Beginners Guide To Software Testing

- Padmini C

Table of Contents:

1. Overview 5
The Big Picture 5
What is software? Why should it be tested? 6
What is Quality? How important is it? 6
What exactly does a software tester do? 7
What makes a good tester? 8
Guidelines for new testers 9
2. Introduction 11
Software Life Cycle 11
Various Life Cycle Models 12
Software Testing Life Cycle 13
What is a bug? Why do bugs occur? 15
Bug Life Cycle 16
Cost of fixing bugs 17
When can testing be stopped/reduced? 18
3. Software Testing Levels, Types, Terms and Definitions 19
Testing Levels and Types 19

Testing Terms 22
Most common software errors 23
Types of errors with examples 23
5. The Test Planning Process 25
What is a Test Strategy? What are its Components? 25
Test Planning – Sample Structure 25
Major Test Planning Tasks 26
6. Test Case Development 27
General Guidelines 27
Test Case – Sample Structure 27
Test Case Design Techniques 28
What is a Use Case? 30
7. Defect Tracking 31
What is a defect? 31
What are the defect categories? 31
How is a defect reported? 32
How descriptive should your bug/defect report be? 32
What does the tester do when the defect is fixed? 33
8. Types of Test Reports 34
9. Software Test Automation 35

Approaches to Automation 36

Choosing the right tool 37

Top Ten Challenges of Software Test Automation 37

10. Introduction to Software Standards 38

Six Sigma 38

ISO 39

11. Software Testing Certifications 40

12. Facts about Software Engineering 41

1. Overview

The Big Picture

All software problems can be termed as bugs. A software bug usually occurs when the software does not do what it is intended to do or does something that it is not intended to do. Flaws in specifications, design, code or other reasons can cause these bugs. Identifying and fixing bugs in the early stages of the software is very important as the cost of fixing bugs grows over time. So, the goal of a software tester is to find bugs and find them as early as possible and make sure they are fixed.

Testing is context-based and risk-driven. It requires a methodical and disciplined approach to finding bugs. A good software tester needs to build credibility and possess the attitude to be explorative, troubleshooting, relentless, creative, diplomatic and persuasive.

As against the perception that testing starts only after the completion of coding phase, it actually begins even before the first line of code can be written. In the life cycle of the conventional software product, testing begins at the stage when the specifications are written, i.e. from testing the *product specifications* or *product spec*. Finding bugs at this stage can save huge amounts of time and money.

Once the specifications are well understood, you are required to design and execute the test cases. Selecting the appropriate technique that reduces the number of tests that cover a feature is one of the most important things that you need to take into consideration while designing these test cases. Test cases need to be designed to cover all aspects of the software, i.e. security, database, functionality (critical and general) and the user interface. Bugs originate when the test cases are executed.

As a tester you might have to perform testing under different circumstances, i.e. the application could be in the initial stages or undergoing rapid changes, you have less than enough time to test, the product might be developed using a life cycle model that does not support much of formal testing or retesting. Further, testing using different operating systems, browsers and the configurations are to be taken care of.

Reporting a bug may be the most important and sometimes the most difficult task that you as a software tester will perform. By using various tools and clearly communicating to the developer, you can ensure that the bugs you find are fixed.

Using automated tools to execute tests, run scripts and tracking bugs improves efficiency and effectiveness of your tests. Also, keeping pace with the latest developments in the field will augment your career as a software test engineer.

What is software? Why should it be tested?

Software is a series of instructions for the computer that perform a particular task, called a program; the two major categories of software are system software and application software. System software is made up of control programs. Application software is any program that processes data for the user (spreadsheet, word processor, payroll, etc.).

A software product should only be released after it has gone through a proper process of development, testing and bug fixing. Testing looks at areas such as performance, stability and error handling by setting up test scenarios under controlled conditions and assessing the results. This is why exactly any software has to be tested. It is important to note that software is mainly tested to see that it meets the customers' needs and that it conforms to the standards. It is a usual norm that software is considered of good quality if it meets the user requirements.

What is Quality? How important is it?

Quality can briefly be defined as "a degree of excellence". High quality software usually conforms to the user requirements. A customer's idea of quality may cover a breadth of features - conformance to specifications, good performance on platform(s)/configurations, completely meets operational requirements (even if not specified!), compatibility to all the end-user equipment, no negative impact on existing end-user base at introduction time.

Quality software saves good amount of time and money. Because software will have fewer defects, this saves time during testing and maintenance phases. Greater reliability contributes to an immeasurable increase in customer satisfaction as well as lower maintenance costs. Because maintenance represents a large portion of all software costs, the overall cost of the project will most likely be lower than similar projects.

Following are two cases that demonstrate the importance of software quality:

Ariane 5 crash June 4, 1996

- Maiden flight of the European Ariane 5 launcher crashed about 40 seconds after takeoff
- Loss was about half a billion dollars
    - Explosion was the result of a software error
    - Uncaught exception due to floating-point error: conversion from a 64-bit integer to a 16-bit signed integer applied to a larger than expected

number
- Module was re-used without proper testing from Ariane 4
- Error was not supposed to happen with Ariane 4
- No exception handler

Mars Climate Orbiter - September 23, 1999

- Mars Climate Orbiter, disappeared as it began to orbit Mars.
- Cost about $US 125-million
- Failure due to error in a transfer of information between a team in Colorado and a team in California
- One team used English units (e.g., inches, feet and pounds) while the other used metric units for a key spacecraft operation.

What exactly does a software tester do?

Apart from exposing faults ("bugs") in a software product confirming that the program meets the program specification, as a test engineer you need to create test cases, procedures, scripts and generate data. You execute test procedures and scripts, analyze standards and evaluate results of system/integration/regression testing. You also...

- Speed up development process by identifying bugs at an early stage (e.g. specifications stage)
- Reduce the organization's risk of legal liability
- Maximize the value of the software
- Assure successful launch of the product, save money, time and reputation of the company by discovering bugs and design flaws at an early stage before failures occur in production, or in the field
- Promote continual improvement

What makes a good tester?

As software engineering is now being considered as a technical engineering profession, it is important that the software test engineer's posses certain traits with a relentless attitude to make them stand out. Here are a few.

- **Know the technology.** Knowledge of the technology in which the application is developed is an added advantage to any tester. It helps design better and powerful test cases basing on the weakness or flaws of the technology. Good testers know what it supports and what it doesn't, so concentrating on these lines will help them break the application quickly.
- **Perfectionist and a realist.** Being a perfectionist will help testers spot the problem and being a realist helps know at the end of the day which problems are really important problems. You will know which ones require a fix and which ones don't.
- **Tactful, diplomatic and persuasive.** Good software testers are tactful and know how to break the news to the developers. They are diplomatic while convincing the developers of the bugs and persuade them when necessary and have their bug(s) fixed. It is important to be critical of the issue and not let the person who developed the application be taken aback of the findings.

- **An explorer.** A bit of creativity and an attitude to take risk helps the testers venture into unknown situations and find bugs that otherwise will be looked over.
- **Troubleshoot.** Troubleshooting and figuring out why something doesn't work helps testers be confident and clear in communicating the defects to the developers.
- **Posses people skills and tenacity.** Testers can face a lot of resistance from programmers. Being socially smart and diplomatic doesn't mean being indecisive. The best testers are both-socially adept and tenacious where it matters.
- **Organized.** Best testers very well realize that they too can make mistakes and don't take chances. They are very well organized and have checklists, use files, facts and figures to support their findings that can be used as an evidence and double-check their findings.
- **Objective and accurate.** They are very objective and know what they report and so convey impartial and meaningful information that keeps politics and emotions out of message. Reporting inaccurate information is losing a little credibility. Good testers make sure their findings are accurate and reproducible.
- **Defects are valuable.** Good testers learn from them. Each defect is an opportunity to learn and improve. A defect found early substantially costs less when compared to the one found at a later stage. Defects can cause serious problems if not managed properly. Learning from defects helps – prevention of future problems, track improvements, improve prediction and estimation.

Guidelines for new testers

- **Testing can't show that bugs don't exist.** An important reason for testing is to prevent defects. You can perform your tests, find and report bugs, but at no point can you guarantee that there are no bugs.
- **It is impossible to test a program completely.** Unfortunately this is not possible even with the simplest program because – the number of inputs is very large, number of outputs is very large, number of paths through the software is very large, and the specification is subjective to frequent changes.
- **You can't guarantee quality.** As a software tester, you cannot test everything and are not responsible for the quality of the product. The main way that a tester can fail is to fail to report accurately a defect you have observed. It is important to remember that we seldom have little control over quality.
- **Target environment and intended end user.** Anticipating and testing the application in the environment user is expected to use is one of the major factors that should be considered. Also, considering if the application is a single user system or multi user system is important for demonstrating the ability for immediate readiness when necessary. The error case of Disney's Lion King illustrates this. Disney Company released its first multimedia CD-ROM game for children, *The Lion King Animated Storybook*. It was highly promoted and the sales were huge. Soon there were reports that buyers were unable to get the software to work. It worked on a few systems – likely the ones that the Disney programmers used to create the game – but not on the most common systems that the general public used.
- **No application is 100% bug free.** It is more reasonable to recognize there are priorities, which may leave some less critical problems unsolved or unidentified. Simple case is the Intel Pentium bug. Enter the following equation into your PC's calculator: (4195835 / 3145727) * 3145727 – 4195835. If the answer is zero, your computer is just fine. If you get anything else, you have an old Intel Pentium CPU with a floating-point division bug.
- **Be the customer.** Try to use the system as a lay user. To get a glimpse of this, get a person who has no idea of the application to use it for a while and you will be amazed to see the number of problems the person seem to come across. As you can see, there is no procedure involved. Doing this could actually cause the system to encounter an array of unexpected tests – repetition, stress, load, race etc.
- **Build your credibility.** Credibility is like quality that includes reliability, knowledge, consistency, reputation, trust, attitude and attention to detail. It is not instant but should be built over time and gives voice to the testers in the organization. Your keys to build credibility – identify your strengths and weaknesses, build good relations, demonstrate competency, and be willing to admit mistakes, re-assess and adjust.
- **Test what you observe.** It is very important that you test what you can observe and have access to. Writing creative test cases can help only when you have the

opportunity to observe the results. So, assume nothing.

- **Not all bugs you find will be fixed.** Deciding which bugs will be fixed and which won’t is a risk-based decision. Several reasons why your bug might not be fixed is when there is no enough time, the bug is dismissed for a new feature, fixing it might be very risky or it may not be worth it because it occurs infrequently or has a work around where the user can prevent or avoid the bug. Making a wrong decision can be disastrous.
- **Review competitive products.** Gaining a good insight into various products of the same kind and getting to know their functionality and general behavior will help you design different test cases and to understand the strengths and weaknesses of your application. This will also enable you to add value and suggest new features and enhancements to your product.
- **Follow standards and processes.** As a tester, your need to conform to the standards and guidelines set by the organization. These standards pertain to reporting hierarchy, coding, documentation, testing, reporting bugs, using automated tools etc.

2. Introduction

Software Life Cycle

The software life cycle typically includes the following: requirements analysis, design, coding, testing, installation and maintenance. In between, there can be a requirement to provide Operations and support activities for the product.

Requirements Analysis. Software organizations provide solutions to customer requirements by developing appropriate software that best suits their specifications. Thus, the life of software starts with origin of requirements. Very often, these requirements are vague, emergent and always subject to change.

Analysis is performed to - To conduct in depth analysis of the proposed project, to evaluate for technical feasibility, to discover how to partition the system, to identify which areas of the requirements need to be elaborated from the customer, to identify the impact of changes to the requirements, to identify which requirements should be allocated to which components.

Design and Specifications. The outcome of requirements analysis is the requirements specification. Using this, the overall design for the intended software is developed.

Activities in this phase - Perform Architectural Design for the software, Design Database (If applicable), Design User Interfaces, Select or Develop Algorithms (If Applicable), Perform Detailed Design.

Coding. The development process tends to run iteratively through these phases rather than linearly; several models (spiral, waterfall etc.) have been proposed to describe this process.

Activities in this phase - Create Test Data, Create Source, Generate Object Code, Create Operating Documentation, Plan Integration, Perform Integration

Testing. The process of using the developed system with the intent to find errors. Defects/flaws/bugs found at this stage will be sent back to the developer for a fix and have to be re-tested. This phase is iterative as long as the bugs are fixed to meet the requirements.

Activities in this phase - Plan Verification and Validation, Execute Verification and validation Tasks, Collect and Analyze Metric Data, Plan Testing, Develop Test Requirements, Execute Tests

Installation. The so developed and tested software will finally need to be installed at the client place. Careful planning has to be done to avoid problems to the user after installation is done.

Activities in this phase - Plan Installation, Distribution of Software, Installation of Software, Accept Software in Operational Environment.

Operation and Support. Support activities are usually performed by the organization that developed the software. Both the parties usually decide on these activities before the system is developed.

Activities in this phase - Operate the System, Provide Technical Assistance and Consulting, Maintain Support Request Log.

Maintenance. The process does not stop once it is completely implemented and installed at user place; this phase undertakes development of new features, enhancements etc.

Activities in this phase - Reapplying Software Life Cycle.

Various Life Cycle Models

The way you approach a particular application for testing greatly depends on the life cycle model it follows. This is because, each life cycle model places emphasis on different aspects of the software i.e. certain models provide good scope and time for testing whereas some others don't. So, the number of test cases developed, features covered, time spent on each issue depends on the life cycle model the application follows.

No matter what the life cycle model is, every application undergoes the same phases described above as its life cycle.

Following are a few software life cycle models, their advantages and disadvantages.

Waterfall Model	Prototyping Model	Spiral Model
Strengths:	**Strengths:**	**Strengths:**
•Emphasizes completion of one phase before moving on •Emphasises early planning, customer input, and design •Emphasises testing as an integral part of the life cycle	•Requirements can be set earlier and more reliably •Requirements can be communicated more clearly and completely between developers and clients •Requirements and design options can be investigated quickly and with low cost •More requirements and	• It promotes reuse of existing software in early stages of development. • Allows quality objectives to be formulated during development. • Provides preparation for eventual evolution of the software product. • Eliminates errors and unattractive alternatives early.

	design faults are caught early	• It balances resource expenditure. • Doesn't involve separate approaches for software development and software maintenance. • Provides a viable framework for integrated Hardware-software system development.
Weakness:	**Weakness:**	**Weakness:**
• Depends on capturing and freezing requirements early in the life cycle •Depends on separating requirements from design • Feedback is only from testing phase to any previous stage • Not feasible in some organizations • Emphasizes products rather than processes	•Requires a prototyping tool and expertise in using it – a cost for the development organization •The prototype may become the production system	• This process needs or usually associated with Rapid Application Development, which is very difficult practically. • The process is more difficult to manage and needs a very different approach as opposed to the waterfall model (Waterfall model has management techniques like GANTT charts to assess)

Software Testing Life Cycle

Software Testing Life Cycle consist of six (generic) phases: 1) Planning, 2) Analysis, 3) Design, 4) Construction, 5) Testing Cycles, 6) Final Testing and Implementation and 7) Post Implementation. Each phase in the life cycle is described with the respective activities.

Planning. Planning High Level Test plan, QA plan (quality goals), identify – reporting procedures, problem classification, acceptance criteria, databases for testing, measurement criteria (defect quantities/severity level and defect origin), project metrics and finally begin the schedule for project testing. Also, plan to maintain all

test cases (manual or automated) in a database.

Analysis. Involves activities that - develop functional validation based on Business Requirements (writing test cases basing on these details), develop test case format (time estimates and priority assignments), develop test cycles (matrices and timelines), identify test cases to be automated (if applicable), define area of stress and performance testing, plan the test cycles required for the project and regression testing, define procedures for data maintenance (backup, restore, validation), review documentation.

Design. Activities in the design phase - Revise test plan based on changes, revise test cycle matrices and timelines, verify that test plan and cases are in a database or requisite, continue to write test cases and add new ones based on changes, develop Risk Assessment Criteria, formalize details for Stress and Performance testing, finalize test cycles (number of test case per cycle based on time estimates per test case and priority), finalize the Test Plan, (estimate resources to support development in unit testing).

Construction (Unit Testing Phase). Complete all plans, complete Test Cycle matrices and timelines, complete all test cases (manual), begin Stress and Performance testing, test the automated testing system and fix bugs, (support development in unit testing), run QA acceptance test suite to certify software is ready to turn over to QA.

Test Cycle(s) / Bug Fixes (Re-Testing/System Testing Phase). Run the test cases (front and back end), bug reporting, verification, and revise/add test cases as required.

Final Testing and Implementation (Code Freeze Phase). Execution of all front end test cases - manual and automated, execution of all back end test cases - manual and automated, execute all Stress and Performance tests, provide on-going defect tracking metrics, provide on-going complexity and design metrics, update estimates for test cases and test plans, document test cycles, regression testing, and update accordingly.

Post Implementation. Post implementation evaluation meeting can be conducted to review entire project. Activities in this phase - Prepare final Defect Report and associated metrics, identify strategies to prevent similar problems in future project, automation team - 1) Review test cases to evaluate other cases to be automated for regression testing, 2) Clean up automated test cases and variables, and 3) Review process of integrating results from automated testing in with results from manual testing.

What is a bug? Why do bugs occur?

A software bug may be defined as a coding error that causes an unexpected defect, fault, flaw, or imperfection in a computer program. In other words, if a program does not perform as intended, it is most likely a **bug**.

There are bugs in software due to unclear or constantly changing requirements, software complexity, programming errors, timelines, errors in bug tracking, communication gap, documentation errors, deviation from standards etc.

- Unclear software requirements are due to miscommunication as to what the software should or shouldn't do. In many occasions, the customer may not be completely clear as to how the product should ultimately function. This is especially true when the software is a developed for a completely new product. Such cases usually lead to a lot of misinterpretations from any or both sides.
- Constantly changing software requirements cause a lot of confusion and pressure both on the development and testing teams. Often, a new feature added or existing feature removed can be linked to the other modules or components in the software. Overlooking such issues causes bugs.
- Also, fixing a bug in one part/component of the software might arise another in a different or same component. Lack of foresight in anticipating such issues can cause serious problems and increase in bug count. This is one of the major issues because of which bugs occur since developers are very often subject to pressure related to timelines; frequently changing requirements, increase in the number of bugs etc.
- Designing and re-designing, UI interfaces, integration of modules, database management all these add to the complexity of the software and the system as a whole.
- Fundamental problems with software design and architecture can cause problems in programming. Developed software is prone to error as programmers can make mistakes too. As a tester you can check for, data reference/declaration errors, control flow errors, parameter errors, input/output errors etc.
- Rescheduling of resources, re-doing or discarding already completed work, changes in hardware/software requirements can affect the software too. Assigning a new developer to the project in midway can cause bugs. This is possible if proper coding standards have not been followed, improper code documentation, ineffective knowledge transfer etc. Discarding a portion of the existing code might just leave its trail behind in other parts of the software; overlooking or not eliminating such code can cause bugs. Serious bugs can especially occur with larger projects, as it gets tougher to identify the problem area.
- Programmers usually tend to rush as the deadline approaches closer. This is the time when most of the bugs occur. It is possible that you will be able to spot bugs of all types and severity.
- Complexity in keeping track of all the bugs can again cause bugs by itself. This gets

harder when a bug has a very complex life cycle i.e. when the number of times it has been closed, re-opened, not accepted, ignored etc goes on increasing.

Bug Life Cycle

Bug Life Cycle starts with an unintentional software bug/behavior and ends when the assigned developer fixes the bug. A bug when found should be communicated and assigned to a developer that can fix it. Once fixed, the problem area should be re-tested. Also, confirmation should be made to verify if the fix did not create problems elsewhere. In most of the cases, the life cycle gets very complicated and difficult to track making it imperative to have a bug/defect tracking system in place.

See Chapter 7 – Defect Tracking

Following are the different phases of a Bug Life Cycle:

Open: A bug is in *Open* state when a tester identifies a problem area

Accepted: The bug is then assigned to a developer for a fix. The developer then accepts if valid.

Not Accepted/Won't fix: If the developer considers the bug as low level or does not accept it as a bug, thus pushing it into Not *Accepted*/*Won't fix* state.

Such bugs will be assigned to the project manager who will decide if the bug needs a fix. If it needs, then assigns it back to the developer, and if it doesn't, then assigns it back to the tester who will have to close the bug.

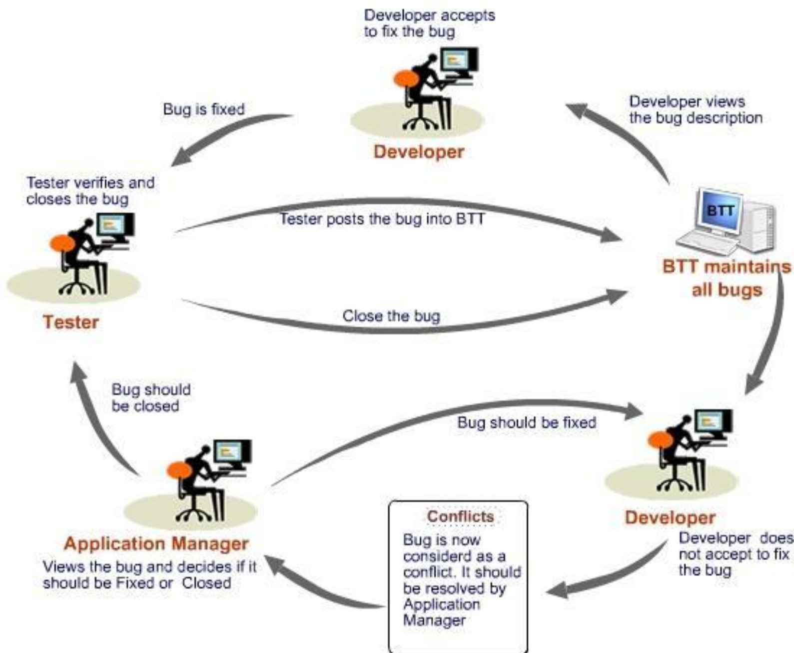
Pending: A bug accepted by the developer may not be fixed immediately. In such cases, it can be put under *Pending* state.

Fixed: Programmer will fix the bug and resolves it as *Fixed*.

Close: The fixed bug will be assigned to the tester who will put it in the *Close* state.

Re-Open: Fixed bugs can be re-opened by the testers in case the fix produces problems elsewhere.

The image below clearly shows the Bug Life Cycle and how a bug can be tracked using Bug Tracking Tools (BTT)



Cost of fixing bugs

Costs are logarithmic; they increase in size tenfold as the time increases. A bug found and fixed during the early stages – requirements or product spec stage can be fixed by a brief interaction with the concerned and might cost next to nothing.

During coding, a swiftly spotted mistake may take only very less effort to fix. During integration testing, it costs the paperwork of a bug report and a formally documented fix, as well as the delay and expense of a re-test.

During system testing it costs even more time and may delay delivery. Finally, during operations it may cause anything from a nuisance to a system failure, possibly with catastrophic consequences in a safety-critical system such as an aircraft or an emergency service.

When can testing be stopped/reduced?

It is difficult to determine when exactly to stop testing. Here are a few common factors that help you decide when you can stop or reduce testing:

- Deadlines (release deadlines, testing deadlines, etc.)
- Test cases completed with certain percentage passed
- Test budget depleted
- Coverage of code/functionality/requirements reaches a specified point
- Bug rate falls below a certain level
- Beta or alpha testing period ends

3. Software Testing Levels, Types, Terms and Definitions

Testing Levels and Types

There are basically three levels of testing i.e. Unit Testing, Integration Testing and System Testing. Various types of testing come under these levels.

Unit Testing

To verify a single program or a section of a single program

Integration Testing

To verify interaction between system components
Prerequisite: unit testing completed on all components that compose a system.

System Testing

To verify and validate behaviors of the entire system against the original system objectives.

Software testing is a process that identifies the correctness, completeness, and quality of software.

Following is a list of various types of software testing and their definitions in a random order:

- **Formal Testing:** Performed by test engineers
- **Informal Testing:** Performed by the developers
- **Manual Testing:** That part of software testing that requires human input, analysis, or evaluation.
- **Automated Testing:** Software testing that utilizes a variety of tools to automate the testing process. Automated testing still requires a skilled quality assurance professional with knowledge of the automation tools and the software being tested to set up the test cases.
- **Black box Testing:** Testing software without any knowledge of the back-end of the system, structure or language of the module being tested. Black box test cases are written from a definitive source document, such as a specification or requirements document.
- **White box Testing:** Testing in which the software tester has knowledge of the back-end, structure and language of the software, or at least its purpose.
- **Unit Testing:** Unit testing is the process of testing a particular complied program,

i.e., a window, a report, an interface, etc. independently as a stand-alone component/program. The types and degrees of unit tests can vary among modified and newly created programs. Unit testing is mostly performed by the programmers who are also responsible for the creation of the necessary unit test data.

- **Incremental Testing:** Incremental testing is partial testing of an incomplete product. The goal of incremental testing is to provide an early feedback to software developers.
- **System Testing:** System testing is a form of black box testing. The purpose of system testing is to validate an application's accuracy and completeness in performing the functions as designed.

- **Integration Testing:** Testing two or more modules or functions together with the intent of finding interface defects between the modules/functions.
- **System Integration Testing:** Testing of software components that have been distributed across multiple platforms (e.g., client, web server, application server, and database server) to produce failures caused by system integration defects (i.e. defects involving distribution and back-
- **Functional Testing:** Verifying that a module functions as stated in the specification and establishing confidence that a program does what it is supposed to do.
- **End-to-end Testing:** Similar to system testing - testing a complete application in a situation that mimics real world use, such as interacting with a database, using network communication, or interacting with other hardware, application, or system.
- **Sanity Testing:** Sanity testing is performed whenever cursory testing is sufficient to prove the application is functioning according to specifications. This level of testing is a subset of regression testing. It normally includes testing basic GUI functionality to demonstrate connectivity to the database, application servers, printers, etc.
- **Regression Testing:** Testing with the intent of determining if bug fixes have been successful and have not created any new problems.
- **Acceptance Testing:** Testing the system with the intent of confirming readiness of the product and customer acceptance. Also known as User Acceptance Testing.
- **Adhoc Testing:** Testing without a formal test plan or outside of a test plan. With some projects this type of testing is carried out as an addition to formal testing. Sometimes, if testing occurs very late in the development cycle, this will be the only kind of testing that can be performed – usually done by skilled testers. Sometimes ad hoc testing is referred to as exploratory testing.
- **Configuration Testing:** Testing to determine how well the product works with a broad range of hardware/peripheral equipment configurations as well as on different operating systems and software.
- **Load Testing:** Testing with the intent of determining how well the product handles competition for system resources. The competition may come in the form of network traffic, CPU utilization or memory allocation.
- **Stress Testing:** Testing done to evaluate the behavior when the system is pushed

beyond the breaking point. The goal is to expose the weak links and to determine if the system manages to recover gracefully.

- **Performance Testing:** Testing with the intent of determining how efficiently a product handles a variety of events. Automated test tools geared specifically to test and fine-tune performance are used most often for this type of testing.
- **Usability Testing:** Usability testing is testing for 'user-friendliness'. A way to evaluate and measure how users interact with a software product or site. Tasks are given to users and observations are made.
- **Installation Testing:** Testing with the intent of determining if the product is compatible with a variety of platforms and how easily it installs.
- **Recovery/Error Testing:** Testing how well a system recovers from crashes, hardware failures, or other catastrophic problems.
- **Security Testing:** Testing of database and network software in order to keep company data and resources secure from mistaken/accidental users, hackers, and other malevolent attackers.
- **Penetration Testing:** Penetration testing is testing how well the system is protected against unauthorized internal or external access, or willful damage. This type of testing usually requires sophisticated testing techniques.
- **Compatibility Testing:** Testing used to determine whether other system software components such as browsers, utilities, and competing software will conflict with the software being tested.
- **Exploratory Testing:** Any testing in which the tester dynamically changes what they're doing for test execution, based on information they learn as they're executing their tests.
- **Comparison Testing:** Testing that compares software weaknesses and strengths to those of competitors' products.
- **Alpha Testing:** Testing after code is mostly complete or contains most of the functionality and prior to reaching customers. Sometimes a selected group of users are involved. More often this testing will be performed in-house or by an outside testing firm in close cooperation with the software engineering department.
- **Beta Testing:** Testing after the product is code complete. Betas are often widely distributed or even distributed to the public at large.
- **Gamma Testing:** Gamma testing is testing of software that has all the required features, but it did not go through all the in-house quality checks.
- **Mutation Testing:** A method to determine to test thoroughness by measuring the extent to which the test cases can discriminate the program from slight variants of the program.
- **Independent Verification and Validation (IV&V):** The process of exercising software with the intent of ensuring that the software system meets its requirements and user expectations and doesn't fail in an unacceptable manner. The individual or group doing this work is not part of the group or organization that developed the software.

- **Pilot Testing:** Testing that involves the users just before actual release to ensure that users become familiar with the release contents and ultimately accept it. Typically involves many users, is conducted over a short period of time and is tightly controlled. (See beta testing)
- **Parallel/Audit Testing:** Testing where the user reconciles the output of the new system to the output of the current system to verify the new system performs the operations correctly.
- **Glass Box/Open Box Testing:** Glass box testing is the same as white box testing. It is a testing approach that examines the application's program structure, and derives test cases from the application's program logic.
- **Closed Box Testing:** Closed box testing is same as black box testing. A type of testing that considers only the functionality of the application.
- **Bottom-up Testing:** Bottom-up testing is a technique for integration testing. A test engineer creates and uses test drivers for components that have not yet been developed, because, with bottom-up testing, low-level components are tested first. The objective of bottom-up testing is to call low-level components first, for testing purposes.
- **Smoke Testing:** A random test conducted before the delivery and after complete testing.

Testing Terms

- **Bug:** A software bug may be defined as a coding error that causes an unexpected defect, fault or flaw. In other words, if a program does not perform as intended, it is most likely a bug.
- **Error:** A mismatch between the program and its specification is an error in the program.
- **Defect:** Defect is the variance from a desired product attribute (it can be a wrong, missing or extra data). It can be of two types – Defect from the product or a variance from customer/user expectations. It is a flaw in the software system and has no impact until it affects the user/customer and operational system. 90% of all the defects can be caused by process problems.

- **Failure:** A defect that causes an error in operation or negatively impacts a user/ customer.
- **Quality Assurance:** Is oriented towards preventing defects. Quality Assurance ensures all parties concerned with the project adhere to the process and procedures, standards and templates and test readiness reviews.
- **Quality Control:** *quality control* or *quality engineering* is a set of measures taken to ensure that defective products or services are not produced, and that the design meets performance requirements.

- **Verification:** Verification ensures the product is designed to deliver all functionality to the customer; it typically involves reviews and meetings to evaluate documents, plans, code, requirements and specifications; this can be done with checklists, issues lists, walkthroughs and inspection meetings.
- **Validation:** Validation ensures that functionality, as defined in requirements, is the intended behavior of the product; validation typically involves actual testing and takes place after
Verifications are completed.

Most common software errors

Following are the most common software errors that aid you in software testing. This helps you to identify errors systematically and increases the efficiency and productivity of software testing.

Types of errors with examples

- **User Interface Errors:** Missing/Wrong Functions Doesn't do what the user expects, Missing information, Misleading, Confusing information, Wrong content in Help text, Inappropriate error messages. Performance issues - Poor responsiveness, Can't redirect output, Inappropriate use of key board
- **Error Handling:** Inadequate - protection against corrupted data, tests of user input, version control; Ignores – overflow, data comparison, Error recovery – aborting errors, recovery from hardware problems.
- **Boundary related errors:** Boundaries in loop, space, time, memory, mishandling of cases outside boundary.
- **Calculation errors:** Bad Logic, Bad Arithmetic, Outdated constants, Calculation errors, incorrect conversion from one data representation to another, Wrong formula, incorrect approximation.
- **Initial and Later states:** Failure to - set data item to zero, to initialize a loop-control variable, or re-initialize a pointer, to clear a string or flag, Incorrect initialization.
- **Control flow errors:** Wrong returning state assumed, Exception handling based exits, Stack underflow/overflow, Failure to block or un-block interrupts, Comparison sometimes yields wrong result, Missing/wrong default, and Data Type errors.
- **Errors in Handling or Interpreting Data:** Un-terminated null strings, overwriting a file after an error exit or user abort.
- **Race Conditions:** Assumption that one event or task finished before another begins, Resource races, Tasks starts before its prerequisites are met, Messages cross

or don't arrive in the order sent.

- **Load Conditions:** Required resources are not available, No available large memory area, Low priority tasks not put off, doesn’t erase old files from mass storage, and doesn’t return unused memory.
- **Hardware:** Wrong Device, Device unavailable, Underutilizing device intelligence, Misunderstood status or return code, Wrong operation or instruction codes.
- **Source, Version and ID Control:** No Title or version ID, Failure to update multiple copies of data or program files.
- **Testing Errors:** Failure to notice/report a problem, Failure to use the most promising test case, Corrupted data files, Misinterpreted specifications or documentation, Failure to make it clear how to reproduce the problem, Failure to check for unresolved problems just before release, Failure to verify fixes, Failure to provide summary report.

5. The Test Planning Process

What is a Test Strategy? What are its Components?

Test Policy - A document characterizing the organization's philosophy towards software testing.

Test Strategy - A high-level document defining the test phases to be performed and the testing within those phases for a program. It defines the process to be followed in each project. This sets the standards for the processes, documents, activities etc. that should be followed for each project.

For example, if a product is given for testing, you should decide if it is better to use black-box testing or white-box testing and if you decide to use both, when will you apply each and to which part of the software? All these details need to be specified in the Test Strategy.

Project Test Plan - a document defining the test phases to be performed and the testing within those phases for a particular project.

A Test Strategy should cover more than one project and should address the following issues: An approach to testing high risk areas first, Planning for testing, How to improve the process based on previous testing, Environments/data used, Test management - Configuration management, Problem management, What Metrics are followed, Will the tests be automated and if so which tools will be used, What are the Testing Stages and Testing Methods, Post Testing Review process, Templates.

Test planning needs to start as soon as the project requirements are known. The first document that needs to be produced then is the Test Strategy/Testing Approach that sets the high level approach for testing and covers all the other elements mentioned above.

Test Planning – Sample Structure

Once the approach is understood, a detailed test plan can be written. Usually, this test plan can be written in different styles. Test plans can completely differ from project to project in the same organization.

IEEE SOFTWARE TEST DOCUMENTATION Std 829-1998 - TEST PLAN

Purpose
To describe the scope, approach, resources, and schedule of the testing activities. To

identify the items being tested, the features to be tested, the testing tasks to be performed, the personnel responsible for each task, and the risks associated with this plan.

OUTLINE
A test plan shall have the following structure:

- Test plan identifier. A unique identifier assign to the test plan.
- Introduction: Summarized the software items and features to be tested and the need for them to be included.
- Test items: Identify the test items, their transmittal media which impact their
- Features to be tested
- Features not to be tested
- Approach
- Item pass/fail criteria
- Suspension criteria and resumption requirements
- Test deliverables

- Testing tasks
- Environmental needs
- Responsibilities
- Staffing and training needs
- Schedule
- Risks and contingencies
- Approvals

Major Test Planning Tasks

Like any other process in software testing, the major tasks in test planning are to – Develop Test Strategy, Critical Success Factors, Define Test Objectives, Identify Needed Test Resources, Plan Test Environment, Define Test Procedures, Identify Functions To Be Tested, Identify Interfaces With Other Systems or Components, Write Test Scripts, Define Test Cases, Design Test Data, Build Test Matrix, Determine Test Schedules, Assemble Information, Finalize the Plan.

6. Test Case Development

A *test case* is a detailed procedure that fully tests a feature or an aspect of a feature. While the test plan describes what to test, a test case describes how to perform a particular test. You need to develop test cases for each test listed in the test plan.

General Guidelines

As a tester, the best way to determine the compliance of the software to requirements is by designing effective test cases that provide a thorough test of a unit. Various test case design techniques enable the testers to develop effective test cases. Besides, implementing the design techniques, every tester needs to keep in mind general guidelines that will aid in test case design:

a. The purpose of each test case is to run the test in the simplest way possible. [Suitable techniques - Specification derived tests, Equivalence partitioning]
b. Concentrate initially on positive testing i.e. the test case should show that the software does what it is intended to do. [Suitable techniques - Specification derived tests, Equivalence partitioning, State-transition testing]
c. Existing test cases should be enhanced and further test cases should be designed to show that the software does not do anything that it is not specified to do i.e. Negative Testing [Suitable techniques - Error guessing, Boundary value analysis, Internal boundary value testing, State-transition testing]
d. Where appropriate, test cases should be designed to address issues such as performance, safety requirements and security requirements [Suitable techniques - Specification derived tests]
e. Further test cases can then be added to the unit test specification to achieve specific test coverage objectives. Once coverage tests have been designed, the test procedure can be developed and the tests executed [Suitable techniques - Branch testing, Condition testing, Data definition-use testing, State-transition testing]

Test Case – Sample Structure

The manner in which a test case is depicted varies between organizations. Anyhow, many test case templates are in the form of a table, for example, a 5-column table with fields:

Test Case ID	Test Case Description	Test Dependency/ Setup	Input Data Requirements/ Steps	Expected Results	Pass/Fail

Test Case Design Techniques

The test case design techniques are broadly grouped into two categories: Black box techniques, White box techniques and other techniques that do not fall under either category.

Black Box (Functional)	**White Box (Structural)**	**Other**
- Specification derived tests - Equivalence partitioning - Boundary Value Analysis - State-Transition Testing	- Branch Testing - Condition Testing - Data Definition - Use Testing - Internal boundary value testing	- Error guessing

Specification Derived Tests

As the name suggests, test cases are designed by walking through the relevant specifications. It is a positive test case design technique.

Equivalence Partitioning

Equivalence partitioning is the process of taking all of the possible test values and placing them into classes (partitions or groups). Test cases should be designed to test one value from each class. Thereby, it uses fewest test cases to cover the maximum input requirements.

For example, if a program accepts integer values only from 1 to 10. The possible test cases for such a program would be the range of all integers. In such a program, all integers up to 0 and above 10 will cause an error. So, it is reasonable to assume that if 11 will fail, all values above it will fail and vice versa.

If an input condition is a range of values, let one valid equivalence class is the range (0 or 10 in this example). Let the values below and above the range be two respective invalid equivalence values (i.e. -1 and 11). Therefore, the above three partition values can be used as test cases for the above example.

Boundary Value Analysis

This is a selection technique where the test data are chosen to lie along the boundaries of the input domain or the output range. This technique is often called as stress testing and incorporates a degree of negative testing in the test design by anticipating that errors will occur at or around the partition boundaries.

For example, a field is required to accept amounts of money between $0 and $10. As a tester, you need to check if it means up to and including $10 and $9.99 and if $10 is acceptable. So, the boundary values are $0, $0.01, $9.99 and $10.

Now, the following tests can be executed. A negative value should be rejected, 0 should be accepted (this is on the boundary), $0.01 and $9.99 should be accepted, null and $10 should be rejected. In this way, it uses the same concept of partitions as equivalence partitioning.

State Transition Testing

As the name suggests, test cases are designed to test the transition between the states by creating the events that cause the transition.

Branch Testing

In branch testing, test cases are designed to exercise control flow branches or decision points in a unit. This is usually aimed at achieving a target level of Decision Coverage. Branch Coverage, need to test both branches of IF and ELSE. All branches and compound conditions (e.g. loops and array handling) within the branch should be exercised at least once.

Condition Testing

The object of condition testing is to design test cases to show that the individual components of logical conditions and combinations of the individual components are correct. Test cases are designed to test the individual elements of logical expressions, both within branch conditions and within other expressions in a unit.

Data Definition – Use Testing

Data definition-use testing designs test cases to test pairs of data definitions and uses. Data definition is anywhere that the value of a data item is set. Data use is anywhere that a data item is read or used. The objective is to create test cases that will drive execution through paths between specific definitions and uses.

Internal Boundary Value Testing

In many cases, partitions and their boundaries can be identified from a functional specification for a unit, as described under equivalence partitioning and boundary value analysis above. However, a unit may also have internal boundary values that can only be identified from a structural specification.

Error Guessing

It is a test case design technique where the testers use their experience to guess the possible errors that might occur and design test cases accordingly to uncover them.

Using any or a combination of the above described test case design techniques; you can develop effective test cases.

What is a Use Case?

A use case describes the system's behavior under various conditions as it responds to a request from one of the users. The user initiates an interaction with the system to accomplish some goal. Different sequences of behavior, or scenarios, can unfold, depending on the particular requests made and conditions surrounding the requests. The use case collects together those different scenarios.

Use cases are popular largely because they tell coherent stories about how the system will behave in use. The users of the system get to see just what this new system will be and get to react early.

7. Defect Tracking

What is a defect?

As discussed earlier, defect is the variance from a desired product attribute (it can be a wrong, missing or extra data). It can be of two types – Defect from the product or a variance from customer/user expectations. It is a flaw in the software system and has no impact until it affects the user/customer and operational system.

What are the defect categories?

With the knowledge of testing so far gained, you can now be able to categorize the defects you have found. Defects can be categorized into different types basing on the core issues they address. Some defects address security or database issues while others may refer to functionality or UI issues.

Security Defects: Application security defects generally involve improper handling of data sent from the user to the application. These defects are the most severe and given highest priority for a fix.

Examples:

- Authentication: Accepting an invalid username/password
- Authorization: Accessibility to pages though permission not given

Data Quality/Database Defects: Deals with improper handling of data in the database.

Examples:

- Values not deleted/inserted into the database properly
- Improper/wrong/null values inserted in place of the actual values

Critical Functionality Defects: The occurrence of these bugs hampers the crucial functionality of the application.

Examples:

- Exceptions

Functionality Defects: These defects affect the functionality of the application.

Examples:

- All Javascript errors
- Buttons like Save, Delete, Cancel not performing their intended functions
- A missing functionality (or) a feature not functioning the way it is intended to
- Continuous execution of loops

User Interface Defects: As the name suggests, the bugs deal with problems related to UI are usually considered less severe.

Examples:

- Improper error/warning/UI messages
- Spelling mistakes
- Alignment problems

How is a defect reported?

Once the test cases are developed using the appropriate techniques, they are executed which is when the bugs occur. It is very important that these bugs be reported as soon as possible because, the earlier you report a bug, the more time remains in the schedule to get it fixed.

Simple example is that you report a wrong functionality documented in the Help file a few months before the product release, the chances that it will be fixed are very high. If you report the same bug few hours before the release, the odds are that it won't be fixed. The bug is still the same though you report it few months or few hours before the release, but what matters is the time.

It is not just enough to find the bugs; these should also be reported/communicated clearly and efficiently, not to mention the number of people who will be reading the defect.

Defect tracking tools (also known as bug tracking tools, issue tracking tools or problem trackers) greatly aid the testers in reporting and tracking the bugs found in software applications. They provide a means of consolidating a key element of project information in one place. Project managers can then see which bugs have been fixed, which are outstanding and how long it is taking to fix defects. Senior management can use reports to understand the state of the development process.

How descriptive should your bug/defect report be?

You should provide enough detail while reporting the bug keeping in mind the people who will use it – test lead, developer, project manager, other testers, new testers assigned etc. This means that the report you will write should be concise, straight and clear. Following are the details your report should contain:

- Bug Title
- Bug identifier (number, ID, etc.)

- The application name or identifier and version
- The function, module, feature, object, screen, etc. where the bug occurred
- Environment (OS, Browser and its version)
- Bug Type or Category/Severity/Priority
  - Bug Category: Security, Database, Functionality (Critical/General), UI
  - Bug Severity: Severity with which the bug affects the application – Very High, High, Medium, Low, Very Low
  - Bug Priority: Recommended priority to be given for a fix of this bug – P0, P1, P2, P3, P4, P5 (P0-Highest, P5-Lowest)
- Bug status (Open, Pending, Fixed, Closed, Re-Open)
- Test case name/number/identifier
- Bug description
- Steps to Reproduce
- Actual Result
- Tester Comments

What does the tester do when the defect is fixed?

Once the reported defect is fixed, the tester needs to re-test to confirm the fix. This is usually done by executing the possible scenarios where the bug can occur. Once retesting is completed, the fix can be confirmed and the bug can be closed. This marks the end of the bug life cycle.

8. Types of Test Reports

The documents outlined in the IEEE Standard of Software Test Documentation covers test planning, test specification, and test reporting.

Test reporting covers four document types:

1. A **Test Item Transmittal Report** identifies the test items being transmitted for testing from the development to the testing group in the event that a formal beginning of test execution is desired

Details to be included in the report - Purpose, Outline, Transmittal-Report Identifier, Transmitted Items, Location, Status, and Approvals.

2. A **Test Log** is used by the test team to record what occurred during test execution

Details to be included in the report - Purpose, Outline, Test-Log Identifier, Description, Activity and Event Entries, Execution Description, Procedure Results, Environmental Information, Anomalous Events, Incident-Report Identifiers

3. A **Test Incident report** describes any event that occurs during the test execution that requires further investigation

Details to be included in the report - Purpose, Outline, Test-Incident-Report Identifier, Summary, Impact

4. A **test summary report** summarizes the testing activities associated with one or more test-design specifications

Details to be included in the report - Purpose, Outline, Test-Summary-Report Identifier, Summary, Variances, Comprehensiveness Assessment, Summary of Results, Summary of Activities, and Approvals.

9. Software Test Automation

Automating testing is no different from a programmer using a coding language to write programs to automate any manual process. One of the problems with testing large systems is that it can go beyond the scope of small test teams. Because only a small number of testers are available the coverage and depth of testing provided are inadequate for the task at hand.

Expanding the test team beyond a certain size also becomes problematic with increase in work over head. Feasible way to avoid this without introducing a loss of quality is through appropriate use of tools that can expand individual's capacity enormously while maintaining the focus (depth) of testing upon the critical elements.

Consider the following **factors that help determine the use of automated testing tools**:

- Examine your current testing process and determine where it needs to be adjusted for using automated test tools.
- Be prepared to make changes in the current ways you perform testing.
- Involve people who will be using the tool to help design the automated testing process.
- Create a set of evaluation criteria for functions that you will want to consider when using the automated test tool. These criteria may include the following:
  - Test repeatability
  - Criticality/risk of applications
  - Operational simplicity
  - Ease of automation
  - Level of documentation of the function (requirements, etc.)
- Examine your existing set of test cases and test scripts to see which ones are most applicable for test automation.
- Train people in basic test-planning skills.

Approaches to Automation

There are three broad options in Test Automation:

Full Manual	Partial Automation	Full Automation
Reliance on manual testing	Redundancy possible but requires duplication of effort	Reliance on automated testing
Responsive and flexible	Flexible	Relatively inflexible
Inconsistent	Consistent	Very consistent
Low implementation cost	-	High implementation cost
High repetitive cost	Automates repetitive tasks and high return tasks	Economies of scale in repetition, regression etc
Required for automation	-	-
Low skill requirements	-	High skill requirements

Fully manual testing has the benefit of being relatively cheap and effective. But as quality of the product improves the additional cost for finding further bugs becomes more expensive. Large scale manual testing also implies large scale testing teams with the related costs of space overhead and infrastructure. Manual testing is also far more responsive and flexible than automated testing but is prone to tester error through fatigue.

Fully automated testing is very consistent and allows the repetitions of similar tests at very little marginal cost. The setup and purchase costs of such automation are very high however and maintenance can be equally expensive. Automation is also relatively inflexible and requires rework in order to adapt to changing requirements.

Partial Automation incorporates automation only where the most benefits can be achieved. The advantage is that it targets specifically the tasks for automation and thus achieves the most benefit from them. It also retains a large component of manual testing which maintains the test team’s flexibility and offers redundancy by backing up automation with manual testing. The disadvantage is that it obviously does not provide as extensive benefits as either extreme solution.

Choosing the right tool

- Take time to define the tool requirements in terms of technology, process, applications, people skills, and organization.
- During tool evaluation, prioritize which test types are the most critical to your success and judge the candidate tools on those criteria.
- Understand the tools and their trade-offs. You may need to use a multi-tool solution to get higher levels of test-type coverage. For example, you will need to combine the capture/play-back tool with a load-test tool to cover your performance test cases.
- Involve potential users in the definition of tool requirements and evaluation criteria.
- Build an evaluation scorecard to compare each tool's performance against a common set of criteria. Rank the criteria in terms of relative importance to the organization.

Top Ten Challenges of Software Test Automation

1. Buying the Wrong Tool
2. Inadequate Test Team Organization
3. Lack of Management Support
4. Incomplete Coverage of Test Types by the selected tool
5. Inadequate Tool Training
6. Difficulty using the tool
7. Lack of a Basic Test Process or Understanding of What to Test
8. Lack of Configuration Management Processes
9. Lack of Tool Compatibility and Interoperability
10. Lack of Tool Availability

10. Introduction to Software Standards

Capability Maturity Model - Developed by the software community in 1986 with leadership from the SEI. The CMM describes the principles and practices underlying software process maturity. It is intended to help software organizations improve the maturity of their software processes in terms of an evolutionary path from ad hoc, chaotic processes to mature, disciplined software processes. The focus is on identifying key process areas and the exemplary practices that may comprise a disciplined software process.

What makes up the CMM? The CMM is organized into five maturity levels:

- Initial
- Repeatable
- Defined
- Manageable
- Optimizing

Except for Level 1, each maturity level decomposes into several key process areas that indicate the areas an organization should focus on to improve its software process.

Level 1 - Initial Level: Disciplined process, Standard, Consistent process, Predictable process, Continuously Improving process

Level 2 – Repeatable: Key practice areas - Requirements management, Software project planning, Software project tracking & oversight, Software subcontract management, Software quality assurance, Software configuration management

Level 3 – Defined: Key practice areas - Organization process focus, Organization process definition, Training program, integrated software management, Software product engineering, intergroup coordination, Peer reviews

Level 4 – Manageable: Key practice areas - Quantitative Process Management, Software Quality Management

Level 5 – Optimizing: Key practice areas - Defect prevention, Technology change management, Process change management

Six Sigma

Six Sigma is a quality management program to achieve "six sigma" levels of quality. It was pioneered by Motorola in the mid-1980s and has spread too many other manufacturing companies, notably General Electric Corporation (GE).

Six Sigma is a rigorous and disciplined methodology that uses data and statistical analysis to measure and improve a company's operational performance by identifying and eliminating "defects" from manufacturing to transactional and from product to service. Commonly defined as 3.4 defects per million opportunities, Six Sigma can be defined and understood at three distinct levels: metric, methodology and philosophy...

Training Sigma processes are executed by Six Sigma Green Belts and Six Sigma Black Belts, and are overseen by Six Sigma Master Black Belts.

ISO

ISO - International Organization for Standardization is a network of the national standards institutes of 150 countries, on the basis of one member per country, with a Central Secretariat in Geneva, Switzerland, that coordinates the system. ISO is a non-governmental organization. ISO has developed over 13, 000 International Standards on a variety of subjects.

11. Software Testing Certifications

Certification Information for Software QA and Test Engineers

CSQE - ASQ (American Society for Quality)’s program for CSQE (Certified Software Quality Engineer) - information on requirements, outline of required 'Body of Knowledge', listing of study references and more.

CSQA/CSTE - QAI (Quality Assurance Institute)'s program for CSQA (Certified Software Quality Analyst) and CSTE (Certified Software Test Engineer) certifications.

ISEB Software Testing Certifications - The British Computer Society maintains a program of 2 levels of certifications - ISEB Foundation Certificate, Practitioner Certificate.

ISTQB Certified Tester - The International Software Testing Qualifications Board is a part of the European Organization for Quality - Software Group, based in Germany. The certifications are based on experience, a training course and test. Two levels are available: Foundation and Advanced.

12. Facts about Software Engineering

Following are some facts that can help you gain a better insight into the realities of Software Engineering.

1. The best programmers are up to 28 times better than the worst programmers.
2. New tools/techniques cause an initial LOSS of productivity/quality.
3. The answer to a feasibility study is almost always “yes”.
4. A May 2002 report prepared for the National Institute of Standards and Technologies (NIST)(1) estimate the annual cost of software defects in the United States as $59.5 billion.
5. Reusable components are three times as hard to build
6. For every 25% increase in problem complexity, there is a 100% increase in solution complexity.
7. 80% of software work is intellectual. A fair amount of it is creative. Little of it is clerical.
8. Requirements errors are the most expensive to fix during production.
9. Missing requirements are the hardest requirement errors to correct.
10. Error-removal is the most time-consuming phase of the life cycle.
11. Software is usually tested at best at the 55-60% (branch) coverage level.
12. 100% coverage is still far from enough.
13. Rigorous inspections can remove up to 90% of errors before the first test case is run.

15. Maintenance typically consumes 40-80% of software costs. It is probably the most important life cycle phase of software.
16. Enhancements represent roughly 60% of maintenance costs.
17. There is no single best approach to software error removal.

Happy Testing!!!