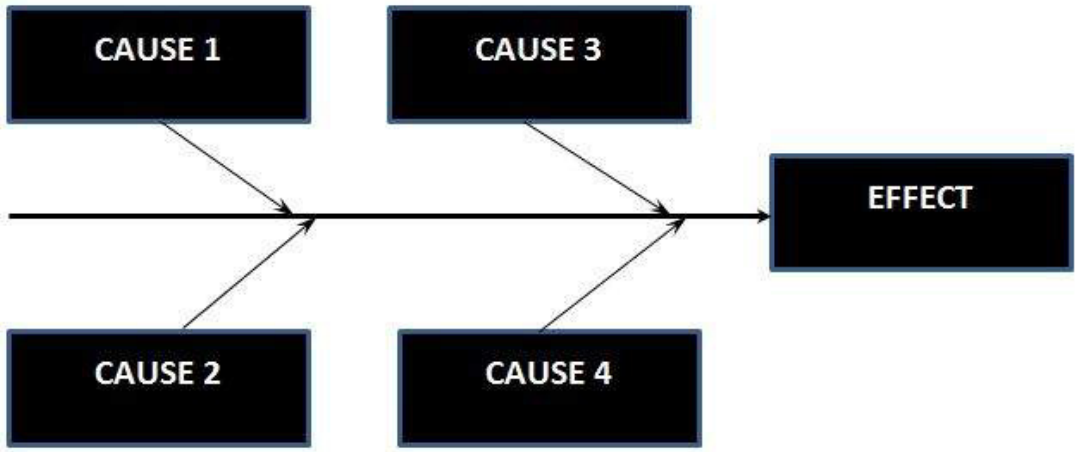
Boundary values for Invalid partition :0.00
Boundary values for valid partition(No Discounts): 1, 50
Boundary values for valid partition(5% Discount): 51, 200
Boundary values for valid partition(10% Discount): 201,500
Boundary values for valid partition(15% Discount): 501, Max allowed number in the software application

What is Cause-Effect Graph?

Cause Effect Graph is a black box testing technique that graphically illustrates the relationship between a given outcome and all the factors that influence the outcome.

It is also known as Ishikawa diagram as it was invented by Kaoru Ishikawa or fish bone diagram because of the way it looks.

Cause Effect - Flow Diagram



Circumstances - under which Cause-Effect Diagram used

- To Identify the possible root causes, the reasons for a specific effect, problem, or outcome.
- To Relate the interactions of the system among the factors affecting a particular process or effect.
- To Analyze the existing problems so that corrective action can be taken at the earliest.

Benefits :

- It Helps us to determine the root causes of a problem or quality using a structured approach.
- It Uses an orderly, easy-to-read format to diagram cause-and-effect relationships.
- It Indicates possible causes of variation in a process.
- It Identifies areas, where data should be collected for further study.

- It Encourages team participation and utilizes the team knowledge of the process.
- It Increases knowledge of the process by helping everyone to learn more about the factors at work and how they relate.

Steps for drawing cause-Effect Diagram:

- **Step 1 :** Identify and Define the Effect
- **Step 2 :** Fill in the Effect Box and Draw the Spine
- **Step 3:** Identify the main causes contributing to the effect being studied.
- **Step 4 :** For each major branch, identify other specific factors which may be the causes of the EFFECT.
- **Step 5 :** Categorize relative causes and provide detailed levels of causes.

What is Syntax Testing?

Syntax Testing, a black box testing technique, involves testing the System inputs and it is usually automated because syntax testing produces a large number of tests. Internal and external inputs have to conform the below formats:

- Format of the input data from users.
- File formats.
- Database schemas.

Syntax Testing - Steps:

- Identify the target language or format.
- Define the syntax of the language.
- Validate and Debug the syntax.

Syntax Testing - Limitations:

- Sometimes it is easy to forget the normal cases.
- Syntax testing needs driver program to be built that automatically sequences through a set of test cases usually stored as data.

SCSX1038 Software Quality Assurance and Testing

Unit - IV

Quality Assurance is *process* oriented and focuses on defect *prevention*, while **Quality control** is *product* oriented and focuses on defect *identification*.

Quality Assurance versus Quality Control comparison

	Quality Assurance	Quality Control
Definition	QA is a set of activities for ensuring quality in the processes by which products are developed.	QC is a set of activities for ensuring quality in products. The activities focus on identifying defects in the actual products produced.
Focus on	QA aims to prevent defects with a focus on the process used to make the product. It is a proactive quality process.	QC aims to identify (and correct) defects in the finished product. Quality control, therefore, is a reactive process.
Goal	The goal of QA is to improve development and test processes so that defects do not arise when the product is being developed.	The goal of QC is to identify defects after a product is developed and before it's released.
How	Establish a good quality management system and the assessment of its adequacy. Periodic conformance audits of the operations of the system.	Finding & eliminating sources of quality problems through tools & equipment so that customer's requirements are continually met.
What	Prevention of quality problems through planned and systematic activities including documentation.	The activities or techniques used to achieve and maintain the product quality, process and service.
Responsibility	Everyone on the team involved in developing the product is	Quality control is usually theresponsibility of a specific

Quality Assurance versus Quality Control comparison

	Quality Assurance	Quality Control
	responsible for quality assurance.	team that tests the product for defects.
Example	Verification is an example of QA	Validation/Software Testing is an example of QC
Statistical Techniques	Statistical Tools & Techniques can be applied in both QA & QC. When they are applied to processes (process inputs & operational parameters), they are called Statistical Process Control (SPC); & it becomes the part of QA.	When statistical tools & techniques are applied to finished products (process outputs), they are called as Statistical Quality Control (SQC) & comes under QC.
As a tool	QA is a managerial tool	QC is a corrective tool
Orientation	QA is process oriented	QC is product oriented

Software Quality Factors

A software quality factor is a non-functional requirement for a software program which is not called up by the customer's contract, but nevertheless is a desirable requirement which enhances the quality of the software program.

Some software quality factors are listed here:

1. **Understandability** is possessed by a software product if the purpose of the product is clear. This goes further than just a statement of purpose - all of the design and user documentation must be clearly written so that it is easily understandable. This is obviously subjective in that the user context must be taken into account, i.e. if the software product is to be used by software engineers it is not required to be understandable to the layman.

2. A software product possesses the characteristic **completeness** to the extent that all of its parts are present and each of its parts is fully developed. This means that if the code calls a sub-routine from an external library, the software package must provide reference to that library and all required parameters must be passed. All required inputdata must be available.

3. A software product possesses the characteristic **conciseness** to the extent that no excessive information is present. This is important where memory capacity is limited, and it is important to reduce lines of code to a minimum. It can be improved by replacing repeated functionality by one sub-routine or function which achieves that functionality. It also applies to documents.

4. A software product possesses the characteristic **portability** to the extent that it can be operated easily and well on computer configurations other than its current one. This is particularly important with PC applications where, for example, a product is expected to work on all 80486 processors.

5. A software product possesses the characteristic maintainability to the extent that it facilitates updating to satisfy new requirements. Thus the software product which is maintainable should be **well-documented,** not complex, and should have spare capacity for memory usage and processor speed.

6. A software product possesses the characteristic **testability** to the extent that it facilitates the establishment of acceptance criteria and supports evaluation of its performance. Such a characteristic must be built-in

during the design phase if the product is to be easily testable - a complex design leads to poor testability.

7. A software product possesses the characteristic usability to the extent that it is **convenient and practicable** to use. This is affected by such things as the human-computer interface. The component of the software which has most impact on this is the graphical user interface (GUI).

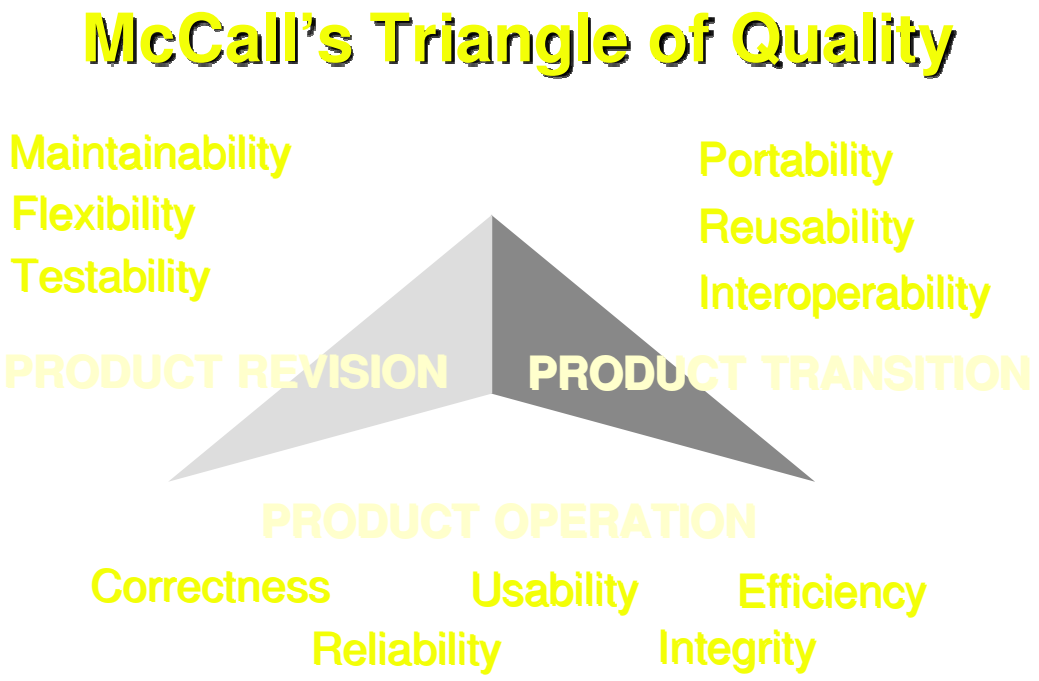
8. A software product possesses the characteristic **reliability** to the extent that it can be expected to perform its intended functions satisfactorily. This implies a time factor in that a reliable product is expected to perform correctly over a period of time. It also encompasses environmental considerations in that the product is required to perform correctly in whichever conditions it finds itself - this is sometimes termed robustness.

McCall's Quality Factors

McCall's quality factors were proposed in the early 1970s. They are as valid today as they were in that time. It's likely that software built to conform to these factors will exhibit high quality well into the 21st century, even if there are dramatic changes in technology.

The factors that affect S/W quality can be categorized in two broad groups:

1. factors that can be directly measured (defects uncovered during testing)
2. factors that can be measured only indirectly (Usability and maintainability)



The S/W quality factors shown above focus on three important aspects of a S/W product:

- Its operational characteristics
- Its ability to undergo change
- Its adaptability to new environments

Referring to these factors, McCall and his colleagues provide the following descriptions:

Correctness: The extent to which a program satisfies its specs and fulfills the customer's mission objectives.

Reliability: The extent to which a program can be expected to perform its intended function with required precision.

Efficiency: The amount of computing resources and code required to perform is function.

Integrity: The extent to which access to S/W or data by unauthorized persons can be controlled.

Usability: The effort required to learn, operate, prepare input for, and interpret output of a program.

Maintainability: The effort required to locate and fix errors in a program.

Flexibility: The effort required to modify an operational program.

Testability: The effort required to test a program to ensure that it performs its intended function.

Portability: The effort required to transfer the program from one hardware and/or software system environment to another.

Reusability: The extent to which a program can be reused in other applications-related to the packaging and scope f the functions that the program performs.

Interoperability: The effort required to couple one system to another.

FURPS:

FURPS is an acronym representing a model for classifying software quality attributes (functional and non-functional requirements):

- **F**unctionality - Capability (Size & Generality of Feature Set), Reusability (Compatibility, Interoperability, Portability), Security (Safety & Exploitability)
- **U**sability (UX) - Human Factors, Aesthetics, Consistency, Documentation, Responsiveness
- **R**eliability - Availability (Failure Frequency (Robustness/Durability/Resilience), Failure Extent & Time-Length

(Recoverability/Survivability)), Predictability (Stability), Accuracy (Frequency/Severity of Error)

- **P**erformance - Speed, Efficiency, Resource Consumption (power, ram, cache, etc.), Throughput, Capacity, Scalability
- **S**upportability (Serviceability, Maintainability, Sustainability, Repair Speed) - Testability, Flexibility (Modifiability, Configurability, Adaptability, Extensibility, Modularity), Installability, Localizability

The model, developed at Hewlett-Packard was first publicly elaborated by Grady and Caswell. *FURPS+* is now widely used in the software industry. The *+* was later added to the model after various campaigns at HP to extend the acronym to emphasize various attributes.

9126 Quality Factors

ISO 9126 is an international standard for the evaluation of softwarequality.

- **Functionality** - A set of attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs.
  - Suitability
  - Accuracy
  - Interoperability
  - Compliance
  - Security

- **Reliability** - A set of attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time.
  - Maturity
  - Recoverability

- **Usability**- A set of attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users.
  - Learnability
  - Understandability
  - Operability

- **Efficiency**- A set of attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions.
    - Time Behavior
    - Resource Behavior

- **Maintainability**- A set of attributes that bear on the effort needed to make specified modifications.
    - Stability
    - Analyzability
    - Changeability
    - Testability

- **Portability**- A set of attributes that bear on the ability of software to be transferred from one environment to another.
    - Installability
    - Replaceability
    - Adaptability

Software Quality Metrics:

Software metrics can be classified into three categories: product metrics, process metrics, and project metrics. Product metrics describe the characteristics of the product such as size, complexity, design features, performance, and quality level. Process metrics can be used to improve software development and maintenance. Examples include the effectiveness of defect removal during development, the pattern of testing defect arrival, and the response time of the fix process. Project metrics describe the project characteristics and execution. Examples include the number of software developers, the staffing pattern over the life cycle of the software, cost, schedule, and productivity. Some metrics belong to multiple categories. For example, the inprocess quality metrics of a project are both process metrics and project metrics. Software quality metrics are a subset of software metrics that focus on the quality aspects of the product, process, and project. In general, software quality metrics are more closely associated with process and product metrics than with project metrics. Nonetheless, the project parameters such as the number of developers and their skill levels, the schedule, the size, and the organization structure certainly affect the quality of the product. Software quality metrics can be divided further into end-product

quality metrics and in-process quality metrics. The essence of software quality engineering is to investigate the relationships among in-process metrics, project characteristics, and end-product quality, and, based on the findings, to engineer improvements in both process and product quality. Moreover, we should view quality from the entire software life-cycle perspective and, in this regard, we should include metrics that measure the quality level of the maintenance process as another category of software quality metrics. In this chapter we discuss several metrics in each of three groups of software quality metrics: product quality, in-process quality, and maintenance quality. In the last sections we also describe the key metrics used by several major software developers and discuss software metrics data collection.

Product Quality Metrics:

The de facto definition of software quality consists of two levels: intrinsic product quality and customer satisfaction.
The metrics we discuss here cover both levels:
* Mean time to failure
* Defect density
* Customer problems
* Customer satisfaction.

Intrinsic product quality is usually measured by the number of “bugs” (functional defects) in the software or by how long the software can run before encountering a “crash.” In operational definitions, the two metrics are defect density (rate) and mean time to failure (MTTF). The MTTF metric is most often used with safetycritical systems such as the airline traffic control systems, avionics, and weapons. For instance, the U.S. government mandates that its air traffic control system cannot be unavailable for more than three seconds per year. In civilian airliners, the probability of certain catastrophic failures must be no worse than 10−9 per hour (Littlewood and Strigini, 1992). The defect density metric, in contrast, is used in many commercial software systems. The two metrics are correlated but are different enough to merit close attention. First, one measures the time between failures, the other measures the defects relative to the software size (lines of code, function points, etc.). Second, although it is difficult to separate defects and failures in actual measurements and data tracking, failures and defects (or faults) have different meanings. According to the IEEE/ American National Standards Institute (ANSI) standard (982.2):
* An error is a human mistake that results in incorrect software.

* The resulting fault is an accidental condition that causes a unit of the system to fail to function as required.
* A defect is an anomaly in a product.
* A failure occurs when a functional unit of a software-related system can no longer perform its required function or cannot perform it within specified limits.

From these definitions, the difference between a fault and a defect is unclear. For practical purposes, there is no difference between the two terms. Indeed, in many development organizations the two terms are used synonymously. In this book we also use the two terms interchangeably. Simply put, when an error occurs during the development process, a fault or a defect is injected in the software. In operational mode, failures are caused by faults or defects, or failures are materializations of faults. Sometimes a fault causes more than one failure situation and, on the other hand, some faults do not materialize until the software has been executed for a long time with some particular scenarios. Therefore, defect and failure do not have a one-to-one correspondence. Third, the defects that cause higher failure rates are usually discovered and removed early. The probability of failure associated with a latent defect is called its size, or "bug size." For special-purpose software systems such as the air traffic control systems or the space shuttle control systems, the operations profile and scenarios are better defined and, therefore, the time to failure metric is appropriate. For general-purpose computer systems or commercial-use software, for which there is no typical user profile of the software, the MTTF metric is more difficult to implement and may not be representative of all customers. Fourth, gathering data about time between failures is very expensive. It requires recording the occurrence time of each software failure. It is sometimes quite difficult to record the time for all the failures observed during testing or operation. To be useful, time between failures data also requires a high degree of accuracy. This is perhaps the reason the MTTF metric is not widely used by commercial developers. Finally, the defect rate metric (or the volume of defects) has another appeal to commercial software development organizations. The defect rate of a product or the expected number of defects over a certain time period is important for cost and resource estimates of the maintenance phase of the software life cycle. Regardless of their differences and similarities, MTTF and defect density are the two key metrics for intrinsic product quality. Accordingly, there are two main types of software reliability growth models—the time between failures models and the defect count (defect rate) models. We discuss the two types of models and provide several examples of each type.

The Defect Density Metric:

Although seemingly straightforward, comparing the defect rates of software products involves many issues. In this section we try to articulate the major points. To define a rate, we first have to operationalize the numerator and the denominator, and specify the time frame. As discussed in Chapter 3, the general concept of defect rate is the number of defects over the opportunities for error (OFE) during a specific time frame. We have just discussed the definitions of software defect and failure. Because failures are defects materialized, we can use the number of unique causes of observed failures to approximate the number of defects in the software. The denominator is the size of the software, usually expressed in thousand lines of code (KLOC) or in the number of function points. In terms of time frames, various operational definitions are used for the life of product (LOP), ranging from one year to many years after the software product's release to the general market. In our experience with operating systems, usually more than 95% of the defects are found within four years of the software's release. For application software, most defects are normally found within two years of its release.

Lines of Code:

The lines of code (LOC) metric is anything but simple. The major problem comes from the ambiguity of the operational definition, the actual counting. In the early days of Assembler programming, in which one physical line was the same as one instruction, the LOC definition was clear. With the availability of high-level languages the one-to-one correspondence broke down. Differences between physical lines and instruction statements (or logical lines of code) and differences among languages contribute to the huge variations in counting LOCs. Even within the same language, the methods and algorithms used by different counting tools can cause significant differences in the final counts. Jones (1986) describes several variations:

* Count only executable lines.
* Count executable lines plus data definitions.
* Count executable lines, data definitions, and comments.
* Count executable lines, data definitions, comments, and job control language.
* Count lines as physical lines on an input screen.
* Count lines as terminated by logical delimiters.

To illustrate the variations in LOC count practices, let us look at a few examples by authors of software metrics. In Boehm's well-known book Software Engineering Economics (1981), the LOC counting method counts lines as physical lines and includes executable lines, data definitions, and comments. In Software Engineering Metrics and Models by Conte et al. (1986), LOC is defined as follows: A line of code is any line of program text that is not a comment or blank line, regardless of the number of statements or fragments of statements on the line. This specifically includes all lines containing program headers, declarations, and executable and non-executable statements. (p. 35) Thus their method is to count physical lines including prologues and data definitions (declarations) but not comments. In Programming Productivity by Jones(1986), the source instruction (or logical lines of code) method is used. The method used by IBM Rochester is also to count source instructions including executable lines and data definitions but excluding comments and program prologues.

The resultant differences in program size between counting physical lines and counting instruction statements are difficult to assess. It is not even known which method will result in a larger number. In some languages such as BASIC, PASCAL, and C, several instruction statements can be entered on one physical line. On the other hand, instruction statements and data declarations might span several physical lines, especially when the programming style aims for easy maintenance, which is not necessarily done by the original code owner. Languages that have a fixed column format such as FORTRAN may have the physical-lines-to-source-instructions ratio closest to one. According to Jones (1992), the difference between counts of physical lines and counts including instruction statements can be as large as 500%; and the average difference is about 200%, with logical statements outnumbering physical lines. In contrast, for COBOL the difference is about 200% in the opposite direction, with physical lines outnumbering instruction statements.

There are strengths and weaknesses of physical LOC and logical LOC (Jones, 2000). In general, logical statements are a somewhat more rational choice for quality data. When any data on size of program products and their quality are presented, the method for LOC counting should be described. At the minimum, in any publication of quality when LOC data is involved, the author should state whether the LOC counting method is based on physical LOC or logical LOC.

Furthermore, as discussed in Chapter 3, some companies may use the straight LOC count (whatever LOC counting method is used) as the denominator for calculating defect rate, whereas others may use the normalized count (normalized to Assembler-equivalent LOC based on some

conversion ratios) for the denominator. Therefore, industrywide standards should include the conversion ratios from highlevel language to Assembler. So far, very little research on this topic has been published. The conversion ratios published by Jones (1986) are the most well known in the industry. As more and more high-level languages become available for software development, more research will be needed in this area.

When straight LOC count data is used, size and defect rate comparisons across languages are often invalid. Extreme caution should be exercised when comparing the defect rates of two products if the operational definitions (counting) of LOC, defects, and time frame are not identical. Indeed, we do not recommend such comparisons. We recommend comparison against one's own history for the sake of measuring improvement over time.

Note: The LOC discussions in this section are in the context of defect rate calculation. For productivity studies, the problems with using LOC are more severe. A basic problem is that the amount of LOC in a softare program is negatively correlated with design efficiency. The purpose of software is to provide certain functionality for solving some specific problems or to perform certain tasks. Efficient design provides the functionality with lower implementation effort and fewer LOCs. Therefore, using LOC data to measure software productivity is like using the weight of an airplane to measure its speed and capability. In addition to the level of languages issue, LOC data do not reflect noncoding work such as the creation of requirements, specifications, and user manuals. The LOC results are so misleading in productivity studies that Jones states "using lines of code for productivity studies involving multiple languages and full life cycle activities should be viewed as professional malpractice" (2000, p. 72). For detailed discussions of LOC and function point metrics, see Jones's work (1986, 1992, 1994, 1997, 2000).

When a software product is released to the market for the first time, and when a certain LOC count method is specified, it is relatively easy to state its quality level (projected or actual). For example, statements such as the following can be made: "This product has a total of 50 KLOC; the latent defect rate for this product during the next four years is 2.0 defects per KLOC." However, when enhancements are made and subsequent versions of the product are released, the situation becomes more complicated. One needs to measure the quality of the entire product as well as the portion of the product that is new. The latter is the measurement of true development quality—the defect rate of the new and changed code. Although the defect rate for the entire product will improve from release to release due to aging, the defect rate of the new and changed code will not improve unless there is real

improvement in the development process. To calculate defect rate for the new and changed code, the following must be available:

* LOC count: The entire software product as well as the new and changed code of the release must be available.

* Defect tracking: Defects must be tracked to the release origin—the portion of the code that contains the defects and at what release the portion was added, changed, or enhanced. When calculating the defect rate of the entire product, all defects are used; when calculating the defect rate for the new and changed code, only defects of the release origin of the new and changed code are included.

These tasks are enabled by the practice of change flagging. Specifically, when a new function is added or an enhancement is made to an existing function, the new and changed lines of code are flagged with a specific identification (ID) number through the use of comments. The ID is linked to the requirements number, which is usually described briefly in the module's prologue. Therefore, any changes in the program modules can be linked to a certain requirement. This linkage procedure is part of the software configuration management mechanism and is usually practiced by organizations that have an established process. If the change-flagging IDs and requirements IDs are further linked to the release number of the product, the LOC counting tools can use the linkages to count the new and changed code in new releases. The changeflagging practice is also important to the developers who deal with problem determination and maintenance. When a defect is reported and the fault zone determined, the developer can determine in which function or enhancement pertaining to what requirements at what release origin the defect was injected.

The new and changed LOC counts can also be obtained via the delta-library method. By comparing program modules in the original library with the new versions in the current release library, the LOC count tools can determine the amount of new and changed code for the new release. This method does not involve the change-flagging method. However, change flagging remains very important for maintenance. In many software development environments, tools for automatic change flagging are also available.

Example: Lines of Code Defect Rates

At IBM Rochester, lines of code data is based on instruction statements (logical LOC) and includes executable code and data definitions but excludes comments. LOC counts are obtained for the total product and for the new and changed code of the new release. Because the LOC count is based on source instructions, the two size metrics are called shipped source instructions (SSI)

and new and changed source instructions (CSI), respectively. The relationship between the SSI count and the CSI count can be expressed with the following formula:

SSI (current release) = SSI (previous release) + CSI (new and changed code instructions for current release) − deleted code (usually very small) − changed code (to avoid double count in both SSI and CSI)

Defects after the release of the product are tracked. Defects can be field defects, which are found by customers, or internal defects, which are found internally. The several postrelease defect rate metrics per thousand SSI (KSSI) or per thousand CSI (KCSI) are:

(1) Total defects per KSSI (a measure of code quality of the total product)
(2) Field defects per KSSI (a measure of defect rate in the field)
(3) Release-origin defects (field and internal) per KCSI (a measure of development quality) (4) Release-origin field defects per KCSI (a measure of development quality per defects found by customers)

Metric (1) measures the total release code quality, and metric (3) measures the quality of the new and changed code. For the initial release where the entire product is new, the two metrics are the same. Thereafter, metric (1) is affected by aging and the improvement (or deterioration) of metric (3). Metrics (1) and (3) are process measures; their field counterparts, metrics (2) and (4) represent the customer's perspective. Given an estimated defect rate (KCSI or KSSI), software developers can minimize the impact to customers by finding and fixing the defects before customers encounter them.

Customer's Perspective:

The defect rate metrics measure code quality per unit. It is useful to drive quality improvement from the development team's point of view. Good practice in software quality engineering, however, also needs to consider the customer's perspective. Assume that we are to set the defect rate goal for release-to-release improvement of one product. From the customer's point of view, the defect rate is not as relevant as the total number of defects that might affect their business. Therefore, a good defect rate target should lead to a release-to-release reduction in the total number of defects, regardless of size. If a new release is larger than its predecessors, it means the defect rate

goal for the new and changed code has to be significantly better than that of the previous release in order to reduce the total number of defects.

Consider the following hypothetical example:

Initial Release of Product Y
KCSI = KSSI = 50 KLOC
Defects/KCSI = 2.0
Total number of defects = 2.0 × 50 = 100

Second Release
KCSI = 20
KSSI = 50 + 20 (new and changed lines of code) − 4 (assuming 20% are changed lines of code) = 66 Defect/KCSI = 1.8 (assuming 10% improvement over the first release)
Total number of additional defects = 1.8 × 20 = 36

Third Release
KCSI = 30
KSSI = 66 + 30 (new and changed lines of code) − 6 (assuming the same % (20%) of changed lines of code) = 90
Targeted number of additional defects (no more than previous release) = 36
Defect rate target for the new and changed lines of code: 36/30 = 1.2 defects/KCSI or lower

From the initial release to the second release the defect rate improved by 10%. However, customers experienced a 64% reduction [(100 − 36)/100] in the number of defects because the second release is smaller. The size factor works against the third release because it is much larger than the second release. Its defect rate has to be onethird (1.2/1.8) better than that of the second release for the number of new defects not to exceed that of the second release. Of course, sometimes the difference between the two defect rate targets is very large and the new defect rate target is deemed not achievable. In those situations, other actions should be planned to improve the quality of the base code or to reduce the volume of postrelease field defects (i.e., by finding them internally).

Function Points:

Counting lines of code is but one way to measure size. Another one is the function point. Both are surrogate indicators of the opportunities for error (OFE) in the defect density metrics. In recent years the function point has been gaining acceptance in application development in terms of both productivity (e.g., function points per person-year) and quality (e.g., defects per function point). In this section we provide a concise summary of the subject.

A function can be defined as a collection of executable statements that performs a certain task, together with declarations of the formal parameters and local variables manipulated by those statements (Conte et al., 1986). The ultimate measure of software productivity is the number of functions a development team can produce given a certain amount of resource, regardless of the size of the software in lines of code. The defect rate metric, ideally, is indexed to the number of functions a software provides. If defects per unit of functions is low, then the software should have better quality even though the defects per KLOC value could be higher—when the functions were implemented by fewer lines of code. However, measuring functions is theoretically promising but realistically very difficult.

The function point metric, originated by Albrecht and his colleagues at IBM in the mid-1970s, however, is something of a misnomer because the technique does not measure functions explicitly (Albrecht, 1979). It does address some of the problems associated with LOC counts in size and productivity measures, especially the differences in LOC counts that result because different levels of languages are used. It is a weighted total of five major components that comprise an application:

* Number of external inputs (e.g., transaction types) × 4
* Number of external outputs (e.g., report types) × 5
* Number of logical internal files (files as the user might conceive them, not physical files) × 10
* Number of external interface files (files accessed by the application but not maintained by it) × 7
* Number of external inquiries (types of online inquiries supported) × 4

These are the average weighting factors. There are also low and high weighting factors, depending on the complexity assessment of the application in terms of the five components (Kemerer and Porter, 1992; Sprouls, 1990):

* External input: low complexity, 3; high complexity, 6

* External output: low complexity, 4; high complexity, 7
* Logical internal file: low complexity, 7; high complexity, 15
* External interface file: low complexity, 5; high complexity, 10
* External inquiry: low complexity, 3; high complexity, 6

The complexity classification of each component is based on a set of standards that define complexity in terms of objective guidelines. For instance, for the external output component, if the number of data element types is 20 or more and the number of file types referenced is 2 or more, then complexity is high. If the number of data element types is 5 or fewer and the number of file types referenced is 2 or 3, then complexity is low.

With the weighting factors, the first step is to calculate the function counts (FCs) based on the following formula:

$$FC = \sum_{i=1}^{5} \sum_{j=1}^{3} w_{ij} \times x_{ij}$$

wherew_{ij} are the weighting factors of the five components by complexity level (low, average, high) and x_{ij} are the numbers of each component in the application. The second step involves a scale from 0 to 5 to assess the impact of 14 general system characteristics in terms of their likely effect on the application. The 14 characteristics are:

1. Data communications
2. Distributed functions
3. Performance
4. Heavily used configuration
5. Transaction rate
6. Online data entry
7. End-user efficiency
8. Online update
9. Complex processing
10. Reusability
11. Installation ease
12. Operational ease
13. Multiple sites
14. Facilitation of change

The scores (ranging from 0 to 5) for these characteristics are then summed, based on
the following formula, to arrive at the value adjustment factor (VAF)

$$\mathrm{VAF} = 0.65 + 0.01 \sum_{i=1}^{14} c_i$$

Where C_i is the score for general system characteristic i. Finally, the number of
function points is obtained by multiplying function counts and the value adjustment
factor:

$$\mathrm{FP} = \mathrm{FC} \times \mathrm{VAF}$$

This equation is a simplified description of the calculation of function points. One should consult the fully documented methods, such as the International Function Point User's Group Standard (IFPUG, 1999), for a complete treatment.

Over the years the function point metric has gained acceptance as a key productivity measure in the application world. In 1986 the IFPUG was established. The IFPUG counting practices committee is the de facto standards organization for function point counting methods (Jones, 1992, 2000). Classes and seminars on function points counting and applications are offered frequently by consulting firms and at software conferences. In application contract work, the function point is often used
to measure the amount of work, and quality is expressed as defects per function point. In systems and real-time software, however, the function point has been slow to gain acceptance. This is perhaps due to the incorrect impression that function points work only for information systems (Jones, 2000), the inertia of the LOC related practices, and the effort required for function points counting. Intriguingly, similar observations can be made about function point use in academic research.

There are also issues related to the function point metric. Fundamentally, the meaning of function point and the derivation algorithm and its rationale may need more research and more theoretical groundwork.

There are also many variations in counting function points in the industry and several major methods other than the IFPUG standard. In 1983, Symons presented a function point variant that he termed the Mark II function point (Symons, 1991). According to Jones (2000), the Mark II function point is now widely used in the United Kingdom and to a lesser degree in Hong Kong and Canada. Some of the minor function point variants include feature points, 3D function points, and full function points. In all, based on the comprehensive software benchmark work by Jones (2000), the set of function point variants now include at least 25 functional metrics. Function point counting can be timeconsuming and expensive, and accurate counting requires certified function point specialists. Nonetheless, function point metrics are apparently more robust than LOC-based data with regard to comparisons across organizations, especially studies involving multiple languages and those for productivity evaluation.

Example: Function Point Defect Rates

In 2000, based on a large body of empirical studies, Jones published the book *Software Assessments, Benchmarks, and Best Practices.* All metrics used throughout the book are based on function points. According to his study (1997), the average number of software defects in the U.S. is approximately 5 per function point during the entire software life cycle. This number represents the total number of defects found and measured from early software requirements throughout the life cycle of the software, including the defects reported by users in the field. Jones also estimates the defect removal efficiency of software organizations by level of the capability maturity model (CMM) developed by the Software Engineering Institute (SEI). By applying the defect removal efficiency to the overall defect rate per function point, the following defect rates for the delivered software were estimated. The time frames for these defect rates were not specified, but it appears that these defect rates are for the maintenance life of the software. The estimated defect rates per function

point are
as follows:

SEI CMM Level 1: 0.75
SEI CMM Level 2: 0.44
SEI CMM Level 3: 0.27
SEI CMM Level 4: 0.14
SEI CMM Level 5: 0.05

PROCESS ASSESSMENT AND IMPROVEMENT

The existence of a software process is no guarantee that software will be delivered on time, that it will meet the customer's needs, or that it will exhibit the technical characteristics that will lead to long-term quality characteristics (Chapters 14 and 16). Process patterns must be coupled with solid software engineering practice (Part 2 of this book). In addition, the process itself can be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering. A number of different approaches to software process assessment and improvement have been proposed over the past few decades:

Standard CMMI Assessment Method for Process Improvement(SCAMPI)— provides a five-step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting, and learning. The SCAMPI method uses the SEI CMMI as the basis for assessment.

CMM-Based Appraisal for Internal Process Improvement (CBA IPI) — provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment.

SPICE (ISO/IEC15504) — a standard that defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process.

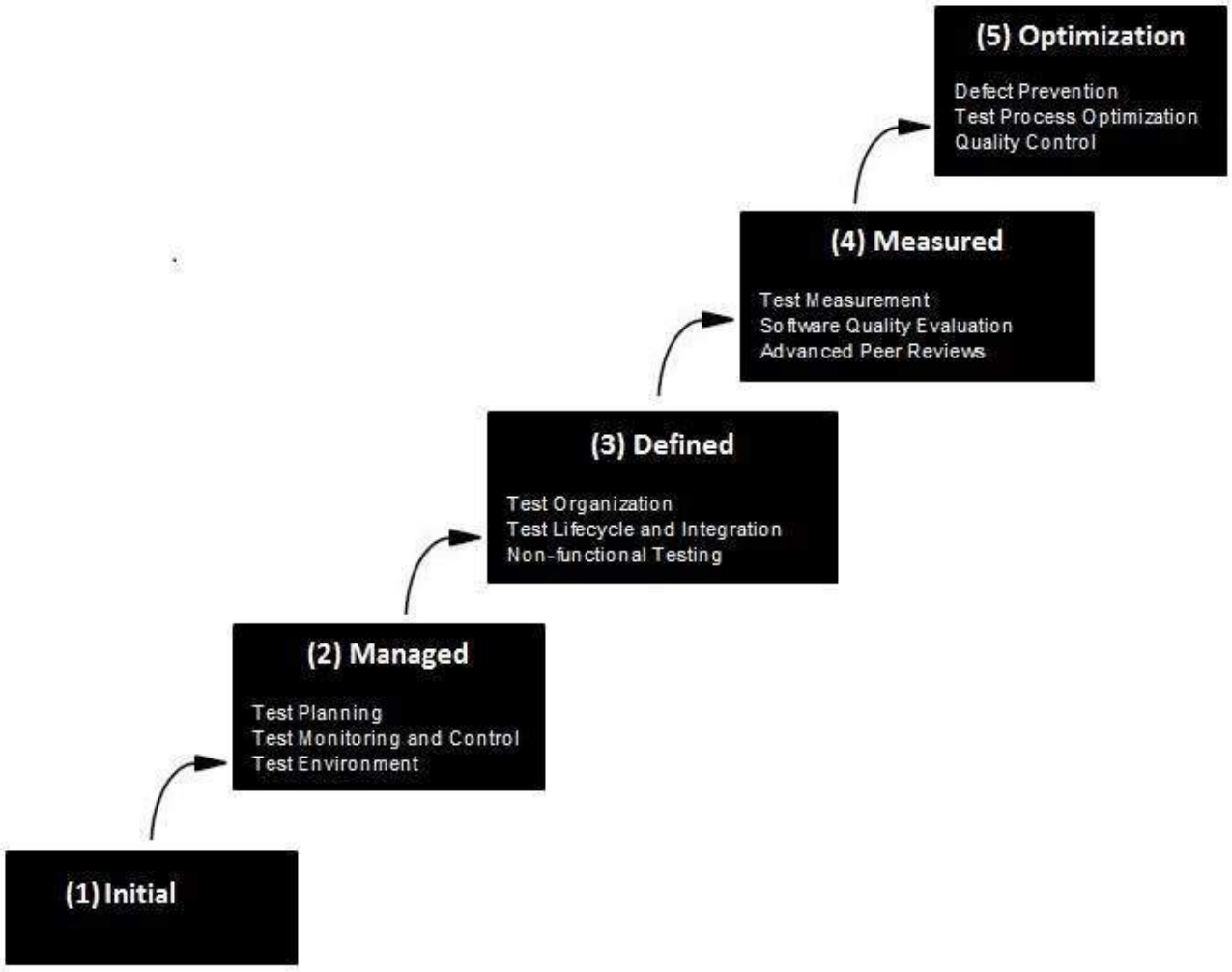
ISO 9001:2000 for Software — a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies.

What is Capability Maturity Model?

The Software Engineering Institute (SEI) Capability Maturity Model (CMM) specifies an increasing series of levels of a software development organization.

The higher the level, the better the software development process, hence reaching each level is an expensive and time-consuming process.

Levels of CMM



- **Level One :Initial** - The software process is characterized as inconsistent, and occasionally even chaotic. Defined processes and standard practices that exist are abandoned during a crisis. Success of the organization majorly depends on an individual effort, talent, and heroics. The heroes eventually move on to other organizations taking their wealth of knowledge or lessons learnt with them.

- **Level Two: Repeatable** - This level of Software Development Organization has a basic and consistent project management processes to track cost, schedule, and functionality. The process is in place to repeat the earlier successes on projects with similar applications. Program management is a key characteristic of a level two organization.

- **Level Three: Defined** - The software process for both management and engineering activities are documented, standardized, and integrated into a standard software process for the entire organization and all projects across the organization use an approved, tailored version of the organization's standard software process for developing, testing and maintaining the application.

- **Level Four: Managed** - Management can effectively control the software development effort using precise measurements. At this level, organization set a quantitative quality goal for both software process and software maintenance. At this maturity level, the performance of processes is controlled using statistical and other quantitative techniques, and is quantitatively predictable.

- **Level Five: Optimizing** - The Key characteristic of this level is focusing on continually improving process performance through both incremental and innovative technological improvements. At this level, changes to the process are to improve the process performance and at the same time maintaining statistical probability to achieve the established quantitative process-improvement objectives.

Why should we assess the testing maturity?

There is no consistency within their organization as to the health and professionalism of the testing process.

An assessment of the testing process using a testing maturity model will

- document the current level.
- highlight the variances between the imagined level and the actual level.
- Provide a road map for making the necessary process improvements.

Testing maturity models:

- Most of the testing maturity models were developed around 1996.
    - Testability Maturity Model.
    - Software Testing Maturity Model.
    - Test Process Improvement.
    - Test Organization Maturity.
    - Testing Assessment Program.
    - Proposed Evaluation and Test SW-CMM Key Process Area.

- None has found much acceptance because of
  - little documentation on the model.
  - theoretical style.

What is the Software Testing Maturity Model?

- SW-TMM is a testing process improvement tool that can be used.
  - either in conjunction with the SW-CMM.
  - or as a stand-alone tool.
- Developed by Dr.Ilene Burnstein of the Illinois Institute of Technology and her associates.
  - publish several articles in professional magazines.
  - institute plan: to release a book on the SW-TMM in 2002.

The five levels of SW-TMM:

Level 1: Initial.
Level 2: Phase Definition.
Level 3: Integration.
Level 4: Management and Measurement.
Level 5: Optimization/Defect Prevention and Quality Control.

Software Testing Maturity Model Level 1

- A chaotic process.
- Not distinguished from debugging and ill defined.
- The tests are developed ad hoc after coding is complete.
- Usually lack a trained professional testing staff and testing tools.
- The objective of testing is to show that the system and software work.

Software Testing Maturity Model Level 2

- Identify testing as a separate function from debugging.
- Testing becomes a defined phase following coding.
- Standardize their process to the point where basic testing techniques and methods are in place.
- The objective of testing is to show that the system and software meets specifications.

Software Testing Maturity Model Level 3

- Integrate testing into the entire life cycle.
- Establish a formal testing organization.
    - establishes formal testing technical trainings.
    - controls and monitors the testing process.
    - begins to consider using automated test tools.
- The objective of testing is based on system requirements.
- Major milestone reached at this level: management recognizes testing as a professional activity.

Software Testing Maturity Model Level 4

- Testing is a measured and quantified process.
- Development products are now tested for quality attributes such as Reliability, Usability, and Maintainability.
- Test cases are collected and recorded in a test database for reuse and regression testing.
- Defects found during testing are now logged, given a severity level, and assigned a priority for correction.

Software Testing Maturity Model Level 5

- Testing is institutionalized within the organization.
- Testing process is well defined and managed.
- Testing costs and effectiveness are monitored.
- Automated tools are a primary part of the testing process.
- There is an established procedure for selecting and evaluating testing tools.

Why do we need to use SW-TMM?

- easy to understand and use.
- provide a methodology to baseline the current test process maturity.
- designed to guide organization.

    - selecting process improvement strategies.
    - identifying critical issues to test process maturity.
- provide a road map for continuous test process improvement.
- provide a method for measuring progress.
- allow organizations to perform their own assessment.
- Organizations that are using SW-CMM.
    - SW-TMM fulfills the design objective of being an excellent companion to SW-CMM.
    - SW-TMM is just another assessment tool and easily incorporated into the software process assessment.
- Organizations that are not using SW-CMM.
    - provide an unbiased assessment of the current testing process.
    - provide a road map for incremental improvements.
    - save testing cost as the testing process moves up the maturity levels.

How do we perform the SW-TMM assessment?

- Prepare for the assessment.
- Conduct the assessment.
- Document the findings.
- Analyze the findings.
- Develop the action plan.
- Write the final report.
- Implement the improvements.

- Prepare for the assessment.
    - choose team leader and members.
    - choose evaluation tools (e.g. questionnaire).
    - training and briefing.
- Conduct the assessment.
    - organization being evaluated gives a presentation.
        - what is important from their perspective.
        - the organization is an integral part of the assessment.
    - conduct interviews with individual.
    - review all testing documentation and procedures to determine the actual testing process currently being used.
- Document the findings.

    - document the organization's current testing process.
    - compile and summarize the questionnaire data.
    - document the interview information.
- Analyze the findings.
    - document the current maturity level.
    - document any areas of disagreement highlighted during the evaluation.
    - identify areas for improvement.
    - prioritize a list of recommended improvement goals.
    - include anticipated benefits resulting from implementation.
- Develop the action plan.
    - Specific actions.
    - Required resources.
    - Schedule for implementation.
    - Cost/Benefit analysis.
- Write the final report.
    - a support documentation for the management briefing.
- Implement the improvements.
    - best to implement the improvements either in a pilot project or in phases.
        - track progress and achievements prior to expanding organization wide.
    - also good in a limited application.
        - easier to fine-tune the new process prior to expanded implementation.

SCSX1038_Software Quality Assurance and Testing
Unit V

INTRODUCTION

Competition to provide specialized products and services results in breakthroughs as well as long-term growth and change. Quality assurance verifies that any customer offering, regardless if it is new or evolved, is produced and offered with the best possible materials, in the most comprehensive way, with the highest standards. The goal to exceed customer expectations in a measurable and accountable process is provided by quality assurance.

DEFINITION OF QUALITY, QA, SQA

Quality:

The term 'quality' is often used in a vague, blurred way. If someone talks about 'working on quality', they may simply mean activities designed to improve the organisation and its services. Quality is essentially about learning what you are doing well and doing it better. It also means finding out what you may need to change to make sure you meet the needs of your service users. Quality is about:

knowing what you want to do and how you want to do it

learning from what you do

using what you learn to develop your organisation and its services

seeking to achieve continuous improvement

Satisfying your stakeholders - those different people and groups with an interest in your organisation.

Quality assurance is the process of verifying or determining whether products or services meet or exceed customer expectations. Quality assurance is a process-driven approach with specific steps to help define and attain goals. This process considers design, development, production, and service.

The four quality assurance steps within the PDCA model stand for -

Plan: Establish objectives and processes required to deliver the desired results.

Do: Implement the process developed.

Check: Monitor and evaluate the implemented process by testing the results against the predetermined objectives

Act: Apply actions necessary for improvement if the results require changes.

PDCA is an effective method for monitoring quality assurance because it analyzes existing conditions and methods used to provide the product or service customers. The goal is to ensure that excellence is inherent in every component of the process. Quality assurance also helps determine whether the steps used to provide the product or service are appropriate for the time and conditions. In addition, if the PDCA cycle is repeated throughout the lifetime of the product or service, it helps improve internal company efficiency.

Quality assurance demands a degree of detail in order to be fully implemented at every step. *Planning*, for example, could include investigation into the quality of the raw materials used in manufacturing, the actual assembly, or the inspection processes used. The *Checking* step could include customer feedback, surveys, or other marketing vehicles to determine if customer needs are being exceeded and why they are or are not. *Acting* could mean a total revision in the manufacturing process in order to correct a technical or cosmetic flaw.

Quality Control : The terms “quality assurance” and “quality control” are often usedinterchangeably to refer to ways of ensuring the quality of a service or product. The terms, however, have different meanings.

Assurance: The act of giving confidence, the state of being certain or the act ofmaking certain.

Quality assurance: The planned and systematic activities implemented in aquality system so that quality requirements for a product or service will be fulfilled.

Control: An evaluation to indicate needed corrective responses; the act ofguiding a process in which variability is attributable to a constant system of

chance causes. Quality control:

Software Quality Assurance

Business software is never built overnight. It takes a lot of planning, consultation and testing to be able to come up with an initial version of the application. If a business hurries up the development of a certain application, they would end up spending more in addressing the problem the application brought than they have earned.

This could even be more frustrating when the software being developed is set to be sold to customers or for public use. But a bug free program is not the only goal of a company. If they follow that concept they would end up developing a very simple program without any special features and would frustrate their users and hinder their productivity.

That is where the concept of software quality assurance comes in. For most developers software quality assurance offers a comprehensive solution in ensuring that the application they are working on is functioning and lives up to expectations. Quality assurance could be found everywhere. Products in the market also undergo quality assurance to make sure the goods are made according to expectations. However, once the products are sent out to the market, they are already done and it is up to the customers on how to be satisfied with the product. On the other hand software quality assurance takes on a different side. Aside from delivering good products, software quality assurance or simply known as SQA follows a set protocols that are universally recognized. By using these protocols users are assured the application they are using are built according to standards and every stage of software development follows the same standards.

Software Quality Assurance was created with the following objectives:
Small to Zero Defects After Installation – One of the biggest goals of SQA is to prevent any possible defects when the output is made. Developers and engineers have to use universally approved steps to ensure that the program was built up to expectations but also to prevent errors in the system. Although some standards allow as much as .04 errors in the system, zero-error is still the system's target. When there's zero-error, the program is more likely to have zero crash scenarios. The ability to handle stress of a program is different from the errors it has but crashes usually comes from defects so prevention of defects will most likely yield a continuously working application.

Customer Satisfaction – Everything else will be just nothing if the customers don'tlike what they see. Part of SQA is to ensure that software development was made according to their needs, wants and exceeding their expectations. Even if the bugs and errors are minimized by the system, customer satisfaction is more important and should be emphasized.

Well Structured – SQA takes care of the stages of application construction. Anyonecould be easily build an application and launch it in their environment without any glitches. However, not everyone could easily build an application that could be understood well. SQA ensures that each application are build in an understandable manner. Their applications could easily be transferred from one developer to another.

QUALITY FACTORS

The various factors, which influence the software, are termed as software factors. They can be broadly divided into two categories. The classification is done on the basis of measurability. The first category of the factors is of those that can be measured directly such as number of logical errors and the second category clubs those factors which can be measured only indirectly for example maintainability but the each of the factors are to be measured to check for the content and the quality control. Few factors of quality are available and they are mentioned below.

Correctness - extent to which a program satisfies its specification and fulfills the client's objective.

Reliability - extent to which a program is supposed to perform its function with the required precision.

Efficiency - amount of computing and code required by a program to perform its function.

Integrity - extent to which access to software and data is denied to unauthorized users.

Usability- labor required to understand, operate, prepare input and interpret output of a program

Maintainability- effort required to locate and fix an error in a program.

Flexibility :effort needed to modify an operational program.

Testability- effort required to test the programs for their functionality.

Portability- effort required to run the program from one platform to other or to different hardware.

Reusability- extent to which the program or it's parts can be used as building blocks or as prototypes for other programs.

Interoperability- effort required to couple one system to another.

Now as you consider the above-mentioned factors it becomes very obvious that the measurements of all of them to some discrete value are quite an impossible task. Therefore, another method was evolved to measure out the quality. A set of matrices is defined and is used to develop expressions for each of the factors as per the following expression

Auditability- ease with which the conformance to standards can be verified.

Accuracy- precision of computations and control

Completeness- degree to which full implementation of functionality required has been achieved.

Conciseness- program's compactness in terms of lines of code.

Consistency- use of uniform design and documentation techniques throughout the software development.

Data commonality- use of standard data structures and types throughout the program.

Error tolerance – damage done when program encounters an error. Execution efficiency- run-time performance of a program.

Expandability- degree to which one can extend architectural, data and procedural design.

Hardware independence- degree to which the software is de-coupled from its operating hardware.

Instrumentation- degree to which the program monitors its own operation and identifies errors that do occur.

Modularity- functional independence of program components. Operability- eases of programs operation.

Security- control and protection of programs and database from the unauthorized users.

Self-documentation- degree to which the source code provides meaningful documentation.

Simplicity- degree to which a program is understandable without much difficulty.

Software system independence- degree to which program is independent of nonstandard programming language features, operating system characteristics and other environment constraints.

Traceability- ability to trace a design representation or actual program component back to initial objectives.

Training- degree to which the software is user-friendly to new users.

There are various 'checklists' for software quality. One of them was given by Hewlett-Packard that has been given the acronym FURPS – for Functionality, Usability, Reliability, Performance and Supportability. Functionality is measured via the evaluation of the feature set and the program capabilities, the generality of the functions that are derived and the overall security of the system.

Considering human factors, overall aesthetics, consistency and documentation assesses usability. Reliability is figured out by evaluating the frequency and severity of failure, the accuracy of output results, the mean time between failure (MTBF), the ability to recover from failure and the predictability of the program.

Performance is measured by measuring processing speed, response time, resource consumption, throughput and efficiency. Supportability combines the ability to extend the program, adaptability, serviceability or in other terms maintainability and also testability, compatibility, configurability and the ease with which a system can be installed.

SOFTWARE QUALITY MATRICS

We best manage what we can measure. Measurement enables the Organization to improve the software process; assist in planning, tracking and controlling the software project and assess the quality of the software thus produced. It is the measure of such specific attributes of the process, project and product that are used to compute the software metrics. Metrics are analyzed and they provide a dashboard to the management on the overall health of the process, project and

product. The kind of metrics employed generally account for whether the quality requirements have been achieved or are likely to be achieved during the software development process. As a quality assurance process, a metric is needed to be revalidated every time it is used. Two leading firms namely, IBM and Hewlett-Packard have placed a great deal of importance on software quality. The IBM measures the user satisfaction and software acceptability in eight dimensions which are capability or functionality, usability, performance, reliability, ability to be installed, maintainability, documentation, and availability. For the Software Quality Metrics the Hewlett-Packard normally follows the five Juran quality parameters namely the functionality, the usability, the reliability, the performance and the serviceability. In general, for most software quality assurance systems the common software metrics that are checked for improvement are the Source lines of code, cyclical complexity of the code, Function point analysis, bugs per line of code, code coverage, number of
cohesion and coupling between the modules etc.

Metrics

There are many forms of metrics in SQA but they can easily be divided into three categories: product evaluation, product quality, and process auditing.

Product Evaluation Metrics – Basically, this type of metric is actually the number ofhours the SQA member would spend to evaluate the application. Developers who might have a good application would solicit lesser product evaluation while it could take more when tackling an application that is rigged with errors. The numbers extracted from this metric will give the SQA team a good estimate on the timeframe for the product evaluation.

Product Quality Metrics – These metrics tabulates all the possible errors in theapplication. These numbers will show how many errors there are and where do they come from. The main purpose of this metric is to show the trend in error. When the trend is identified the common source of error is located. This way, developers can easily take care of the problem compared to answering smaller divisions of the problem. There are also metrics that shows the actual time of correcting the errors of the application. This way, the management team who are not entirely familiar with the application.

Process Audit Metrics – These metrics will show how the application works. Thesemetrics are not looking for errors but performance. One classic example of this type of metric is the actual response time compared to the stress placed on the application. Businesses will always look for this metric since they want to make sure the application will work well even when there are thousands of users of the application at the same time.

There are lots of options on what standard to be used in developing the plan for Software Quality Assurance. But on metrics, the numbers are always constant and will be the gauge whether the application works as planned.

Common software metrics include:

- Bugs per line of code
- Code coverage
- Cohesion
- Coupling
- Cyclomatic complexity
- Function point analysis
- Number of classes and interfaces
- Number of lines of customer requirements
- Order of growth
- Source lines of code
- Robert Cecil Martin's software package metrics

Software Quality Metrics focus on the process, project and product. By analyzing the metrics the organization the organization can take corrective action to fix those areas in the process, project or product which are the cause of the software defects.

The de-facto definition of software quality consists of the two major attributes based on intrinsic product quality and the user acceptability. The software quality metric encapsulates the above two attributes, addressing the mean time to failure and defect density within the software components. Finally it assesses user requirements and acceptability of the software. The intrinsic quality of a software product is generally measured by the number of functional defects in the software, often referred to as

bugs, or by testing the software in run time mode for inherent vulnerability to determine the software "crash" scenarios.

Maintainability : Maintainability is the ease with which a program can be correctif an error occurs. Since there is no direct way of measuring this an indirect way has been used to measure this. MTTC (Mean time to change) is one such measure. It measures when a error is found, how much time it takes to analyze the change, design the modification, implement it and test it.

Integrity : This measure the system's ability to with stand attacks to its security. Inorder to measure integrity two additional parameters are threat and security need to be defined. Threat – probability that an attack of certain type will happen over a period of time. Security – probability that an attack of certain type will be removed over a period of time. Integrity = Summation [(1 - threat) X (1 - security)]

Usability : How usable is your software application ? This important characteristic ofyour application is measured in terms of the following characteristics:

- Physical / Intellectual skill required to learn the system
- time required to become moderately efficient in the system.
- the net increase in productivity by use of the new system.
- subjective assessment(usually in the form of questionnaire on the new system).

Standard for the Software Evaluation

In context of the Software Quality Metrics, one of the popular standards that addresses the quality model, external metrics, internal metrics and the quality in use metrics for the software development process is ISO 9126.

Defect Removal Efficiency

Defect Removal Efficiency (DRE) is a measure of the efficacy of your SQA activities.. For eg. If the DRE is low during analysis and design, it means you should spend time improving the way you conduct formal technical reviews.

$$DRE = E / (E + D)$$

Where E = No. of Errors found before delivery of the software and D = No. of Errors found after delivery of the software.

Ideal value of DRE should be 1 which means no defects found. If you score low on DRE it means to say you need to re-look at your existing process. In essence DRE is a indicator of the filtering ability of quality control and quality assurance activity. It encourages the team to find as many defects before they are passed to the next activity stage. Some of the Metrics are listed out here:

Test Coverage = Number of units (KLOC/FP) tested / total size of the system Number of tests per unit size = Number of test cases per KLOC/FP Defects per size = Defects detected / system size Cost to locate defect = Cost of testing / the number of defects located Defects detected in testing = Defects detected in testing / total system defects Defects detected in production = Defects detected in production/system size Quality of Testing = No. of defects found during Testing/(No. of defects found during testing + No of acceptance defects found after delivery) *100 System complaints = Number of third party complaints / number of transactions processed Effort Productivity = Test Planning Productivity = No of Test cases designed / Actual Effort for Design and Documentation Test Execution Productivity = No of Test cycles executed / Actual Effort for testing Test efficiency= (number of tests required / the number of system errors).

SOFTWARE PROCESS IMPROVEMENT

Process Areas in Capability Maturity Model (CMM)

The Capability Maturity Model Integration (CMMI), based process improvement can result in better project performance and higher quality products. A Process Area is a cluster of related practices in an area that, when implemented collectively, satisfy a set of goals considered important for making significant improvement in that area.

In CMMI, Process Areas (PAs) can be grouped into the following four categories to understand their interactions and links with one another regardless of their defined level:

Process Management : It contains the cross-project activities related to defining, planning, resourcing, deploying, implementing, monitoring, controlling, appraising, measuring, and improving processes. Process areas are :

- Organisational Process Focus.

- Organisational Process Definition.
- Organisational Training.
- Organisational Process Performance.
- Organisational Innovation and Deployment.

Project Management : The process areas cover the project management activitiesrelated to planning, monitoring, and controlling the project. Process areas are :

- ✓ Project Planning.
- ✓ Project Monitoring and Control.
- ✓ Supplier Agreement Management.
- ✓ Integrated Project Management for IPPD (or Integrated Project Management).
- ✓ Risk Management.
- ✓ Integrated Teaming.
- ✓ Integrated Supplier Management.
- ✓ Quantitative Project Management.

Engineering : Engineering process areas cover the development and maintenanceactivities that are shared across engineering disciplines. Process areas are :

- ✓ Requirements Development.
- ✓ Requirements Management.
- ✓ Technical Solution.
- ✓ Product Integration.
- ✓ Verification.
- ✓ Validation.

Support : Support process areas cover the activities that support productdevelopment and maintenance. Process areas are :

- ✓ Process and Product Quality Assurance.
- ✓ Configuration Management.
- ✓ Measurement and Analysis.
- ✓ Organisational Environment for Integration.
- ✓ Decision Analysis and Resolution.
- ✓ Causal Analysis and Resolution.

PROCESS AND PRODUCT QUALITY ASSURANCE (PPQA)

PROCESS AREA IN CMMI

The purpose of Process and Product Quality Assurance (PPQA) is to provide staff and management with objective insight into processes and associated work products. A Support Process Area at Maturity Level 2.

The Process and Product Quality Assurance process area involves the **following activities:**

- ✓ Objectively evaluating performed processes, work products, and services against applicable process descriptions, standards, and procedures.
- ✓ Identifying and documenting noncompliance issues.
- ✓ Providing feedback to project staff and managers on the results of quality assurance activities.
- ✓ Ensuring that noncompliance issues are addressed.

The Process and Product Quality Assurance process area supports the delivery of high-quality products and services by providing project staff and managers at all levels with appropriate visibility into, and feedback on, processes and associated work products throughout the life of the project.

Specific Goals and Practices

SG 1 Objectively Evaluate Processes and Work Products

Adherence of the performed process and associated work products and services to applicable process descriptions, standards, and procedures is objectively evaluated.

SP 1.1 Objectively Evaluate Processes.

Objectively evaluate the designated performed processes against the applicable process descriptions, standards, and procedures. Objectivity in quality assurance evaluations is critical to the success of the project. A description of the quality assurance reporting chain and how it ensures objectivity should be defined.

SP 1.2 Objectively Evaluate Work Products and Services.

Objectively evaluate the designated work products and services against the applicable process descriptions, standards, and procedures. The intent of this subpractice is to provide criteria, based on business needs, such as the following:

- What will be evaluated during the evaluation of a work product?
- When or how often a work product will be evaluated?
- How the evaluation will be conducted?
- Who must be involved in the evaluation?

SG 2 Provide Objective Insight

Noncompliance issues are objectively tracked and communicated, and resolution is ensured.

- SP 2.1 Communicate and Ensure Resolution of Noncompliance Issues.

Communicate quality issues and ensure resolution of noncompliance issues with the staff and managers. Noncompliance issues are problems identified in evaluations that reflect a lack of adherence to applicable standards, process descriptions, or procedures. The status of noncompliance issues provides an indication of quality trends. Quality issues include noncompliance issues and results of trend analysis.

When local resolution of noncompliance issues cannot be obtained, use established escalation mechanisms to ensure that the appropriate level of management can resolve the issue. Track noncompliance issues to resolution.

THE SEI PROCESS CAPABILITY MATURITY MODEL, ISO, SIX-SIGMA

SEI = 'Software Engineering Institute' at Carnegie-Mellon University; initiated by the U.S. Defense Department to help improve software development processes.

CMM = 'Capability Maturity Model', now called the CMMI ('Capability Maturity Model Integration'), developed by the SEI. It's a model of 5 levels of process 'maturity' that determine effectiveness in delivering quality software. It is geared to large organizations such as large U.S. Defense Department contractors. However, many of the QA processes involved are appropriate to any organization, and if reasonably applied can be helpful. Organizations can receive CMMI ratings by undergoing assessments by qualified auditors.

Level 1 - characterized by chaos, periodic panics, and heroic efforts required byindividuals to successfully complete projects. Few if any processes in place; successes may not be repeatable.

Level 2 - software project tracking, requirements management, realistic planning, andconfiguration management processes are in place; successful practices can be repeated.

Level 3 - standard software development and maintenance processes are integratedthroughout an organization; a Software Engineering Process Group is is in place to oversee software processes, and training programs are used to ensure understanding and compliance.

Level 4 - metrics are used to track productivity, processes, and products. Projectperformance is predictable, and quality is consistently high.

Level 5 - the focus is on continouous process improvement. The impact of newprocesses and technologies can be predicted and effectively implemented when required.

Perspective on CMM ratings: During 1997-2001, 1018 organizations were assessed. Of those, 27% were rated at Level 1, 39% at 2, 23% at 3, 6% at 4, and 5% at 5. (For ratings during the period 1992-96, 62% were at Level 1, 23% at 2, 13% at 3, 2% at 4, and 0.4% at 5.) The median size of organizations was 100 software engineering / maintenance personnel; 32% of organizations were U.S. federal contractors or agencies. For those rated at Level 1, the most problematical key process area was in Software Quality Assurance.

ISO = 'International Organisation for Standardization' - The ISO 9001:2008 standard (which provides some clarifications of the previous standard 9001:2000) concerns quality systems that are assessed by outside auditors, and it applies to many kinds of production and manufacturing organizations, not just software. It covers documentation, design, development, production, testing, installation, servicing, and other processes. The full set of standards consists of: (a)Q9001-2008 - Quality Management Systems: Requirements; (b)Q9000-2005 - Quality Management Systems: Fundamentals and Vocabulary; (c)Q9004-2009 –

Quality Management Systems: Guidelines for Performance Improvements. To beISO 9001 certified, a third-party auditor assesses an organization, and certification is typically good for about 3 years, after which a complete reassessment is required. Note that ISO certification does not necessarily indicate quality products - it indicates only that documented processes are followed.

ISO 9126 is a standard for the evaluation of software quality and defines six high level quality characteristics that can be used in software evaluation. It includes functionality, reliability, usability, efficiency, maintainability, and portability.

IEEE = 'Institute of Electrical and Electronics Engineers' - among other things, creates standards such as 'IEEE Standard for Software Test Documentation' (IEEE/ANSI Standard 829), 'IEEE Standard of Software Unit Testing (IEEE/ANSI Standard 1008), 'IEEE Standard for Software Quality Assurance Plans' (IEEE/ANSI Standard 730), and others.

ANSI = 'American National Standards Institute', the primary industrial standards body in the U.S.; publishes some software-related standards in conjunction with the IEEE and ASQ (American Society for Quality).

Six Sigma is a methodology of quality management that gives a company tools for business processes improvement. This approach allows to manage quality assurance and business processes more effectively, and reduce costs and increase company profits. *The fundamental principle* of Six Sigma approach is the customer satisfaction through implementing defects-free business processes and products (3.4 or fewer defective parts per million).*Six Sigma* approach determines factors that are important for product and service quality. This approach contributes to reduction of the business process deviation, improvement of opportunities and increase of production stability.

There are five stages in Six Sigma Project:

1. **Defining**

   The first stage of Six Sigma project is to define the problem and deadlines to solve this problem. The team of specialists considers a business process (e.g., production process) in details and identifies defects that should be erased. Then the team generates a list of tasks to improve the business process, project boundaries, customers, their product and service requirements and expectations.

2. **Measuring**

   On the second stage the business process is to be measured and current performance is to be defined. The team collects all data and compares it to customer requirements and expectations. Then the team prepares measures for future large-scale analysis.

3. **Analyzing**

   As soon as the data is put together and the whole process is documented, the team starts analysis of the business process. The data collected on stage two “Measuring” are determine root reasons of defects/problems and identify gaps between current performance and new goal performance. Usually the team specialists begin with defining the fields in which employees make mistakes and cannot take effective control of the process.

4. **Improving**

   On this stage the team analyzes the business process and works up some recommendations, solutions and improvements to erase defects/problem or achieve desired performance level.

5. **Controlling**

   On the final stage of Six Sigma Project the team creates means of control of the business process. It allows the company to hold and extend scale of transformations.

   Other software development/IT management process assessment methods besides CMMI and ISO 9000 include SPICE, Trillium, TickIT, Bootstrap, ITIL, MOF, and CobiT.

PROCESS CLASSIFICATION

The processes can be classified to structure service level agreements. The process classes shown in table have been identified so far in past and current projects. Depending on the concrete service there may be some of the classes missing but the complexes the service the more of them are necessary.

Services must satisfy some quality criteria to reach a service level acceptable to the customer. For example, in the presented scenario the response time must be small enough to query information during the sales conversation. Management processes must reach reasonable service levels, too, e.g. the problem resolution process must fix an error in a certain amount of time.

Table: Process Classes

Class	*Description*
provisioning	installation of the service
usage	normal usage of the service
operation	service management of the provider
maintenance	plan able activities needed for service sustainment
accounting	collection and processing of accounting data
change management	extension, cancellation and change of service elements
problem management	notification, identification and resolution of service malfunction
SLA management	management of SLA contents
customer care	service reviews and user support
termination	De-installation of the service

Before the service can be used it must be installed. The *provisioning* class includes installation, test, acceptance and if necessary migration processes. Timeliness is an important quality parameter for processes in this class. *Usage* is the most common interaction. Service levels define for example the response time of the service.

Operation processes are invisible to the customer most of the time, but quality parameters are needed for them anyway, for example, the amount of time to detect a malfunction of a POP for ISDN calls is a quality criteria of service operation. As the service is usually at least limited during *maintenance* activities the customer wants to limit e.g. the time slot for maintenance. The method used for *accounting* purposes is an important fact because it influences the price of the service.

Minor changes of the service are not uncommon in long-term relationships. This could be the extension of resources or addition of further dealers in the presented scenario. Thus a *change management* is necessary. A well working *problemmanagement* is a very important factor in outsourcing. Problems can be solved faster ifboth partners are working together smoothly using well defined processes.

Often the requirements of the customer change during lifetime of the SLA. If these changes are known but not the point in time it is useful to specify a *SLAmanagement* with processes for announcing the need, negotiating and changing theSLA. Communication on strengths and weaknesses of the service, its quality or future optimizations should be some of the subjects of regular *customer care* contacts. Besides, a help desk or other support offers are usually necessary for complex services.

Termination of processes for a relationship usually need to be specified ifequipment of the provider is installed at customer locations or if support of the provider is needed for migration to the following service.

Additionally, it is useful to define some basic processes which are needed in many classes. Such processes are for example documentation, feedback and escalation mechanisms.

Summary

Quality is the activities designed to improve the organization and its services. Quality is essentially about learning what you are doing well and doing it better. Quality assurance is the process of verifying or determining whether products or services meet or exceed customer expectations. Quality assurance is a process-driven approach with specific steps to help define and attain goals. This process considers design, development, production, and service. Quality control is the observation techniques and activities used to fulfill requirements for quality. The various factors, which influence the software, are termed as software factors. They can be broadly divided into two categories. The classification is done on the basis of measurability. The first category of the factors is of those that can be measured directly such as number of logical errors and the second category clubs those factors which can be measured only indirectly for example maintainability but the each of the factors are to be measured to check for the content and the quality control. Two leading firms namely, IBM and Hewlett-Packard have placed a great deal of importance on software quality. Software Quality Metrics focus on the process, project and product. In operational terms, the two metrics are often described by terms namely the defect density (rate) and mean time to failure (MTTF).Although there are many measures of software quality, correctness, maintainability, integrity and usability provide useful insight. Defect Removal Efficiency (DRE) is a measure of the efficacy of your SQA activities.DRE is a indicator of the filtering ability of quality control and quality assurance activity. Process Areas in Capability Maturity Model (CMM). The Capability Maturity Model Integration (CMMI), based process improvement can result in better project performance and higher quality products. SEI = 'Software Engineering Institute', ISO = 'International Organization for Standardization', IEEE = 'Institute of Electrical and Electronics Engineers', ANSI = 'American National Standards Institute' and Six Sigma are few standards for Software Engineering and Software Quality Assurance.

Software Quality Assurance

INTRODUCTION

SQA should be utilized to the full extend with traceability of errors, cost effiecient . There are the principles behind a highly efficient SQA procedure. Every business should use SQA and use it to the best as they can. They can easily guide the application to be the best procedure. Everything should be “spick and span” when SQA has been executed. Since it is internationally recognized, the application should not only have a local impact but global acceptance at the same time. Businesses that develop and application and setting it for global standard will have a high reputation and the salability is even better. SQA definitely has benefits that will launch the company towards the global standard it needs.

NEED OF SQA

There are so many reasons why a company should consider SQA. It is all about business survival and SQA is just one of the many tools the company should effectively use. And just like a tool, it has to be effectively used to its maximum. If the tool is not used to its full extent, the tool will just be a financial burden to the company. SQA should also be utilized at the same way – to its full extent.

One of the reasons why businesses and companies opt out in using SQA is their inability to realize who effective an SQA when properly used. The following are the reasons why SQA should be used by any company before releasing their application to their intended users:

Traceability of Errors

Without using SQA, developers can still locate the errors of the application. Since their familiarity with the application is impeccable, they will have the ability to find a remedy to the problem. SQA on the other hand does not just look for answers to their problems.

SQA looks for the reason why the error occurred. Looking for errors is really easy but tracing the roots of the problem is another thing. The SQA team should be able to tell which practice has started the problem. Most of the time, the errors is not rooted on the execution of the coding but it could be found in the philosophy in developing the application.

Cost Efficient

This idea is saving money is very debatable in SQA. Using SQA means you will have to hire another set of people to work with you. Most of the SQA today are 3rd party companies that are only there to work on the specific project. Manpower will always cost money. Manpower does not even cover the cost of the tools the SQA team might use. Unfortunately most of the SQA tools are expensive especially the most efficient tools.

On the other hand, consider the idea of locating the bugs and errors in the application before they are released. Every application that are used in a business setting or sold to the consumers should have an ideal of 0% error. Even though this is rarely attained, SQA should be able to reduce errors to less than 1%.

Considering the effect on productivity together with customer satisfaction, you can always tell that preventing the problem is always better compared to answering them. SQA could practically save your business and boost it to success. Without them, there is a higher possibility that the application will have errors which will discourage your customers from purchasing the application. In a business setting, an erroneous application will lead to slower productivity since instead of helping their customers; they might end up troubleshooting their application most of the time.

Flexible Solutions

SQA could easily provide solutions to the problem since they look for the root of the problem instead of just answering them. By providing the root of the problem, they have the ability to provide solutions to these problems fast. But instead of one solution only, the SQA team should be able to provide more than one solution.

Finding the root of the problem means they can easily provide more than one

solution. Most of the SQA companies offer web-based solutions to their concerns and aside from that, on-site solutions are also provided by most companies to ensure that the company could have as much options for solutions.

Better Customer Service

One of the ultimate goals of any businesses is to provide the best customer service possible. SQA could help these companies realize that goal. Most of the transactions in today's business world today are IT related so an efficient application should be employed in the first place. In order to do that, the application should go under rigorous test. Testers could do that but again, that is only looking for the answer of the problem.

SQA goes more than simple solution to the problem. The SQA team takes note at every step of the application development process. Through this, they will know what are the practices that might go out against the industry standards.

Aside from monitoring, SQA also has a good reputation in looking after the development of the application. There are universal standards that guide every SQA team on how to monitor and guide the application for its success. Every step of the way has been standardized.

Even the recommended documents are carefully planned out to ensure that that feed back is working as planned. The end result is a globally recognized product developed with certain standards. The SQA methodologies are ISO certified which means, the practice of monitoring the development of a software or application is recognized by some of the world's biggest companies.

Innovation and Creativity

Since the SQA team is here to standardize how their work is done, it still fosters innovation and creativity for developing the product. Everyone is given a free hand in developing the application and the SQA team is there to help them standardize the ideas.

Most of the time, developers would feel constricted in building applications because of the metrics proposed by the developers. On the other hand, developers should have the ability to be more creative as long as the SQA team is there. The SQA team on the other hand has the ability to foster drive as everything will be under the guidance of the team.

Software Principles

The SQA team also has to follow certain principles. As a provider of quality assurance for procedures and application, they need to have a strong foundation on what to believe in and what to stand for. These principles will ensure that the application will live up to the expectations of the client and the users. Like the application, the SQA principles run on the background but eventually dictate how the system will be tackled. The following are some of the most powerful principles that can be used for proper execution of software quality assurance:

Feedback – In gist, the faster the feedback the faster the application will move forward. An SQA principle that uses rapid feedback is assured of success. Time will always be the best friend and the most notorious enemy of any developer and it's up to the SQA team to give the feedback as soon as possible. If they have the ability to get the feedback of their applications as soon as possible, then the chance of developing a better application faster is possible.

Focus on Critical Factor – This principle has so many meanings; first it just means that some of the factors of the software being developed are not as critical compared to other. That means SQA should be focused on the more important matters.

Secondly, SQA's measurement should never be universal in the sense that every factor in the application should not have the same treatment. One great example of this is the treatment of the specific functions compared to the skin or color of the interface. Clearly, the function should have more focus compared to a simple skin color.

Multiple Objectives – This is partly a challenge as well as risk for the SQA team. Atthe start of the SQA planning, the team should have more than one objective. If you think about it, it could be very dangerous however it is already a common practice. But what is emphasized here is that each objective should be focused on. As much as possible a matrix should be built by the SQA so that it could track the actual actions that relates to the objective.

Evolution – Reaching the objective is really easy but every time something newhappens, it should be always noted. Evolution is setting the benchmark in each development. Since the SQA team is able to mark every time something new is done, evolution is monitored.

The good thing about this principle is for future use. Whenever a benchmark is not reached, the SQA team should be able to study their previous projects. Evolution should be able to inform and educate the SQA team while working on the project.

Quality Control – By the name itself, Quality Control is the pillar for Software QualityAssurance. Everything needs to have quality control – from the start to the finish. With this principle there has to be an emphases on where to start. The biggest and the tightest quality control should be executed as early as possible.

For example, when the SQA team receives the SR (software requirements document) the intensity of quality control should be at the start. Of course quality control will still be executed until the end but developers should take into account that anything that starts out real bad could never take off. It's better to know what's wrong at first than to find that out later.

Motivation – There is not substitute than to have the right people who has the will todo their job at all times. When they have the right mindset the willingness to do it, everything will just go through. Work will definitely be lighter, expertise will be seen and creativity is almost assured when everyone has the drive and passion in their line of work. Quality assurance is a very tedious task and will get the most out of the person if they are not dedicated to their line of work.

Process Improvement – Every project of the SQA team should be a learningexperience. Of course each project will give us the chance to increase our experience of SQA but there's more to that. Process improvement fosters the development of the

actual treatment of the project. Every project has a unique situation that will give the SQA team a chance to experience something new. This “new” will never be translated to something good if they are not documented for future references. Learning should not only be based on individual experience but also on company’s ability to adapt to the new situation and use it for future references.

Persistence – There is no perfect application. The bigger they get, the more errorthere could be. The SQA team should be very tenacious in looking for concerns in every aspect of the software development process. Even with all the obstacles everyone would just have to live with the fact that every part should be scrutinized without hesitation.

Different Effects of SQA – SQA should go beyond software development. A regularSQA will just report for work, look for errors and leave. The SQA team should be role models in business protocols at all times. This way, the SQA does not only foster perfection in the application but also in their way of life. That seemed to be quite off topic but believe me; when people dress and move to success, their work will definitely reflect with it.

Result-focused – SQA should not only look at the process but ultimately its effect to the clients and users. The SQA process should always look for results whenever a phase is set.

These are the principles that every SQA plan and team should foster. These principles tell encourages dedication towards work and patience not necessarily for perfection but for maximum efficiency.

SQA ACTIVITIES

SQA is composed of a variety of tasks linked with 2 dissimilar constituencies: SQA group that has responsibility for quality assurance planning, record keeping, oversight, analysis and reporting and the other are software engineers, who do technical work. The character of a SQA group is to help the software team in attaining a high-class product.

The SQA group prepares a SQA plan that identifies:

- Assessments to be carried out,
- Reviews and audits to be executed,
- Standards those are relevant to the project,
- Measures for error reporting and tracking,
- Credentials to be produced by the SQL group, and
- Amount of feedback provided to the software project team.

BUILDING BLOCKS OF SQA

SQA Project Metrics

The application is only as good as its numbers. It is a harsh reality but everything has to come down to the numbers. Although we can enumerate tens or hundreds of features in a single application, it will all be for nothing if the metrics do not live up according to expectation. The SQA team should ensure that the expected metrics will be posted. At the same time, the SQA team should also select the right tool to gauge the application.

There are hundreds of applications that can measure the application but it is

quite rare to find the right application to provide the right information.

To fully gauge the software’s ability, a number of metrics should be attained. A regular testing application will just show errors of the software as well its ability to withstand constant and heavy traffic. However when you take a closer look at the application, you will realize these are not the only numbers that you need to know if the application actually works. The SQA team should know the numbers that needs to be shown to the developers and to the management.

Metrics

In the previous chapter, the metrics provided were referred to the SQA’s ability to influence the application. This time, the metrics will only refer to the application alone. To gauge the actual application, the metrics are divided into four categories. Each category has attributes with specific metrics ensuring that the attribute is achieved.

Quality of Requirements – These set of metrics will show how well the application is planned. Among the requirements of quality first and foremost is completeness. Remember that requirements are geared to answer all the problems and concerns of the users. Therefore, its main aim is to answer all the concerns.

One of metrics found in this category is the completeness of the requirements. As this is in the planning stage, the ability to be understood is also important. The SQA team should gauge if the document are well written and useful.

Lastly, the requirements that were written by the developers and the intended users should be traceable. No one can just place a requirement out of whim since each of the requirements should have a logical root or explanation.

Product Quality – These set of metrics will gauge the ability of the developersto formulate codes and functions. The SQA team will first gauge how simple the application has been written. The software has to be written in a simple manner. It might sound contradicting that even a highly complex application could come in as simple.

What the SQA team is looking for from the application is the logical formulation of the codes. When the application is logically coded, it has two repercussions which are also gauged by SQA.

First is the maintainability. If the application could be easily understood, troubleshooting is very easy. Reusability of the application is also emphasized. Since the codes could easily be distinguished, developers can use parts of the application to be implemented to another program.

SQA Software and Tools

In quality assurance, it is always important to get all the help we could get. In other industries, developers could easily check the products manually and discard those that do not meet the standard. The length and the width of the product are checked to maintain standardization of the product. Others use special machines to check the product. With tools and machines, they can easily set a standard with their products.

That also goes the same with software and applications. Although it does not use physical machines, applications go through rigorous testing before they are released to the public even for beta testing. The tools used in SQA are generally testing tools wherein an application is run through a series of tests to gauge the performance of the application.

The tools used in SQA vary in purpose and performance. These applications range from testing the code or running the application under great stress. These tools are employed to test the application and produce numbers and statistics regarding the actual application. Through these numbers, the SQA team and their developers will know if the application has lived up according to the targeted performance.

Like most developers each SQA team has their preferred tools for application testing. Based on their belief and expertise, the SQA team will usually give the owners or business managers a free hand on what type of testing tool to use.

Notable SQA Tools

The following are some of the renowned SQA tools and applications. There are

still hundreds out there but the following tools have been around for years and have been used by thousands or probably millions of testers.

WinRunner – Developed by HP, WinRunner is a user friendly application thatcan test the applications reaction from the user. But other than measuring the response time, WinRunner can also replay and verify every transaction and interaction the application had with the user. The application works like a simple user and captures and records every response the application does.

LoadRunner – Developed by HP, LoadRunner is one of the simple applicationsthat can test the actual performance of the application. If you are looking for a program to test your application's tolerance to stress, LoadRunner is your tool. It has the ability to work like thousands of users **at the same time –** testing the stress of the application.

QuickTest Professional – If you have worked with WinRunner you surely have bumped in with this tool. Built by HP, QuickTest emulates the actions of users and exploits the application depending on the procedure set by testers. It can be used in GUI and non-GUI websites and applications. The testing tool could be customized through different plug-ins.

Mercury Test Director – An all-in-one package, this web-based interface couldbe used from start to end in testing an application or a website. Every defect will be managed according to their effect to the application. Users will also have the option to use this exclusively for their application or use it together with wide array of testers.

Silktest – Although available in limited operating system, Silktest is a very smarttesting tool. Silktest lists all the possible functions and tries to identify the function one by one. It can be implemented in smaller iterations as it translate the available codes into actual objects.

Bugzilla – Developed by Mozilla, this open source testing tool works as thename suggests. Bugzilla specializes in detecting bugs found in the application or website. Since the application is open-source it can be used freely and it is availability in different OS makes it even a viable alternative for error tracking. The only downside is it has a long list of requirements before it could run.

Application Center Test – Also known as ACT, this testing tool was developedby Microsoft using ASP.NET. This application is primarily used for determining the capacity of the servers that handle the application. Testers can test the server by asking constant requests. A customized script either from VB or JS could be used to test the server's capacity.

OpenSTA – Another open source tool, testers can easily launch the applicationand use it for testing the application's stress capacity. The testing process could be recorded and testing times could be scheduled. Great for websites that needs daily maintenance.

QARun – Instead of an application, QARun is actually a platform where you canbuild your own testing application. QARun could be easily integrated with the application so that it could easily sync with the releases and checks the application or website every time something new is introduced.

The documentation that supports the application is also gauged and the comment within the coding is also gauged. It is not important if there lots of comments but what is important is that the comments are clear and could be found in every function.

Implementation Capability –

The general coding behavior of the developers is also gauged by the SQA team. The metrics used in this classification is based on the SDLC used by the developers. The SQA team will rate the developer's ability to finish each action in the stage of the SDLC on time.

The staff hours are rated according to the need of the stage. Each phase of the SDLC will have their time requirement – some will just need a day or two while others will require a month to finish. The time required in every stage is set by the project manager.

Aside from the time required for completion, the developers are also rated if they ever finish a task at all. There are tasks that are necessary for the application to run while there are tasks that could be offset by another task. Although that's a convenience, it will actually trigger a domino effect so the whole operation could be placed in jeopardy.

Software Efficiency – Last but maybe the most important is to ensure that the application do not have any errors at all. This is basically a major pillar to any application. Without errors, the software will never give any hackers a chance to infiltrate the system. Usually, the SQA team will develop test cases wherein the error is highlighted. The developers are then rated how fast they could locate the root of the problem and how fast they could built a fix to the problem.

These are the metrics that every software developer will have to go through. The better the rating, the better the application would work.

Selecting the Right Quality Model

There are so many quality models to choose from. Each of them has their own advantage and disadvantages. It is up to the SQA team to select which application that will give the clients a good idea how the application works.

In order to select a good Quality Model is no trick actually. Every application developed has a "core" or a feature that differentiates the application from other software.

The SQA team should just focus on that feature and look for the quality model that can fully gauge the feature. It is a very simple idea but will do wonders especially when the SQA team is trying to prove the validity of the application against the feature needed by the clients.

The only disadvantage of this technique is that the developers might be focusing too much in the feature. However, all of the Quality Models could easily gauge every aspect of the application. That means every SQA team can easily gauge the software without any problem. Finding errors in the application is a great challenge to the SQA team. But with the right tools the software could be easily gauged and the application will work as expected.

Selecting Right SQA Tool

The tools that are listed above are only a very small chunk of the hundreds of testing tools available as a free or licensed application. Among the dilemmas of the SQA team is to select the proper testing tool for the particular website or application.

In selecting the right tool, testers and SQA managers should always start with their objective. Although usually the testers' objective is to know every bug and error related to the application, testers and developers should also consider writing benchmarks. Through this benchmark, they can assure the application works as expected. Every tester should understand that each application might require a completely different testing tool. Experience will always tell the testers that a simple automation tool will never work in complex applications.

Aside from this consideration, testers also have the choice to use more than one testing tool. But before getting all the possible testing tools, the SQA team should consider what metrics they are looking for. By determining the numbers they need and the benchmark they are looking for they can easily select the testing tools they can use.

The success of SQA can be determined based on the applications they use. In fact, it can be safely said that the SQA team is only as good as the software they use. Although they can check the application manually, it will take time and it will always be prone to human error. With the right testing tool, the application is tested faster with better accuracy.

2.4.5 SQA Planning and Requirements

The scope of Software Quality Assurance or SQA starts from the planning of the application until it is being distributed for the actual operations. To successfully monitor the application build up process, the SQA team also has their written plan. In a regular SQA plan, the team will have enumerated all the possible functions, tools and metrics that will be expected from the application. SQA planning will be the basis of everything once the actual SQA starts.

Without SQA planning, the team will never know what the scope of their function

is. Through planning, the client's expectations are detailed and from that point, the SQA team will know how to build metrics and the development team could start working on the application.

When the SQA team starts working, one of the first things they will have to tackle is to determine the developer team's initial output. The intended users or at least part of them will be forwarding information which tells of their expectations of the new software called UR (User Requirements), this document will be the bases.

What is left is to determine the software requirements in the intended application. In business or in any intended application, software requirements are very important since these requirements will tell how an application would work. This will also be the ground work for developers. From these requirements they will know what language, SDLC program and other protocols that will be used to build the applications.

While the developers work on determining the requirements, the SQA team will be monitoring their work to ensure the requirements are according to the expectations.

Technical Activities

One of the responsibilities of the SQA team from the developers is the required user requirements document. The UR document is produced by the intended users who are tapped for this project. This document will inform the developers what the users are expecting from the application. The SQA team will ensure that the document exists before they actually work on the software requirements. The UR document will be the bases for the SR document so it will never make sense if the SR document is produced without consulting the user requirements document first. The SQA team should also ensure that the UR document should be of quality. To develop an efficient software requirement document, the developers should use a known method to determine this type requirement for software development. There are lots of software requirements analysis methods that could be used but developers do not just pick one they prefer. The SQA team should ensure that the analysis method should be reputable or recognized. As much as possible the analysis method should have been published.

Aside from a published analysis method for software requirements, it is also ideal that the analysis method will be supported by different Computer-Aided Software Engineering (CASE) tools. CASE tools are used to develop models for software development and software requirements could easily be determined by with this application. Although it is not required for financial and time frame reasons, CASE tools should be highly considered by developers. The intended metrics for performance is also detailed in software requirements. Above all, metrics should be emphasized since it will tell how fast or how efficient the application should perform. Based on this metrics, the SQA team should be able to determine their testing tools. These documents should reflect the principles and the metrics of the clients and the intended users are looking for.

SQA Management Plans

In the planning and requirements phase, there will be four plans that will be created. These plans will be used in the next stage of software development which is the architectural phase. The Software Quality Assurance Plan (SQAP) for architectural design (AD) is basically the following list of activities to ensure that the preparation of the architectural plan is a success. If any tool or application will be used, the SQA team should point this out in this phase.

- Software Project Management Plans (SPMP) for Architecture Design.

  The SQA team should ensure that this management plan will have the specific budget for developing the software. Aside from being specific, the SQA team should also ensure that the estimate should be obtained using scientific methods.

- Software Configuration Management Plans (SCMP) for Architectural Design.

  SCMP is the plan on how the software will be eventually configured. In this phase, the SQA team should ensure that the configuration should have been established at the start.

- Software Verification Management Plan (SVMP) for Architectural Design.

  Like most of the management plans, this one has already been established. But extra information should be sought after by the SQA team. Since this document will be used in the architectural design phase more detailed information is needed. A test plan should be made to ensure that the Architectural Design

phase is a success. Also, the SQA team should ensure a general testing system should be established.

Output and Principles

At the end of the planning phase, developers should be able to produce a document that has the details needed for the Architectural Design phase. The SQA team should check if the document has the following characteristics:

- **Consistency –** This is especially applicable if the output does not use anyCASE tools. Case tools should easily figure out inconsistencies and it is up to the SQA team to determine this consistency if no CASE tools was used in this phase.

- **Verifiable –** Each specific requirement in this phase should be tested either by apopular tool or verifiable facts from the past.
- **Complete –** The SR or software requirements is the end of the requirementphase. To declare that this phase is complete, the SQA team should check with the UR (user requirements) and see to it that all the things needed by the user are answered. If possible, a matrix should be created to track each answer of the SR to the UR built by the users.

In the end, the SQA team, the developers and representatives of the users will come together for a formal review of the documents. By sitting together, everyone will know if the things needed in the software are covered or will come together with the application.

SQA PLANNING AND STANDARDS

Planning is one of the most important aspects of Software Quality Assurance. The entire operation of the SQA team depends on how well their planning is done. In smaller businesses, planning might not really dictate the flow of SQA but in larger businesses, SQA Planning takes on center stage. Without it, each component or department that works on the application will be affected and will never function.

In gist, SQA Planning tackles almost every aspect of SQA's operation. Through planning, each member and even non-member of the SQA team is clearly defined. The reason for this is very simple: when everyone knows their role and boundaries, there is no overlapping of responsibilities and everyone could concentrate on their roles.

But SQA Planning is not only a document that tells who gets to do the specific task. The stages in are also detailed. The whole SQA team will be very busy once the actual testing starts but with SQA, everyone's work is clearly laid out. Through planning, the actual state of the application testing is known. Again in smaller businesses, the planning maybe limited to the phase of the application testing but when outlined for corporations, the scenario changes and only through planning that everyone will know where they are and where they are going in terms of **SQA.**

SQA Planning is not just a simple document where objectives are written and stages are clearly stated. Because of the need to standardize software development ensuring the limitation of error, a scientific approach is recommended in developing an SQA plan. Certain standards such as IEEE Std 730 or 983.

SQA Plan Content

An SQA Plan is detailed description of the project and its approach for testing. Going with the standards, an SQA Plan is divided into four sections:

- Software Quality Assurance Plan for Software Requirements;
- Software Quality Assurance Plan for Architectural Design;
- Software Quality Assurance Plan for Detailed Design and Production and;
- Software Quality Assurance Plan for Transfer

In the first phase, the SQA team should write in detail the activities related for software requirements. In this stage, the team will be creating steps and stages on how they will analyze the software requirements. They could refer to additional documents to ensure the plan works out.

The second stage of SQA Plan or the SQAP for AD (Architectural Design) the

team should analyze in detail the preparation of the development team for detailed build-up. This stage is a rough representation of the program but it still has to go through rigorous scrutiny before it reaches the next stage.

The third phase which tackles the quality assurance plan for detailed design and actual product is probably the longest among phases. The SQA team should write in detail the tools and approach they will be using to ensure that the produced application is written according to plan. The team should also start planning on the transfer phase as well.

The last stage is the QA plan for transfer of technology to the operations. The SQA team should write their plan on how they will monitor the transfer of technology such as training and support.

SQA LIFECYCLE STANDARDS

Software Quality Assurance procedures have finally been standardized and have been virtually perfected after years of planning on how to perfect the application standardization. Through experience, the company was able to place in writing how to develop a plan for software development. Because it has been standardized, the application that was developed using SQA could be recognized worldwide because it has been made according to the standards. Along with the standards, the metrics are also standardized. More than anything else written in the report, the clients who are looking for an efficient application looks for the numbers more than anything else. Metrics will tell whether the application has been developed according to plan and could perform when released or sold to their intended users.

SQA Standards

IEEE (Institute of Electric and Electronic Engineers) – This standard was established by the organization with the same name. This organization was established in 1963 and the IEEE standards for software development starting in 1986. There are two types of IEEE standards for Software Quality Application: the Standard 730-1989 which was developed in 1989 and the ANSI/IEEE Standard 983 – 1986 which was the original version developed in 1986. IEEE is very popular especially for SQA Planning and development.

ISO (International Standards Organization) – One of the oldest standardization organizations in the world, ISO were established in 1947 and have established itself to be the standardized company not only in software development but also in business plans. Because it was internationally recognized it has become a powerful standard for different business uses.

DOD (US Department of Defense) – The government has also developed theirown standardization scheme especially for developing technology. They have evolved from ISO 9000 and developed a specialized standard on their own. There are currently two DOD standards: the MIL-STD-961E which refers to the program specific standards and the MIL-STD-962D which stands for general standardization. Both of these formats were used applied by the DOD starting August 1, 2003.

ANSI (American National Standards Institute) – Working with US-based companies, ANSI has become the bridge of small US based companies to international standards so that they could achieve international recognition. ANSI covers almost anything in the country, products technology and application. ANSI ensures that the products developed in the US could be used in other countries as well.

IEC (International Electro technical Commission) – Stated June of 1906, the commission has dedicated itself to the standardization of electrical materials and its development. Today it is usually associated with ISO since it has become a part of technical industries. IEC is known for standardizing electronics through the International Electro technical Vocabulary which is used by all electronic industries until today.

EIA (Electronic Industries Alliance) - EIA is a coming together of different electronic manufacturers in US. This organization set the standards for electronic products for the country and has been accepted by thousands of companies worldwide.

SQA Management Standards

Aside from internationally recognized SQA standards, there are specific standards that were developed to cater specifically for the management of software development:

ISO 9001 and ISO 9000-3 – These certifying organizations was establishedspecifically for software development. This standard encourages leadership and could be integrated continuously even when the product has been developed and released to its users. Good supplier relations are also emphasized as technology is not only developed in-house.

SW-CMM (Software Capability Maturity Model) – Developed in 1980, SW-CMM has become the standards for large scale software development companies. It has drawn support because this development model was established by the developers for the developers. It believes in quantitative methods to develop and maintain productivity. SW-CMM has a five-level model to gauge the applications maturity and establish a detailed plan to further enhance them. The best draw so far of SW-CMM is that it does not care about SDLC model, tool and documentation standard, promoting creativity for software development.

ISO/IEC 15504 Process Assessment Model and SPICE (Simulation Program with Integrated Circuit Emphasis) – Aiming for international acceptance, this type of SQA supports a specific type of testing standard for a better application. Called SPICE, this application could test each part of the application. SPICE is also used to asses the performance of circuits in electronic products.

SUMMARY

SQA has benefits that will launch the company towards the global standard it needs.SQA should also be utilized at the same way to its full extent.SQA is needed to trace the errors, find the cost affiant, to give flexible solutions and better customer service. SQA team has to follow certain principles like Feedback, Focus on Critical Factor, Multiple Objectives, Evolution, and Quality Control. Motivation, Process Improvement, Persistence, Different Effects of SQA and Result focused.SQA is composed of a variety of tasks linked with 2 dissimilar constituencies: SQA group that has responsibility for quality assurance planning, record keeping, oversight, analysis and reporting and the other are software engineers, who do technical work. The SQA team should ensure that the expected metrics will be posted. At the same time, the SQA team should also select the right tool to gauge the application. Notable SQA tools are WinRunner, LoadRunner, QuickTest Professional, Mercury TestDirector, Silktest and Bugzilla, Application Center Test, OpenSTA, QARun. Selecting the Right Quality Model and Right SQA tool is responsibility of SQA team. SQA management plans involve SQAP, SPMP, SCMP, SVMP. SQA standards are IEEE, ISO, DOD,ANSI, ICE, EIA. SQA Management Standards are ISO, SW-CMM, ISO/IEC.

Software Reliability

INTRODUCTION

According to ANSI, Software Reliability is defined as: the probability of failure-free software operation for a specified period of time in a specified environment. Although Software Reliability is defined as a probabilistic function, and comes with the notion of time, we must note that, different from traditional Hardware Reliability, Software Reliability is not a direct function of time. Electronic and mechanical parts may become "old" and wear out with time and usage, but software will not rust or wear-out during its life cycle. Software will not change over time unless intentionally changed or upgraded.

RELIABILITY MEASURES

Software Reliability is an important to attribute of software quality, together with functionality, usability, performance, serviceability, capability, install ability, maintainability, and documentation. Software Reliability is hard to achieve, because the complexity of software tends to be high. While any system with a high degree of complexity, including software, will be hard to reach a certain level of reliability, system developers tend to push complexity into the software layer, with the rapid growth of system size and ease of doing so by upgrading the software. For example, large next-generation aircraft will have over one million source lines of software on-board; next-generation air traffic control systems will contain between one and two million lines; the upcoming international Space Station will have over two million lines on-board and over ten million lines of ground support software; several major life-critical defence systems will have over five million source lines of software. While the complexity of software is inversely related to software reliability, it is directly related to other important factors in software quality, especially functionality, capability, etc. Emphasizing these features will tend to add more complexity to software.

Software failure mechanisms

Software failures may be due to errors, ambiguities, oversights or misinterpretation of the specification that the software is supposed to satisfy, carelessness or incompetence in writing code, inadequate testing, incorrect or unexpected usage of the software or other unforeseen problems. While it is tempting to draw an analogy between Software Reliability and Hardware Reliability, software and hardware have basic differences that make them different in failure mechanisms. Hardware faults are mostly *physical faults*, while software faults are *design faults*, which are harder to visualize, classify, detect, and correct. Design faults are closely related to fuzzy human factors and the design process, which we don't have a solid understanding. In hardware, design faults may also exist, but physical faults usually dominate. In software, we can hardly find a strict corresponding counterpart for "manufacturing" as hardware manufacturing process, if the simple action of uploading software modules into place does not count. Therefore, the quality of software will not change once it is uploaded into the storage and start running. Trying to achieve higher reliability by simply duplicating the same software modules will not work, because design faults cannot be masked off by voting.

A partial list of the distinct characteristics of software compared to hardware is listed below:

Failure cause: Software defects are mainly design defects.

Wear-out: Software does not have energy related wear-out phase. Errors can occur without warning.

Repairable system concept: Periodic restarts can help fix software problems.

Time dependency and life cycle: Software reliability is not a function of operational time.

Environmental factors: Do not affect Software reliability, except it might affect program inputs.

Reliability prediction: Software reliability cannot be predicted from any physical

basis, since it depends completely on human factors in design.

Redundancy: Cannot improve Software reliability if identical software components are used.

Interfaces: Software interfaces are purely conceptual other than visual.

Failure rate motivators: Usually not predictable from analyses of separate statements.

Built with standard components: Well-understood and extensively-tested standard parts will help improve maintainability and reliability. But in software industry, we have not observed this trend. Code reuse has been around for some time, but to a very limited extent. Strictly speaking there are no standard parts for software, except some standardized logic structures.

The bathtub curve for Software Reliability

Over time, hardware exhibits the failure characteristics shown in Figure 1, known as the bathtub curve. Period A, B and C stands for burn-in phase, useful life phase and end-of-life phase. A detailed discussion about the curve can be found in the topic Traditional Reliability.

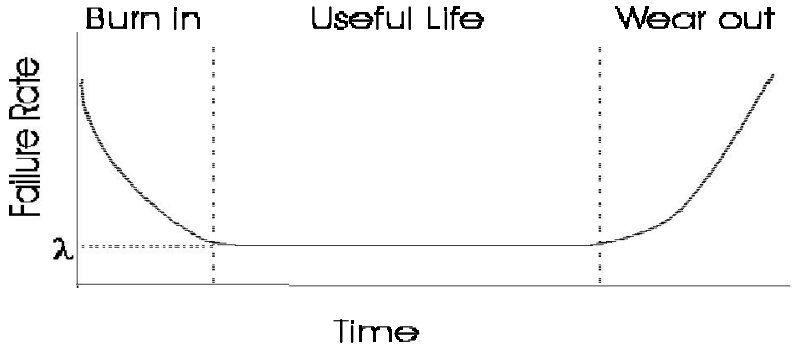


Figure 1. Bath tub curve for hardware reliability

Software reliability, however, does not show the same characteristics similar as hardware. A possible curve is shown in Figure 2 if we projected software reliability on the same axes. There are two major differences between hardware and software curves. One difference is that in the last phase, software does not have an increasing failure rate as hardware does. In this phase, software is approaching obsolescence; there is no motivation for any upgrades or changes to the software. Therefore, the failure rate will not change. The second difference is that in the useful-life phase, software will experience a drastic increase in failure rate each time an upgrade is made. The failure rate levels off gradually, partly because of the defects found and fixed after the upgrades.

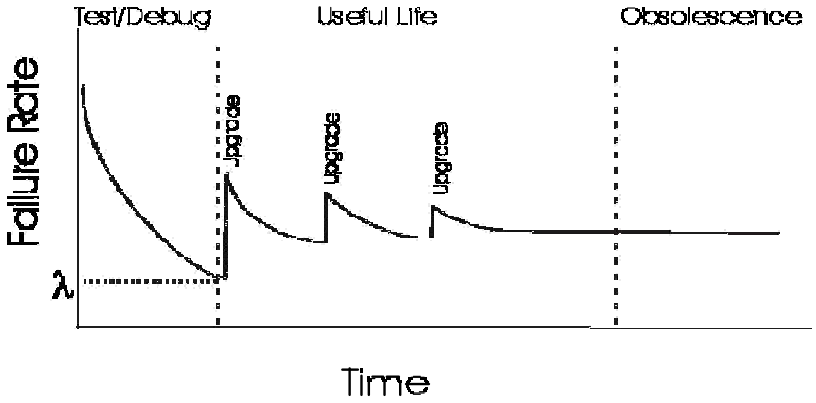


Figure 2. Revised bathtub curve for software reliability

The upgrades in Figure 2 imply feature upgrades, not upgrades for reliability. For feature upgrades, the complexity of software is likely to be increased, since the functionality of software is enhanced. Even bug fixes may be a reason for more software failures, if the bug fix induces other defects into software. For reliability upgrades, it is possible to incur a drop in software failure rate, if the goal of the upgrade is enhancing software reliability, such as a redesign or reimplementation of some modules using better engineering approaches, such as clean-room method.

A proof can be found in the result from Ballista project, robustness testing of off-the-shelf software Components. Figure 3 shows the testing results of fifteen POSIX compliant operating systems. From the graph we see that for QNX and HP-UX, robustness failure rate increases after the upgrade. But for SunOS, IRIX and Digital UNIX, robustness failure rate drops when the version numbers go up. Since software robustness is one aspect of software reliability, this result indicates that the upgrade of those systems shown in Figure 3 should have incorporated reliability upgrades.

Available tools, techniques, and metrics

Since Software Reliability is one of the most important aspects of software quality, Reliability Engineering approaches are practiced in software field as well. *Software Reliability Engineering* (SRE) is the quantitative study of the operationalbehavior of software-based systems with respect to user requirements concerning reliability.

SOFTWARE RELIABILITY MODELS

A proliferation of software reliability models have emerged as people try to understand the characteristics of how and why software fails, and try to quantify software reliability. Over 200 models have been developed since the early 1970s, but how to quantify software reliability still remains largely unsolved. Interested readers may refer to as many models as there are and many more emerging, none of the models can capture a satisfying amount of the complexity of software; constraints and assumptions have to be made for the quantifying process. Therefore, there is no single model that can be used in all situations. No model is complete or even representative. One model may work well for a set of certain software, but may be completely off track for other kinds of problems.

Most software models contain the following parts: assumptions, factors, and a mathematical function that relates the reliability with the factors. The mathematical function is usually higher order exponential or logarithmic.

Software modelling techniques can be divided into two subcategories: prediction modelling and estimation modelling. Both kinds of modelling techniques are based on observing and accumulating failure data and analyzing with statistical inference. The major difference of the two models is shown in Table 1.

ISSUES	PREDICTION MODELS	ESTIMATION MODELS
DATA REFERENCE	Uses historical data	Uses data from the current software development effort
WHEN USED IN DEVELOPMENT CYCLE	Usually made prior to development or test phases; can be used as early as concept phase	Usually made later in life cycle(after some data have been collected); not typically used in concept or development phases
TIME FRAME	Predict reliability at some future time	Estimate reliability at either present or some future time

Table 1. Difference between software reliability prediction models and software reliability estimation models

Software Reliability Metrics

Measurement is commonplace in other engineering field, but not in software engineering. Though frustrating, the quest of quantifying software reliability has never ceased. Until now, we still have no good way of measuring software reliability.

Measuring software reliability remains a difficult problem because we don't have a good understanding of the nature of software. There is no clear definition to what aspects are related to software reliability. We cannot find a suitable way to measure software reliability, and most of the aspects related to software reliability. Even the most obvious product metrics such as software size have not uniform definition.

It is tempting to measure something related to reliability to reflect the characteristics, if we cannot measure reliability directly. The current practices of software reliability measurement can be divided into four categories:

Product metrics

Software size is thought to be reflective of complexity, development effort and reliability. Lines Of Code (LOC), or LOC in thousands (KLOC), is an intuitive initial approach to measuring software size. But there is not a standard way of counting. Typically, source code is used(SLOC, KSLOC) and comments and other non-executable statements are not counted. This method cannot faithfully compare software not written in the same language. The advent of new technologies of code reuses and code generation technique also cast doubt on this simple method.

Function point metric is a method of measuring the functionality of a proposed software development based upon a count of inputs, outputs, master files, inquires, and interfaces. The method can be used to estimate the size of a software system as soon as these functions can be identified. It is a measure of the functional complexity of the program. It measures the functionality delivered to the user and is independent of the programming language. It is used primarily for business systems; it is not proven in scientific or real-time applications.

Complexity is directly related to software reliability, so representing complexity is important. Complexity-oriented metrics is a method of determining the complexity of a program’s control structure, by simplifies the code into a graphical representation. Representative metric is McCabe's Complexity Metric.

Test coverage metrics are a way of estimating fault and reliability by performing tests on software products, based on the assumption that software reliability is a function of the portion of software that has been successfully verified or tested. Detailed discussion about various software testing methods can be found in topic Software Testing.

Project management metrics

Researchers have realized that good management can result in better products. Research has demonstrated that a relationship exists between the development process and the ability to complete projects on time and within the desired quality objectives. Costs increase when developers use inadequate processes. Higher reliability can be achieved by using better development process, risk management process, configuration management process, etc.

Process metrics

Based on the assumption that the quality of the product is a direct function of the process, process metrics can be used to estimate, monitor and improve the reliability and quality of software. ISO-9000 certification, or "quality management standards", is the generic reference for a family of standards developed by the International Standards Organization (ISO).

Measure	Metrics
1. Customer satisfaction index	Number of system enhancement requests per year Number of maintenance fix requests per year User friendliness: call volume to customer service hotline User friendliness: training time per new user Number of product recalls or fix releases (software vendors) Number of production re-runs (in-house information systems groups)
2. Delivered defect quantities	Normalized per function point (or per LOC) At product delivery (first 3 months or first year of operation) Ongoing (per year of operation) By level of severity By category or cause, e.g.: requirements defect, design defect, code defect, documentation/on-line help defect, defect introduced by fixes, etc.
3. Responsiveness (turnaround time) to users	Turnaround time for defect fixes, by level of severity Time for minor vs. major enhancements; actual vs. planned elapsed time (by customers) in the first year after product delivery
7. Complexity of delivered product	McCabe's cyclomatic complexity counts across the system Halstead's measure Card's design complexity measures Predicted defects and maintenance costs, based on complexity measures
8. Test coverage	Breadth of functional coverage Percentage of paths, branches or conditions that were actually tested Percentage by criticality level: perceived level of risk of paths The ratio of the number of detected faults to the number of predicted faults.
9. Cost of defects	Business losses per defect that occurs during operation Business interruption costs; costs of work-around Lost sales and lost goodwill Litigation costs resulting from defects Annual maintenance cost (per function point) Annual operating cost (per function point) Measurable damage to your boss's career
10. Costs of quality activities	Costs of reviews, inspections and preventive measures Costs of test planning and preparation Costs of test execution, defect tracking, version and change control Costs of diagnostics, debugging and fixing Costs of tools and tool support Costs of tools and tool support Costs of test case library maintenance Costs of testing & QA education associated with the product Costs of monitoring and oversight by the QA organization (if separate from the development and test organizations)
11. Re-work	Re-work effort (hours, as a percentage of the original coding hours) Re-worked LOC (source lines of code, as a percentage of the total delivered LOC) Re-worked software components (as a percentage of the total delivered components)
12. Reliability	Availability (percentage of time a system is available, versus the time the system is needed to be available) Mean time between failure

	(MTBF) Mean time to repair (MTTR) Reliability ratio (MTBF / MTTR) Number of product recalls or fix releases Number of production re-runs as a ratio of production runs

Fault and failure metrics

The goal of collecting fault and failure metrics is to be able to determine when the software is approaching failure-free execution. Minimally, both the number of faults found during testing (i.e., before delivery) and the failures (or other problems) reported by users after delivery are collected, summarized and analyzed to achieve this goal. Test strategy is highly relative to the effectiveness of fault metrics, because if the testing scenario does not cover the full functionality of the software, the software may pass all tests and yet be prone to failure once delivered. Usually, failure metrics are based upon customer information regarding failures found after release of the software. The failure data collected is therefore used to calculate failure density, Mean Time Between Failures (MTBF) or other parameters to measure or predict software reliability.

Software Reliability Improvement Techniques

Good engineering methods can largely improve software reliability. Before the deployment of software products, testing, verification and validation are necessary steps. Software testing is heavily used to trigger, locate and remove software defects. Software testing is still in its infant stage; testing is crafted to suit specific needs in various software development projects in an ad-hoc manner. Various analysis tools such as trend analysis, fault-tree analysis, Orthogonal Defect classification and formal methods, etc, can also be used to minimize the possibility of defect occurrence after release and therefore improve software reliability.

After deployment of the software product, field data can be gathered and analyzed to study the behavior of software defects. Fault tolerance or fault/failure forecasting techniques will be helpful techniques and guide rules to minimize fault occurrence or impact of the fault on the system.