SOFTWARE TESTING

ONE SHOULD HAVE A STRONG FOUNDATION

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

Introduction

What is Software Testing?

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

Software testing is primarily a broad process that is composed of several interlinked processes – verifying software completeness in regards to functional/business requirements, identifying technical defects and assessing software usability, performance, security, localization, compatibility, installation, etc.

- *to find whether the software met the specified requirements.*
- *process of preventing (pro-active) and then finding (reactive) defects in the software product.*
- *verify software completeness – functional, usability, security, compatibility, etc.*
- *provide stakeholders with information about the quality of the software product or service under test.*

Yes, software testing is **ALL** of this. It is **NOT** just about finding defects!

Why Software Testing?

Software Testing serves **dual purpose** – Ensuring that the product is built as per the requirements (well done!) and finding the shortcomings in parallel (the improvement areas). An enterprise can bring value to their customers only when the product delivered is ideal. And to achieve that, organizations have to make sure that users don't face any issues while using their product.

Software Testing is inherently considered a negative capability. Yes, we help to prevent, identify & rectify the defects but we are not done yet. What about the **positive outlook?** It's not like we always show defects. The most relevant purpose of Software Testing is to find success, i.e., confirm that the system is functioning as it was expected to work. Requirements are the starting point for any software development & testing project. Apart from the defects, we as testers need to make sure that the software, product or service is implemented as per the business requirements – be it functional, performance, security, usability, reliability, etc.

To summarize, you need software testing to *ensure that the software meets the requirements (customer satisfaction) + help in improving the overall product (prevent or identify the gaps/defects).*

Common Understanding

- **End-user**: Person who will actually the software or product. Just like we are the end-users for mobile banking app or net banking website.

- **Tech Company**: A technology company that will actually build the software/product. In above example, say a project team from Infosys will build the mobile app/website for HDFC Bank. Infosys here might also be called a 'vendor'.

- **Client/Customer**: Company, or an Individual who has the idea about what needs to be built and is ready to pay for it. In above example, Bank is the client/customer of the Tech company, like HDFC, ICICI, etc.

- **Project Team**: A team within the technology company that is responsible for delivering the software/product. Team includes everyone – business analyst, developer, testers, managers, etc.

- **Stakeholders**: EVERYBODY involved in the production of software/product, except for the end-users. i.e., Vendor, Client, Team, and any third-party.

The 7 Software Testing Principles

Principle 1 & 2

"Testing shows the presence of defects"

With Software Testing, we find defects but ***that doesn't mean post-testing the application is defect-free***. As we say in life – 'There is always a scope for improvement', similarly even after rigorous testing there is always a scope of defect occurring or further optimizing the application. The goal of testing is clear – to identify 'as many' defects as possible.

Even after rigorous testing, why do you think organizations have the famous Bug Bounty program encouraging end-users to identify *still-unidentified* defects.

Note: This doesn't mean you can use this principle as an excuse for poor quality, instead focus on designing the test cases that have maximum coverage and identify 'maximum' defects!

"Exhaustive testing in not possible"

Exhaustive as in complete or in-depth testing is not possible. Why? You don't have all the money, time & resources at your perusal – the budget is limited, timelines are defined, resource count is fixed, P&C are quite huge, etc. Even a simple text-box with 5 possible values need data tests, UI tests, security tests, compatibility tests, etc.

Keeping different factors in mind, it is not possible to test all the possible combinations, flows, data & scenarios.

What's the way out then? **Prioritize** based on the Risk analysis! Use different Test Design techniques to optimize test cases. Focus is to derive optimal amount of testing required based on the risks & priorities, such that it uncovers maximum defects focusing on the most important aspects first.

Principle 3, 4 & 5

"Early testing"

Software Testing should **start as early as possible** in the Software Development Life Cycle. Say client asks to develop a login page for his application. Developers build a page with Username & Password fields and deploy it for testing. During testing you observe that there is no 'Forgot Password' link provided. Come on, it's obvious! What do you do? Simple – raise a defect >> client says it's obvious >> requirements are detailed >> developer changes the code to fix it >> you retest it & close.

Now imagine Testing team is involved from the project start itself. What do you do? As soon as requirements are received, you raise an issue with business team about the ambiguous, missing or confusing requirements >> Business team details out the requirements clearly. Result – It is easier & cheaper to fix the issue if identified earlier in the cycle. Hope you got the point.

"Defect clustering"

Ever heard about the 'Pareto principle', also known as the 80–20 rule? It states that for many events, roughly 80% of the effects come from 20% of the causes. Applying this principle to Software Testing – **80% of the defects are found in 20% of the code** modules. Yeah! That's roughly true.

"The pesticide paradox"

If you keep running the same set of tests over and over again the software gets immune to testing, i.e., as the system evolves, many of the previously reported defects will have been fixed and the **old test cases cannot find any new defects**.

What's the way out then? To overcome this "Pesticide Paradox", the test cases need to be regularly reviewed & revised, adding new & different test cases to help find more defects.

Principle 6 & 7

"Testing is context dependent"

Context as in the environment, i.e., application type. All the developed software's are not identical. You might use different methodologies, approach, method, techniques and types of testing depending upon the application type, whether it's a Game, Mobile App, Cloud-based, Web-based, desktop application, embedded system, etc. After all, testing an ATM machine will be different from testing an e-Commerce site.

"Absence of errors fallacy"

Imagine you developed an e-commerce system & tested it completely. All the identified defects have been fixed & retested. 99% of the defects have been rectified. Management is pretty sure about the product quality with respect to defects. Now when the system is demonstrated to client, what if you get feedback saying "**Though it is so-called defect-free, but still this is not what I wanted**. I wanted a simple UI that can handle the user load"?

Yeah! Everybody in the Test team was confident about the product 'quality' (absence of errors) but at the end it proved to be false (fallacy) – the system is not usable; it doesn't fulfil client's expectations.

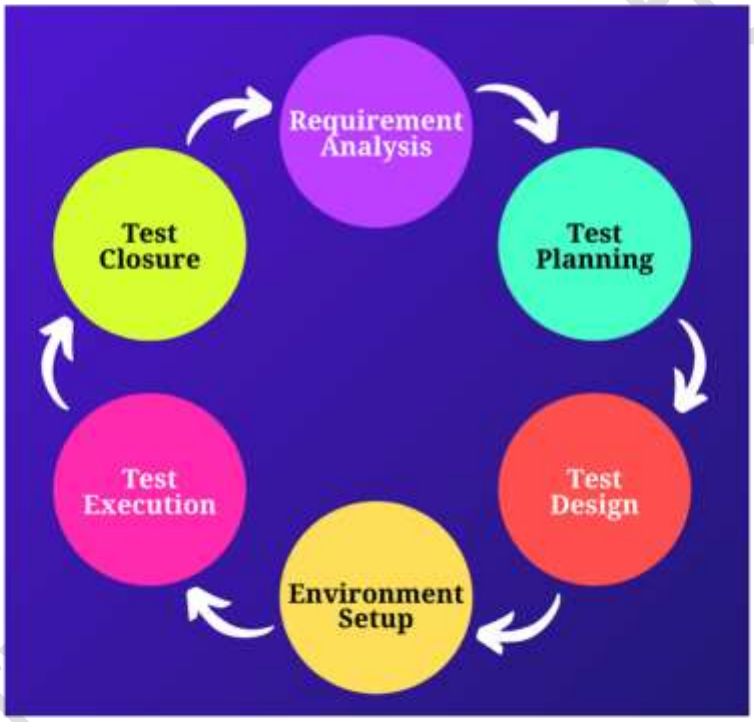
The lesson: Client's requirement & expectations are as important as product quality. And it might change over time. So don't just focus on 'defects' but also on the actual client requirements.

SDLC, STLC & Models

SDLC & STLC

SDLC: Software Development Life Cycle, SDLC for short, is a well-defined, structured sequence of stages in software engineering **to develop** the intended software product. Phases involve: Requirement gathering and analysis >> Design >> Implementation & Coding >> Testing >> Deployment >> Maintenance.

STLC: Software Testing Life Cycle, popularly known as STLC is a well-defined process with different structured sequence of phases **to test** the intended software. All the activities performed during testing of a software product.



1. **Requirement Analysis**: requirements are analysed and validated and the scope of testing is defined.

2. **Test planning**: Test plan & strategy is defined, estimation of test effort along with automation strategy and tool selection is done.

3. **Test Design**: Test cases are designed; test data is prepared and automation scripts are implemented.

4. **Test environment setup**: test environment closely simulating the real-world environment is prepared.

1. **Test execution**: The test cases are executed, bugs are reported and retested once resolved.

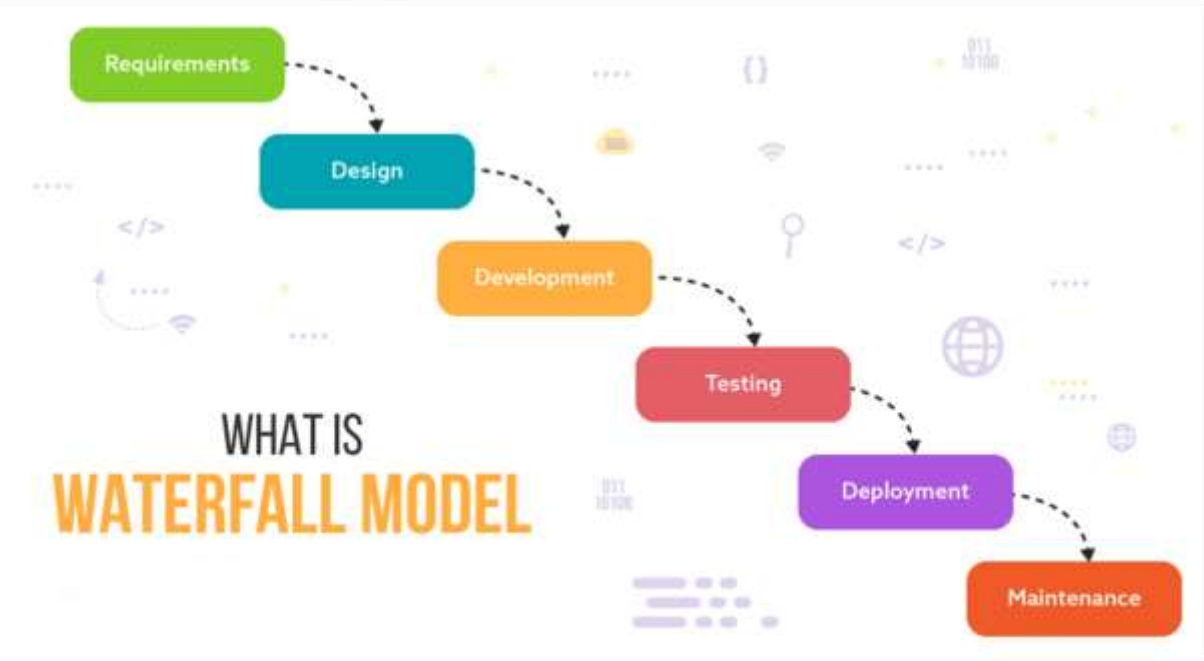
2. **Test closure and reporting**: A test closure report is prepared to have the final test results summary, learning, and test metrics.

Note: This STLC was more applicable in Waterfall model where each phase output was input to the next phase. With Agile, there is no hard-line segregation of these phases. But still, it is good to know about STLC from interview perspective.

SDLC: Waterfall Model

The sequential phases in Waterfall model are,

- **Requirements gathering and Analysis**: the requirements are defined, discussed, analysed, clarified and documented.
- **Design**: With requirements as input, specify hardware and software requirements and define the overall system architecture.
- **Implementation** (Code and Unit Test): With inputs from system design, develop-unit test-integrate the software.
- **Testing**: Testing of individual components, integration and complete system for the systematic discovery and debugging of defects.
- **Deployment**: After testing-fixing-retest, the software/application is deployed in the production environment for use by end-users.
- **Maintenance**: Support and maintenance to fix production issues (if any), provide user trainings, release minor enhancements or defect patches.



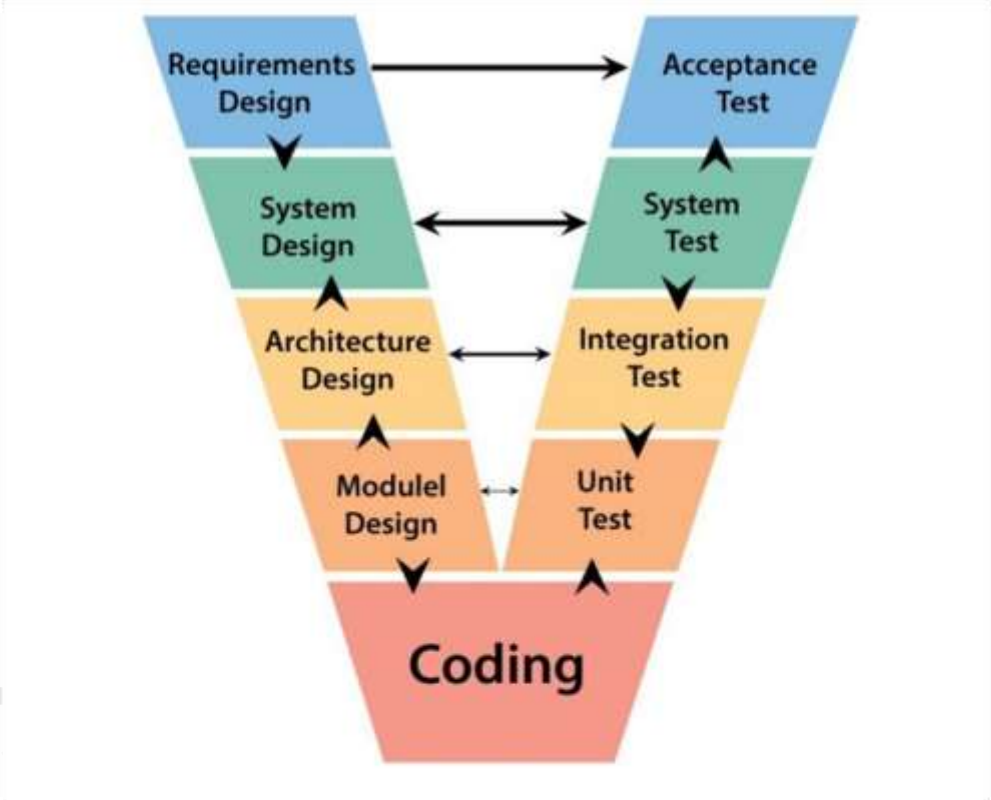
The next phase is started only after the defined set of goals are achieved for previous phase and it is signed off, so the name Waterfall Methodology.

SDLC: V-Model

The V Model gets its name from the fact that the process is often mapped out as a flowchart that takes the form of the letter V. Introduction of V Model actually proved the implementation of testing right from the requirement phase.

There are several **Verification & Validation phases in the V Model**, each of these are explained in detail below,

- Verification: Business Requirements analysis >> Functional Specifications >> High-level Design >> Detailed Design
- Validation: Acceptance Tests >> System Testing >> Integration Testing >> Unit Testing
- Join: The Coding phase

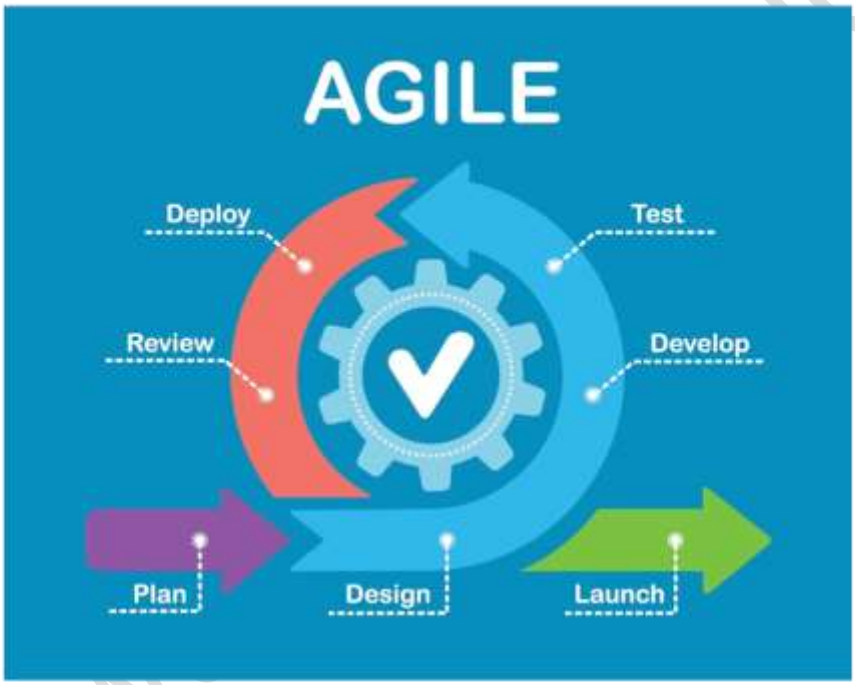


Verification: Producer view of quality, i.e., whether a product, service or system confirms to design standards. In simple terms – 'Are we building the product right?' And how to ensure that? Simple – just verify the intermediary products (like documents) at each phase adhere to the guidelines or requirements imposed at the beginning of the project. Reviewing these documents can find out many hidden anomalies that if found or fixed in the later phase of development cycle, can turn out

to be costly. Verification is more inclined towards development best-practices and conformance to requirements.

Validation: Consumers view of quality, i.e., whether a product, service or system is fit for use for end-users. In simple terms – 'Are we building the right product?' And how to ensure that? Software Testing. A verified product is of no use if it's not defect-free, or cannot be used by end-users. Validation ensures that the product is fit for use and satisfy the business needs. Validation is more inclined towards Testing and end-user perspective.

SDLC: Agile



A testing practice that follows the rules and principles of agile. Unlike Waterfall method, Agile testing can begin at the start of the project with continuous integration. Agile Testing is not sequential like Waterfall but continuous, i.e., testing happens in parallel with feature development as part of every sprint - goes hand in hand with development work and provides an ongoing feedback loop into the quality. It is a collaborative effort between testers, developers, product owners and even customers.

Types of Software Testing

Two broad Types

Types of Software Testing define the different aspects of the software which you are going to cover as part of your Test efforts, i.e., the objective. Will it be only Functional? Or are you going to measure the system performance as well? What about the database schema? What if the Banking application you are testing is secure or not? Will it work on Mobiles as well?

Broadly, Software Testing can be distinguished as either **Functional or Non-functional**, i.e., you need to validate the functional aspects like functions, compatibility, user interface, etc. OR you need to validate the performance, security aspects.

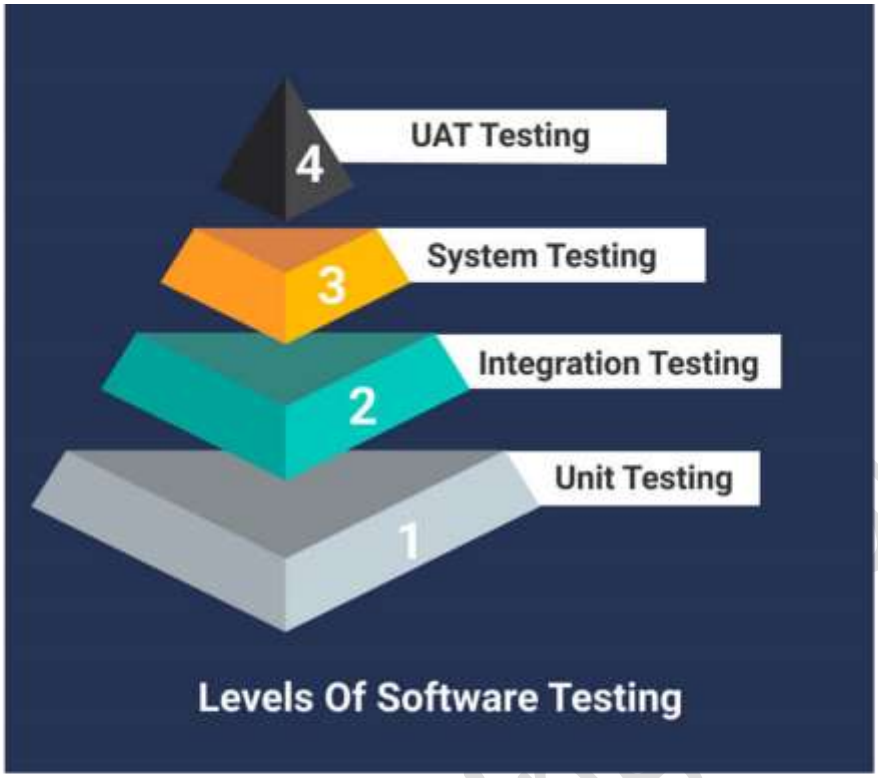
Functional Testing: The most-essential types of Software Testing, Functional testing focuses on testing the software against design document and user requirements - that it correctly performs all its required functions.

Non-Functional Testing: Here, you are not testing the software for any functionality. Instead, the focus is on non-functional aspects like performance, load, accessibility, localization, security, reliability, recovery, etc.

List of Testing Types

Functional & Non-functional are the root types of Software Testing and every other type falls in either of the one category.

Functional Testing | Database Testing | ETL Testing | Performance Testing | Security Testing | Penetration Testing | Vulnerability Testing | Recovery Testing | Data & Database Integrity Testing | GUI (Graphical User Interface) Testing | Compatibility Testing | Usability Testing | Accessibility Testing | Portability Testing | Scalability Testing | Stability Testing | Fault Tolerance, Error handling & Fail-over Testing | Internationalization / Localization Testing | Conformance & Compliance Testing | etc.



Integration Testing (QA)

Testing of the interfaces between different components, so as to confirm that different components can work together in harmony. E.g., Order tracking module expects User details to be passed to it in a certain format >> What if the interface isn't working correctly? I.e., There is a difference between the expected and actual formats used for communication? Integration testing aims to uncover any defects in the way components interact with each other.

System Testing (QA)

Rigorous Testing of the integrated system as a whole, to ensure that the overall product meets the business requirements specified. Generally, this is the most important of all the Software Testing Levels as the end-user is not bothered about the individual components or the integration. He/She is just bothered about the system as a whole – if it works as expected. System testing is typically performed by a specialized testing team and in an environment that is very close to the production environment.

User Acceptance Tests (Business Users)

Whenever we purchase an electronic or electrical item, we always ask the seller to first give a demo so as to check if it's working fine. In case of Software – it is not used by the client, instead by the end-users, i.e., client's business reputation is at stake if it doesn't work. In this case how do you think client would accept your tested system without a walk-through? As the name suggests – Business Users walk-through the system during the UAT phase to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery / release (mainly from user's point of view)

Alpha & Beta Testing

For Commercial off the shelf (COTS) software's that are meant for the mass market testing needs (to be done by the potential users), there are two additional types of acceptance testing:

- **Alpha testing**: Done by potential users, at the developer's site. Potential users or members are invited to use the system and report defects.
- **Beta testing**: No matter how carefully you test, there's nothing like the real-world test of having other people use the software. A Beta release means rolling out not-so-perfect copy of the software (in real-time environment) to a larger set of Users before you actually 'Go Live'. Beta testers (client users or set of actual end-users) try the software, use it consistently for a specific period of time and report back any issues, bugs or feedback about it. It is also known as pre-release testing.

Quick Ref: System testing, Integration testing and E2E Testing?

These are different levels at which testing can be performed, with a bottom-up approach. Start with individual system testing, post that check the integration between different systems. Lastly, check for the end-to-end flow when all systems are connected up-and-running. Example,

Let's take the typical payments' processing landscape. Different systems include,

- Front-Office [FO]: Client-facing application from where the user initiates a payment.
- Middle-office [MO]: System which acts as a router to send the payment to appropriate back-office for processing.
- Back-Office [BO]: The actual payment processing happens in the back-office, i.e., including clearing and settlement. There might be different systems for processing, clearing & settlement, reporting, etc.

Taking this as example,

- System Testing: Testing of individual systems and sub-systems.
- Integration Testing: Testing of integration [data exchange] between FO-MO, MO-BO, BO-FO, and sub-systems as well.
- E2E Testing: Considering the complete landscape, testing for E2E payment processing originating from FO and ending at reporting via MO and BO.

Most of the projects have multiple systems that would interact to achieve the end goal. Hence Testing can be done at different levels.

Testing Ways/Approach

Static Testing

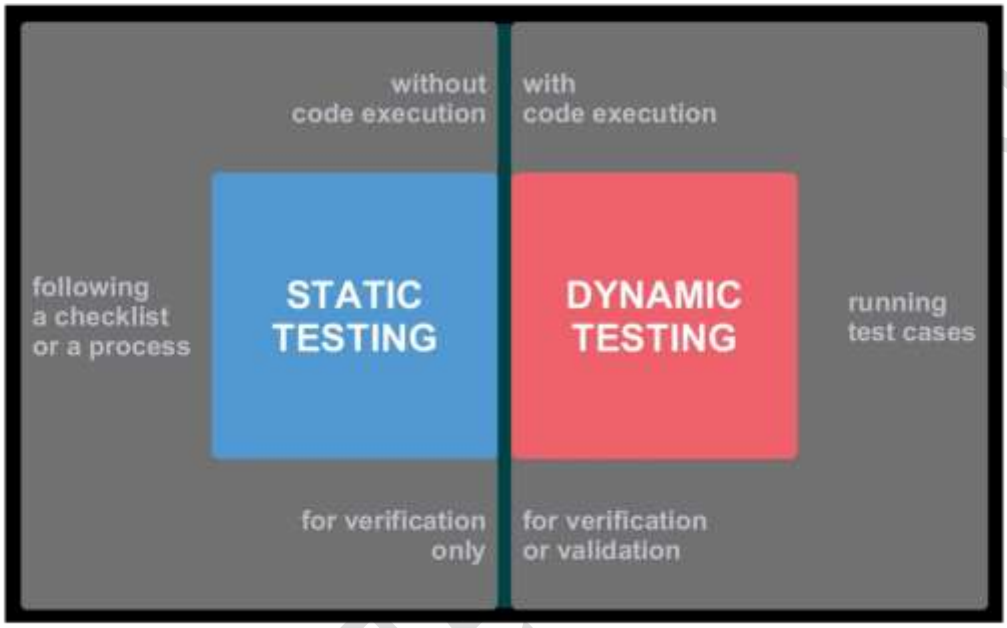
In Static Code Testing we don't execute the code, instead it is checked manually or via tools for any design defects. As you might have guessed – Static Testing is not just limited to the code, we can even analyse the associated documentation like requirement & design documents to identify any potential errors or standard violations. Static testing is also known as Dry run testing.

Main objective of Static Code testing is to improve the quality of software products by finding errors in early stages of the development cycle. Some of the defect types that are easier to find during static code testing are: Programming standards violations, missing requirements, design defects, non-maintainable code, inconsistent interface specifications, variable with an undefined value, variables that are declared but never used, Unreachable code (or) Dead Code, Security vulnerabilities or Syntax violations.

- **Informal Reviews**: No process is followed to find errors in the document. Just review the document and give informal comments on it.
- **Technical Reviews**: A technical team (mostly consisting of peers) review the technical specification of the software product and checks whether it is suitable for the project. The aim is to find any discrepancies in the specifications and standards followed.
- **Walk-through**: A step-by-step presentation by the author of a document in order to gather information and to establish a common understanding of its content. Participants can ask questions if any. A Scribe makes note of review comments.
- **Inspection**: The most formal review technique and therefore always based on a documented procedure. Reviewers have checklist to review the work products. They record the defects such as violation of development standards or non-conformance to higher level documentation and inform the participants to rectify those errors. The meeting is led by trained moderator.
- **Static code analysis**: This is systematic review of the software source code without executing the code. It checks the syntax of the code, coding standards, code optimization, etc. This is also termed as white box testing. This review can be done at any point during development.

Dynamic Testing

Dynamic as in active – the code is executed. How? By using the application, exercising the different functional or non-functional flows of the Application-under-test. The objective – to confirm that the application is working in accordance with the client requirements.



Test planning

Test Strategy

'Strategy', as the name suggests – **what's your approach to testing** a software/product? Before commencement of any Testing project the management first devises a Test Strategy covering the Methodology, Test Approach, Scope of testing, Types of testing to be covered, Environment details, Resource requirements, Test Cycles, RAID (Risk, Assumption, Issues & Dependency), etc.

The first step would be to understand the high-level client requirements and the type of software/product [web/mobile/desktop/cloud-hosted/etc.]. and then the next steps follow,

- Test methodology: Are you following agile or a waterfall? How is your test cycle organized?
- Testing Types – functional, performance, security, database, API testing, etc.
- Tools: What tools you would be using. Say, Selenium, Rest Assured, JMeter, etc.
- Test Management: How will you manage requirements, cases, defects, traceability, etc. – HPE ALM, JIRA, Inhouse tools, etc.?
- Team: What all team roles and expertise are required.
- Test Environment: What will be the build deployment process from dev to production.
- Test Approach: How will you focus on Unit tests, static analysis, dynamic testing, risk-based approach, etc.
- Hardware Requirements: Will you use any stub/drivers? Or a mobile emulator/test lab? Virtual machines for test automation? Etc.

Test Plan

The Test Plan document list your Test approach & all the details included in the Test Strategy – together with the timelines. As the name suggest – it's a plan about your Testing. How have you organized your Testing efforts? A Test plan should be able to **answer the very basic structure of your QA activities**, i.e. What, When, Where, Why, Which, How, Who and What Else.

- What: Scope defines the features, functional or non-functional requirements that will be tested. Additionally, a Test Plan also explicitly mentions the features that are considered to be out-of-scope!
- When: The Timelines. It can be a cycle/sprint structure or the different test phases within waterfall along with the milestones.
- Where: Details about the Test Environment.
- Why: The objective of your planned Test Efforts. To build customer confidence? Catch maximum defects? Provide optimum quality? Sign-off the Product quality?
- Which: Which all Testing types are to be undertaken? And which all tools will be used? Etc.
- How: What is the Test methodology? Test approach…Etc.
- Who: What are the roles and responsibilities of different team members like Test Manager, QA Analyst, Automation Engineer, Configuration Manager, Developers, Installation Team, Business Analyst, etc.?
- What Else: All other important things like defect management, deliverables, risks & assumptions and open items (if any).

The idea is simple, focus on the value it offers – boil Test planning down to only the essentials and cut all fat and fluff. A comprehensive Plan gives customers the confidence that an efficient Test process is adopted to ensure optimum Quality!

Estimations

"Estimation is the process of finding an estimate, or approximation, which is a value that is usable for some purpose even if input data may be incomplete, uncertain, or unstable." – Wikipedia

Applying this to Software Testing,

- **Estimate value**: the time & effort it will take to test a particular requirement.
- **Some purpose**: to plan the scope, timelines, efforts & resources it will take to test a requirement.
- **Input data**: User requirements
- **Incomplete, uncertain or unstable**: Estimation is done based on initial requirements which might not be clear and will be refined as the project progresses. The actual effort might differ as we start testing, due to unforeseen circumstances like downtime, changing requirements, external factors, etc.

There are different estimation techniques based on the methodology a project adopts. E.g., Use case point estimation in case of waterfall and Story point estimation in case of agile.

Requirements

Requirement Analysis

IEEE defines requirements analysis as,

- *The process of studying user needs to arrive at a definition of a system, hardware or software requirements.*
- *The process of studying and refining system, hardware or software requirements. Requirement's analysis helps to understand, interpret, classify, and organize the software requirements in order to assess the feasibility, completeness, and consistency of the requirements.*

You cannot build the right software/product unless the requirements are clear. Right?

There are situations where the requirements might be missing, or there might be an ambiguity (unclear), or some requirements need discussions for further clarification.

- **Missing**: Login Page has a username/password & login option – but Forgot password option is missing.
- **Ambiguous**: Client needs a textbox but it didn't mention the values it should accept. E.g., Amount field shall accept only numeric, Password field might have combinations, Name field should not accept special chars, etc.

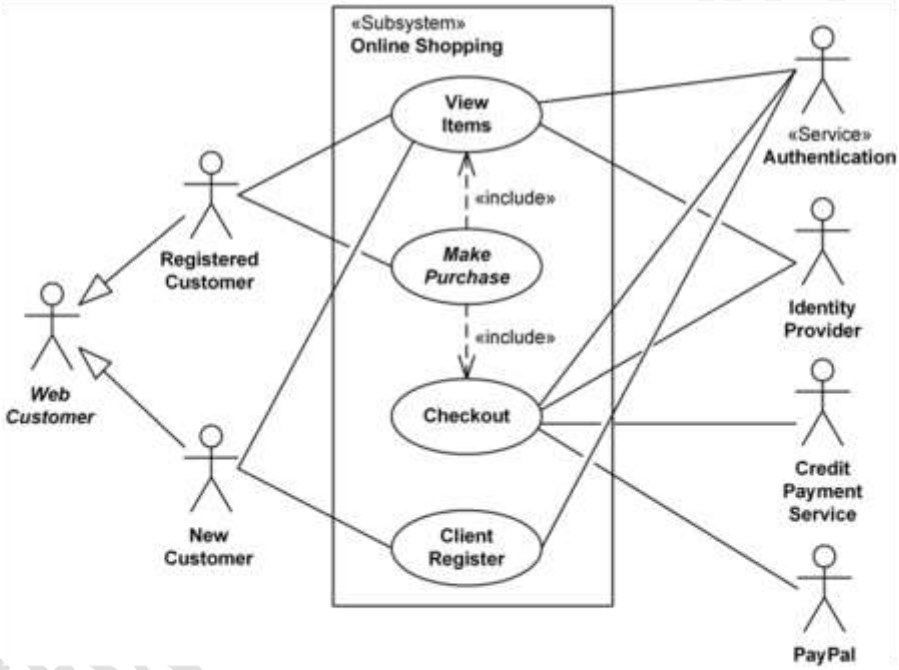
Requirement analysis is the activity of going through the requirements one-by-one and **identify any gaps/misses/ambiguity that should be first clarified** before starting development. Waterfall model had a separate phase dedicated to requirement analysis whereas in agile requirement discussions will happen as part of backlog grooming & sprint planning.

Requirement analysis requires a close coordination between ALL the stakeholders – client, business analyst, product owner, development team and testers.

Use Cases [Waterfall model]

A set of **interactions between a system and one or more actors**, with actors being people, other systems, or both. It's a complete specification of all possible scenarios. They are usually created as documents, and generally include this kind of information:

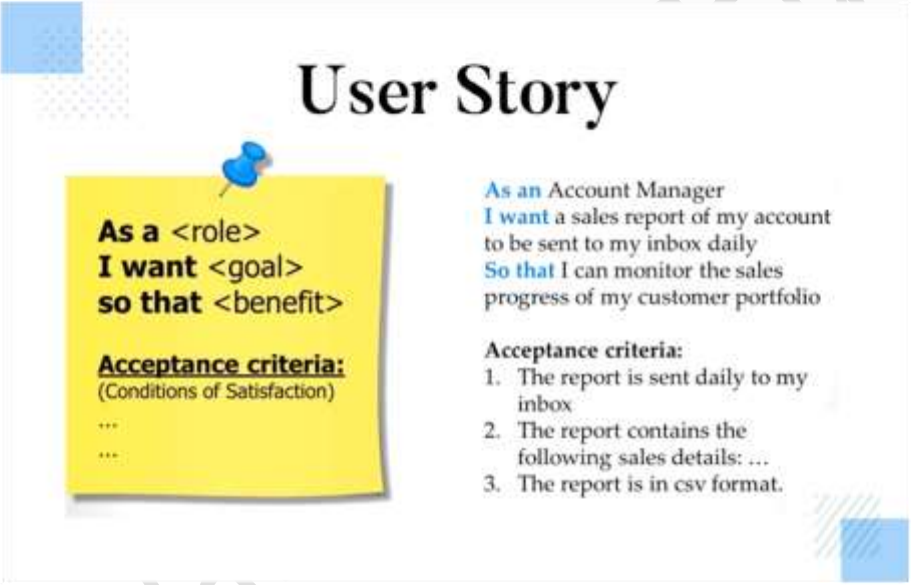
- Use case title & description
- Actor/user
- Preconditions
- Standard path or main success scenario
- Alternate paths or extensions
- Post conditions



Use cases focus on how your system interacts with other systems and actors and are typically extremely detailed. Traditionally, these are written by a single person or a small group of people and require someone else to sign off on them.

Epic – User Story – Tasks [Agile]

- **Epic** - something so big it probably won't fit into a sprint and should be broken down into stories. Epics are usually defined during initial product roadmap and decomposed into stories in the product backlog as more is learned.
- **User Story** - something actionable and small enough to fit in a sprint. Stories should deliver a functionality to the customer that is valuable and complete by the end of an iteration.
- **Tasks** - decomposed parts of a story that get into HOW the story will be completed. Tasks can be hour estimated if desired. Tasks are usually defined by the people doing the work (developers, QA, etc), whereas stories and epics are generally created by the customer or the product owner representing the customer.



Product Backlog [Agile]

It's a document that acts as a **single source of requirements for everything needed** in the product. It includes new features, changes to existing features, bug fixes, infrastructure setups, etc. It is constantly evolving and is never complete. The Product Owner manages the Product Backlog, including how it's made available to the scrum team, its content, and how it's ordered and prioritized, everything. The Product Owner and the rest of the scrum team work together to review the Product Backlog and make adjustments as and when necessary. Each requirement in the product backlog have a,

- Description
- Order based on priority
- Estimate
- Value to the business

Test Design | Part-I

What & How of Test Design

Simple put – writing the test cases based on client requirements. This will include writing any test automation scripts as well. But it's not as simple as it sounds. A proper process has to be followed for Test Design to ensure that none of the client requirement is missed and we have an optimum test coverage before advancing to the Test execution. Test design also depends on the types of testing that are in scope of the project. Generally, below aspects needs to be covered in test design,

- **Positive Testing**: The system is validated against straight-forward user actions using valid input data. The intention is to check if the system behaves as expected for a valid input (& not showing an error). Positive Testing proves that the application-under-test meets the requirements and specifications.
- **Negative Testing**: Just the reverse – The system is validated against invalid data. The intention is to analyse the stability of application-under-test when invalid input data is entered. The application shouldn't crash, rather in most cases should display a proper error message.
- **Alternate flows**: End-user is not a robot to follow a test case or a pre-defined written workflow. He/she can perform any random actions – cancel the request in between, save & come later for completion, click on help, check action history, edit and fill additional info, wrong password by mistake, forgot password, payment failure, anything. Always try to cover as many alternate flows as possible.

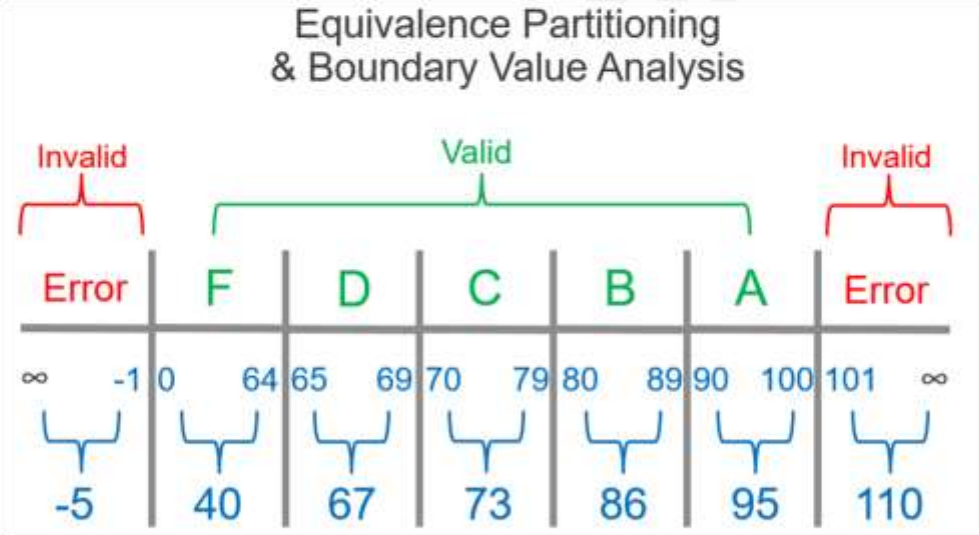
Negative Testing & covering the alternate flows is an important aspect when trying to find defects in an application. It ensures that the application can gracefully handle invalid input or unexpected user behaviour. How would you feel if the page is not getting submitted & no error is displayed? Or a payment fails in the back-end without any error being displayed?

Test data preparation comes in handy here – blank value, boundary limits, invalid input, special characters, etc. As a tester always check for graceful error messages when inputting invalid data. Or when performing an invalid operation.

Test Design Techniques

As the name suggests – techniques we as testers use every day to derive effective test cases from the requirements. The most common ones,

- **Equivalence Partitioning**: Grouping the inputs with the same attributes to partitions, i.e., partitions that would exhibit equal results. Pick one condition out of each partition, which covers all possible scenarios, to execute test cases. This helps reduce the number of test cases. E.g., testing an input field – all alphabets, all digits, all special chars, combination, blank, null, etc. might become your partitions.
- **Boundary Value Analysis**: Include values at the boundaries. If the input is within the boundary value, it is considered 'Positive testing.' If the input is outside - 'Negative testing.' The goal is to select test cases to execute boundary values. Baseline: Boundaries are an area in which testing is more likely to yield defects. E.g., testing an input field – blank, full length, null, exceeding length, etc.



- **Experience-based [Exploratory]**: Domain experts perform testing just by exploring the functionalities of the application without having the knowledge of the exact requirements. Testers can explore and learn the system while using these techniques. High severity bugs are found very quickly. E.g., based on your experience, you might want to cover different buckets of test cases – positive input [happy flow], alternate flows, negative inputs [error messages], no input/null/etc., browser combinations, etc.

Test Conditions

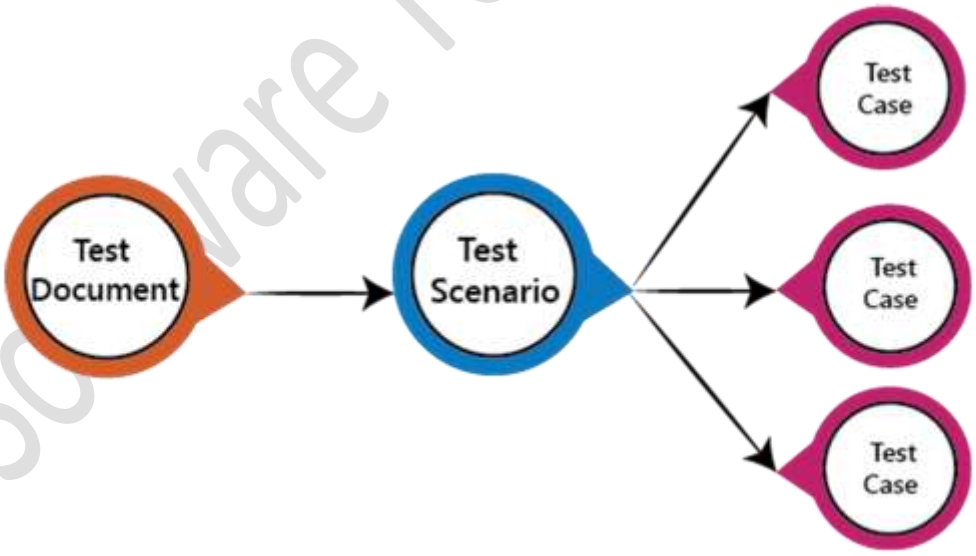
Testing terminologies can sometimes confuse even the most experienced of IT professionals. To a layman Test conditions, Test Scenarios, Test cases and Test suite might seem similar but there is a subtle difference. Each of these implies a different level of detail and is used for a different purpose.

Test Condition [*Functionalities*]

"An item or event of a component or system that could be verified by one or more test cases, e.g., a function, transaction, feature, quality attribute or structural element. For example: Test Object: Login form."

What all conditions (functionalities) are you going to test? Test Condition is nothing but the high-level pieces of a puzzle, i.e., different events/features of an application-under-test. E.g., Home Page rendering | Different sections displayed – Header/Content/Footer | Main Menu Options & Navigation | Sub-menu Options & Navigation | All Navigation (All hyperlinks, some 50/100+) | Search functionality | Slider functionality | Location Detection | Any particular Business Logics | Language Translations | Device/Browser compatibility | Page Load performance.

It is advisable to treat Test conditions as a chance to generate Test ideas, where you can think of ALL the functionalities/scope which can be tested – high-level things to be verified.



Test Scenarios

Test Scenario [*Business Flows* within Functionalities]

"A business process flow is composed of Stages, and within each stage there are Steps to complete."

It's time to move from high-level to mid-level Test coverage. Test scenarios help in achieving a good test coverage by breaking down a functionality in different possible business flows, i.e., different ways to accomplish a task. It's a short description about the business flow under test. Put yourself in the end-user shoes and figure out the real-world scenarios which can be performed on the application.

Below can be a set of Test scenarios for Test Condition – Different sections displayed – Header/Content/Footer.

- Verify that Header is displayed at the top of the web page.
- Verify the UI of the Header, text-colour-font-size-etc.
- Verify the logo displayed in the header.
- Verify that the location is auto-detected & displayed correctly in the header.
- Verify the Category menu.
- Verify the Sign In menu & options displayed.
- Verify the 'Prime' menu options & navigation.
- Verify the 'List' menu option & navigation.
- Verify the 'Cart' option displayed in the header.
- Verify the Search menu and the Search functionality using all categories.
- Verify the Offer (if any) displayed at the top right corner.
- Verify other navigation options available.

Test Design | Part-II

Test Cases

Test Cases [*How* to Test Business Flows]

"A test case is a document which consists of a set of conditions or actions which are performed on the software application in order to verify the expected functionality of the feature. Here we describe the end-to-end logical flow of a specific requirement with test data, prerequisites and expected results."

Consists of a set of conditions or actions which are performed on the software application in order to verify the expected functionality of the feature. Here we describe the end-to-end logical flow of a specific requirement with test data, prerequisites and expected results. It consists of a Test case description, execution pre-condition, steps to follow, a set of input values, expected results and executed post condition. Test cases are derived from Test scenarios and one Test scenario may result in multiple test cases.

Let's try to create a Test case for the Search functionality, the steps would be -

- Open Amazon website in a web browser | Website is displayed successfully
- Verify the Search box at the top | Search text-box is displayed with category options at the left & search icon at the right.
- Verify the categories displayed for the Search option | Below list of categories are displayed.
- Select 'Amazon Fashion' category | Category is selected successfully.
- Enter the Search criteria as below & click Search icon | Search results are displayed.

Test Data for search box input: Valid Fashion brand, valid fashion clothing, valid different-category input, only numeric values, alphanumeric values including special chars, Unicode chars, Blank value, NULL value, JavaScript insertion, all spaces, etc.

Quick Ref: What will be your top 5 to 10 scenarios to test ABC application?

One of the questions frequently asked in any interview. Either they will open the application like e-commerce/travel/etc. OR just ask you to give scenarios verbally.

Before you start giving the test scenarios, **take a pause and think about the application and its modules/functionalities**.

- Homepage rendering on different browsers – covering both web and mobile.
- Does it require registration? Sign-Up scenario.
- Most of the applications need a login, verify login with different role users [authorization + authentication] – sign-in and sign-out.
- Testing of 'Search' within the application, i.e., search results with different test data combinations.
- Test to check any broken links, i.e., different categories listed, header & footer notes, etc.
- Any integration with outside-apps like OTP/Email verification/etc.

Apart from these, interviewers are impressed if you cover other aspects as well apart from just functional cases,

- UI/UX design, i.e., easy-of-navigation, colour-scheme, image-rendering, scroll bars, font-size, neat-and-clean vs. cluttered, etc.
- Localization and Globalization testing, i.e., if the application is to be used in different geographies/languages.
- Page rendering performance measurement
- Cross-browser testing
- Security aspects like cookies, certificate, secure connection, etc.
- Tests on handheld devices, say a smartphone.

Requirement Traceability Matrix

"A traceability matrix is a document that co-relates any two baseline documents (section to section) that require a many-to-many relationship to check the completeness of the relationship, i.e., to ensure that everything within both documents is related in any way"!

Purpose: Identification of anything that is not linked in two documents, which require further analysis.

Applying the concept of Traceability matrix to requirements, i.e., one of the reference documents will always be 'Requirements Specification'. In simple terms, Requirement Traceability Matrix (commonly known as RTM) is a tool that traces requirement throughout the validation process, i.e., Requirement Analysis, Test Design, Test execution & Closure phase.

How?

Simple mapping – Requirement IDs >> Test Scenario IDs >> Test cases IDs >> Defect IDs (including final status). The RTM can be created and maintained in an automated tool (such as HP QC), in an Excel spreadsheet, or MS Word table.

- Requirement ID
- Requirement description
- Testable (Y / N)
- Module / Sub-Module
- Scenario IDs
- Test Cases ID
- Defect IDs
- Status
- Remarks

Purpose: To trace that all requirements are covered in Test Design, Test execution & Closure.

- Client Confidence: First & foremost is to build the client's confidence that the product is being developed & tested as per the requirements
- Ensure Test Coverage: All requirements have been covered in Test Design & Execution. The aim of any testing project should be 100% test coverage. Requirement Traceability Matrix helps is uncovering the gaps.
- Track Change Requests: Maintained throughout the lifecycle of the release, changes to the requirements are also recorded and tracked in the Requirement Traceability Matrix.
- Risk Management: ensuring every requirement is testable & eventually tested
- Maintainability: Any change in requirements can be tracked to corresponding required change in Test Design
- Product Quality: The Test coverage & corresponding defect status gives confidence to the client about the product's quality

Test Data

How do you test a textbox? Yeah! By entering & submitting different text. The permutations & combinations of this 'text' used for testing purpose is known as 'Test Data', i.e., data used for testing purpose. For testing a textbox, we will first prepare the permutations & combinations of test data – numeric, alphabets, alphanumeric, max length, special characters, etc. As the application functionalities grow, so does the corresponding Test data to-be-used for testing. It may be any kind of input to the application-under-test (User credentials, Environmental data, Setup data, Input / Output data, Uploaded files, Database tables, Transitional data, etc.)

Test Environment

What is Test Environment?

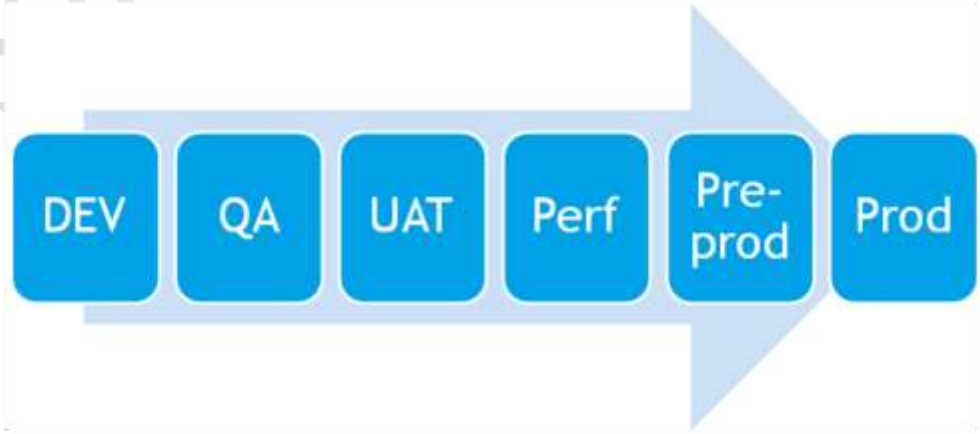
Environment – as in the atmosphere and the surrounding. Now what's surrounding a software? Yeah! Database, Front-end files, operating system, Hardware configurations, Server and Network, etc. All the hardware and software requirements for a particular application to work as expected constitutes 'Environment' in Software Engineering.

Real-world example,

What to do if you want to make an online transfer? Yeah! We login to our Bank's net-banking website and initiate the transaction. Hope you already know that the net-banking application that you use via web is 'already tested'. Now where does this 'testing' happen? Or rather where does the 'development' happen? Say the Bank want to change the User Interface. Developers cannot start changing the interface straight-away for the application being used by millions of people, right? The current application needs to be running on the web, and development & testing need to happen on some other 'environment' before the changed-application is deployed for public use.

In simple terms – Test Environment is nothing but a replica of actual production environment (being used by end-users) with close-enough hardware and software configurations, where the testing would happen for the developed application. Test environment is where all the action happens – experiments, defect identification, fixing, retesting, regression and final sign-off. Behind each successful test environment there is normally a build team responsible for putting the environment together.

Once the testing is completed in this environment >> Test team will sign-off the application for actual production deployment.



Different Environments

- **Development Environment (Dev)**: where actual development (including defect fixes) happens. Unit tests are done by developers in this environment. After successful unit tests, the build is promoted to next environment for Testing.
- **Test Environment**: Also known as QA environment – where testing happens. All the test cases are executed, exploratory tests are performed, defect retesting and regression. Finally Test team provides a sign-off for the build to be deployed in next higher environment for UAT.
- **Performance Test Environment**: for performance testing. This is different from general Test environment because for effective performance measurement we need a replica of production environment (i.e., the servers count, capacity, memory, etc.). Generally, a Test environment is a toned-down replica of production environment with fewer servers, capacity, memory, etc.
- **Acceptance Test Environment (UAT)**: After successful System testing, the build is deployed in UAT environment for User Acceptance tests (performed by Business team). Once UAT team confirms (A Go / No-Go decision) the application is deployed in next higher environment (known as application Go-live).
- **Production Environment (Live)**: Also known as 'Live environment' – this is where the end-user accesses the application. Real world example – We (end-users) access our Bank's net-banking application in 'production environment' to initiate transfers.

Depending on the criticality and the type of application-under-test (AUT) there might be few variations like a staging environment (migrate/test on real production data), pre-production environment (where you can test user profiles, security, disaster recovery and back-up) and training environments (where client wants the organization's user training to happen before actual go-live).

Test Execution

What is Test execution?

Now the test design (test cases) is ready and the environment is also setup. The next logical step is to execute the test cases against the application-under-test. Go on to execute the Test cases to find defects. If required, do some ad-hoc tests as well. The bottom line is – To confirm application is working as per requirements and to find defects!

Pre-requisite: Software Build

What do you think it takes to build a software? Yeah! The code. But it isn't just one single code file, generally there are multiple source code files. Now these source code files need to be compiled & combined into a single executable file which can then be deployed by the release team. This process of combining multiple source code files into a single executable file is known as 'Software Build'. As you might have guessed, before being delivered to the client a software undergoes multiple changes (defect fixes), i.e., multiple 'Builds'!

Entry & Exit Criteria

"Defines the conditions to be satisfied in order for testing to begin and when to stop the testing".

Entry and Exit criteria vary for organization and projects. But few items that can be considered are,

Entry criteria,

- Requirements are clear & complete.
- Test design is completed for all requirements that are in scope.
- Software is deployed in test environment.
- & The list goes on…

Exit Criteria,

- All test cases have been executed.
- All-important defects have been fixed, retested and closed.
- Regression testing is completed with >95% pass result.
- In agile, 'Definition of Done' is met for all user stories in scope for current sprint.
- & The list goes on…

Test Execution Cycles

Smoke Testing

What if developers are too reckless? There is a defect at the very first step – User is unable to login itself. Yeah! You will say what about Unit testing, but usually developers don't follow the rules every time ☺ Smoke testing is the preliminary testing to reveal simple failures severe enough to reject a prospective software release. In other terms you can call it as 'Confidence testing' or the 'Build verification testing'. Smoke tests cover the **MOST CRITICAL functionalities**. The purpose is to reject a 'critical defect' build before the Test team starts detailed functional tests. Before starting the day, a daily build and smoke test is among industry best practices.

Note: The term "smoke test" refers to powering on a device simply to make sure it does not start smoking (indicating a major problem). It originated in the testing of electronic hardware.

Sanity Testing

First of all, irrespective of Testing, Sanity check is a basic concept – a simple check to see if the produced output is rational (that the product's creator was thinking rationally, applying sanity). Extending the concept to software, after every change in a build, Sanity testing is performed **to ascertain that the particular changes are working as expected** post which detailed tests are performed. If sanity test fails, the build is rejected to save the time and costs involved in a more rigorous testing.

Note: Sanity in general refers to the soundness, rationality and healthiness of the human mind.

System Testing

Execution of all the test cases written in the test design phase in order to confirm that the software is developed as per the requirements and also to find all possible issues/defects which can be fixed & retested before the software build is deployed to next environment.

Regression Testing (No Side-effects)

It's like **checking for side-effects**. The code is developed >> Build is deployed >> Sanity is performed >> Full testing is done >> Defects are logged, fixed & retested. What else? Yeah! Truth is stranger than fiction, and so is the Software. You never know a change in one function (defect fix or enhancement or change request) can impact multiple areas of the software. It's our duty as a Test team to ensure everything (apart from the change) impacted is working as expected. That's called regression testing to ensure there are no side-effects of the code change.

Quick-Ref: Manual and Test Automation

As you know by now, Test execution means to actually use the application in the test environment with respect to requirements - & confirm the correctness or report any issues.

Now this can be done either by a user, just like you & me, by opening the application and using it based on the test design and experience. I.e., without using any software tools. Just like your end-user will use it. This is popularly known as 'manual testing' though there is NO documented definition.

Test Automation, on the other hand, is to use software tools to use the application. Yes, there are tools which can mimic the user actions on application. Test automation can be done on application UI, API level, Database level or even at Unit tests level. In simple terms, you write the code in order to perform some action using some data and then to validate the response. It is known as 'Test script' (synonymous to test case). With agile, the focus is to deliver a working software quickly in iterations – which is possible only when you use test automation wherever applicable, as much as possible. Hence test automation is a must-have skill for any tester.

Note: There is always a debate around "*There is nothing called manual testing*". **It is just "Software Testing"** – use application as per requirements and also use tools whenever it can speed up the testing.

Defect Management

What is Defect?

"Anything that deviates from the agreed requirements is a defect".

It can be as simple as change in button colour to as severe as application crash on user action. A security breach into the application or the application hangs on user click. Amount debited twice or transferred without any authentication. It could be anything.

- **Mistake**: There was a slipup by the developer hence the issue.
- **Error/Bug**: Due to developer's mistake, there is an error/bug residing in the code.
- **Defect**: Once the bug is identified during testing, it is logged as a 'Defect' in the tracking system.
- **Failure**: The outcome of any defect, i.e., when the application is behaving in a way it is NOT supposed to. Literally a disappointment or a let-down!

Generally, every requirement (use case/user story) has an associated acceptance criteria which is a checklist. Anything that doesn't fulfil the acceptance criteria is a defect.

Defect Lifecycle

Different statuses that a 'defect' undergoes during its 'life cycle'.

Note: Defect Life cycle varies from organization to organization and is governed by the software testing process the organization or project follows and/or the Defect tracking tool being used. Let's look at the industry-standard.

New: Say you identify a defect in the Application-under-test (AUT). What's next? Yeah! Logging it in a defect management tool. By default, whenever you log a defect, its status is 'NEW'. Mind you, it doesn't mean it's a valid defect and will be fixed – the analysis is not yet done!

Assigned: As a Lead whenever you notice a 'New' defect in the system the next step is to analyse the defect for its correctness. If valid, assign it to a particular developer and change the status to 'Assigned'.

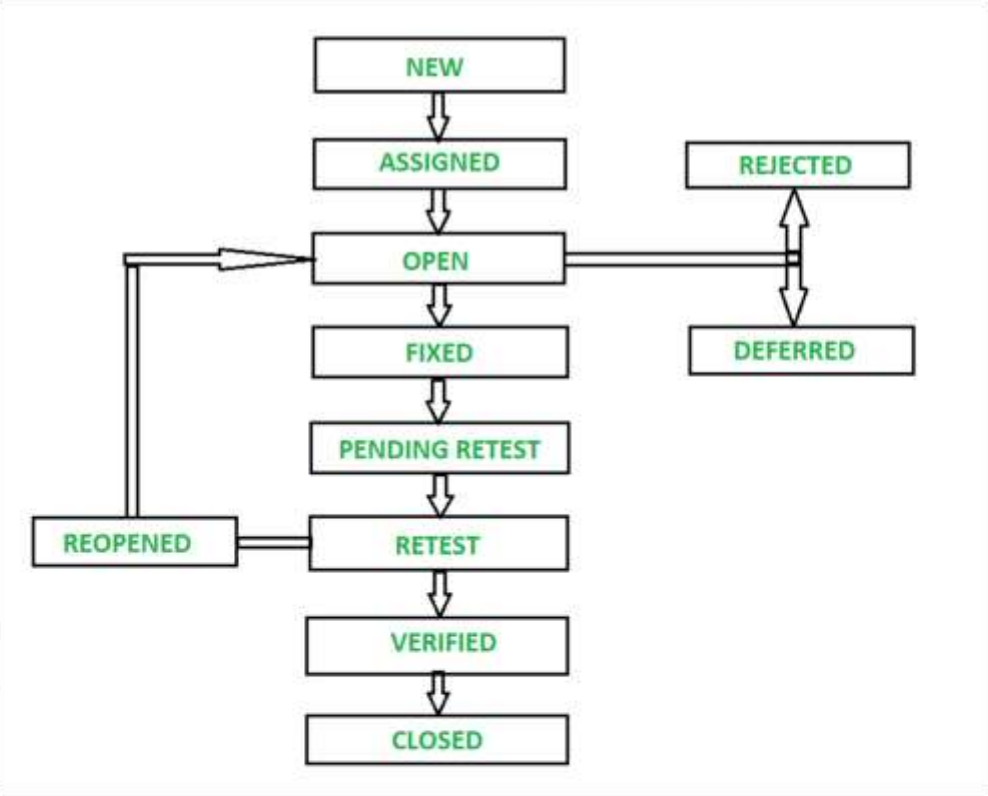
a. **Deferred**: Deferred as in delayed or postponed. After initial analysis it is observed that a particular defect will not be fixed as of now, i.e., in this release. It will be fixed in some future release. Therefore, the Lead will change its status to 'Deferred'. And why do you think a defect will be deferred? There are multiple factors and there can be multiple reasons – very low priority, lack of time, not part of current requirement, technical dependencies, etc. Generally, it's a standard practice

to get sign-off from the business team before marking a defect as 'Deferred'. After all, they know the best about the business impact.

b. **Rejected:** After all, 'To err is human'. Even a tester can make mistakes. What if you used a wrong test data? What if your application understanding is wrong? Didn't understand the requirement? Defect is already logged by some other tester? The functionality is out-of-scope? It is not a requirement? Wrong observation? Yeah! What if you as a tester have raised an invalid defect? Obviously, it will be rejected. If after analysis it is observed that this is NOT a defect, then Lead will change the status to 'Rejected' mentioning the details in the comment section.

Open: Once a defect is assigned to a particular developer – he/she starts working on the defect fix and changes the status to 'Open'. Working as in analysing the defect, investigating its root cause and rectifying the code.

Fixed: Once the code changes are done & verified in the development environment – the developer changes the status to 'Fixed'. What's next? The fixed code needs to be deployed in the Test environment post which it will be verified by the Test team.



Ready for Test: What's the difference from 'Fixed'? The Environment. Once the rectified code is deployed in the Test environment – it is available for the Test team to verify, i.e., retest. Post deployment, the developer changes the status to 'Ready for Test' and assign it back to the tester (who logged it). Tester verifies that the fixed code is working fine and the defect is no more reproducible.

Closed: If the rectified code is working as expected, i.e., the defect is no more reproducible – the tester changes the status to 'Closed'. It means that the defect is fixed & retested successfully and is no longer reproducible. Happy Ending!

a. **Reopen**: During retest you find that the defect is still present even after the fix. What to do now? Simple, just change the status to 'Reopen' and assign it back to the developer. The lifecycle starts again.

Defect Priority & Severity

Priority: The 'urgency' to fix the defect.

How urgent is it to fix the issue from **business perspective**? Unable to login? Top priority. Misspelt brand name? Top priority since you don't want to tarnish the brand image. Error in payment processing? Top priority. 'Help' URLs not accessible? Low priority since it can be fixed later, it is used very rarely.

Severity: The 'brutality' of the defect.

What is the impact of issue from **functionality perspective**? Unable to login? Critical severity. Misspelt brand name? Low severity since it doesn't impact any functionality. Error in payment processing? Critical severity.

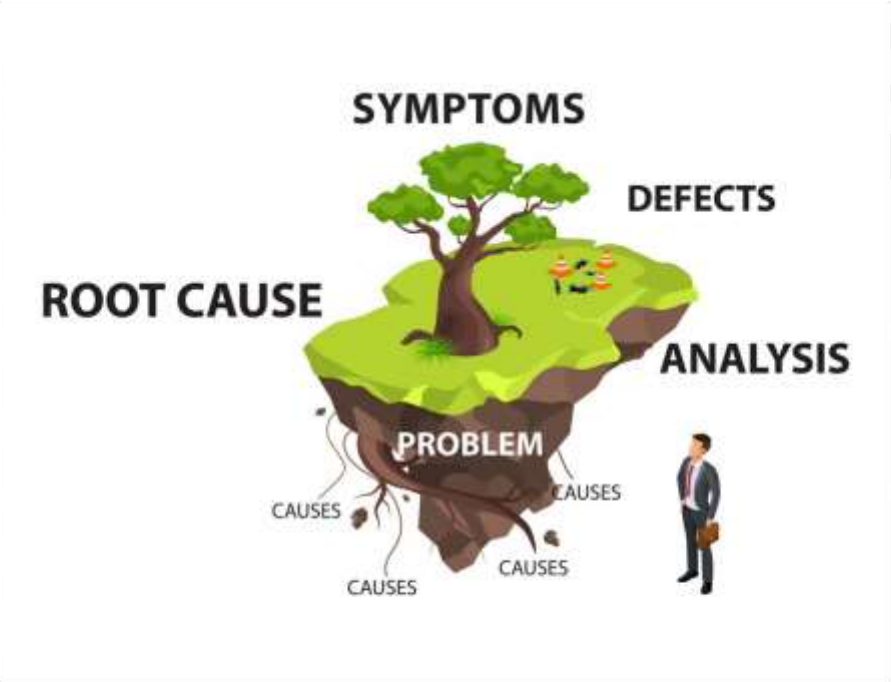
- High-Priority-High-Severity: Login issues, payment issues, etc.
- Low-Priority-Low-Severity: One of the random labels in some page is misspelt.
- High-Priority-Low-Severity: Brand name misspelt. Data is saved but with an error message.
- Low-Priority-High-Severity: The application crashes at boundary values, say very long username/passwords. Which are rare.

It is important to clearly understand & update defect's severity & priority, from both business and functionality point of views.

Root Cause Analysis

"a systematic process for identifying "root causes" of problems to find a way to prevent them in future".

In terms of testing, it is mostly employed for defects-RCA. Many defects can be prevented if action-items from an RCA are actioned upon. Some common root cause can be – environmental, configurations, permissions, invalid test data, incorrect test, ambiguous requirements, code issue, etc.



Defects falling under certain categories can be prevented from the start itself, hence saving both time & efforts and focusing more on valuable testing. In that sense, RCAs help in continuous improvement of the overall process.

Test Reporting

Daily Status Report

As the name suggests, it shows the daily progress of the testing efforts.

- **Highlights**: Important pointers to be conveyed about today's status.
- **Progress**: What progress has been made against the plan, i.e., it can be execution count, stories tested, value added, etc.
- **Defects List**: All defects raised along with respective priority, status, etc.
- **RAIDs**: What are the highlights, i.e., any risks, assumptions, issues or dependencies considered.
- **Downtime**: Did you have any downtime due to environment unavailability, OR a blocker?

Generally, there are tools available in the market used by project teams for ALL reporting purpose. JIRA is one such tool which is used for Agile project management covering requirements, progress, and defect management. HPE ALM/QC is yet another commercial tool used for all the test management activities starting release plan – requirements – test design – defect management. Reports can be quickly seen within the tool OR can be exported to share in an email.

Defect report

The defect report is the most effective manner of **communicating, tracking, and explaining defects** to managers and development staff. Contents,

- Defect ID and Status
- Summary
- Description
- Severity & Priority
- Build/Platform/Version
- Steps to reproduce
- Actual results
- Expected results
- Related Requirement ID
- Supporting documents

Important pointers,

- Take note of any configuration, user role, or setup changes that have been made to the QA system - if yes, mention it.
- Sometimes it requires evidence; can be error message text, actual error log, or screenshot/video.

In this way, anybody can quickly get a solid idea of what the defect is and how does it impact the application.

Defects Metrics | Part-I

Total Defect Metrics

The first & foremost defect metrics to look at is – "Total defects". The relative assessment of Total defects vs. Module size & complexity can tell you a lot of things about the effectiveness of testing efforts.

Defects Distribution by Priority & Severity

Total defects give a good idea about the test efforts, but it doesn't reflect on the 'quality' of testing or the build. A more appropriate metrics in this case would be "Defects Distribution by Priority & Severity". Priority and Severity distribution serves as an input to both Business as well as Project team, which helps them gauge the actual "Quality" of the Test team's efforts. Additionally, critical & high defects can be presented as a percentage,

(# Critical or High Defects / Total Defects) * 100

Defect Density Metric | Test Design Efficiency

Most of the testing terminologies can be understood by their straight-forward English meaning. Density as in Thickness or Concentration. On a similar line we can also identify the percent concentration of defects within a software build. How?

Defect Density = (No. of Defects identified / size) * 100

Here "Size" can be considered as the number of requirements or test cases. Hence the Defect Density is calculated as number of defects identified per requirement or test case. Example:

- Total test cases: 1200
- Passed: 900
- Failed: 300 (or Total defects = 300)
- Defect Density = (300 / 1200) * 100 = 25%

Now what does this mean? Simple, 25% of your test cases failed during execution OR 25% of your test design was able to catch relevant defects.

Defect Removal Efficiency (DRE) | Positive outlook

Why do we do Testing? Yeah! To identify defects and improve the Software quality by removing (fixing) them. One of the most important defect metrics, Defect removal efficiency is a measure of Test team's competence to remove (identify) maximum defects before a software is moved to the subsequent stage. Here defects that matter are the ones caught by either the test team or by other users in the next phase.

Defect Removal Efficiency = #Defects found during testing / (#Defects found during testing + #Defects found during next phase) * 100

DRE is the percentage of defects that have been removed during an activity. Say the Test team identified 100 defects during system testing. After fixing, retest & regression the build is promoted for User acceptance tests. The UAT team identifies 25 more defects in the build which could have been identified during system testing. What does it mean? The system testing team could not find these 25 defects and they were NOT removed from the system. Hence,

Defect Removal Efficiency = [100 / (100 + 25)] * 100 = 80%

Means that the System testing team was able to remove (identify) only 80% of the defects, rest 20% were identified during the subsequent phase. As you might have guessed, DRE can be (or rather should be) computed for each SDLC phase and plotted on a bar graph to show the relative defect removal efficiencies for each phase.

Defects Metrics | Part-II

Defect Leakage metrics | Negative outlook

Defect Leakage, as in what percentage of defects actually leaked from the current testing phase to the subsequent phase. As you might have guessed, Defect leakage (defects passed to next phase) is just the reverse of Defect removal (defects identified in current phase) efficiency. Defect leakage is another defect metrics but a deadly one, i.e., defect leakage percentage should be minimal in order to prove test team's worth!

Defect Leakage = #Defects found during next phase / (#Defects found during testing + #Defects found during next phase) * 100

Say the Test team identified 100 defects during system testing. After fixing, retest & regression the build is promoted for User acceptance tests. The UAT team identifies 25 more defects in the build which could have been identified during system testing. What does it mean? The system testing team could not find these 25 defects and they were LEAKED to the next phase. Hence,

Defect Leakage = [25 / (100 + 25)] * 100 = 20%

Means that 20% of the defects were leaked by the Testing team to the next phase, or in other words test team wasn't able to identify these 20% of the defects. Similar to DRE, Defect Leakage can be (or rather should be) computed for each SDLC phase and plotted on a bar graph to show the relative defect leakage for each phase.

Cost of finding a Defect

How to measure test effectiveness with respect to the money spent? Yeah! The cost of finding a defect!

Cost of finding a defect = Total effort (money) spent on testing / total defects found

Note: Total effort can be calculated by considering total resources, the duration and the billing rate.

Defect Age

Like everything in this universe, defects too have a life-cycle – from birth (new) till death (closed). Defect age is a measure of its oldness, i.e., how much time has elapsed since it was identified and NOT closed.

Defect Age (in Time) = Current Date (or Closed date) – Defect detection date

Note: Defect age can be calculated in hours or days. It is a measure to gauge the responsiveness of the development/testing team. Lesser the age better the responsiveness.

Weighted Defect Density

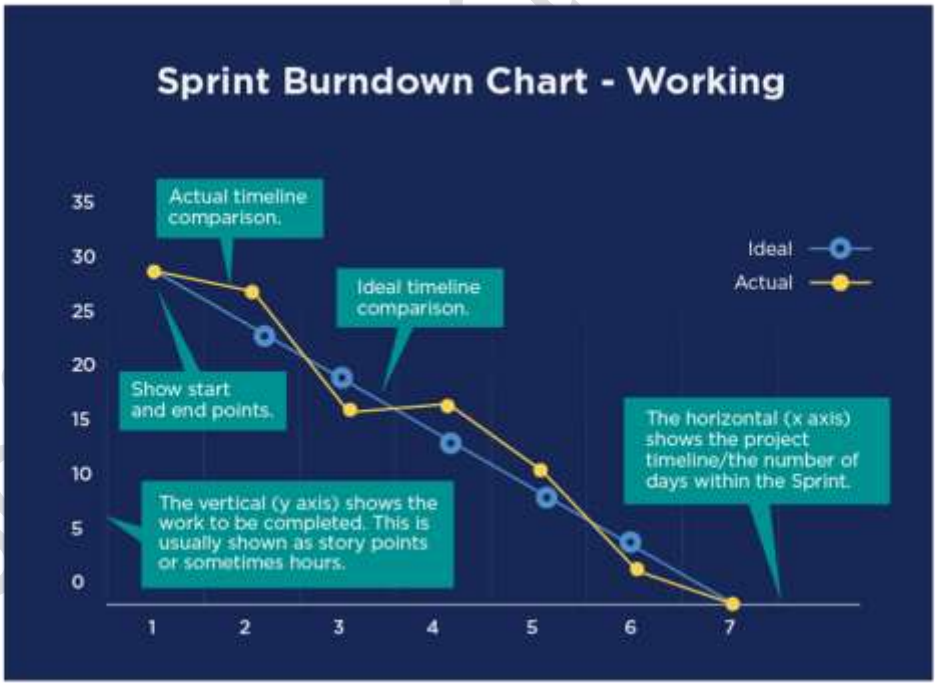
We already discussed the Defects Distribution by Priority & Severity. Now how to find the defect density with respect to severity or priority? i.e., what was the average defect severity per test case (or requirement) created.

Weighted Defect Density = [(5*High + 3*Medium + 1*Low Defects) / Size] * 100

Here "Size" can be considered as the number of requirements or test cases. Weights 5, 3 and 1 are assigned based on the defect severity of High, Medium and Low.

Burndown/Burnup charts [in Agile]

Burndown and burnup charts are two types of charts that project managers use to track and communicate the progress of their projects. A burn down chart shows **how much work is remaining** to be done in the project. Its purpose is to enable that the project is on the track to deliver the expected solution within the desired schedule. A burn up chart shows **how much work has been completed**, and the total amount of work. There are two main lines shown on the chart: one for the total project work planned, and the other for tracking the work completed to date.



Team Velocity [in Agile]

Agile Velocity is an extremely simple method for measuring the **rate at which scrum teams consistently deliver business value**. In other words - How much product backlog effort a team can handle in one sprint? It's the rate at which a team delivers stories from the product backlog, i.e., sum of estimates of delivered (i.e., accepted) features per iteration. It can be measured in story points, days, ideal days, or hours that the Scrum team delivers – all of which are considered acceptable.

A team delivers 20 and 30 story points in the first & second iteration respectively. The average velocity = Sum of Story points delivered / Number of iterations = (20 + 30)/2 = 25 Story points!

Test Closure

Test Summary Report

Finally, the QA team has done all the testing, reported all the defects, retested fixed defects and are done with a round of regression testing. The actual testing is COMPLETED.

But **how do stakeholders know about the final outcome of Testing**? Or what are the important points that testing covered which can be referred to build customer confidence and take appropriate decision to go-live? Few of the pointers would be,

- **Demo** of the developed & tested product. The team gives a demo of the product [to product owner & customer] in order to build confidence that it is actually what was expected (& without any major defects).
- **Test Summary report** is shared which clearly highlights the requirements that have been tested, list of defects found-fixed-and-closed, any pending minor defects, test coverage, test automation reports, recommendation about promoting the build to next environment, any risks we foresee, etc.

Overall, Test closure phase is to provide important information (related to product/software) to all the stakeholders which can help them take an informed decision about deploying to next environment or any feedback that they might have.

Go-No Go Meeting

By Definition: Before each public release all stakeholders meet to determine if the release criteria are met and we are good to deploy in production environment.

- Participants: make sure there is a representation (SPOCs) from all teams involved in the project - Dev, QA, release, DB, infra, client, etc. SPOCs as in who are the decision-makers.
- Evaluation: The most important aspect of a Go/No-Go meeting is to identify if there are any blockers for the particular release. I.e., blocking defects not fixed, not all requirements are demoed, QA test execution is not yet completed, release team & required infra is not yet ready, etc. It can be anything that can have an impact on production deployment and subsequent end-user usage.
- The Exit criteria OR checklist can vary on a project-to-project basis but few items to include can be requirements coverage, QA completion (including Unit test & UAT), Open defects, Risks (if any), infra readiness, team's availability, post-deployment support agreement, Roll-back plan, etc.

- Documentation: A Go/No-Go meeting should be documented in the form of evaluation criteria/checklist discussed and their corresponding RAG status (with comments) leading to the final Go/No-Go decision.
- Communication: The final outcome of the meeting should be communicated to ALL the stakeholders involved via email.

Retrospective

The team discusses what went right, what went wrong, and how to improve. They decide on how to fix the problems and create a plan for improvements to be enacted during the next sprint. Regardless of how good a team is, there is always room for improvement.

Production Go-live

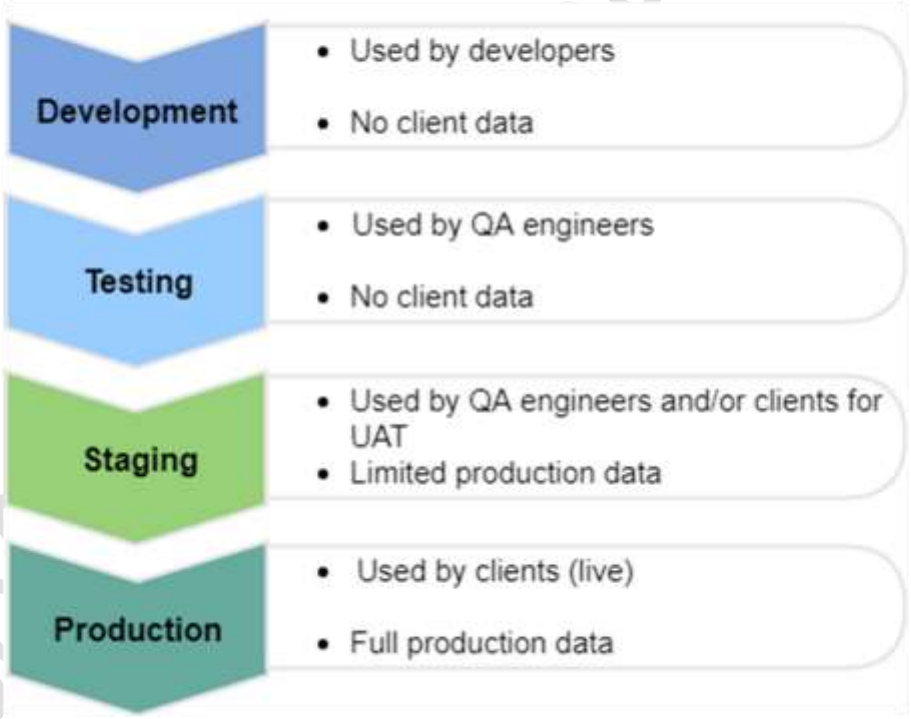
The D-Day

Finally, the D-Day has arrived. The culmination of weeks (or months) of project team's efforts. The discussed – designed - developed - & - tested product/software is ready to be used by end-users.

Go-live is the time at which something becomes available for use. In software development, go-live is the point at which **code moves from the test environment to the production environment**. As a verb, go-live means to make such an event happen.

At the time of the go-live, a system is officially and formally available to users who can then start using the new system.

E.g., you must have seen Mobile App updates, that's production go-live that a new updated version of the app is now available for users.



Fallback Testing

"Contingency plan for reverting to the old system or to an alternative emergency solution in the event of a failure of the new system during installation"

You deploy a solution in production environment and boom! It stops working. What do you do? Simple – revert to the old system (fallback). But that's not so easy. Scripts should be written to support the rollback of all changes done while installing the new build. And Fallback testing ensures that the old system works fine after rollback, that all new changes have been reverted successfully.

Why is it important? In case of rollback, you don't want any residual changes impacting the functioning of old existing system.

Test Management Tool

What & Why

Whether it is about capturing requirements, designing test cases, test execution reports, informing other team-members about Testing progress etc. a test management tool is mandatory.

Test Management Tools allow you and your team to manage requirements, test cases, test execution, defect tracking, reporting, monitor metrics, and integration with third-party tools.

Why do we need it?

Simple, because capturing & tracking ALL this information without using tools is really inefficient & time-consuming way. These tools help a test team to focus on main tasks at hand – building & testing a software that customer wants.

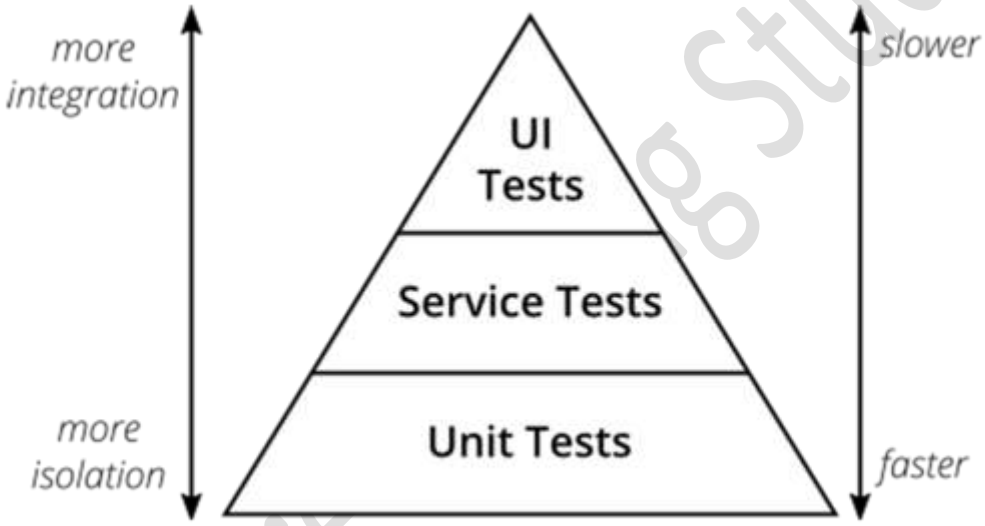
There are 100s of Test management tools available, both free & commercial. Some of the popular ones would be - Microfocus ALM, TestRail, TestLink, Zephyr, qTest, PractiTest, etc. JIRA is yet another popular tool which is used for overall Agile Project management.

Testing Glossary Part-I

The Testing Pyramid

The "Test Pyramid" is a metaphor that tells us to group software tests into buckets of different granularity. It also gives an idea of how many tests we should have in each of these groups.

The assumption here is that automated unit tests are cheap, easy, fast to run and isolated, compared to those slow, brittle, hard to write end-to-end tests that require a full working system and a web browser or mobile device. The foundation of a test effort should be unit tests, with fewer service tests and very few end-to-end tests, creating a bit of a pyramid.



Risk-Based and Database Testing

Risk-based Testing

As the name suggests, RBT is a testing approach to prioritize the testing efforts based on the likelihood and impact of risks in different functional software modules. E.g., For Amazon, getting the homepage right is of utmost importance. Then comes all other features. So basically, as a test team, you prioritize the test execution efforts based on which modules have the highest importance OR are in the high-risk areas.

Database Testing

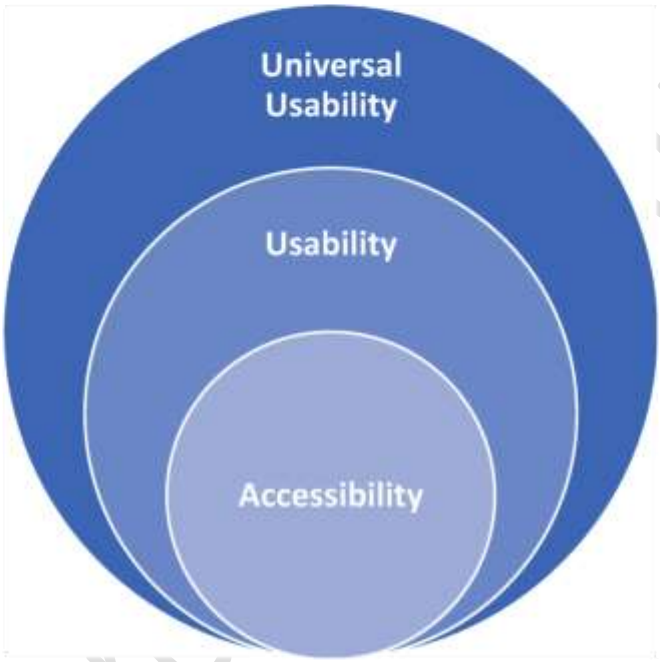
Also known as back-end testing, it focuses on testing the database & not the Web UI. It could be about validating the database schema, triggers, tables design, data create-read-update-delete operations, data security, database performance, etc.

Usability and Accessibility Testing

Usability Testing

In simple terms – testing the 'ease of use'. Why do you think some websites are more user-friendly? It involves checking if a new user can understand, learn & use the application easily, help contents, system navigation, pleasing design, etc.

Note: Usability testing is from client or end-user's perspective, i.e., it's subjective and depends on the targeted end-user or customer. User interviews, surveys, video recording of user sessions, and other techniques can be used.



Accessibility Testing

Access as in the 'right to use', i.e., is your software accessible by ALL? Say, what if a blind person wants to access your software. Did you ensure that the software is accessible by the disabled? E.g., Audio support for blind users, colour and contrast for the colour blind, font size for visually impaired, speech recognition, etc. Testing objective is to ensure the 'application accessibility by ALL'!

Portability, Conformance & Compliance Testing

Portability Testing

Portability, or in other words movability or transferability of a software or application. Movability as in the changes to the environment such as hardware, operating system or browser. The software may be installed in more than one environment or its executables may be created and run on different platforms. This form of testing is important if the customer intends to use the software application on more than one platform.

Conformance & Compliance Testing

Conformance testing verify the implementation conformance to industry standards, i.e., producing tests that verify that the software implementation conforms to industry standards with respect to different aspects like portability, inter-operability, and/or compatibility.

Compliance Testing checks whether the system was developed in accordance with standards, procedures, and guidelines. E.g., regulatory-compliance testing facilitates resolution of compliance issues and submission of necessary documentation to meet regulatory guidelines and mandates.

Internationalization & Localization

What if your software is hard-coded to support just one language? Say English. How will you target German (or any other local-language) end-users into using your software? The way out – Internationalize your software, i.e., remove the hard-coding and make the design generic such that it can support any number of languages and other geography-based features like currency, date format, etc. Officially *"**Internationalization** is the process of designing a software application so that it can potentially be adapted to various languages and regions without engineering changes."*

Now that you have internationalized the software, you need to add region-specific details for your target geographies. Say you want to add text-translations for Portuguese, German, French but not Chinese. I.e., though it's internationalized, but your software is localized for Portugal, Germany and France but not for China. Officially *"**Localization** is the process of adapting internationalized software for a specific region or language by adding locale-specific components and translating text."*

Testing Glossary Part-II

Performance Testing – Load, Volume and Stress

One of the most important and common out of all types of QA testing. The objective is not functional but to uncover performance issues such as network delay, data rendering, database transaction processing, load balancing between servers, throughput, response time, etc. (speed and efficiency of the system). Performance testing is critical for business intensive applications such as stock trade, financial transactions, etc. It's a broader set of Test type including Load, Stress & Volume tests – generally carried out using automation tools.

Load Testing

Load as in testing the 'capacity' of the application under test. Didn't get it? The common scenario in most of the IT firms – the company portal cannot be accessed as soon as you get the news that appraisal letters have been released. The reason – Load testing wasn't performed. The portal couldn't handle the Users' rush or concurrent access to the system.

The objective is to check the behaviour of the software under normal and over peak load conditions – at what point the system's response time degrades or fails. Load testing is usually performed using automated testing tools like JMeter, Load Runner, and Silk Performer – several virtual users can be created and then a script can be executed to check how the software behaves when multiple users try to access the system concurrently.

Volume Testing

Volume (meaning 'Bulk') testing is carried out to find the response of the software with different sizes of the data being sent/received or to be processed. E.g., If you were to be testing Microsoft word, volume testing would ensure if word can open, save and work on files of different sizes (10 to 100 MB).

Stress Testing

When do you get stressed? Yeah! When we can't handle the pressure. To a certain limit, we can handle the tension but beyond it – stressed.

Similarly, a software is built to handle a certain load, beyond that it breaks. The objective of 'Stress Testing' is to test the software under abnormal load conditions (beyond the acceptable limit) and then its performance is monitored to observe how the software would behave at breakpoint. There are many ways of creating abnormal conditions – the database can be turned off and on, complex database queries, continuous input to system, database load or network ports can be shut or restarted randomly. A graceful degradation under load leading to non-catastrophic failure is the desired result. The most common way of performing stress testing is by starting processes that would consume a lot of resources. Stress testing enables to check some of the quality attributes like robustness and reliability.

Compatibility Testing

Taking a cue from human relationships (people need to be compatible with each-other to make a cohesive society), what is it that Software interacts with? Yeah! The hardware resources like memory, disk drives & CPU, the Operating system, databases, web servers, application servers, different browsers, etc. that a customer may use. What if it fails in some interaction?

The objective of compatibility testing is to **ensure that the software (or its latest version) is well-suited for its surroundings** – hardware, servers, OS, CPU, etc. It determines the minimal and optimal configuration of hardware and software for the application under test to perform perfectly. The most popular form of compatibility is – Browser compatibility (application works fine in all in-scope browsers) and versions' compatibility (newer version is backward compatible or not). Compatibility testing can be performed manually or can be driven by an automated functional or regression test suite.

Security Testing

What if any one logging to Facebook can access your profile data & can post on your behalf? Critical security breach, right?

As the name suggests – the objective of security testing is to **verify how secure the software is to external or internal threats from humans and malicious programs**. E.g., software's authorization mechanism, how strong is authentication, maintaining data confidentiality & integrity, etc. In other words, Security testing assures that the program is accessed by the authorized personnel only. Authentication (login credentials are checked) and authorization (privileges to access restricted features) are considered to be two very important aspects of software testing. Security is especially important for applications that require firewalls, encryption, user authentication, financial transactions, or access to databases with sensitive data. Security testing requires good knowledge of application, technology, networking & security testing tools.

Web Application Security Testing Methodology

Besides authentication and authorization, security testing also deals with confidentiality, integrity, availability, non-repudiation, software data security, SQL insertion flaws, cross-site scripting attacks and other security related concern which is a complete different subject matter expertise by itself.

Penetration and Vulnerability Testing

Penetration Testing

Also known as PenTest in short, it's a type of security testing. The objective is to gauge how secure software and its environments (hardware, operating system and network) are when subject to attack by an external or internal intruder. Intruder can be a human/hacker or malicious programs. Pentest uses methods to forcibly intrude (by brute force attack) or by using a weakness (vulnerability) to gain access to a software or data or hardware with an intent to expose ways to steal, manipulate or corrupt data, software files or configuration. Penetration Testing is a way of ethical hacking, an experienced Penetration tester will use the same methods and tools that a hacker would use but the intention of Penetration tester is to identify vulnerability and get them fixed before a real hacker or malicious program exploits it.

Vulnerability Testing

Vulnerability or weakness in the software system that exposes its important functionality or data to unauthorized access. Vulnerability testing involves identifying & exposing the software, hardware or network vulnerabilities that can be exploited by hackers and other malicious programs likes viruses or worms. With increase in the number of hackers and malicious programs, Vulnerability Testing has become one of the critical types of QA testing for success of a business.

Recovery and Scalability Testing

Recovery Testing

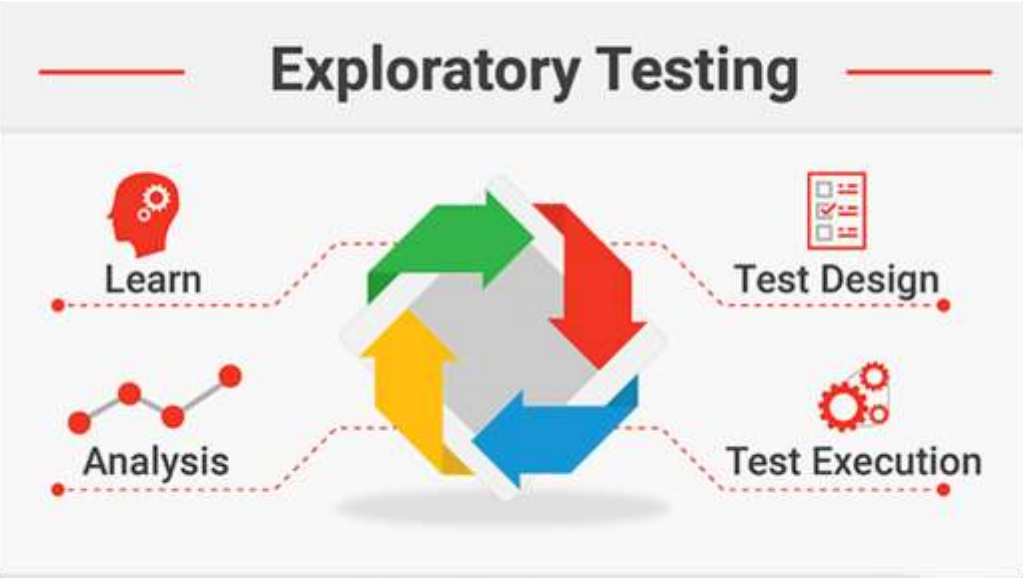
What if the system crashes? Even though it won't after all the testing is completed, but still we need to be prepared for the worst, i.e., testing how well a system recovers from crashes, hardware failures, or other catastrophic problems.

Scalability Testing

The objective is to test software's 'Scalability' – the ability of the software to scale up with increased users, increased transactions, increase in database size etc., i.e., how much more workload the software can support with expanding user base, transactions, data storage etc.

Exploratory testing

An approach to software testing that is often described as **simultaneous** learning, test design, and execution. No matter how many test cases of how many types you've created, you will run out of formally planned tests. Keep testing. Run new tests as you think of them, without spending much time preparing or explaining the tests. Trust your instincts.



Conclusion

Domain Knowledge

Domain is basically a grouping based on the type of business rather than technology/platform. Almost all IT companies organize their projects in different verticals based on different domains. Some of the common domains are Banking, Retail, Insurance, Healthcare, E-commerce, Telecom, etc.

In simple terms – "**Testing is context driven**". It's good if you already know the 'context' to be tested. Anyone can 'Check' but not 'Test'. The difference is same as a novice driver and an experienced one. Novice driver focus on just driving with the fear of road whereas experienced driver can gauge the air pressure, engine sound, music connections, rear-view, appropriate acceleration in traffic, etc.

- Suggestive: The job will be done anyway, but a domain expert can be suggestive. He / She can provide feedback, suggest for improvements, and be an active participant in a lot of decision-making meetings.
- Prioritize: A Domain expert can help prioritize the test areas most relevant to the application.
- Requirements: Quick understanding of requirements and thorough analysis.
- Effective: Better simulate the business flows and end-user actions, i.e. effective test scripts.
- Roles: Testers with domain expertise can play multiple roles like requirement analysis, client interaction, pitching for new business, training, etc.
- Defect Triage: Knowing how the application will be used and how it is expected to perform, will tell QA whether a given defect is trivial, significant, or critical.
- Client Relationship: Helps in building Client relationship since domain experts are likely to use business language while reporting & communication.

How will you test an e-commerce website if you don't know about portfolio management, order management, payment gateways, user management, etc.? How will you test a banking application if you don't know how a payment is processed end-to-end? When testing a healthcare product, it's good to know about terms like provider, inpatient, outpatient, and anything involving medicine will require the user to be aware of ICD's (International Classification of Diseases). Hope you got the point!

Become a Good Tester

A good Tester is built on not just one, but many important traits.

- **Curiosity**. Or in other words - an appetite to learn. Always curious about anything new (or different).
- **Technical**. A good Tester is technically strong. Technically doesn't mean only programming - he/she understands the Tech stack.
- **Observation**. A keen observation to identify the anomaly.

All these are important, but - "**Being Logical**" has the utmost importance. We as Testers have to understand the business, technology, product & process. Being logical helps in every area.

One of the important purposes of Software Testing existence is to verify & validate that the ultimate product delivered is useful & meets (rather exceeds) the business requirements. As a Software Tester, **voice your opinions** about the usefulness & clarity of the requirements and propose value-add that can enhance the overall user experience. Be the voice of quality. Raise any risks that you foresee. As a tester, we need effective communication with all the stakeholders involved – business, client, development, vendors, leadership, etc. Silently finding & fixing the defects is not enough, be the voice that is heard & responded to. **Happy Testing!**