SDLC (Software Development Life Cycle)

The Software Development Life Cycle (SDLC) is a structured process followed by software development teams to ensure the efficient production of high-quality software. It consists of several distinct phases, beginning with **requirement gathering and analysis**, where developers work closely with stakeholders to understand business needs and document them in detail. This is followed by the **planning phase**, where the scope of the project is defined, resources are allocated, timelines are estimated, and risks are assessed. The third phase is **design**, where the system's architecture is created. This includes both high-level design, such as overall structure and technology choices, and low-level design, which focuses on logic, database schema, and user interfaces.

Once the design is approved, the **development or implementation phase** begins. Here, the actual coding takes place according to the specified design. After the software is built, it enters the **testing phase**, where it is rigorously evaluated through various testing methods like unit testing, integration testing, system testing, and user acceptance testing to ensure that the product is bug-free and meets all requirements. After successful testing, the software is moved to the **deployment phase**, where it is released for use in a live environment. Finally, the **maintenance phase** involves ongoing support to fix issues, implement updates, and add new features as needed.

There are several SDLC models that organizations can choose from based on their needs, including the **Waterfall model** (a linear and sequential approach), the **Agile model** (an iterative and flexible approach focused on collaboration and customer feedback), the **Spiral model** (which combines iterative development with risk analysis), and the **V-Model** (which emphasizes validation and verification alongside development). More modern approaches, such as **DevOps**, focus on continuous integration and delivery to streamline the development-to-deployment pipeline. Overall, SDLC ensures that software development is carried out in a planned, disciplined, and repeatable manner, leading to reliable and efficient systems.

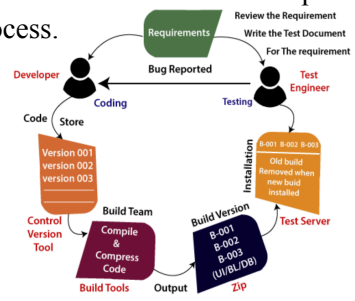
Software Testing Life Cycle (STLC)

The **Software Testing Life Cycle (STLC)** is a systematic process followed by quality assurance teams to ensure that software applications are thoroughly tested and meet the desired quality standards. It defines the phases involved in the testing of software, from planning to test closure. The cycle begins with the **requirement analysis** phase, where testers study and understand the testing requirements based on the software requirement specifications. This helps identify what needs to be tested and the scope of the testing effort. Once the requirements are clear, the **test planning** phase starts, where the test strategy is defined, resources are allocated, tools are selected, and the schedule is prepared. A detailed **test plan document** is created during this phase.

The next step is **test case design and development**, where test cases and test scripts are created based on the requirements and planning. Test data is also prepared during this phase. Once test cases are ready, the team proceeds with the **test environment setup**, which ensures that all software, hardware, and network configurations are in place for executing the test cases. After the environment is ready, the **test execution** phase begins, where testers run the test cases, record the outcomes, and report any defects found during the

testing. The development team then works to fix these issues. Once testing is complete and all major bugs are resolved, the cycle moves to the **test closure** phase, where testing activities are wrapped up, and final reports are prepared. Lessons learned are documented for future projects, and a test summary report is shared with stakeholders.

STLC plays a crucial role in delivering high-quality software products by ensuring that bugs and issues are identified and resolved before the product reaches end-users. Each phase in STLC is carried out in a planned and structured manner, often aligned with the corresponding phases in the Software Development Life Cycle (SDLC), ensuring a seamless and efficient software delivery process.



Software Build Process

Manual Testing:

Manual Testing is a type of software testing in which test cases are executed manually by a tester without using any automation tools. The main goal of manual testing is to identify bugs, issues, or unexpected behaviors in a software application by simulating user interactions and checking if the software functions according to the specified requirements. Testers follow a set of pre-written test cases or exploratory testing techniques to validate various aspects of the application such as functionality, usability, and performance. This process requires a deep understanding of the application's requirements and user flows. Manual testing is particularly useful in the early stages of development, during UI/UX validation, or in scenarios where automation is not feasible or cost-effective.

The **types of manual testing** include **Black Box Testing**, **White Box Testing**, and **Grey Box Testing**. Black Box Testing focuses on verifying the software's functionality without knowledge of the internal code, while White Box Testing involves testing with knowledge of the source code, and Grey Box Testing combines both approaches. Manual testing also includes various levels such as **Unit Testing**, **Integration Testing**, **System Testing**, and **User Acceptance Testing (UAT)**. Though manual testing is time-consuming and prone to human error compared to automated testing, it remains an essential part of the testing life cycle, especially for exploratory, ad-hoc, and usability testing where human judgment is crucial.

Agile Methodologies :

Agile methodology is a dynamic and customer-focused approach to project management and software development, designed to respond effectively to changing requirements and market needs. Unlike traditional Waterfall methods, where all project phases are planned and completed sequentially, Agile emphasizes delivering small, incremental improvements through iterative cycles, often called "sprints" or "iterations." Agile is based on four key values outlined in the Agile Manifesto: valuing individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over strictly following a plan. These

values are supported by twelve principles, which encourage continuous delivery of valuable software, embrace changes even late in development, promote close collaboration between business and technical teams, and emphasize building projects around motivated individuals. Agile also encourages sustainable work pace, technical excellence, and simplicity by maximizing the amount of work not done.

The typical Agile lifecycle includes several stages: starting with the **concept phase**, where the team identifies needs and defines initial goals; the **inception phase**, where teams are formed, technologies chosen, and an initial backlog of work is created; **iteration planning**, where detailed planning for the first sprint occurs; **iteration execution**, where developers design, code, and test small features; **review and demo**, where completed features are showcased to stakeholders for feedback; **retrospective**, where the team reflects on successes and areas for improvement; and finally, **release**, where working software is delivered to the user. Agile projects usually involve constant feedback loops, allowing for priorities and approaches to shift as new information becomes available.

There are several popular frameworks that implement Agile principles, including **Scrum**, **Kanban**, **Extreme Programming (XP)**, **Lean**, and **Scaled Agile Framework (SAFe)**. Scrum, perhaps the most well-known, organizes work into sprints of two to four weeks, with clearly defined roles like Product Owner, Scrum Master, and Development Team, and ceremonies like Daily Stand-ups, Sprint Reviews, and Sprint Retrospectives. Kanban, in contrast, focuses on visualizing work through boards and limiting work-in-progress to improve flow and efficiency. XP is heavily focused on technical practices like test-driven development and continuous integration, while Lean emphasizes minimizing waste and maximizing value to the customer.

Agile's major benefits include faster time-to-market, improved product quality through continuous testing and integration, better alignment with customer needs, and highly motivated teams who take ownership of their work. However, Agile also comes with challenges. It demands high levels of collaboration and communication, can be difficult in geographically dispersed teams, and may lead to issues if team members or stakeholders are not committed to active participation. Additionally, budgeting and long-term forecasting can be more complex in Agile compared to traditional approaches. Nonetheless, when applied well, Agile leads to products that are more aligned with user expectations, encourages innovation through quick adaptation, and creates a culture of continuous learning and improvement within organizations.

Drawbacks of the Waterfall Model:

1. **Inflexible to Changes**
   Once a phase is completed, it's difficult to go back and make changes. If requirements evolve (which they often do), it’s tough and expensive to adapt.
2. **Late Testing Phase**
   Testing happens after implementation. If bugs or issues are found late, they can be more costly and time-consuming to fix.
3. **Delayed Feedback**
   Clients or end-users don’t see a working product until the end, which can result in misaligned expectations or unmet needs.

4. **athuAssumes Stable Requirements**
   It works best when all requirements are clearly defined upfront—but that's rarely the case in real projects.
5. **Risk of Wasted Work**
   If a major issue is discovered late (like a fundamental misunderstanding of the client's needs), large chunks of work might have to be redone.
6. **Not Ideal for Complex or Long-Term Projects**
   For large projects, keeping track of everything through linear steps becomes inefficient and risky.
7. **Minimal User Involvement After Initial Stage**
   Users give input at the start, but often aren't involved again until testing—leading to disconnects between what was built and what is actually needed.
8. **Slower Time to Market**
   Because it follows a sequential process, delivery can be slower compared to iterative methods like Agile.

Advantages of the Waterfall Model:

1. **Simple and Easy to Understand**
   The linear structure makes it straightforward. Each phase has clear goals and deliverables, which is great for teams that prefer clarity and order.
2. **Well-Defined Phases**
   Since each phase is completed before the next begins, there's a strong sense of organization and milestone-based progress.
3. **Better for Fixed Requirements**
   If the project scope and requirements are very clear and unlikely to change, Waterfall can be very efficient.
4. **Strong Documentation**
   It emphasizes documentation at every stage, which makes it easier to maintain, scale, and transfer knowledge across teams or over time.
5. **Easy to Manage**
   The clear structure and sequential process make it easier for managers to track progress, assign tasks, and estimate timelines and budgets.
6. **Ideal for Smaller Projects**
   It works well for small to medium-sized projects where goals are clearly defined and unlikely to shift.
7. **Early Design Planning**
   Because design happens upfront, developers have a full understanding of the system architecture before coding begins.
8. **Testing is Comprehensive**
   Since testing is done after development, it often includes full system and integration testing, which can be more thorough.

Disadvantages of the Waterfall Model:

1. **Hard to Go Back**
   Once a phase is completed, going back to make changes is difficult, expensive, and time-consuming.
2. **Inflexible to Change**
   It assumes requirements are fixed early on, which is rarely the case in real-world projects.
3. **Late Client Involvement**
   Clients often don't see the product until the end, which increases the risk of it not meeting their needs.
4. **Late Discovery of Issues**
   Testing comes after development, so bugs or design flaws may not be caught until late in the process.
5. **No Working Software Until Late**
   You don't get a functional version of the software until near the end, making it harder to evaluate progress.
6. **Costly for Large Projects**
   Errors discovered late in big projects can lead to high costs for rework and delays.
7. **Not Suitable for Complex/Uncertain Projects**
   For dynamic or evolving projects, Waterfall struggles to adapt.
8. **Limited User Feedback**
   Lack of ongoing feedback can lead to a disconnect between what's built and what the user really wants.

V-Model

The **V-Model** is a software development methodology that extends the **Waterfall model** by emphasizing **testing at each development stage**. It's called the "V" model because the process is visualized in a **V-shape**, where each development phase on the left has a corresponding testing phase on the right.

Structure of the V-Model

Left Side: Development Phases

1. **System Design**
   → High-level architecture.

2. **Architectural Design**
   → How components interact.
3. **Module Design**
   → Design of individual units.
4. **Coding**
   → Actual implementation.

Right Side: Corresponding Testing Phases

1. **Acceptance Testing**
   ← Validates against requirements.
2. **System Testing**
   ← Verifies complete system functionality.
3. **Integration Testing**
   ← Tests interactions between modules.
4. **Unit Testing**
   ← Tests individual units/modules.

Key Characteristics

- **Each phase has a testing counterpart.**
- **Verification** = Left side (Are we building the product right?)
- **Validation** = Right side (Are we building the right product?)
- Testing is **planned early**, alongside development.

Advantages

- Simple and structured.
- Emphasizes early testing (defect prevention).
- Clear milestones and deliverables.
- Works well for well-defined, fixed-scope projects.

Disadvantages

- Rigid — not ideal for projects with changing requirements.
- Costly to adapt if changes are needed late.
- Not suitable for complex or agile development environments.

When to Use the V-Model?

- In projects where **requirements are fixed**.
- For **critical systems** (like medical, aerospace, defense).
- When a **high level of documentation and traceability** is needed.

Static Testing:

Static Testing is a type of software testing where the code, requirements, or design documents are **reviewed without executing the program**. It helps detect errors early in the development lifecycle.

Key Features

- Done **before code is run**.
- Focuses on **prevention of defects**.
- Includes **manual reviews** and **automated code analysis**.

Types of Static Testing

1. **Manual Reviews**
    - Requirement reviews
    - Design reviews
    - Code walkthroughs and inspections
2. **Automated Static Analysis**
    - Tools analyze source code for:

    - Syntax errors
    - Security issues
    - Coding standard violations

Advantages

- Finds bugs early → **low cost to fix**
- Improves **code quality**
- No need to run software
- Helps ensure **compliance with standards**

Disadvantages

- Can't catch **runtime errors**
- Requires **experienced reviewers**
- Might miss complex logic bugs

Common Tools

- **SonarQube**
- **ESLint** (for JavaScript)
- **Checkstyle** (for Java)

Dynamic Testing:

Dynamic Testing is a type of software testing that involves **executing the code** to validate its **functionality**, **performance**, and **behavior**. It helps detect **runtime errors** and ensures the software works as expected.

Key Features

- Requires the **software to run**.
- Focuses on **identifying defects** during execution.
- Checks both **functional** and **non-functional** aspects.

Types of Dynamic Testing

1. **Functional Testing**
   Verifies if the software meets the specified requirements.
   Includes:
   Unit Testing
   Integration Testing
   System Testing
   User Acceptance Testing (UAT)

2. **Non-Functional Testing**
   Evaluates performance, security, usability, etc.
   Includes:
   Performance Testing
   Security Testing
   Compatibility Testing

Advantages

- Detects **runtime and logic errors**.
- Validates the **actual behavior** of the system.
- Ensures the software is **user-ready**.

Disadvantages

- Requires a **working system** or prototype.
- Can be **time-consuming and costly**.
- Errors found late may be **more expensive to fix**.

Common Tools

- **Selenium** – UI testing
- **JUnit/TestNG** – Unit testing
- **Postman** – API testing
- **JMeter** – Performance testing

Spiral Model: An Overview

The Spiral Model was introduced by **Barry Boehm** in 1986. It's called "spiral" because the process is represented as a spiral with many loops. Each loop represents a **phase** of the software process, and each loop is split into **four key quadrants**.

Structure of the Spiral Model

Each spiral (iteration) consists of four main phases:

1. Planning (Determine Objectives)

- Identify objectives, requirements, and constraints.
- Stakeholders are consulted.
- Cost estimation and scheduling are done.

2. Risk Analysis (Identify & Resolve Risks)

- Critical phase: analyze risks and explore alternatives.
- Prototypes may be built to reduce risk.
- Technical feasibility, schedule risk, and cost risks are evaluated.

3. Engineering (Develop & Verify Product)

- Actual development and testing happen here.
- Can include designing, coding, and unit testing.
- A prototype or increment may be built, depending on the phase.

4. Evaluation (Customer Evaluation & Planning Next Phase)

- The customer evaluates the output.
- Feedback is gathered and decisions are made to move to the next spiral loop.
- Based on feedback, planning for the next phase begins

Repeating the Spiral

Each loop through the spiral adds features, refines requirements, and brings the product closer to completion. With every loop:

- The product becomes more refined.
- Risks are addressed progressively.
- Requirements become clearer through continuous user involvement.

Advantages of Spiral Model

- **Great for high-risk projects** – focuses heavily on risk assessment.
- **Early detection of issues** – thanks to repeated prototyping and evaluation.
- **Flexibility to changes** – because requirements are refined with each iteration.
- **Customer involvement** – ensures the end product matches user needs.
- **Incremental delivery** – useful features can be released sooner.

Disadvantages of Spiral Model

- **Complex and costly to manage** – not ideal for small/simple projects.
- **Requires expertise in risk management** – not all teams are equipped for this.
- **Can take a long time** – especially if too many spirals are needed.
- **No fixed end-point** – if poorly managed, the project can spiral indefinitely.

When to Use the Spiral Model

- Large, complex projects
- Projects with unclear or evolving requirements
- High-risk projects (e.g., critical systems)
- Where frequent user feedback is essential

The Role of Verification and Validation in the Software Development Life Cycle

The **Software Development Life Cycle (SDLC)** is a series of steps that guide the development of software, from initial concept through to deployment. **Verification** and **Validation** are integral at different stages of the SDLC, ensuring that the software is not only technically sound but also aligned with the business needs and user expectations.

1. **Verification in the SDLC**:

   Verification happens at various stages of the SDLC, such as during requirements gathering, design, coding, and unit testing. For example:

Requirement Phase: Verifying whether the requirements are clearly defined and whether they align with the project goals.

Design Phase: Ensuring that the software architecture and design meet the requirements.

Development Phase: Code reviews and inspections are done to ensure that the code is aligned with design specifications and doesn't introduce new errors.

2. Verification is typically done by the development team or QA engineers and is more focused on **processes** rather than the end-user's experience. Common verification activities include:

   Static Analysis: Analyzing the code or design without executing it.
   Code Reviews: Developers reviewing each other's code to catch potential issues early.
   Unit Testing: Testing individual components of the software to ensure they work as expected.individual components of the software to ensure they work as expected.

3. **Validation in the SDLC**:

   Validation typically takes place toward the end of the SDLC. It involves **dynamic testing** where the software is actively tested to ensure it meets business objectives and user needs. It validates that the right functionality is being delivered.

   - **System Testing**: After integration, the software is tested as a whole to check if all components work together.
   - **User Acceptance Testing (UAT)**: This is a key validation step where the end users or stakeholders test the system to ensure it meets their needs and expectations.
   - **Beta Testing**: The software is released to a small group of users to test it in a real-world environment before it is fully deployed.

4. The primary goal of validation is to ensure that the product provides the expected value and solves the user's problem in the intended way. Validation is often performed by the QA team, business analysts, or sometimes directly by users.

Common Challenges in Verification and Validation

1. **Challenges in Verification**:

    - **Incomplete Requirements**: If the initial requirements are vague or incomplete, it becomes difficult to verify if the software meets them.
    - **Changing Requirements**: During development, if the requirements change, it may require constant re-verification to ensure that the software is still on track.
    - **Complexity in Design**: When the software design is highly complex, verifying all aspects of it (such as edge cases) can be very time-consuming and prone to human error.

2. **Challenges in Validation**:

    - **Incomplete Test Coverage**: Sometimes, it's challenging to create exhaustive test cases that simulate real-world user interactions, leading to gaps in validation.
    - **Misaligned Expectations**: The development team may build a system that works according to technical specifications but doesn't meet the user's expectations or solve their problems as anticipated.
    - **Time Constraints**: Validation often requires thorough testing, but due to project deadlines, validation testing may be rushed or skipped, leading to undetected issues in the product.

Real-World Example: E-commerce Website

- **Verification**:
    - The **requirements document** specifies that the website must handle user registrations, display products, and process payments securely.
    - In the **design phase**, the development team creates detailed wireframes and database schemas, which are reviewed for accuracy.
    - During **coding**, the developers verify that the user registration system checks for proper email format, password strength, and duplicate user entries.
    - **Unit testing** verifies individual components, like adding products to the cart and processing payment methods.

- **Validation**:

    - Once the website is developed, the team performs **system testing** to ensure that all components work together.
    - In **user acceptance testing (UAT)**, real users try out the website to see if they can navigate it easily, find products, and check out without any issues. The goal is to validate if the website meets their needs and expectations, not just if it works according to the requirements.
    - Finally, **beta testing** might be performed, where a small group of users accesses the site in a real-world scenario to spot any further usability issues or bugs.

Both **verification** and **validation** are essential in ensuring that the software is both functional and user-friendly.

- **Verification** ensures that the software is built right, preventing early errors and reducing the cost of fixing issues later in the development process.
- **Validation** ensures that the software meets the real-world needs of users, guaranteeing its relevance, usability, and overall success in solving the problem it was created for.

Without **verification**, the product could be flawed from a technical standpoint, leading to costly post-launch bugs or failure. Without **validation**, the product might be built according to specs but fail to meet the expectations or solve the actual problems of the users, leading to poor adoption or user dissatisfaction.

Verification is the process of ensuring that the software is built correctly and according to the predefined specifications and design documents. It focuses on **conformance** to the requirements set out during the design phase, ensuring that each part of the software works as intended in isolation and in combination. This process typically takes place throughout the software development life cycle and is performed at different stages—such as during design reviews, code reviews, and unit testing.

Key Aspects of Verification:

Ensures that the product is being developed in line with the original requirements and design documents.

Focuses on the correctness of the software.

Typically involves activities such as inspections, walkthroughs, and reviews.

Performed through **static methods** (e.g., reviews, static analysis) and **dynamic methods** (e.g., unit testing).

The main question verification answers is: **"Are we building the product right?"**

Validation, on the other hand, is about ensuring the software meets the needs and expectations of the end-users and that it solves the intended problem. While verification checks if the product meets its technical specifications, validation focuses on checking if the product performs as required in a real-world environment. This is typically done through **system testing**, **acceptance testing**, and **user testing** to ensure that the software delivers value to the customer or user

Key Aspects of Validation:

Focuses on the end-user's needs and whether the product delivers the expected functionality.

Ensures the software meets the real-world requirements, as opposed to just technical specifications.

Typically involves activities such as user acceptance testing (UAT), system testing, and beta testing.

Performed through **dynamic testing methods** like functional testing, usability testing, and performance testing.

The main question validation answers is: **"Are we building the right product?"**

Conclusion

In conclusion, while **Verification** and **Validation** are related, they serve different purposes in software testing. Verification ensures that the product is being developed correctly according to technical specifications, whereas validation ensures that the product is the right one to meet the users' needs and

solve the problem at hand. Both are necessary to create high-quality, functional software that satisfies both technical requirements and end-user expectations.

Quality Assurance (QA)

Quality Assurance is a **process-focused** practice aimed at **preventing defects** in the software development lifecycle. QA ensures that the processes used to manage and create deliverables are effective and followed properly.

Key Points:

- Focuses on improving development and testing processes.
- Involves activities like process definition, audits, training, and standards enforcement.
- Works throughout the software development life cycle (SDLC).
- Examples of QA practices include peer code reviews, process checklists, and development standards.

Goal: Build a process that consistently delivers high-quality products.

Quality Control (QC)

Quality Control is a **product-focused** activity that involves **detecting and fixing defects** in the final product. It ensures the software product meets the quality standards and requirements before it is released.

Key Points:

- Involves identifying bugs and issues in the software through testing activities.
- Typically occurs after development is complete.
- Uses techniques like functional testing, regression testing, performance testing, etc.
- Relies on both manual and automated testing methods.

Goal: Ensure the product meets the specified requirements and is free of bugs.

QA vs QC – Quick Comparison

Feature	Quality Assurance (QA)	Quality Control (QC)

Focus	Process	Product
Objective	Prevent defects	Detect defects
Activity Type	Proactive	Reactive
Involves	Process audits, standards	Testing, inspections
When It Occurs	Throughout development	After development is complete
Responsible Team	QA Team	QC/Test Team

Manual testing plays a crucial role in the software development lifecycle by ensuring that the application behaves as expected under various conditions. It involves testers manually executing test cases without the help of automation tools, relying on their experience, intuition, and understanding of the system. Within manual testing, **verification** and **validation** are essential activities that contribute to overall software quality. **Verification** is the process of checking whether the software is being developed according to the specified requirements and design documentation. It answers the question, “Are we building the product right?” This can involve activities like reviewing requirement documents, design specs, code reviews, and conducting walkthroughs with the development team.

Validation, in contrast, focuses on evaluating the actual functionality of the software by running test cases to ensure it meets user expectations and real-world scenarios. It answers the question, “Are we building the right product?” During validation, testers perform different types of testing such as functional testing, usability testing, system testing, and acceptance testing to confirm the software's behavior and performance under expected and unexpected conditions.

Manual testing is especially useful in exploratory testing, usability testing, and situations where human judgment is critical. While it can be time-consuming and prone to human error compared to automated testing, it remains a valuable method for discovering issues that automation might miss. Together, verification and validation in manual testing ensure that the software is not only technically sound but also user-friendly, reliable, and ready for release.

Manual testing involves testers manually executing test cases to identify defects in software. **Verification** ensures the software is being built according to the design and requirements, answering, “Are we building the product correctly?” It includes activities like code reviews and design inspections. **Validation**, on the other hand, ensures the software meets the user’s needs and behaves as expected, answering, “Are we

building the right product?" This involves functional and usability testing to confirm the software works in real-world scenarios. Both verification and validation are essential in ensuring the software's quality and reliability.

Conclusion:

Manual testing, through the processes of verification and validation, plays a critical role in ensuring software quality. Verification ensures that the software is being developed correctly according to specifications, while validation ensures that it meets user expectations and functions as intended. Together, they help identify defects, improve the user experience, and ensure that the software is reliable, functional, and ready for deployment. Although manual testing can be time-consuming, its thoroughness and human judgment make it an indispensable part of the software development life cycle.

Unit Testing:

Unit testing is the process of testing **individual components or functions** of a program in isolation to make sure they work as expected.A **unit test** checks whether a **small piece of code (a unit)**—typically a function or method—works correctly in isolation. It helps catch bugs early and makes future code changes safer.

Box Testing:

- Whitebox Testing
- Blackbox Testing
- Greybox Testing

White-box testing (a.k.a. glass-box or clear-box):

  - You **do** know the internal structure.
  - You write tests based on the code logic, branches, loops, conditions, etc.
  - Often used in **unit testing**.

- **Black-box testing**:
    - You **don't** know the internal structure.
    - You test based on **inputs and expected outputs**.
    - More common in **integration** and **system testing**.
- **Gray-box testing**:
    - A mix — you have partial knowledge of the internals.

Integration Testing in Manual Testing

In manual testing, **integration testing** is performed by human testers to verify that different components or modules of an application interact with each other as expected. Unlike automated testing, which is typically run by tools, manual integration testing involves real-time interaction with the system and validating its behavior.Manual integration testing generally involves testing the **interaction between modules** and **external dependencies** such as databases, APIs, or third-party services.

Manual Integration Testing:

Manual **integration testing** is the process of testing the interaction between different modules or components in an application, ensuring that they work together as expected. This type of testing is done **manually**, meaning the tester does not use any automated test scripts or tools but relies on the application interface and expected outcomes.

Key Focus Areas of Manual Integration Testing:

- **Module Interaction:** Testing how different parts of the application work together.
- **Data Flow:** Ensuring that data is correctly passed between modules.
- **Third-party Integrations:** Testing interactions with external systems like APIs, payment gateways, or databases.

- **Interface Consistency:** Verifying that modules share data and functionality in a consistent manner.
- **Error Handling:** Ensuring proper error handling when one component fails to communicate with another.

Steps in Manual Integration Testing

Manual integration testing involves several key steps to ensure the system functions as expected when different components are integrated:

1. Understanding the System Architecture:

- Before performing manual integration testing, it's crucial to understand how different components interact with each other.
- Review **system designs**, **documentation**, and **flow diagrams** to understand how modules are connected and what data flows between them.

2. Prepare Test Cases:

- Develop test cases that focus on verifying the interactions between integrated modules or components. Each test case should focus on:
    - **Correct data flow** between modules
    - **Expected outcomes** for both success and failure scenarios
    - **Boundary conditions**, such as data size or limits
- **Test Case Example**:

    - **Test Case 1: User Registration + Payment Processing**
        - **Pre-condition**: User is registered in the system.
        - **Test**: Verify that the payment processing module works after the user is registered.

    - **Expected Outcome**: Payment should be successfully processed with correct transaction details.

3. Set Up the Test Environment:

- Ensure that all the integrated modules are available for testing.
- If testing involves external systems (e.g., APIs, databases), ensure they are up and running and simulate real-world conditions where possible.

4. Perform the Test:

- Execute the test cases manually, interacting with the system in the same way end users or administrators would.
- During execution, observe how the system behaves when modules are integrated. Ensure data is flowing between components correctly and that the interactions occur as expected.

5. Check for Error Handling and Logging:

- During testing, intentionally break the integration (e.g., disconnect a service or introduce incorrect data) to check how the system handles failures.
- Make sure the system responds appropriately with clear error messages and that logs are correctly generated.

6. Test for Performance and Data Integrity:

- Verify if the system behaves normally under load, and ensure that data is consistent and accurate across modules.

- Perform some **stress testing** to observe the behavior when multiple modules interact under heavy load.

7. Report Issues:

- Any inconsistencies, failures, or unexpected behaviors should be documented.
- Make sure to include details like:
    - What component or module caused the issue
    - Steps to reproduce the issue
    - Screenshots, if applicable
    - Expected vs. actual results

8. Re-test After Fixes:

- Once issues are fixed, the integration tests should be re-executed to confirm that the problems have been resolved.

- Ensure that the fixes don't introduce new issues or affect other parts of the system.

Common Scenarios in Manual Integration Testing

Here are a few examples of scenarios where manual integration testing is useful:

1. User Registration + Payment Integration:

Imagine an e-commerce website where users can register and make payments.

- **Objective**: Ensure the payment module processes successfully after a user registers.

- **Manual Test Steps**:

    - Register a new user.
    - Try to make a payment using the registered account.
    - Verify that the payment is processed and that the transaction details are recorded.

2. API Integration with a Frontend:

Consider a scenario where a web application fetches data from an API.

- **Objective**: Ensure the frontend correctly fetches data from the API and displays it.

- **Manual Test Steps**:

    - Open the web application.
    - Trigger an action that calls the API (e.g., clicking a "Load Data" button).
    - Verify that the data fetched from the API appears on the user interface correctly.

3. Database and Application Integration:

In a scenario where your application interacts with a database for CRUD operations (Create, Read, Update, Delete), integration tests can check for correct data flow.

- **Objective**: Verify that the application correctly stores, retrieves, and updates data in the database.

- **Manual Test Steps**:

    - Create a new record via the application.
    - Check the database to confirm the record is created.
    - Update the record in the application and verify the database reflects the change.

External System Integration in Manual Testing

Some integrations involve external systems, such as third-party APIs, external databases, or payment systems. Here's how you can handle such integrations in manual testing:

Mocking or Simulating External Systems:

- If the external system is unavailable or cannot be used in a test environment, you can mock or simulate the interaction. For example:

    - Mock the response of an external API by simulating what the response would be.
    - Use tools like **Postman** to send mock API requests to simulate how the system should respond.

End-to-End Testing:

- When integrating with external systems, it's also essential to conduct **end-to-end testing**, where you test the entire system as a whole to ensure that the external system works seamlessly with your application.

Manual Integration Testing Best Practices

1. **Define Clear Test Cases**: Write detailed test cases that outline each step and expected outcome for every integration scenario.

2. **Prioritize Critical Interactions**: Focus on testing the core integrations first, such as database interactions or payment gateways.

3. **Test Realistic Scenarios**: Always test scenarios that closely mirror how users or other systems will interact with your application in the real world.

4. **Check Error Handling**: Manually test how your system responds when modules fail to interact correctly (e.g., a network failure, database down, etc.).

5. **Test for Data Integrity**: Ensure that data remains consistent when passed between integrated components.

6. **Retest After Fixes**: After defects are fixed, make sure to re-test the integration to ensure it works as expected.

Summary of Manual Integration Testing:

- **Manual Integration Testing** verifies that different modules or components work together as expected in a system.

- It is done **manually** by testers, who interact with the system and validate the flow of data and behavior between components.

- It is useful for testing **real-world scenarios** where multiple systems or services interact, including **third-party integrations**.

- Manual tests should be planned in detail, focusing on the **core interactions** between components and **error handling**.

Visual Consistency

- **Expected Outcome**: The box should appear with clear boundaries, using only the colors white, black, and grey in its design.

- **Testing Steps**:

    - Verify that the colors are correctly displayed (white for the background, black for the borders or text, and grey for other elements such as shadows, gradients, or text).
    - Check that the box does not display unintended colors (like blue or red).
    - Validate that the shades of grey used are consistent across different elements.

Color Contrast

- **Expected Outcome**: The contrast between the white, black, and grey colors should adhere to accessibility standards, ensuring that content (like text or icons) is legible.

- **Testing Steps**:

    - Check if the text or icons inside the box have sufficient contrast against the background.
    - Use an accessibility tool or manual calculation to verify contrast ratios (WCAG guidelines recommend a minimum ratio of 4.5:1 for normal text and 3:1 for large text).
    - Ensure that grey text (if used) is distinguishable from the background.

Responsiveness

- **Expected Outcome**: The white, black, and grey box should adapt well to various screen sizes.
- **Testing Steps**:
    - Resize the browser window or test on multiple devices to ensure the box maintains its proportions and the colors stay consistent.
    - Check that the box does not distort when resized and that any text or elements inside it stay aligned and are still legible.

Hover and Focus States

- **Expected Outcome**: If the box is interactive (e.g., a button, input box, or clickable element), the hover and focus states should change in a way that aligns with the design guidelines.

- **Testing Steps**:

    - Hover over the box and check if it changes color (e.g., a grey or darker shade appears on hover).
    - Ensure that the color change is smooth and visually distinct.
    - Check the focus state (especially for accessibility) to ensure it's clearly indicated (e.g., a border or outline).

Text or Content Alignment

- **Expected Outcome**: Any text or content inside the box should be properly aligned, legible, and formatted according to the design specs.

- **Testing Steps**:

    - Verify that the text is aligned according to the design (centered, left-aligned, right-aligned, etc.).
    - Check that there are no overlaps or clipping of text or images inside the box.
    - Ensure that any images or icons inside the box don’t distort when resizing.

Interaction Testing (if applicable)

- **Expected Outcome**: If the box contains interactive elements (e.g., buttons, checkboxes), ensure they function as expected when clicked or interacted with.

- **Testing Steps**:

- Click on any interactive elements inside the box and check if the expected action is triggered.
- If it's a form input field, verify that the text input works correctly with the white background.
- Ensure any icons or buttons inside the box behave as intended.

Boundary and Alignment Checks

- **Expected Outcome**: The white, black, and grey box should align with other surrounding UI elements or the grid layout, maintaining consistency.

- **Testing Steps**:

  - Check that the box maintains the correct margins, padding, and alignment relative to surrounding elements.
  - Verify that the box size matches the design specs and is centered if needed.

Cross-Browser/Device Testing

- **Expected Outcome**: The box should look consistent across different browsers and devices.

- **Testing Steps**:

  - Test the white, black, and grey box in different browsers (Chrome, Firefox, Safari, Edge, etc.).

- Test the box on various devices (mobile, tablet, desktop) to ensure it appears consistently and functions properly.

Performance and Loading

- **Expected Outcome**: The colors and box design should load smoothly without performance issues.

- **Testing Steps**:

  - Ensure there are no delays or lag when rendering the box (especially if it's part of a larger design or complex UI).
  - Test in low-network or slow connection conditions to see if it still loads properly.

Edge Case Testing

- **Expected Outcome**: The box should behave predictably in edge cases.

- **Testing Steps**:

  - Test for extremely small or large text inside the box.
  - Verify behavior when the box contains no content (should the box still be visible, or should it collapse?).
  - Test the box with special characters, long strings of text, or unexpected inputs.

Example Test Cases:

Test Case 1: Box Color

- **Description**: Ensure the box is displayed in white with black borders and a grey shadow.
- **Steps**:
    1. Open the page containing the box.
    2. Inspect the colors used in the box (check background, border, shadow).
- **Expected Result**: White background, black border, grey shadow.

Test Case 2: Hover Effect

- **Description**: Check the hover effect for the box.
- **Steps**:
    1. Hover over the box.
    2. Observe the color change.
- **Expected Result**: Box turns to a darker grey on hover.

Test Case 3: Content Alignment

- **Description**: Ensure text is centered within the box.
- **Steps**:
    1. Verify that text inside the box is centered.
- **Expected Result**: Text is aligned properly within the box.

GUI Testing:

(Graphical User Interface Testing) is a type of software testing that focuses on validating the visual elements and user interactions within an application. Its primary goal is to ensure that the interface behaves as expected and meets design specifications, providing a smooth and intuitive experience for users. This involves checking everything from buttons, text fields, and menus to pop-ups, layouts, and navigation flows. GUI testing also covers aspects such as correct alignment of elements, responsiveness across different screen sizes, consistency in fonts and colors, and accurate feedback when users perform actions—like seeing a tooltip on hover or an error message when submitting an incomplete form.

The testing can be performed manually by testers simulating user actions or automated using tools like Selenium (for web), Appium (for mobile), and TestComplete (for desktop and web). For example, an automated GUI test might open a login page, enter credentials, click a login button, and verify that the user is redirected to the dashboard. GUI testing is crucial for identifying usability issues, broken interface elements, and inconsistencies across platforms or browsers. It also often includes accessibility testing, ensuring that the interface works for users with disabilities. To maintain efficiency and scalability, testers often use frameworks like the Page Object Model and integrate tests into CI/CD pipelines. GUI testing is essential not only for confirming that a system functions properly but also for ensuring that users have a seamless and visually coherent experience.

This type of testing is especially important in applications where **visual correctness**, **workflow accuracy**, and **user navigation** are central—such as e-commerce sites, banking apps, and SaaS dashboards. GUI testing is also useful for catching issues introduced by frequent UI updates, browser incompatibilities, or changes in screen resolution. For instance, a layout that looks perfect on a desktop browser might break on a mobile device if responsive design is not implemented or tested correctly.

GUI Testing can be performed at different levels. During early development, developers might do quick manual checks. As the system matures, testers automate frequent checks

using tools like **Selenium**, **Cypress**, or **Playwright** for web apps, and **Appium** for mobile apps. These tools simulate user interactions such as clicking buttons, entering data, navigating pages, or validating output. Automation not only speeds up regression testing but also helps detect visual regressions and unexpected behavior after code changes.

A challenge in GUI testing is maintaining the test scripts, especially when UI elements or layouts frequently change. To handle this, testers often adopt practices like the **Page Object Model**, where each page or component is represented in code, allowing tests to be updated more easily when the UI changes.

In summary, GUI Testing ensures that a software application's interface not only functions correctly but also delivers a reliable and user-friendly experience across various devices and conditions. It's a vital component of the software testing lifecycle, particularly for user-facing products, and contributes directly to customer satisfaction, product usability, and brand reputation.

GUI Testing is often seen as the "final line of defense" before a product reaches users, as it's the layer that users interact with directly. Even if the underlying system is perfectly functional, a poor user interface—such as confusing navigation, unreadable text, or unresponsive buttons—can make an application feel broken. This makes GUI testing critical in **customer-facing software**, where first impressions and usability determine user retention and satisfaction.

One of the key components of GUI testing is **verifying user workflows**. This involves testing multi-step processes such as logging in, searching for a product, adding it to a cart, and completing a purchase. Each step needs to function not only in isolation but also as part of the complete user journey. Inconsistent behavior across these flows—such as a button that fails only after the user adds multiple items—can often only be caught through comprehensive GUI testing.

Cross-platform testing is another essential part of GUI testing. With the proliferation of devices, screen sizes, and browsers, a UI that looks perfect in Chrome on a desktop might be unusable in Safari on an iPhone. GUI testing should therefore account for **responsive design**, **browser compatibility**, and **device adaptability** to ensure consistency and accessibility for all users. Tools like **BrowserStack** and **Sauce Labs** are often used to simulate various environments and automate tests across them.

GUI Testing is also closely tied to **non-functional testing**, such as usability and accessibility testing. For example, it includes checking whether elements are labeled correctly for screen readers, whether buttons can be operated via keyboard shortcuts, and whether there is adequate contrast for users with visual impairments. These are not strictly "functional" issues, but they are crucial for compliance with standards like WCAG and for offering an inclusive user experience.

Despite its importance, GUI testing comes with several **challenges**. The most notable is **flakiness**—tests that fail intermittently due to timing issues, dynamic content, or animation delays. To mitigate this, modern GUI automation frameworks include features like **smart waits**, **element stability detection**, and **retry mechanisms**. Additionally, **visual testing tools** like **Percy** and **Applitools** can capture screenshots and perform pixel-level comparisons to detect visual bugs that might not be caught through traditional test assertions.In enterprise environments, GUI testing is often integrated into a **CI/CD pipeline**, so every code change triggers automated UI tests. This ensures that regressions are caught early and developers receive instant feedback. To balance speed and depth, teams may run a quick set of **smoke GUI tests** on every build and schedule deeper regression tests overnight or weekly.

In conclusion, GUI Testing is not just about clicking buttons or verifying color schemes—it's about validating the complete visual and interactive experience of a software product. It bridges the gap between technical correctness and user satisfaction,

ensuring that what the user sees and interacts with aligns with expectations and functions smoothly across all usage scenarios.

Usability Testing

Usability Testing is a type of software testing that focuses on evaluating how easy, intuitive, and efficient a system or application is for end-users. It aims to ensure that the product offers a smooth, satisfying user experience (UX) and that users can perform tasks without confusion, errors, or frustration.

Purpose of Usability Testing

The goal of usability testing is to ensure that users can easily interact with the application, navigate through it intuitively, and complete tasks with minimal effort. This type of testing uncovers issues related to design, interaction, and workflow, which might not be immediately apparent through functional testing.

By performing usability testing, you can:

- Improve the overall user experience.
- Identify areas where users encounter difficulties or confusion.
- Enhance the efficiency of users in completing tasks.
- Ensure the interface is clear, intuitive, and accessible to a wide audience.

Key Elements of Usability Testing

1. **Ease of Use**
    - Can users navigate the system without confusion? Are common tasks straightforward?
2. **Efficiency**
    - How long does it take a user to perform a task? Is the workflow optimized?
3. **Learnability**

- How quickly can users learn how to use the application? Are instructions or onboarding processes clear?

4. **Satisfaction**
    - How satisfied are users with the design and overall experience? Does the system meet user expectations and needs?

Types of Usability Testing

1. **Moderated Usability Testing**
    - A facilitator guides the user through tasks and observes their actions in real time. This allows the tester to ask questions and clarify when needed.

2. **Unmoderated Usability Testing**
    - Users complete tasks independently while the test is recorded. This is typically done remotely using testing platforms like **UserTesting** or **Lookback**.

3. **Remote Usability Testing**
    - Conducted from a distance, typically over the internet, allowing for testing with users from various geographical locations.

4. **In-Person Usability Testing**

    - The facilitator and participants are in the same physical location, and the facilitator can observe participants directly.

5. **Exploratory Usability Testing**
    - Users are given a set of goals and encouraged to freely explore the application, without any specific guidance, to see how they interact with the

UI.

Usability Testing Process

1. **Define Testing Goals**
    - Identify what you want to learn from the testing. This could be specific tasks or general feedback on the user experience.

2. **Recruit Participants**
    - Select real users who match your target audience. Participants should represent various demographics or personas that will be interacting with the product.

3. **Create Tasks and Scenarios**
    - Develop tasks that users will complete, such as “Search for a product,” “Complete a purchase,” or “Sign up for an account.” Scenarios should be as close to real-world tasks as possible.

4. **Conduct the Test**
    - Observe users as they interact with the system, taking note of where they struggle, their reactions, and their comments.

5. **Analyze Data**
    - After the test, review the results to identify common issues or areas where users had difficulty. Look at completion rates, error rates, and feedback from users.

6. **Report Findings and Make Improvements**
    - Provide a detailed report with actionable recommendations. Address any usability problems, and iterate on the design.

Usability Testing Metrics

- **Task Success Rate**: Percentage of users who successfully complete a task.
- **Time on Task**: The time it takes for users to complete a specific task.
- **Error Rate**: The number of errors users make while completing tasks.
- **User Satisfaction**: Often measured using surveys or ratings (e.g., 1-10 scale, SUS—System Usability Scale).
- **System Usability Scale (SUS)**: A standardized questionnaire used to measure perceived usability.

Tools for Usability Testing

Tool	Description
UserTesting	Provides both moderated and unmoderated remote usability tests, with real-time video feedback from users.
Lookback	Records user interactions with the product while capturing their screen and voice, ideal for remote testing.

Hotjar	Provides heatmaps, session recordings, and feedback polls to understand user behavior.
Crazy Egg	Offers heatmaps and visual analytics to help identify user clicks, scrolls, and patterns on the site.
UsabilityHub	Helps collect feedback on design elements (like navigation, layout, and buttons) with short tasks.

Usability Testing Techniques

1. **Think-Aloud Protocol**
    - Participants verbalize their thought process as they navigate through the interface. This provides valuable insights into their reasoning and decision-making.

2. **A/B Testing**
    - Comparing two versions of a UI (e.g., two different button designs) to see which one users prefer or which leads to better performance.

3. **Card Sorting**
    - Users are asked to organize information or elements on the screen into categories that make sense to them. This helps improve the information architecture of the app or website.

4. **Tree Testing**
    - Used to evaluate the structure of menus or navigation. Users are asked to find specific content or features in an app's menu structure, which helps

identify problems in how the content is organized.

Benefits of Usability Testing

1. **Improved User Experience**: Direct feedback from users ensures the product meets their needs and is intuitive to use.
2. **Higher User Satisfaction**: Identifying and fixing usability issues improves the overall satisfaction of users and fosters positive reviews.
3. **Increased Conversion Rates**: A better user experience typically leads to higher conversions, whether that means more purchases, sign-ups, or other goal completions.
4. **Reduced Support Costs**: By designing a more intuitive product, users are less likely to require extensive customer support.
5. **Greater Accessibility**: Usability testing helps identify and remove barriers that might make it difficult for users with disabilities to interact with the system.

Challenges in Usability Testing

1. **Recruiting the Right Users**: Finding participants who truly represent your target audience can be challenging. Testers must be carefully selected to match real-world users.
2. **Time and Resources**: Conducting usability testing, particularly in-person or moderated sessions, can require significant time and resources to arrange.

3. **Bias in Results**: Facilitators might unintentionally influence users, and users might be hesitant to voice negative feedback, so it’s essential to maintain objectivity during the test.
4. **Interpreting Results**: Sometimes, usability issues are subtle, making them harder to pinpoint. It’s important to analyze both qualitative and quantitative data to draw meaningful conclusions.

Why Usability Testing Matters

Ultimately, **usability testing** ensures that a product is user-centric, which is critical for any software or application aimed at a wide audience. By focusing on the **user experience**, it helps you build products that not only function correctly but also **delight** users, leading to better adoption, higher retention, and more positive reviews. Whether you're developing a website, mobile app, or enterprise software, usability testing should be a fundamental part of the software development lifecycle.

Functional Testing

Functional Testing is a type of software testing that focuses on verifying that a software application behaves according to the specified requirements and performs the intended functions correctly. Unlike non-functional testing, which looks at aspects like performance, security, or usability, **functional testing** is concerned with ensuring that the system performs its core functions correctly. It validates the system’s features, behavior, and interactions from a user’s perspective.

Purpose of Functional Testing

The primary goal of **functional testing** is to ensure that each function of the software application operates in conformance with the business requirements. This type of testing checks if the application performs the required tasks, processes data as expected, and returns the correct outputs for a given input.

Functional testing is often carried out from the **user's perspective** to ensure the application behaves as expected in various scenarios. By identifying defects or bugs early, functional testing contributes to delivering a product that functions as intended and meets the end user's needs.

What Functional Testing Covers

1. **Basic Functionality**: Ensures the software does what it is supposed to do, such as performing calculations, generating reports, and handling user input.
2. **Input and Output Validation**: Verifies that the software correctly processes inputs, including validating form fields, and generates expected outputs.
3. **Business Logic**: Ensures the business rules (like calculations, logic, or workflows) are implemented correctly within the system.
4. **User Interface Behavior**: Checks that UI elements, such as buttons, forms, and links, work as expected when users interact with them.
5. **Data Integrity**: Verifies that data is correctly stored, retrieved, and processed by the system without corruption.
6. **Error Handling**: Ensures that the system handles errors correctly, such as by providing appropriate error messages when invalid input is given.

Types of Functional Testing

1. **Unit Testing**:
    - **Focus**: Validates individual components or units of code in isolation to ensure that they work as expected.
    - **Example**: Testing a function that adds two numbers to ensure the correct sum is returned.
2. **Integration Testing**:
    - **Focus**: Verifies that different components or systems work together as intended.
    - **Example**: Testing whether a login form correctly integrates with a user authentication service.

3. **System Testing**:

    - **Focus**: Tests the entire system as a whole, including all components, to ensure that it behaves as expected when all parts work together.
    - **Example**: Testing an e-commerce website to check if users can search for products, add them to the cart, and complete a purchase.

4. **Smoke Testing**:

    - **Focus**: A basic set of tests to check whether the system's core features work, often conducted after a build is deployed.
    - **Example**: Verifying that a web page loads, buttons are clickable, and forms can be submitted.

5. **Sanity Testing**:

    - **Focus**: A subset of regression testing to ensure that new features or bug fixes work without introducing new issues.

- **Example**: After fixing a bug in a payment gateway, sanity testing would verify that payments can still be processed.

6. **Regression Testing**:

   - **Focus**: Ensures that changes or additions to the code have not caused existing functionality to break.
   - **Example**: Testing that a shopping cart still works after a new product filter feature is added.

7. **User Acceptance Testing (UAT)**:

   - **Focus**: Verifies that the system meets the business requirements and that the user is satisfied with the functionality.
   - **Example**: A client testing an app to see if it fulfills their specific business needs before accepting it for release.

Functional Testing Process

1. **Requirement Analysis**:
   - Begin by reviewing the functional requirements of the system or application to understand what the software is supposed to do.

2. **Test Planning**:
   - Create a test plan that includes the scope of testing, the testing approach, the resources required, and the test cases to be executed.

3. **Test Case Design**:
    - Based on the requirements, design detailed test cases. Each test case should include the input data, expected output, and steps for executing the test.

4. **Test Execution**:
    - Execute the test cases and document the results. If the actual result matches the expected result, the test passes; otherwise, it fails.

5. **Defect Reporting**:
    - Report any defects or issues found during testing, including details on how to reproduce them, their severity, and potential impact.

6. **Retesting and Regression**:
    - After defects are fixed, retest the affected areas. Conduct regression testing to ensure that the fixes haven't introduced new issues.

Functional Testing Metrics

1. **Test Case Coverage**:
    - The percentage of functionality tested against the total functionality described in the requirements.

2. **Defect Density**:
    - The number of defects found in functional components relative to the size of the code or functionality.

3. **Test Pass Rate**:

- The ratio of test cases that pass to the total number of test cases executed.

4. **Defect Severity**:
    - The impact of defects discovered during testing, ranging from critical defects to minor issues.

Popular Tools for Functional Testing

Tool	Use Case	Notes
Selenium	Web application testing	Open-source, supports multiple browsers and languages.
QTP (UFT)	Functional testing for web, desktop, and mobile	Commercial, highly versatile with great support for scripting.
JUnit	Unit testing for Java applications	Popular for writing and running tests in Java.
TestComplete	Automated testing for desktop, web, and mobile apps	Supports record-and-playback for functional tests.
Appium	Mobile app functional testing	Supports automation of native, hybrid, and mobile web apps.
Postman	Functional API testing	Best for testing REST APIs and web services.

Example Functional Test Case

Test Case ID	**TC-001**
Title	Test Login Functionality
Description	Ensure that users can log in with valid credentials.
Preconditions	User is registered with the system and has valid login credentials.
Test Steps	1. Open the login page. 2. Enter a valid username and password. 3. Click the "Login" button.
Expected Result	User is successfully logged in and redirected to the dashboard page.
Actual Result	(To be filled after execution)
Status	Pass/Fail
Remarks	(Any notes, e.g., specific environment details, etc.)

Why Functional Testing Matters

Functional testing is crucial because it ensures that the **core functionality** of a software product works as expected. Without functional testing, users could encounter critical issues such as broken features, incorrect results, or system crashes. This can lead to negative user experiences, financial losses, and a damaged reputation. By identifying and

fixing defects early, functional testing helps ensure that the product is stable, reliable, and ready for use.

In summary, **functional testing** focuses on the essential aspects of an application's operations, ensuring that all features are working correctly and as intended by the user. It forms the backbone of quality assurance, ensuring that the application provides the correct output, handles inputs correctly, and meets the business requirements.

Non-Functional Testing

Non-Functional Testing refers to a type of software testing that focuses on the non-functional aspects of an application or system. Unlike functional testing, which validates whether the system performs its intended functions correctly, **non-functional testing** assesses how well the system performs in terms of qualities such as **performance**, **scalability**, **usability**, **reliability**, **security**, and **compatibility**. Essentially, it ensures the system meets the criteria for quality attributes that affect the user experience but aren't directly related to specific functions.

Purpose of Non-Functional Testing

The purpose of non-functional testing is to ensure that the system meets the expected standards and behaves in a way that can handle real-world usage and requirements. It checks how the system performs under various conditions, how it handles unexpected user behavior, and whether it can scale or support a large number of users, among other things. While functional testing ensures the application "does what it should," non-functional testing ensures the system "performs how it should."

Types of Non-Functional Testing

1. **Performance Testing**

    - **Objective**: Validates the responsiveness, speed, and stability of the application under various conditions.
    - **Types**:
        - **Load Testing**: Checks the system's performance under normal and peak load conditions.
        - **Stress Testing**: Tests the application's behavior when it's subjected to extreme load conditions, typically beyond its intended capacity, to check for failures.
        - **Scalability Testing**: Assesses the system's ability to scale up or down to handle increased load without sacrificing performance.
        - **Spike Testing**: Evaluates how the system handles sudden, sharp increases in load.
    - **Example**: Testing how a website handles 1000 users simultaneously accessing the homepage.

2. **Security Testing**

    - **Objective**: Ensures that the system is secure and protects against threats such as unauthorized access, data breaches, and vulnerabilities.
    - **Types**:
        - **Penetration Testing**: Simulating attacks on the system to identify security vulnerabilities.
        - **Vulnerability Scanning**: Automated tools are used to identify security vulnerabilities.
        - **Authentication Testing**: Ensuring that users can only access data or features based on their authentication and authorization levels.

        - **Data Encryption Testing**: Verifying that sensitive data is securely encrypted during storage or transmission.
    - **Example**: Verifying that a user cannot access another user's private information by bypassing login credentials.

3. **Usability Testing**

    - **Objective**: Evaluates the ease of use, user satisfaction, and overall experience of the application.
    - **Focuses on**:
        - User-friendliness
        - Navigation clarity
        - Accessibility features
    - **Example**: Ensuring that users can easily navigate an app and complete common tasks without encountering confusion.

4. **Compatibility Testing**

    - **Objective**: Ensures the system works across different environments, platforms, devices, and browsers.
    - **Types**:
        - **Browser Compatibility**: Verifying that the application works across different web browsers (Chrome, Firefox, Safari, etc.).
        - **Operating System Compatibility**: Ensuring that the application functions properly across different OS (Windows, macOS, Linux, etc.).
        - **Device Compatibility**: Testing mobile apps to ensure they work across different devices (iOS, Android) and screen sizes.

- **Example**: Verifying that a website works on both desktop and mobile browsers.

5. **Reliability Testing**
   - **Objective**: Assesses the application's ability to function correctly over time under varying conditions.
   - **Focuses on**:
     - System uptime
     - Error recovery
     - Handling of unplanned events or failures
   - **Example**: Testing how a cloud service recovers after a server crash and whether it continues operating normally.

6. **Scalability Testing**
   - **Objective**: Evaluates how well the system can handle an increasing load or growing data volume over time.
   - **Focuses on**:
     - Database performance as data grows
     - Application's ability to manage more users or transactions
   - **Example**: Testing an e-commerce platform's ability to scale during high-traffic events like Black Friday.

7. **Accessibility Testing**
   - **Objective**: Ensures the system is usable by people with various disabilities, complying with standards such as WCAG (Web Content Accessibility Guidelines).
   - **Focuses on**:
     - Screen reader compatibility
     - Keyboard navigation for users with mobility impairments

        - Text contrast for visually impaired users
    - **Example**: Ensuring that a website is navigable and usable for someone who is blind or has low vision using only a keyboard or screen reader.
8. **Maintainability Testing**
    - **Objective**: Assesses how easily the software can be maintained or updated, including bug fixes and new features.
    - **Focuses on**:
        - Code clarity and structure
        - Ease of making changes without breaking the system
    - **Example**: Testing whether adding new features to a mobile app causes any regressions in existing functionality.

9. **Localization Testing**
    - **Objective**: Ensures that the system works correctly for users in different regions, languages, and cultural contexts.
    - **Focuses on**:
        - Translation accuracy
        - Regional settings (e.g., time zones, currency, number formats)
        - Cultural relevance of content and images

    - **Example**: Ensuring a shopping app shows prices in local currency and uses the correct date format depending on the region.

Non-Functional Testing Process

1. **Requirement Analysis**

- Understanding the non-functional requirements of the application (e.g., performance, security, usability).

2. **Test Planning**
   - Creating a detailed plan that outlines the testing objectives, tools, techniques, and timelines.

3. **Test Design**
   - Developing test cases and scenarios that focus on the non-functional aspects, such as load testing or security vulnerability scanning.

4. **Test Execution**
   - Running the tests according to the plan, capturing relevant data (e.g., response times, system behavior under stress), and identifying issues.

5. **Defect Reporting**
   - Documenting any issues found, their severity, and the impact they could have on users or the system's performance.

6. **Analysis and Optimization**
   - Analyzing the results and making necessary improvements or optimizations based on findings (e.g., increasing server capacity to handle more traffic).

7. **Retesting**
   - After issues are addressed, retesting to ensure the changes have improved the system and no new issues were introduced.

Why Non-Functional Testing Matters

1. **User Satisfaction**
    - Non-functional testing ensures that the system performs well in terms of speed, reliability, security, and usability, directly impacting user satisfaction.

2. **Scalability and Growth**
    - It helps ensure that the application can scale with increasing demand, whether from more users or growing amounts of data, without degrading performance.

3. **Security and Privacy**
    - Ensures that the application is protected from potential attacks, safeguarding sensitive user data and building trust with users.

4. **Regulatory Compliance**
    - Non-functional testing, particularly **accessibility testing**, ensures that the system meets regulatory requirements, such as accessibility laws or data protection regulations.

5. **Competitive Advantage**
    - A product that performs well, is secure, and offers a smooth experience across devices, platforms, and regions can provide a competitive advantage in the marketplace.

Tools for Non-Functional Testing

Tool	ɔn-Functional Testing	Description

JMeter	Performance Testing	Open-source tool for load and stress testing.
LoadRunner	Performance Testing	A comprehensive tool for load, stress, and scalability testing.
OWASP ZAP	Security Testing	A security testing tool for finding vulnerabilities in web applications.
Appium	Compatibility Testing (Mobile)	Automates mobile applications for cross-platform testing.
Wave	Accessibility Testing	An accessibility evaluation tool for websites.
Postman	Security Testing (API)	Used for testing APIs and verifying data encryption and authentication.
BrowserStack	Compatibility Testing	Allows testing across multiple browsers and devices.
SonarQube	Maintainability Testing	Used for code quality analysis, highlighting maintainability issues.

Conclusion

Non-functional testing ensures that a software system not only works as expected in terms of functionality but also performs well, is secure, scalable, and accessible, and provides an optimal experience for users. While functional testing guarantees that the system does what it should, non-functional testing ensures it does so in a way that meets the quality standards of the industry and user expectations. Non-functional testing is crucial for

delivering software that's reliable, secure, and usable under a variety of conditions and environments.

Regression Testing

Regression testing in manual testing refers to the process of re-executing previously completed test cases to ensure that recent code changes have not adversely affected the existing functionality of the software.

Key Concepts:

1. **Purpose**:
    - To verify that new features, bug fixes, or code changes haven't introduced new defects.
    - To ensure the stability and integrity of the application.

2. **When to Perform It**:
    - After bug fixes.
    - After enhancements or new features are added.
    - During major software releases or maintenance cycles.

3. **Manual Regression Testing Process**:
    - **Select test cases** from previous test cycles that cover core functionality and areas most affected by the change.
    - **Prioritize** test cases based on criticality and impact.
    - **Execute** the selected test cases manually.
    - **Log defects** if any failures are found.
    - **Track results** and compare with previous results.

4. **Challenges**:

- Time-consuming and repetitive.
- Prone to human error.
- Difficult to maintain as the application grows.

5. **Best Practices**:
    - Maintain a **regression test suite** that's updated regularly.
    - Use **checklists or spreadsheets** to track regression coverage.
    - Focus on **critical and high-risk areas**.
    - Consider **automation** for repetitive regression tests over time.

Types of Regression Testing (in Manual Testing)

1. **Corrective Regression Testing**:
    - No changes in the existing code.
    - Re-run existing test cases to ensure stability.
2. **Progressive Regression Testing**:
    - When new test cases are added for a new feature.
    - Ensures new code does not affect existing functionality.
3. **Selective Regression Testing**:
    - Execute a **subset** of test cases (based on impact analysis).
    - Saves time compared to full regression.

4. **Complete Regression Testing**:
    - Run **all test cases** in the regression suite.
    - Usually done before major releases or after large code changes.

5. **Partial Regression Testing**:
    - Focus on **modules impacted** by the code changes.
    - More targeted than complete regression.

Advantages of Manual Regression Testing

- **No setup needed**: Ideal when automation tools are not available.
- **Human judgment**: Testers can identify UI issues, usability problems, or edge cases that automation may miss.
- **Flexibility**: Can adapt test steps easily for complex or new scenarios.

Disadvantages of Manual Regression Testing

- **Time-consuming** and labor-intensive.
- **Error-prone**, especially during repetitive test execution.
- **Not scalable** for large projects with frequent changes.
- **Higher cost** in the long term due to manual effort.

Example of Manual Regression Testing Scenario

Imagine you're testing an e-commerce website:

Initial Feature:

- User can add products to the cart.

New Change:

- A wishlist feature is added.

Manual Regression Steps:

1. Check if "Add to Cart" still works correctly.
2. Verify that the cart updates the total price correctly.
3. Ensure user login, logout, and checkout functions are unaffected.
4. Test that the UI elements remain consistent.

You'd re-run all or selected existing test cases related to those functions manually and log any issues you encounter.

Types of Regression Testing

1. **Corrective Regression Testing**
    - **Purpose**: To verify the system still works after **no code changes** have been made to the existing functionality.
    - **When to Use**: When the software environment or test data changes, but the application code remains the same.
    - **Example**: Rerunning existing test cases after a database upgrade.

2. **Progressive Regression Testing**

    - **Purpose**: To test new functionality along with existing features to ensure integration is smooth.
    - **When to Use**: When **new features** are added and test cases for them are being developed.
    - **Example**: Adding a new login method (e.g., social login) and verifying that standard login still works.

3. **Selective Regression Testing**
    - **Purpose**: To run only a **subset of test cases** that are most likely to be affected by the changes.
    - **When to Use**: When time or resources are limited, and you want to test the most relevant parts of the application.
    - **Example**: Running test cases related to payment when the checkout module is updated.

4. **Partial Regression Testing**

- **Purpose**: To test only the **modified part of the application** and its immediate dependencies.
- **When to Use**: After making minor code changes or bug fixes.
- **Example**: Fixing a calculation bug in a tax module and retesting only that part and related modules.

5. **Complete (Full) Regression Testing**
   - **Purpose**: To test the **entire application** thoroughly.
   - **When to Use**: Before major releases, after major code changes, or system migrations.
   - **Example**: After re-architecting the backend or upgrading the platform/framework.

6. **Unit Regression Testing**
   - **Purpose**: To test **individual units of code** in isolation.
   - **When to Use**: During unit testing by developers after a change to a function or method.
   - **Example**: Modifying a specific function and testing it independently using mock data.

Smoke Testing

Smoke testing in manual testing is a quick, initial check performed **without automation**, usually by QA testers, to ensure that the **core and critical functionalities** of the application work properly after a new build is released.

Smoke Testing in Manual Testing:

1. **Purpose**: To verify whether the software build is stable enough for further testing.

2. **Scope**: Only major functionalities (e.g., login, navigation, dashboard loading).
3. **Execution**: Testers manually perform a predefined set of high-level test cases.
4. **Frequency**: Done every time a new build is deployed.
5. **Outcome**: If smoke testing fails, the build is rejected and sent back to developers.

Example (Manual Smoke Testing Scenario):

Imagine you're testing a banking app:

- Can the app open without crashing?
- Can the user log in?
- Is the account balance displayed correctly?
- Can the user log out? If all these pass, the build is considered stable enough for deeper functional or regression testing.

More Characteristics of Manual Smoke Testing:

1. **Build Verification Test (BVT)**
   Smoke testing is also called a **Build Verification Test** because it ensures the build is good enough for more detailed testing.
2. **Time-Saving**
   Since only critical functionalities are checked, smoke testing is quick and saves time by catching major issues early.
3. **Performed by QA Team**
   Manual testers usually execute smoke tests before starting formal test cycles.
4. **No Detailed Test Cases Required**
   Testers often use a **high-level checklist** instead of step-by-step test cases.

Manual Smoke Testing Checklist – Example (E-commerce Website)

Functionality	Smoke Test Step	Expected Result
Homepage Load	Open the homepage	Homepage loads without errors
Login	Enter valid credentials and login	User successfully logs in
Product Search	Search for a product	Results are displayed
Add to Cart	Add an item to the cart	Item appears in the cart
Checkout Page	Navigate to checkout	Checkout page loads
Logout	Click on logout	User is logged out

Advantages of Manual Smoke Testing:

- Detects showstopper bugs early
- Saves time and effort in deeper testing
- Builds confidence that the system is testable

Limitations:

- Doesn’t catch deeper functional bugs
- Can be repetitive without automation
- Relies on tester experience and consistency

Criteria	Smoke Testing	Sanity Testing
Purpose	To check overall system stability	To verify specific functionality after changes
Level of Testing	High-level testing	Deep-level testing
Scope	Broad and shallow	Narrow and focused
Performed When	After receiving a new build	After bug fixes or minor updates
Type of Testing	Usually scripted with major test cases	Often unscripted or very focused test cases
Goal	“Is the build stable enough for further testing?”	“Does this particular feature or fix work properly?”
Test Coverage	Covers major modules or critical paths	Covers only the affected areas

Sanity Testing

Sanity testing is a type of software testing performed to verify that a specific function or bug fix works correctly **after changes have been made** to the code. It ensures that the new functionality behaves as expected **without doing a full regression test**.

Key Features of Sanity Testing:

- **Purpose**: To check the accuracy of a recent fix or feature.
- **Scope**: Very narrow—only the areas related to the change.
- **Speed**: Fast and focused; usually completed quickly.
- **Documentation**: Often informal or not documented heavily.
- **Who Performs It**: Usually done by testers or QA after a new build is received.
- **Automation**: Typically manual, but can be automated for repeatability.

Example of Sanity Testing:

Imagine a **banking app** where a bug was fixed in the **fund transfer** feature:

- You test just the **fund transfer functionality**.
- You **don't** test unrelated modules like login, balance check, or profile update.
- You verify if money transfers correctly, confirmation messages show up, and no errors occur.

When to Use Sanity Testing:

- After bug fixes

- After small updates or enhancements
- When there’s no time for full regression testing

Limitations:

- Doesn’t cover the whole application
- Might miss side effects of code changes in unrelated areas

When is Sanity Testing Performed?

- After a **bug fix** is deployed.
- After a **minor code change** or feature update.
- When **time is limited**, and full regression testing isn't feasible.
- When QA needs to **quickly validate** whether the fix works and the application is still functional.

Key Characteristics of Sanity Testing:

Feature	Description
Test Level	Usually done at the **system or integration level**
Documentation	Often **not formally documented**
Execution	Quick and **manual testing**, unless automated for repetition
Test Cases	Not detailed—focused only on changed parts

Build Decision	Helps decide whether to **accept or reject** the build for further testing

Benefits of Sanity Testing:

- **Saves time** by focusing only on relevant features.
- Quickly **verifies fixes** without exhaustive testing.
- Helps **avoid wasting effort** on unstable builds.
- Acts as a **checkpoint** before deeper testing.

Examples of Sanity Testing:

Example 1: E-commerce Website

- **Fix**: "Add to Cart" button was not working.
- **Sanity Test**: Tester checks only if "Add to Cart" now works, and the item reflects in the cart. No need to test payment or search.

Example 2: Mobile Banking App

- **Fix**: Error in the "Change Password" feature.
- **Sanity Test**: Tester logs in, goes to settings, changes the password, and confirms it updates. No need to test fund transfer or login history.

Example 3: Social Media Platform

- **Fix**: Hashtag search wasn't showing results.
- **Sanity Test**: Tester enters a few hashtags, checks if results now appear. They don't test posting, commenting, or notifications.

Exploratory Testing

Exploratory testing in manual testing is a process where testers actively explore the application without predefined test cases. They use their knowledge, experience, and intuition to test the software in a flexible and unstructured way. This is particularly useful when documentation is incomplete or when quick feedback is needed.

1. What Is Exploratory Testing in Manual Testing?

In manual exploratory testing, the tester manually navigates through the application to discover issues that might not be caught with scripted or automated tests. Unlike automated testing, which follows strict rules and sequences, exploratory testing allows the tester to learn and adapt as they go

2. Purpose of Exploratory Testing

The main purpose of exploratory testing is to:

- Find hidden bugs and edge cases.
- Understand the software's behavior.
- Test how real users might interact with the system.
- Fill gaps that structured testing may miss.

It is especially helpful in:

- Early development stages.
- Agile environments with frequent updates.
- Testing new or unfamiliar features.

3. Key Features of Exploratory Testing

Feature	Description

Unscripted	No predefined test cases; testers explore freely.
Simultaneous Thinking	Testing, learning, and planning happen together.
Manual Execution	Testers interact directly with the UI and system.
Experience-Based	Relies on tester's domain knowledge and intuition.
Time-Boxed	Often done in short, focused sessions (e.g., 60–90 minutes).

4. Tester's Role and Skills Needed

The success of exploratory testing depends heavily on the **tester's mindset and skills**:

- Analytical thinking
- Domain knowledge
- Curiosity and creativity
- Attention to detail
- Ability to observe and adapt quickly

Since testers are designing and executing tests simultaneously, they must be comfortable making decisions on the spot.

5. Real-World Example

Imagine you are testing a **food delivery app** manually:

- You log in with various types of accounts (new user, returning user, guest).

- You place an order and then cancel it halfway.
- You enter invalid promo codes or try pasting emojis in address fields.
- You rapidly switch between screens to see if the app crashes.

You're not following any written script—you're exploring how the app reacts to real-life behavior.

6. How to Document Exploratory Testing

Although exploratory testing is informal, **documentation is still important**. You can use:

- **Session Notes**: Write down what you tested, what you found, and what you plan to test next.
- **Bug Reports**: For each issue found, include clear steps to reproduce, screenshots, and severity.
- **Charters**: A short statement describing what the session will focus on, e.g., "Explore user registration with edge inputs."

Example Charter: **Goal**: Explore login functionality
Focus Areas: Valid/invalid credentials, password reset, session timeout
Time-box: 60 minutes

7. Advantages of Exploratory Testing in Manual Testing

- **Finds unexpected bugs** that scripted testing may miss.
- **Adapts quickly** to changes in the software.
- **Saves time** in fast-paced environments like Agile.
- Encourages **creative testing** that simulates real user behavior.
- Does **not require extensive documentation** up front.

8. Limitations

- May miss some test coverage if not carefully tracked.

- Depends heavily on the tester's skill and knowledge.
- Difficult to repeat exactly unless well-documented.
- Can be harder to communicate results without proper notes or tools.

9. Tools That Can Support Manual Exploratory Testing

While exploratory testing is manual, you can use tools to enhance it:

- **JIRA / Bugzilla** – for logging bugs
- **TestRail / Zephyr** – for session notes or test charters
- **Screen recorders** (e.g., Loom, OBS Studio) – to record sessions
- **Note-taking tools** (e.g., Notion, OneNote) – for quick observations

10. When to Use Exploratory Testing

Use exploratory testing when:

- You have limited time.
- You're testing a new or unfamiliar feature.
- There's incomplete or missing documentation.
- You want to explore usability and user behavior.
- You need to complement existing scripted tests.

Conclusion

Exploratory testing in manual testing is a powerful approach that leverages the tester's thinking, experience, and instinct to uncover hidden bugs and usability issues. While it lacks the structure of scripted testing, it makes up for it in flexibility, creativity, and effectiveness—especially in dynamic development environments. To be most effective, testers should document their findings, stay focused, and combine this technique with other types of testing for full coverage.

Ad hoc Testing

Ad hoc testing in manual testing is an informal and unstructured software testing technique where testers try to break the application without following any specific test cases or documentation. The goal is to find defects through random checking and creative input, often relying on the tester's intuition, experience, and understanding of the system.

Key Characteristics of Ad Hoc Testing:

- **Unstructured:** No predefined test cases or documentation.
- **Improvisational:** Testers explore the system on the fly.
- **Performed after formal testing:** Often done after scripted testing to uncover issues that structured testing might miss.
- **Requires product knowledge:** More effective when testers are familiar with the application.
- **Bug discovery focus:** Ideal for catching unexpected issues or edge cases.

Common Techniques:

- **Error Guessing:** Testers use their intuition and experience to guess the problematic areas.
- **Monkey Testing:** Inputting random data to see if the system crashes or misbehaves.

Advantages:

- Quick and cost-effective.
- Helps catch hidden or unexpected bugs.
- Encourages exploratory thinking.

Disadvantages:

- Not repeatable or documented.
- Difficult to track and manage coverage.
- Relies heavily on tester skill.

When to Use Ad Hoc Testing

1. **After formal testing** is done, to find additional defects.
2. **When there's limited time** and you can't prepare formal test cases.
3. **During exploratory phases**, like early builds or prototypes.
4. **To verify a bug fix**, especially if regression testing isn't feasible at the moment.
5. **For UI and usability feedback**, especially if you're trying to simulate real user behavior.

Types of Ad Hoc Testing

Here are a few **variations or styles** within ad hoc testing:

1. **Buddy Testing**
    - Two people work together (often a developer and a tester) to find defects.
    - Tester can guide the dev while exploring possible issues.
2. **Pair Testing**
    - Similar to buddy testing, but both are testers.
    - They share ideas and test collaboratively.
3. **Monkey Testing**
    - Random inputs and actions are given to the system to see if it breaks.
    - Example: Rapidly clicking buttons or entering gibberish in forms.

Real-World Examples

Here are some **specific examples** of ad hoc testing:

- **Login Page:**

    - Enter a 1000-character password or special characters to check how the system reacts.
    - Try logging in without an internet connection.
- **E-commerce App:**
    - Add an item to the cart, go to checkout, then remove it and see if total updates correctly.
    - Try checking out with zero quantity.
- **Banking Application:**
    - Try initiating a transfer with the same account as sender and receiver.
    - Refresh the page during a transaction.

Even though ad hoc testing is informal, you can still make it more effective:

- **Take quick notes** of what you test so bugs can be reproduced.
- Focus on **high-risk areas** or complex logic.
- Use your domain knowledge to predict failure points.
- Log bugs clearly so they're actionable.

Positive Testing in manual testing means that a human tester manually checks whether the application behaves **correctly when given valid input or performing expected user actions**.

- **Positive testing**: Focuses on testing the **normal, expected behavior** of the application using correct input data.
- **Manual testing**: The tester performs all actions **without automation tools**.

So, **positive testing in manual testing** is when a tester manually runs test cases using valid data to confirm the system works as intended.

Example Scenarios

1. Login Page

- Input:
  - Username: `user@example.com`
  - Password: `CorrectPassword123`
- Action: Click "Login"
- Expected result: User is successfully logged in and redirected to the dashboard.

2. Registration Form

- Input:
  - Name: `Alice`
  - Email: `alice@example.com`
  - Password: `SafePass123`
- Action: Submit the form
- Expected result: Account is created, and confirmation message is shown.

3. Shopping Cart

- Action: Add a valid item to the cart and proceed to checkout.
- Expected result: The correct item and price appear in the cart, and checkout proceeds without issues.

Done in Manual Testing

1. Tester reads the test case with valid data.
2. Manually navigates through the application (e.g., using a browser).
3. Enters correct input.
4. Observes and verifies that the application behaves as expected.

Negative Testing is a type of software testing where the goal is to ensure the application behaves **correctly when given invalid, unexpected, or incorrect input** — essentially, trying to **break** the system or check how it handles errors.

What Is Negative Testing?

It focuses on verifying that:

- The application **doesn't crash** with invalid data.
- **Error messages** or validations appear when needed.
- The system handles **edge cases and bad input** gracefully.

Examples of Negative Testing

1. Login Form

- Input:
  - Username: `user@example.com`
  - Password: `wrongpassword`
- Expected result: "Incorrect password" error message appears, and user is not logged in.

2. Registration Form

- Input:
  - Email: `invalid-email-format`
  - Password: `123`
- Expected result: Validation error for email format and password strength.

3. Search Field

- Input: `#$%^&*()`

- Expected result: No results found or a message like “Please enter a valid search term.”

Negative Testing Matters

- It helps identify how the system handles **invalid inputs or misuse**.
- Ensures the app doesn't behave unpredictably or crash.
- Strengthens the system’s **stability, security, and reliability**.

Negative Testing Techniques

- **Boundary Value Testing** – Use values just outside valid input ranges.
- **Invalid Data Input** – Use wrong formats or unexpected data types.
- **Empty Fields** – Submit forms without filling required inputs.
- **Injection Attacks** – Test with script or SQL-like input to ensure proper input handling.

Manual Negative Testing: In manual testing, a tester will:

1. Intentionally use invalid or incorrect data.
2. Observe the system’s behavior.
3. Check for appropriate error handling, alerts, or graceful failures.

Positive testing checks if the application works correctly with valid input, ensuring expected behavior. **Negative testing**, on the other hand, verifies that the application handles invalid or incorrect input properly, showing appropriate error messages without crashing. Both are essential to confirm system reliability and robustness.

Difference from Positive Testing:

Feature	Positive Testing	Negative Testing
Input type	Valid / expected	Invalid / unexpected
Goal	Confirm system works correctly	Ensure system handles errors properly
Outcome expected	Successful operation	Error message, graceful rejection