# Module - 2 (Manual Testing)

### What is Exploratory Testing?

Exploratory Testing is a dynamic and informal software testing technique where the tester actively explores the system while simultaneously learning about it, designing test cases, and executing them. Unlike scripted testing, which follows predefined steps, exploratory testing is **unscripted**, allowing testers to adapt based on real-time findings. This approach is especially useful when requirements are vague, documentation is limited, or time is constrained.

In exploratory testing, the tester’s **creativity, domain expertise, intuition, and critical thinking** play a major role. It involves investigating the software to uncover bugs that traditional methods might miss. Testers often define “charters” to guide their sessions and document their observations, bugs, and coverage.

This method is highly effective for identifying usability issues, edge cases, and integration problems. It’s commonly used during **early test phases**, during **agile sprints**, or in **post-fix smoke/sanity testing**. While it may appear ad hoc, structured exploratory testing can be managed with techniques like Session-Based Test Management (SBTM).

The main advantage of exploratory testing is the ability to quickly find critical defects while minimizing preparation time. However, it requires experienced testers and may be harder to reproduce issues if not documented properly.

### What is a Traceability Matrix?

A Traceability Matrix (often called RTM — Requirements Traceability Matrix) is a document used in the software development life cycle to ensure that all **business requirements are properly covered by test cases**. It is used to **track and map** requirements to corresponding test cases, ensuring that nothing is missed during validation and verification processes.

The primary purpose of a traceability matrix is to validate that each requirement is accounted for through proper testing. It also helps identify missing test cases or overlooked functionalities. There are typically three types of traceability:

* **Forward Traceability**: Mapping requirements to test cases.
* **Backward Traceability**: Mapping test cases back to requirements.
* **Bidirectional Traceability**: Mapping both ways to maintain complete traceability.

A well-maintained RTM is especially useful during **impact analysis** when a requirement changes — testers can quickly identify which test cases need updates. It also supports audit processes and quality assurance by showing how thoroughly the product has been tested.

RTMs are often maintained in spreadsheets or test management tools like JIRA, TestRail, or Quality Center. Including columns like requirement ID, description, test case ID, status, and comments helps ensure thorough test coverage and traceability across project stages.

### What is Boundary Value Testing?

Boundary Value Testing (BVT) is a black box testing technique that focuses on the **edge conditions** of input domains. Errors often occur at the boundaries rather than the center of input ranges, making boundary testing a highly effective way to find defects. The idea is to test values at the **minimum, maximum, just below, just above**, and **on the boundary**.

For example, if an input accepts values from 1 to 100, the test cases might include 0, 1, 2, 99, 100, and 101. These boundary values are more prone to defects due to incorrect use of inequality operators (<, <=, >=, >), rounding errors, or missing checks.

BVT is widely used in **functional testing**, especially for validating user input fields, form submissions, API parameters, and any scenario involving numerical ranges or limits. It ensures that the software handles the **extremes of input data** properly without crashing or misbehaving.

One of the key advantages is that it helps **reduce test effort** while still providing good coverage, particularly for input validation. It complements equivalence partitioning and together they form a strong foundation for effective black box test design.

### What is Equivalence Partitioning Testing?

Equivalence Partitioning is a black box testing technique that aims to reduce the number of test cases while ensuring effective coverage. It involves dividing input data into **equivalence classes or partitions**, where each class represents a set of valid or invalid inputs that the system should treat the same way.

For instance, if a form accepts values from 1 to 100, then:

* Inputs from 1 to 100 form a **valid partition**
* Values less than 1 (e.g., 0, -1) form one **invalid partition**
* Values greater than 100 (e.g., 101, 200) form another **invalid partition**

From each partition, only one representative value is chosen for testing, as the system is expected to handle all values in that class similarly. This reduces redundancy and testing time.

This technique is effective for **input field validation**, **form testing**, and **decision-based logic**, where inputs can be grouped by expected behavior. It’s especially powerful when used along with boundary value testing.

Equivalence Partitioning helps **identify defects related to input handling** and ensures that all logical paths are tested without excessive effort. It also supports requirement-based testing, helping validate that each functional input condition is handled appropriately.

### What is Integration Testing?

Integration Testing focuses on verifying the **interaction between multiple modules or components** of a system. After unit testing verifies that individual components work correctly in isolation, integration testing checks how they behave when combined. The goal is to identify interface defects, data flow errors, or unexpected behavior resulting from module interactions.

There are several approaches:

* **Top-Down Integration**: Starts from the top-level module and integrates downwards using stubs.
* **Bottom-Up Integration**: Starts with low-level modules and integrates upwards using drivers.
* **Big Bang Integration**: Combines all modules at once and tests the whole system.
* **Sandwich/Hybrid Integration**: Combines top-down and bottom-up approaches.

Integration testing is essential in **complex systems**, especially when modules come from different developers or teams. It ensures modules pass correct data to each other, function calls are correctly handled, and error handling works across module boundaries.

Tools like JUnit, TestNG, Postman (for APIs), and integration test frameworks are commonly used. Proper integration testing helps uncover issues **early in the development cycle**, reducing the risk of major failures in system or acceptance testing phases.

### What Determines the Level of Risk?

The level of risk in software testing is influenced by the **likelihood of a failure occurring** and the **impact it would have** if it did. Risk-based testing helps prioritize what to test and how rigorously.

**Factors that determine risk include:**

* **Complexity**: Complex modules are more error-prone.
* **Usage Frequency**: Frequently used features have higher risk.
* **Business Criticality**: Features affecting finance, security, or compliance carry greater risk.
* **Historical Defects**: Modules with many past bugs are likely to fail again.
* **External Interfaces**: APIs, databases, or third-party services increase risk.
* **Changes Made**: Modified or newly added code tends to be unstable.

Risk is usually assessed using a **Risk Matrix**, combining impact and probability (e.g., High/High = Critical). This matrix guides where to focus testing efforts — critical, high-risk areas receive **more thorough and earlier testing**, while low-risk areas may be tested lightly or later.

Understanding and analyzing risk ensures **efficient use of resources**, reduces project delays, and improves software quality. Risk-based test strategies are essential in large, time-constrained projects.

### What is Alpha Testing?

Alpha Testing is a type of **internal acceptance testing** performed by the development or quality assurance team before the product is released to external customers. It is typically conducted **at the developer’s site**, in a **controlled environment**, and focuses on identifying bugs, missing features, or usability issues that could affect end-users.

This phase of testing simulates real-world usage using **white box and black box testing techniques**. Testers may perform exploratory, functional, regression, and performance testing during this stage. Feedback from alpha testing helps refine the application and improve stability before it reaches customers.

Alpha testing occurs **before beta testing** and often involves multiple iterations. Once major issues are resolved, the product may be passed to a limited group of internal users or stakeholders for further evaluation.

The main benefits of alpha testing are that it allows for **early bug detection**, helps ensure the software meets business requirements, and provides insights into the user experience. Since it’s conducted in-house, issues can be quickly logged and resolved.

This testing phase is particularly important for identifying flaws in core functionality, navigation, performance, and design before external users interact with the product.

### What is Beta Testing?

Beta Testing is the final phase of software testing before a product’s official release. It involves **real users testing the software in a real-world environment**. These users are typically external customers or selected participants who use the software and provide feedback regarding usability, functionality, and defects.

Beta testing allows developers to understand how the product behaves under real usage conditions, including different platforms, devices, and networks. It reveals bugs that were not found during internal testing, especially those related to performance, UI/UX, and compatibility.

There are two types of beta testing:

* **Closed Beta**: Limited group of users, often under NDA.
* **Open Beta**: Released to a wider audience for public feedback.

Beta testers report bugs, suggest improvements, and assess whether the product meets their needs. The development team then fixes critical issues before launching the final version.

The main goal of beta testing is to ensure **product readiness**, improve user satisfaction, and reduce the likelihood of failure post-launch. It also acts as a **marketing tool**, creating early user engagement.

Beta testing helps verify that the product is robust, user-friendly, and market-fit.

### What is Component Testing?

Component Testing, also known as **unit testing**, involves testing individual software components or modules in isolation to ensure they function correctly. This is the **lowest level of testing** in the software testing life cycle and is typically performed by developers.

Each component is tested independently from the rest of the system, often using **test stubs and drivers** to simulate interactions with other parts of the application. The goal is to verify the correctness of **logic, functionality, input/output handling**, and **error messages** within the module.

For example, a function that calculates discounts might be tested with different input values to ensure the output is accurate.

Component testing can be **automated** using tools such as JUnit, NUnit, or PyTest. Developers write test cases for each method, class, or function, which are run every time code is changed, ensuring no regressions are introduced.

The benefits include **early bug detection**, improved code quality, and faster debugging. By isolating defects within specific modules, component testing makes it easier to fix issues before they propagate to other parts of the system.

It is essential for maintaining **code reliability**, especially in complex or large-scale systems.

### What is Functional System Testing?

Functional System Testing is a **black box testing** technique that validates the complete and integrated system against its functional requirements. It ensures that the system behaves as expected for **various business scenarios**, without concern for internal code or architecture.

Testers focus on inputs, expected outputs, and user interactions. Commonly tested areas include **user login, workflows, transactions, form validations, business logic**, and **error handling**. Test cases are derived from requirement specifications, user stories, and acceptance criteria.

Functional system testing simulates real-world use cases and helps identify:

* Missing or incorrect functionality
* Data processing errors
* UI navigation issues
* Security violations

This type of testing is often automated using tools like Selenium, Katalon, or TestComplete, especially in regression testing. Manual testing is still widely used for exploratory, ad hoc, or UI-based validation.

Functional testing is critical because it directly impacts **user experience** and **business compliance**. By validating features from a user perspective, it ensures that the software delivers value and performs intended tasks reliably.

It is typically performed after integration testing and before user acceptance testing (UAT).

### What is Non-Functional Testing?

Non-Functional Testing assesses attributes that describe **how a system operates** rather than what it does. These attributes include **performance, scalability, reliability, usability, maintainability, security, and compatibility**.

Unlike functional testing, which checks if features work correctly, non-functional testing ensures that the system meets quality benchmarks and performs well under varying conditions. For example:

* **Performance Testing** checks response times and throughput.
* **Load Testing** evaluates behavior under expected user load.
* **Stress Testing** reveals how the system handles extreme conditions.
* **Usability Testing** checks ease of use and accessibility.
* **Security Testing** ensures data protection and access control.

Non-functional testing is often done using specialized tools:

* LoadRunner, JMeter for performance
* Burp Suite for security
* Lighthouse for usability and accessibility

It plays a critical role in **user satisfaction and product success**, especially for applications with high availability or user expectations, such as e-commerce sites or banking apps.

Proper non-functional testing helps detect bottlenecks, minimize risks, and ensures the software remains stable, scalable, and secure under real-world conditions.

### What is GUI Testing?

GUI (Graphical User Interface) Testing is a form of software testing that focuses on verifying the visual and interactive elements of an application. The primary goal is to ensure that the user interface functions correctly, looks consistent, and meets the design specifications. Testers validate components like buttons, menus, input fields, labels, icons, and layout across different devices and screen resolutions.

GUI testing covers both **functionality and usability**. For example, a login button must be clickable and trigger the correct function, and text boxes must accept and validate appropriate input. It also checks alignment, colors, fonts, error messages, and responsiveness.

GUI tests can be **manual or automated**. Tools like Selenium, TestComplete, or Katalon Studio are commonly used for automated GUI testing. It’s crucial to test UI elements across different browsers and operating systems to ensure **cross-platform compatibility**.

Since GUI is the part of the application directly visible to users, defects here can lead to a poor user experience and reduced customer satisfaction. GUI testing is especially important for consumer-facing apps, web applications, and mobile software where user interaction is critical.

Effective GUI testing helps build polished, professional, and accessible interfaces that meet user expectations.

### What is Adhoc Testing?

Adhoc Testing is an informal and unstructured testing approach where testers **explore the application without predefined test cases or documentation**. It is typically performed with the aim of discovering defects through **intuition, experience, and creativity**, rather than following a formal test plan. This type of testing is especially useful when time is limited or when certain areas need to be quickly validated.

Adhoc testing does not require extensive preparation. Instead, testers use their understanding of the application to explore its functionality in a **random or spontaneous manner**, often targeting areas where bugs are likely to be found based on past experience.

While some view it as unstructured, experienced testers often bring valuable insights, making it a powerful tool for uncovering **hidden defects**, especially UI glitches, logical inconsistencies, or integration errors.

It is most effective when performed after formal testing phases like functional or regression testing. One challenge of adhoc testing is the **lack of documentation**, which can make it difficult to reproduce bugs unless findings are immediately recorded.

Despite its informal nature, adhoc testing plays a crucial role in **identifying last-minute defects**, validating fixes, and ensuring overall application stability from a user’s perspective.

### What is Load Testing?

Load Testing is a type of non-functional testing that evaluates how a system behaves under **expected user load conditions**. Its primary objective is to verify the system’s **performance, stability, and response time** when subjected to normal or slightly above-normal usage levels. Load testing ensures that the software application can handle the intended number of concurrent users or transactions without performance degradation.

For example, in an e-commerce application, load testing checks how the system performs when 1,000 users browse products or 500 users perform simultaneous checkouts. It helps identify bottlenecks in areas like database queries, server response, or network bandwidth.

Tools like Apache JMeter, LoadRunner, and Gatling are commonly used to simulate virtual users and generate load on the system. Metrics monitored during load testing include **CPU usage, memory consumption, response time, throughput**, and **error rates**.

Load testing is typically conducted in a controlled environment and is crucial for validating **scalability and capacity planning**. It helps ensure user satisfaction by detecting performance issues before they affect real users.

Ultimately, load testing contributes to building **reliable, high-performing systems**, especially for applications expected to serve large audiences, such as banking, retail, or social platforms.

### What is Stress Testing?

Stress Testing is a type of non-functional testing that evaluates a system’s behavior under **extreme or beyond-normal load conditions**. The goal is to determine the application’s **robustness, stability, and error-handling capacity** when resources are stretched to the limit or intentionally overwhelmed.

This form of testing simulates scenarios such as an unusually high number of concurrent users, excessive data input, or sudden traffic spikes. The intention is not just to measure performance, but to observe **how the system fails**—whether it crashes, slows down, or recovers gracefully.

For example, an online banking application might be stress-tested with ten times the usual transaction load to see how it copes. Testers monitor **response times, error messages, server crashes**, and **data integrity** during this phase.

Tools like JMeter, LoadRunner, and BlazeMeter are often used to conduct stress tests. The outcome provides insights into the **breakpoint of the system**, revealing weaknesses in server configurations, database handling, or memory usage.

Stress testing is essential for applications where uptime is critical, such as financial systems, healthcare platforms, or government portals. It ensures the system won’t collapse under unexpected demand and helps in **planning for disaster recovery** and **failover strategies**.

### What is White Box Testing? List the types.

White Box Testing is a software testing technique that involves testing the **internal logic, structure, and code** of an application. Unlike black box testing, which focuses on functionality, white box testing requires knowledge of the underlying source code and is usually performed by developers.

It ensures that all code paths are properly executed and that logical conditions and loops behave as expected. This technique is useful for identifying **logic errors, code inefficiencies, unreachable code, and security vulnerabilities**.

Key types of white box testing include:

* **Unit Testing**: Testing individual methods or function.
* **Statement Testing**: Ensures each line of code is executed at least once.
* **Branch Testing**: Checks that every decision point (e.g., if-else) is tested for both true and false conditions.
* **Path Testing**: Covers all possible paths through a program.
* **Loop Testing**: Verifies correct functioning of loops (e.g., for, while).
* **Condition Testing**: Focuses on testing all logical conditions in a decision-making statement.

White box testing is often automated using tools like JUnit, NUnit, or PyTest. It is a critical part of **test-driven development (TDD)** and contributes to building high-quality, bug-free software from the ground up.

### What is Black Box Testing? What are the techniques?

Black Box Testing is a software testing technique where the tester evaluates the **functionality of the application without any knowledge of its internal code or structure**. The focus is on **inputs and outputs**—testers verify that the software produces the correct output for a given input, based on the functional specifications.

It is primarily used during **system testing, acceptance testing**, and **functional testing**. Black box testing is especially effective for validating user interfaces, APIs, workflows, and overall behavior of the application from the end-user perspective.

Common black box testing techniques include:

* **Equivalence Partitioning**: Divides input data into valid/invalid groups to reduce test cases.
* **Boundary Value Analysis**: Focuses on input edges where bugs often occur.
* **Decision Table Testing**: Uses logical rules and conditions to determine test cases.
* **State Transition Testing**: Tests how the system moves from one state to another based on events or inputs.
* **Use Case Testing**: Based on user scenarios and business flows.

Black box testing can be manual or automated (e.g., Selenium, TestComplete). It helps uncover **functional defects, UI issues, and integration problems** without the need to inspect code.

### Mention What Are the Categories of Defects?

Defects in software can be categorized based on their nature, impact, and where they occur in the software lifecycle. Understanding defect categories helps teams **track, prioritize, and fix issues efficiently**.

Common categories include:

1. **Functional Defects**: When a feature doesn’t behave as intended. Example: A login button doesn’t submit credentials.
2. **Performance Defects**: Related to speed, response time, and resource usage. Example: Page load takes 10 seconds instead of 2.
3. **UI/UX Defects**: Errors in layout, alignment, font, color, or navigational flow. Example: Overlapping buttons on mobile view.
4. **Compatibility Defects**: Occur when software behaves differently on different browsers, OS, or devices.
5. **Security Defects**: Issues that expose the system to risks like unauthorized access, data breaches, etc.
6. **Integration Defects**: Arise when modules fail to communicate properly. Example: Incorrect data flow between frontend and backend.
7. **Data Defects**: Include data corruption, loss, or incorrect database handling.
8. **Boundary/Validation Defects**: Errors caused by missing or incorrect input validations.

Categorizing defects helps in **root cause analysis, release planning**, and improving the **overall quality process**.

### What is Big Bang Testing?

Big Bang Testing is an **integration testing approach** where all individual modules are combined and tested **as a complete system**, in one go, without testing components incrementally. This method waits until all parts are developed, then integrates them simultaneously to test the final system.

While simple in concept, Big Bang testing is risky. If defects are found, it becomes **difficult to trace the root cause**, since all modules are integrated at once. There’s often little time left to fix critical issues, making this method unsuitable for large, complex systems.

It is best used in **smaller projects** or when dependencies between modules are low. Since there is no gradual build-up or intermediate testing, this approach demands **thorough system-level test cases** to ensure everything functions correctly post-integration.

Benefits of Big Bang Testing include:

* No need for stubs/drivers
* Entire system is tested in a real environment

Drawbacks:

* Late bug detection
* High debugging cost
* Poor error isolation

Big Bang Testing may be used when deadlines are tight or in proof-of-concept projects, but most modern development follows more structured integration methods like **top-down**, **bottom-up**, or **incremental testing** for better defect isolation and risk management.

### What is the Purpose of Exit Criteria?

Exit Criteria define the **conditions that must be met before a testing phase or the entire testing process can be concluded**. It serves as a formal checkpoint to ensure that the application is sufficiently tested and ready for release or the next phase.

Exit criteria are determined during the **test planning stage** and vary depending on project size, risk level, and client expectations. These criteria help assess test completeness and product quality.

Common exit criteria include:

* **Test case execution**: A certain percentage (e.g., 95%) of test cases must be executed.
* **Defect thresholds**: No critical or high-severity defects remain open.
* **Code coverage**: Minimum unit/integration test coverage (e.g., 80%) achieved.
* **Passed tests**: A specified pass rate for critical test cases.
* **Regression testing completed**
* **Sign-off from stakeholders**

Exit criteria ensure testing is not stopped prematurely and that the system meets predefined **quality benchmarks**. If exit criteria aren’t met, testing continues or defects must be deferred formally.

This concept promotes transparency, **risk management**, and consistent decision-making before delivering the product to production or clients.

### When Should Regression Testing Be Performed?

Regression Testing should be performed whenever **code changes are made** to ensure that new updates haven’t adversely affected existing functionalities. This is essential because a bug fix or new feature can unintentionally introduce issues in previously working areas of the software.

**Common scenarios for regression testing**:

* After bug fixes or patches
* After enhancements or new feature development
* After environment or configuration changes
* After performance or security upgrades
* During continuous integration builds

Regression testing can be **manual or automated**. Automation is ideal for large applications where the same test cases need to be rerun multiple times. Tools like Selenium, QTP, and TestNG help maintain a robust regression test suite.

It is crucial in agile and DevOps environments where frequent releases and quick feedback loops are required. Regression tests help detect side effects early, reduce risk, and ensure consistent user experience.

Good regression testing practices include:

* Prioritizing critical functionality
* Maintaining reusable test scripts
* Updating the test suite with new defect scenarios

Ultimately, regression testing **protects software stability** and is a backbone of reliable, scalable development processes.

### What are the 7 Principles of Software Testing?

The **seven principles of software testing**, defined by ISTQB, guide testing practices across industries:

1. **Testing Shows Presence of Defects**: Testing can reveal defects but cannot prove the absence of all bugs.
2. **Exhaustive Testing is Impossible**: It’s not feasible to test every input combination; instead, use risk-based and prioritized testing.
3. **Early Testing Saves Time and Money**: Start testing in early stages (like during requirement reviews) to detect defects sooner.
4. **Defect Clustering**: A small number of modules typically contain most of the bugs (Pareto principle – 80/20 rule).
5. **Pesticide Paradox**: Repeating the same tests won’t find new bugs; tests need to be regularly updated.
6. **Testing is Context Dependent**: The type, depth, and method of testing depend on the project (e.g., medical vs gaming apps).
7. **Absence-of-Errors Fallacy**: A bug-free program that doesn’t meet user needs is still a failure.

These principles highlight the **limitations, strategies, and value** of testing. Applying them ensures more **focused, effective, and efficient** test planning and execution.

### Difference Between QA, QC, and Tester

**Quality Assurance (QA)** is a **process-oriented** practice that ensures software development and maintenance processes are followed to meet quality standards. QA focuses on preventing defects by improving development workflows, documentation, and standards compliance.

**Quality Control (QC)** is **product-oriented**, focusing on identifying defects in the actual product through inspections and testing. QC is a subset of QA and is performed after development to verify the final output meets quality expectations.

A **Tester** is a professional responsible for **executing test cases**, finding bugs, and validating application behavior. Testers often operate under QC but also contribute to QA by reporting process gaps or inconsistent requirements.

* **QA**: Prevents defects (proactive), includes reviews, audits, and process improvement.
* **QC**: Detects defects (reactive), includes actual testing and inspections.
* **Tester**: Executes the QC process, reports issues, and ensures compliance with requirements.

In summary:

* QA = Process to build quality
* QC = Process to verify quality
* Tester = Person who performs testing activities

### Difference Between Smoke and Sanity Testing

**Smoke Testing** is a preliminary test to check whether the basic functionalities of an application are working. It is typically conducted after a new build is deployed and focuses on critical paths like login, home page loading, or core transactions. If smoke testing fails, the build is rejected.

**Sanity Testing** is a narrow, focused testing phase done after minor bug fixes or enhancements to verify that specific functionalities or issues have been resolved and that no new bugs were introduced in related areas.

**Key Differences**:

* **Scope**: Smoke is broad; sanity is focused.
* **Goal**: Smoke checks build stability; sanity validates changes.
* **Timing**: Smoke occurs early in the testing cycle; sanity comes later, often after regression.
* **Automation**: Smoke is often automated; sanity is mostly manual.

Both tests act as **gatekeepers**, reducing wasted time on unstable builds or low-quality releases.

### Difference Between Verification and Validation

**Verification** is the process of evaluating work-products (documents, code, design) to check whether they meet specified requirements **without executing the code**. It answers the question, “Are we building the product right?”

**Validation** checks the final product by **executing the software** to confirm it meets user needs and business requirements. It answers the question, “Are we building the right product?”

**Key Differences**:

* **Verification**: Static process; includes reviews, walkthroughs, inspections.
* **Validation**: Dynamic process; involves actual testing.
* **Focus**: Verification ensures correctness; validation ensures usefulness.

Both are essential. Verification catches issues early, saving time and cost; validation ensures customer satisfaction and real-world usability.

### Explain Types of Performance Testing

Performance Testing evaluates how a system performs under various conditions. Key types include:

1. **Load Testing**: Checks system behavior under expected user loads.
2. **Stress Testing**: Tests the system under extreme or breaking-point conditions.
3. **Spike Testing**: Measures system reaction to sudden, extreme spikes in traffic.
4. **Endurance Testing (Soak Testing)**: Tests system stability under continuous load for extended periods.
5. **Scalability Testing**: Assesses the system’s ability to scale up/down in response to increasing workloads.
6. **Volume Testing**: Involves large volumes of data to test database and storage performance.

These tests ensure systems are **responsive, reliable, and scalable**, reducing the risk of performance issues in production.

### What is Error, Defect, Bug, and Failure?

* **Error**: A human mistake made during coding or design (e.g., a typo or logic error).
* **Defect**: A deviation in the code or product caused by an error. It’s detected during testing.
* **Bug**: An informal term for a defect found in the software.
* **Failure**: When the system behaves incorrectly in production due to an unresolved defect.

In short:  
 **Error → Defect → Bug → Failure (if not fixed before release)**

Understanding the distinction helps with accurate communication, bug reporting, and quality analysis.

### Difference Between Priority and Severity

* **Severity** refers to **how badly the defect impacts the system’s functionality**. It is assigned by testers.
* **Priority** indicates **how urgently the defect needs to be fixed**. It is usually set by the project or product manager.

**Examples**:

* High Severity, High Priority: Login failure on a live site.
* High Severity, Low Priority: A crash in a rarely used feature.
* Low Severity, High Priority: Spelling mistake on the homepage.
* Low Severity, Low Priority: Minor UI issue in a non-critical area.

Proper classification ensures effective **triaging and resource allocation** during defect resolution.

### What is Bug Life Cycle?

The **Bug Life Cycle** (or Defect Life Cycle) represents the various stages a software bug goes through from discovery to closure. It helps track, prioritize, and manage defects efficiently within the development and testing process.

**Typical stages in a bug life cycle:**

1. **New**: The tester reports a defect and logs it in the tracking tool.
2. **Assigned**: The bug is assigned to a developer for analysis.
3. **Open**: The developer starts working on the issue.
4. **In Progress/Fixed**: The defect is fixed by the developer.
5. **Ready for Retest**: The fix is submitted for verification.
6. **Retesting**: QA re-tests to verify the fix.
7. **Reopen** (if not fixed) or **Verified** (if fixed).
8. **Closed**: The bug is resolved and verified by QA.
9. **Deferred**: Fix is postponed to a later release.
10. **Rejected/Invalid**: If the issue is not a defect or is not reproducible.

Tools like **JIRA, Bugzilla, and Mantis** help manage the bug life cycle. Having a structured life cycle ensures transparency, proper communication, and efficient resolution, which directly contributes to software quality and user satisfaction.

### Difference Between Functional and Non-Functional Testing

**Functional Testing** verifies that the system functions according to the specified requirements. It answers the question, “Does the system do what it is supposed to do?” This type of testing includes validating user inputs, business processes, workflows, and output. Common types include unit testing, integration testing, system testing, and user acceptance testing.

Example: Checking whether the login page accepts valid credentials and navigates to the dashboard.

**Non-Functional Testing**, on the other hand, evaluates how well the system performs under various conditions. It focuses on attributes such as performance, security, usability, scalability, and reliability. It answers, “How well does the system perform?”

Example: Measuring how fast the dashboard loads with 1,000 concurrent users.

| **Criteria** | **Functional Testing** | **Non-Functional Testing** |
| --- | --- | --- |
| Focus | What the system does | How the system performs |
| Based on | Business requirements | Quality attributes |
| Tools (examples) | Selenium, TestNG | JMeter, LoadRunner, Burp Suite |

Both are essential for delivering a **high-quality, user-friendly** product. Functional testing ensures features work; non-functional testing ensures performance, security, and user experience.

### Difference Between STLC and SDLC

**SDLC (Software Development Life Cycle)** is the process of developing software from idea to deployment and maintenance. It includes phases like:

1. Requirements Gathering
2. Design
3. Development
4. Testing
5. Deployment
6. Maintenance

It governs the **entire lifecycle of a software product**, involving developers, business analysts, and project managers.

**STLC (Software Testing Life Cycle)** is a subset of SDLC, focusing solely on the **testing process**. It includes:

1. Requirement Analysis
2. Test Planning
3. Test Case Design
4. Environment Setup
5. Test Execution
6. Test Closure

STLC ensures that testing is **systematic, efficient, and traceable**.

| **Aspect** | **SDLC** | **STLC** |
| --- | --- | --- |
| Scope | Entire software development | Testing-specific processes |
| Involves | Developers, PMs, Designers | QA/Testers |
| Output | Functional software | Tested and validated product |

STLC is part of the SDLC testing phase, but it has its own life cycle to plan, execute, and report on testing activities. **Both work together** to ensure the software is built correctly and meets quality expectations.

### Difference Between Test Scenario, Test Case, and Test Script

* **Test Scenario**: A high-level idea or functionality to be tested. It describes *what* to test. Example: “Verify login functionality with valid and invalid inputs.”
* **Test Case**: A detailed document that describes the input, expected output, steps to be followed, and the actual result. It includes test data, preconditions, postconditions, and status. Example:  
  + Step 1: Enter valid username
  + Step 2: Enter valid password
  + Step 3: Click login
  + Expected result: User navigates to dashboard
* **Test Script**: A set of instructions or code (automated or manual) to perform testing. It is used especially in **automation testing**. Example: A Selenium script in Java that enters credentials and validates dashboard access.

| **Term** | **Description** | **Level** |
| --- | --- | --- |
| Scenario | What to test | High-level |
| Case | How to test | Medium-level |
| Script | Actual code or instructions to test | Execution-level |

* All three are integral to **test planning and execution**, helping structure testing efforts from idea to automation.

### What is a Test Plan? What Information Should Be Covered?

A **Test Plan** is a comprehensive document that outlines the **scope, objectives, approach, schedule, and resources** required for testing activities in a project. It serves as a blueprint for the QA team, ensuring structured, traceable, and goal-oriented testing.

**Key components of a test plan include:**

* **Test Objectives**: What needs to be validated.
* **Test Scope**: In-scope and out-of-scope items.
* **Test Strategy**: Types of testing (manual/automated), techniques, tools used.
* **Test Deliverables**: Test cases, defect logs, reports.
* **Environment Details**: Hardware, software, test data, and tools required.
* **Schedule**: Timelines, milestones, testing phases.
* **Roles and Responsibilities**: Team members and their tasks.
* **Entry/Exit Criteria**: When to start and stop testing.
* **Risk Management**: Risks, assumptions, and mitigation plans.

A well-defined test plan ensures **resource optimization, quality tracking**, and **early risk identification**. It acts as a communication tool between stakeholders, testers, and developers. Test plans can be created using tools like Microsoft Word, Excel, or within test management tools like JIRA, TestRail, or Zephyr.

### Bug Categories Are…

Bugs can be categorized based on their **nature, impact, and location**. Common categories include:

1. **Functional Bugs**: Errors in core application functionality (e.g., login failure).
2. **UI/UX Bugs**: Design, alignment, font, or color issues (e.g., unreadable text).
3. **Performance Bugs**: Delays, hangs, or high resource usage.
4. **Security Bugs**: Vulnerabilities like XSS, SQL injection, or data leakage.
5. **Compatibility Bugs**: Issues on different devices, browsers, or OS.
6. **Integration Bugs**: Errors during interaction between modules or third-party systems.
7. **Data Bugs**: Incorrect or corrupt data handling.
8. **Boundary Bugs**: Defects at the edge of input ranges.
9. **Localization Bugs**: Errors in language, currency, or date formats for different regions.

Categorizing bugs helps in **reporting trends**, prioritizing defect resolution, and focusing future test efforts. It also enables better **root cause analysis**, helping development teams improve code quality and prevent recurring issues.

### Advantages of Bugzilla

**Bugzilla** is an open-source bug tracking tool developed by Mozilla. It is widely used to report, track, and manage software defects. Its key advantages include:

1. **Free and Open Source**: No licensing cost; customizable as per project needs.
2. **Email Notifications**: Automatic alerts to stakeholders on bug updates.
3. **Advanced Search and Reporting**: Custom queries, filters, and scheduled reports.
4. **Permission and Role Control**: Restrict access based on user roles.
5. **Audit Trail**: Maintains a history of all changes made to a bug.
6. **Linking and Dependencies**: Track relationships between bugs and their impact.
7. **Custom Fields**: Tailor the bug form based on project needs.
8. **Time Tracking**: Log estimated and actual time for issue resolution.
9. **Integrations**: Supports integration with version control and CI tools.

Bugzilla provides a **robust, scalable, and flexible** platform for managing defects across small and large teams. It is especially useful in projects requiring transparency, auditability, and structured defect lifecycle control.

### What are the Different Methodologies in Agile Development Model?

Agile is a flexible software development approach focused on **iterative progress, customer collaboration, and rapid delivery**. Within Agile, there are several popular methodologies, each with unique practices but all adhering to Agile principles:

1. **Scrum**: The most widely used Agile framework. It organizes work into **Sprints** (2–4 weeks), and includes roles like **Product Owner**, **Scrum Master**, and **Development Team**. Work is planned via a **Product Backlog** and tracked daily using stand-up meetings.
2. **Kanban**: Emphasizes **visual task management** using a board with columns like “To Do,” “In Progress,” and “Done.” It promotes continuous delivery and limits Work In Progress (WIP).
3. **Extreme Programming (XP)**: Focuses on **technical excellence**, with practices like **pair programming**, **test-driven development (TDD)**, and frequent releases.
4. **Lean**: Originating from manufacturing, Lean focuses on **eliminating waste**, delivering only what is necessary, and improving efficiency.
5. **Crystal**: A lightweight methodology that varies based on team size and criticality, promoting **frequent delivery** and **team communication**.
6. **Feature-Driven Development (FDD)**: Builds software around **features** by following a five-step process starting from domain modeling to design and build by feature.

Each methodology shares Agile's core values but tailors practices based on team needs, project complexity, and organizational culture.

### Explain the Difference Between Authorization and Authentication in Web Testing. What are the Common Problems Faced in Web Testing?

**Authentication** is the process of verifying a user's identity—*Who are you?* It ensures only valid users can access the system, typically via **username/password, OTPs, biometrics**, etc.

**Authorization** determines what an authenticated user can access—*What are you allowed to do?* For example, an admin can access user management features, while a regular user cannot.

In **web testing**, both need thorough validation to prevent unauthorized access and data breaches.

**Common problems faced in web testing include**:

* **Broken links** and navigation errors.
* **Cross-browser issues** due to rendering differences.
* **Poor responsiveness** on mobile devices.
* **Security vulnerabilities** like XSS, CSRF, and SQL injection.
* **Session management issues** (timeouts, cookie theft).
* **Slow performance** due to unoptimized resources.
* **Form validation errors** or missing client/server-side checks.
* **Broken functionality** due to JavaScript or plugin incompatibility.

Web testing requires validating **UI, functionality, performance, compatibility, and security**. Testers use tools like Selenium for automation, OWASP ZAP for security, and Lighthouse for performance auditing. Testing both authentication and authorization is critical to protect sensitive data and prevent misuse.

### When to Use Usability Testing?

**Usability Testing** is conducted to evaluate how easily users can interact with a system or product. It answers key questions like: *Is the product intuitive? Can users complete tasks efficiently and without frustration?*

This testing is typically used:

* **During early design stages**: Wireframes or prototypes can be tested for basic interaction flow.
* **Before product launch**: To gather user feedback and make last-minute UI/UX improvements.
* **When introducing major changes**: A redesign or feature overhaul warrants usability checks.
* **If users report confusion or abandonment**: Frequent user complaints or drop-offs indicate usability flaws.

Usability testing is crucial for:

* Consumer-facing apps (e.g., e-commerce, banking, healthcare)
* Websites with complex navigation
* Mobile apps, where screen space is limited

Users are observed while performing tasks. Their **time to complete tasks, error rate, satisfaction levels**, and behavioral cues are analyzed. Tools like Maze, UserTesting.com, or moderated sessions help gather insights.

The key goal is to **ensure ease of use, accessibility**, and **a smooth user journey**. A product may function correctly but still fail if users find it confusing. That’s why usability testing is essential to deliver user-centered, accessible software.

### What is the Procedure for GUI Testing?

**GUI (Graphical User Interface) Testing** verifies that the visual elements of an application—such as buttons, menus, forms, and layouts—function and appear as intended. It ensures the interface is user-friendly, responsive, and consistent across devices and platforms.

**Procedure for GUI Testing:**

1. **Understand Requirements & Design**: Review UI specifications, style guides, and mockups to know how the interface should behave and look.
2. **Create GUI Test Cases**:  
   * Validate all UI elements (text boxes, buttons, checkboxes, dropdowns).
   * Test navigation flow, labels, alignments, and visibility.
   * Include both positive and negative input cases.
3. **Test Visual Elements**:  
   * Check fonts, colors, sizes, spacing, and themes.
   * Ensure consistency across pages/screens.
4. **Test Functional Aspects**:  
   * Ensure UI components trigger the correct backend operations.
   * Verify tooltips, error messages, and confirmations.
5. **Test Responsiveness**:  
   * Resize the screen and verify layout adjustments.
   * Check behavior across resolutions and screen orientations.
6. **Cross-Browser & Cross-Device Testing**:  
   * Test UI across browsers (Chrome, Firefox, Safari) and devices (mobile, tablet, desktop).
7. **Accessibility Checks**:  
   * Use tools to ensure compliance with standards (e.g., WCAG).
   * Validate keyboard navigation and screen reader support.
8. **Use Automation Tools** (for regression): Selenium, Katalon Studio, or TestComplete.

The goal is to ensure **a seamless, intuitive, and visually correct interface** that provides a high-quality user experience.