

# Assume you are expert in answering manual testing questions in interview and fully experienced in it then answer the question "What is Software Testing?"

Absolutely! Here's a strong, interview-ready answer to "What is Software Testing?" based on real industry experience:

---

## Answer:

Software Testing is the process of evaluating a software application to ensure that it meets the specified requirements and is free of defects. The goal is to identify any gaps, errors, or missing functionality compared to the expected behavior.

It involves executing the software under controlled conditions to verify whether it behaves as intended. Testing helps improve the quality of the product by finding and fixing bugs early in the development cycle, reducing the risk of failure in production.

There are two main categories of software testing:

- **Manual Testing** – where testers execute test cases without using automation tools.
- **Automation Testing** – where scripts or tools are used to perform tests.

Additionally, testing can be functional (validating what the system does) or non-functional (validating how the system performs under certain conditions like load, stress, security, etc.).

In real-world projects, software testing is not just about finding bugs—it's also about preventing them, ensuring customer satisfaction, and verifying that the software is reliable, secure, and performs well under expected and unexpected conditions.

---

Let me know if you'd like the same answer in a more concise or fresher style for a different kind of interview (like a startup or senior role).

# What is the difference between Verification and Validation?

Absolutely! Here's how you can confidently and clearly answer the interview question:

---

**Q: What is the difference between Verification and Validation?**

**Answer:**

Verification and Validation are two essential aspects of software testing, and while they are often used together, they serve different purposes:

---

## **Verification – Are we building the product right?**

- **Definition:** Verification is the process of checking **whether the software meets the specified requirements and design specifications**.
  - **Focus:** It focuses on the **process** used to build the product.
  - **Activities:**
    - Reviews
    - Inspections
    - Walkthroughs
    - Desk-checking
  - **Performed During:** Early phases of the Software Development Life Cycle (SDLC) – like requirement analysis, design phase.
  - **Example:** Reviewing requirement documents, checking if design documents align with the requirements.
- 

## **Validation – Are we building the right product?**

- **Definition:** Validation is the process of **evaluating the actual product** to check whether it **meets the business needs and expectations**.
  - **Focus:** It focuses on the **final product**.
  - **Activities:**
    - Functional testing
    - System testing
    - User Acceptance Testing (UAT)
  - **Performed During:** After the development is complete or during the later phases of SDLC.
  - **Example:** Testing the application to ensure it behaves as expected when used by end-users.
- 

### **Key Difference:**

- **Verification** is about **preventing defects** – it's a static process.
  - **Validation** is about **finding defects** – it's a dynamic process.
- 

Let me know if you want a shorter version or one tailored for freshers or senior roles!

## What are the different levels of testing?

Absolutely! Here's a strong and professional answer to the interview question "**What are the different levels of testing?**" from a manual testing expert's point of view:

---

### **Answer:**

In software testing, there are **four main levels of testing**, each focusing on different aspects of the software system. These are:

---

## 1. Unit Testing

- **Purpose:** To test individual components or units of code, such as functions, methods, or classes.
- **Performed by:** Typically done by **developers** during the development phase.
- **Focus:** Verifies that each small unit performs as expected.
- **Example:** Testing a function that calculates the total price after applying a discount.

---

## 2. Integration Testing

- **Purpose:** To test the **interaction between integrated units or components**.
- **Performed by:** Developers or testers.
- **Focus:** Ensures that combined components communicate and work together correctly.
- **Types:** Can be **Top-down**, **Bottom-up**, or **Big bang**.
- **Example:** Testing the interaction between login and dashboard modules.

---

## 3. System Testing

- **Purpose:** To test the **entire application** as a whole in an environment that closely mirrors production.
- **Performed by:** QA or testing team.
- **Focus:** Validates the complete and fully integrated software product against the specified requirements.
- **Example:** Verifying that the e-commerce website functions correctly from login to order placement.

---

## 4. Acceptance Testing

- **Purpose:** To validate whether the software meets the **business requirements** and is ready for release.
- **Performed by:** QA team, **end-users**, or **clients**.
- **Types:**
  - **Alpha testing:** Done internally by QA before releasing to users.
  - **Beta testing:** Done by selected real users in a production-like environment.
- **Example:** Client testing the final product before sign-off.

---

### **Final Note:**

Each level of testing builds upon the previous one, helping to catch defects early and ensure a high-quality product. As a manual tester, it's crucial to understand the objective and scope of each level to design effective test cases and strategies.

---

Let me know if you'd like this formatted for quick revision or practice flashcards!

## What are the types of testing (Manual vs Automation)?

Absolutely! Here's a strong, interview-style answer to the question “**What are the types of testing (Manual vs Automation)?**” — answered from the perspective of a confident and experienced manual tester:

---

### **Answer:**

In software testing, we primarily categorize testing into two types: **Manual Testing** and **Automation Testing**, each serving different purposes depending on the project requirements, budget, and timelines.

---

### ◆ **Manual Testing:**

Manual testing is the process of executing test cases **manually**, without using any automation tools. It's best suited for:

- **Exploratory Testing:** where the tester's intuition and experience are key.
- **Usability Testing:** to evaluate user-friendliness, UI/UX.
- **Ad-hoc Testing:** where informal and unplanned testing is done.
- **Short-term Projects:** where automation would be more time-consuming and costly.

### **Key Points:**

- Involves human observation to find unexpected behavior or user interface issues.
  - Requires more time and effort but offers better flexibility and critical thinking.
  - Used to verify complex scenarios that require judgment and human intuition.
- 

### ◆ **Automation Testing:**

Automation testing involves using **tools or scripts** to perform tests automatically. It's highly beneficial when:

- Test cases are **repetitive** and **time-consuming**.
- The project requires **regression testing** frequently.
- There's a need for **faster feedback** in CI/CD pipelines.

### **Common Tools:**

- Selenium, Cypress, Playwright, QTP/UFT, TestNG, JUnit, etc.

#### **Key Points:**

- Improves accuracy and efficiency.
- Saves time on long-term projects.
- Initial investment is high, but it reduces manual effort in the long run.

---

#### ♦ **Conclusion:**

Both Manual and Automation testing are **complementary**, not competitive. As a skilled manual tester, I understand where human judgment is necessary and also when to recommend automation to improve efficiency and coverage. The goal is to ensure **maximum quality** in the most **efficient way** possible.

---

Would you like this framed specifically for a **fresher-level**, **experienced-level**, or **QA lead-level** interview?

## What is the difference between Functional and Non-functional testing?

Absolutely! Here's how you can confidently answer the interview question "**What is the difference between Functional and Non-functional testing?**" in a clear and structured way:

---

#### **Answer:**

The key difference between **Functional** and **Non-functional testing** lies in **what they validate** in a software application.

#### **Functional Testing:**

- **Focus:** Validates **what the system does** — its **features and functionalities**.

- **Objective:** To ensure that the software behaves as expected according to the **requirements or business logic**.
  - **Examples:**
    - Login functionality
    - User registration
    - Search, filters, add to cart, etc.
  - **Techniques used:**
    - Smoke testing
    - Sanity testing
    - Regression testing
    - Integration testing
    - System testing
  - **Performed using:** Manual test cases or automation tools like Selenium.
- 

## **Non-Functional Testing:**

- **Focus:** Validates **how the system performs** — its **quality attributes**.
- **Objective:** To ensure the software is usable, reliable, secure, and performs well under various conditions.
- **Examples:**
  - Performance Testing (load, stress, scalability)
  - Security Testing
  - Usability Testing



- Compatibility Testing
  - **It does not test specific features**, but rather focuses on **performance, stability, and user experience**.
  - **Tools used:** JMeter, LoadRunner, K6, etc.
- 

### In Simple Terms:

- **Functional Testing = Are we building the right system?**
  - **Non-Functional Testing = Is the system usable and efficient under expected and unexpected conditions?**
- 

Let me know if you want this tailored for freshers, experienced testers, or for a resume too.

## What is SDLC and STLC?

Absolutely! Here's a clear and professional way to answer the interview question "**What is SDLC and STLC?**" from the perspective of an experienced manual tester:

---

### What is SDLC?

**SDLC (Software Development Life Cycle)** is a structured process followed by software development teams to design, develop, test, and deploy software efficiently and with high quality. It outlines the complete journey of a software product from **requirement gathering to maintenance**.

#### ◆ Phases of SDLC:

1. **Requirement Gathering and Analysis** – Understanding what the client needs.
2. **Design** – Creating architecture and design of the system.
3. **Implementation / Coding** – Developers write the actual code.

4. **Testing** – Testers verify that the software meets the requirements.
5. **Deployment** – The software is released to production.
6. **Maintenance** – Post-deployment support and updates.

As a manual tester, I get involved from the requirement phase (for reviewing requirements) and actively during the testing phase to ensure the product is defect-free before deployment.

---

## ✓ What is STLC?

**STLC (Software Testing Life Cycle)** is a set of steps carried out by the QA team to ensure the quality of the software. It specifically focuses on **testing activities** within the SDLC.

### ♦ Phases of STLC:

1. **Requirement Analysis** – Understand what needs to be tested.
2. **Test Planning** – Define scope, strategy, and resources.
3. **Test Case Design** – Write detailed test cases.
4. **Test Environment Setup** – Prepare hardware/software for testing.
5. **Test Execution** – Run test cases and report defects.
6. **Test Closure** – Evaluate the testing cycle, document lessons learned.

STLC is a part of SDLC, but it is purely focused on testing. It ensures that the testing is done systematically and no critical step is missed.

---

## 🔄 Key Difference Between SDLC and STLC:

Aspect	SDLC	STLC
Focus	Entire software development	Testing activities only
Involves	Developers, Testers, BA, etc.	Only QA/Testers

Output      Working Software

Quality-assured, tested  
software

---

If you'd like, I can also give you a **short and beginner-friendly version** of the answer for fresher-level interviews. Want that too?

## What is a Test Case and Test Scenario?

Absolutely! Here's a professional and interview-ready answer based on real-world manual testing experience:

---

### Q: What is a Test Case and Test Scenario?

#### Answer:

A **Test Case** is a detailed set of actions or steps that are executed to verify a specific functionality of an application. It includes specific inputs, execution conditions, and expected results. A test case ensures that a particular feature works as intended and helps identify any defects.

Typical components of a test case include:

- Test Case ID
- Test Description
- Pre-conditions
- Test Steps
- Test Data
- Expected Result
- Actual Result
- Status (Pass/Fail)

#### Example:

*Test Case to verify login with valid credentials:*

- **Precondition:** User is on the login page
  - **Steps:** Enter valid username and password → Click on login
  - **Expected Result:** User should be navigated to the homepage
- 

A **Test Scenario**, on the other hand, is a high-level description of a functionality or feature to be tested. It represents a real-world use case and helps ensure end-to-end testing. Test scenarios are more focused on *what* to test rather than *how* to test.

**Example:**

*Test Scenario for login functionality:*

- Verify login with valid credentials
  - Verify login with invalid password
  - Verify login with blank fields
  - Verify "Forgot Password" functionality
- 

**Key Difference:**

- A **Test Scenario** is a broad concept or idea; it's a "what to test".
- A **Test Case** is a specific and detailed "how to test".

In real-world projects, especially Agile environments, we often derive multiple test cases from a single test scenario to ensure comprehensive test coverage.

---

Let me know if you'd like this tailored more towards a fresher or an experienced tester's point of view!

# What is the difference between Test Plan and Test Strategy?

Absolutely! Here's a professional and interview-ready answer to the question:

---

**Question:** *What is the difference between a Test Plan and a Test Strategy?*

**Answer:**

The **Test Plan** and **Test Strategy** are both essential components of software testing documentation, but they serve different purposes and are created at different levels.

## **Test Strategy:**

- **Definition:** A high-level document that outlines the general approach and testing objectives of the organization across projects.
  - **Scope:** It is **organization-wide or project-wide** and remains static for multiple projects.
  - **Purpose:** To define the **overall testing approach**, including:
    - Testing types (functional, non-functional, regression, etc.)
    - Test levels (unit, integration, system, UAT)
    - Tools to be used
    - Risk management approach
    - Entry and exit criteria
  - **Prepared By:** Usually created by **Test Managers or Project Managers**.
  - **Static Document:** Doesn't change frequently.
- 

## **Test Plan:**

- **Definition:** A project-specific document that outlines how testing will be carried out for a particular product or release.
  - **Scope:** Focuses on a **specific project or module**.
  - **Purpose:** To provide detailed information such as:
    - Features to be tested and not tested
    - Test objectives
    - Resource planning
    - Schedule and timelines
    - Roles and responsibilities
    - Deliverables
  - **Prepared By:** Usually created by **Test Leads or QA Engineers**.
  - **Dynamic Document:** Can be updated as the project progresses.
- 

### **Key Difference Summary:**

Aspect	Test Strategy	Test Plan
Level	Organizational or project-wide	Project-specific
Purpose	Defines <b>overall approach</b>	Defines <b>how testing will be done</b>
Prepared by	Test Manager / Project Manager	Test Lead / QA Engineer
Nature	Static (rarely changes)	Dynamic (updated as needed)
Detail Level	High-level	Detailed

---

### **Example:**

If you're testing an e-commerce platform, the **Test Strategy** would mention testing tools like Selenium, coverage of functional and performance testing, and general policies. The **Test Plan**

would specify that the "Add to Cart" feature will be tested in Sprint 3 using Chrome and Firefox, assign a tester, and outline specific test cases.

---

Let me know if you'd like a sample format of a test plan or test strategy too!

## What is a Defect/Bug? How do you report it?

Absolutely! Here's a strong and professional way to answer the interview question:

---

### What is a Defect/Bug? How do you report it?

**Answer:**

A **defect** or **bug** is any variance between the **expected result** and the **actual result** in a software application. It occurs when the application does not behave as intended or does not meet the specified requirements.

For example, if the requirement states that a user should be able to log in with valid credentials, but the system displays an error instead, that's a defect.

---

### How do I report a defect?

As a manual tester, I follow a systematic defect reporting process using tools like **JIRA**, **Bugzilla**, or **TestLink**. Here's how I report it:

1. **Reproduce the Issue** – First, I make sure the defect is reproducible and not a one-time glitch.
2. **Check for Duplicates** – I verify whether the issue is already logged by someone else to avoid duplication.
3. **Log the Defect** – If it's a valid and new defect, I log it with the following details:
  - **Title/Summary** – A concise title that clearly describes the issue.
  - **Description** – Detailed steps to reproduce the defect.

- **Expected Result** – What the application should do.
  - **Actual Result** – What the application is currently doing.
  - **Severity** – The business impact (e.g., Critical, Major, Minor).
  - **Priority** – How urgently it needs to be fixed.
  - **Environment** – OS, browser, device, version, etc.
  - **Attachments** – Screenshots or video recordings for clarity.
  - **Module Name** – The affected module or feature.
4. **Assign to Developer/Lead** – Based on the workflow, I assign it to the appropriate developer or team lead for further action.
  5. **Follow-Up** – I track the status of the defect through its life cycle (New → Open → In Progress → Fixed → Retested → Closed/Reopened).

---

### **Real-World Tip:**

In my experience, writing **clear and detailed defect reports** helps developers understand the issue quickly, leading to faster resolution and fewer follow-up questions.

---

Let me know if you'd like a real-world example or a sample defect report template!

## What is the difference between Severity and Priority?

In manual testing, **Severity** and **Priority** are two important terms used to categorize and prioritize defects (bugs). They both refer to different aspects of a defect, and understanding the difference between them is crucial for effective bug management.

### 1. Severity:

- **Definition:** Severity refers to the **extent of the impact** a defect has on the functionality of the software or system. It measures how critical the defect is in terms of its effect on



the application.

- **Assessment:** It is typically assessed by the **test team** based on the technical impact of the defect.
- **Example:** A defect that causes the application to crash would generally be considered **high severity**.

### Severity Levels:

- **Critical/Blocker:** Causes system crash or loss of functionality.
- **High:** Major functionality is broken but the system can still run.
- **Medium:** Minor impact on functionality; does not halt workflow.
- **Low:** Cosmetic issues or minor inconvenience that doesn't affect functionality.

## 2. Priority:

- **Definition:** Priority refers to the **urgency** with which a defect needs to be fixed, based on the business impact or the severity of the defect in the context of the release schedule.
- **Assessment:** It is usually decided by the **project manager or business stakeholders**, considering factors like customer needs, release deadlines, and business requirements.
- **Example:** A defect in the login functionality would typically be **high priority**, regardless of its severity, because it prevents users from accessing the system.

### Priority Levels:

- **High:** The defect needs to be fixed immediately because it affects critical functionality or impacts the business.
- **Medium:** The defect should be fixed in the current release but isn't as urgent.
- **Low:** The defect can be fixed in future releases or after other high-priority tasks.

### Key Differences:

Aspect	Severity	Priority
<b>Meaning</b>	Impact of the defect on the system	Urgency to fix the defect from a business perspective
<b>Focus</b>	Technical impact	Business and user impact
<b>Decided By</b>	Testers or developers	Project managers, business stakeholders
<b>Example</b>	A crash in the application	A minor UI glitch in a rarely used feature

### Example Scenario:

- **Bug:** A typo in a non-essential page on the website.
  - **Severity:** Low (doesn't impact functionality).
  - **Priority:** Low (it's not urgent to fix and has minimal business impact).
- **Bug:** A defect causing the website to crash on a login page.
  - **Severity:** High (affects critical functionality).
  - **Priority:** High (needs immediate fixing to restore access for users).

In summary:

- **Severity** indicates the **degree of impact** of a defect on the system.
- **Priority** indicates the **urgency** to fix the defect based on business needs.

Both terms work together to help teams address defects in an effective and timely manner.

## What is a Test Plan and what does it include?

A **Test Plan** is a detailed document that outlines the strategy, objectives, resources, schedule, and scope of testing activities for a software project. It serves as a blueprint for the testing process, guiding the testing team in systematically executing tests and ensuring that the software meets the required quality standards.

**A typical Test Plan includes the following sections:**

**1. Test Plan Identifier:**

- A unique identifier for the test plan document, making it easily referable.

**2. Introduction:**

- A brief overview of the testing project, including the software to be tested, its purpose, and the scope of testing.

**3. Test Objectives:**

- Clearly defined goals for the testing phase, such as verifying functionality, ensuring performance, or checking security.

**4. Scope of Testing:**

- Specifies the areas of the software that will be tested (in-scope) and those that will not be tested (out-of-scope).

**5. Test Criteria:**

- **Entry Criteria:** Conditions that must be met before testing can begin (e.g., test environment setup, code completion).
- **Exit Criteria:** Conditions that must be satisfied to conclude testing (e.g., all critical bugs fixed, test cases executed).

**6. Test Deliverables:**

- The documents and artifacts that will be produced during the testing process, such as test cases, test scripts, defect reports, and test summary reports.

**7. Test Environment:**

- Description of the hardware, software, network configurations, and other environmental setups required for testing.

**8. Test Approach:**

- The overall strategy for testing, including the types of testing to be performed (e.g., functional, performance, security, etc.), testing levels (unit, integration, system, etc.), and the techniques or methods to be used.

**9. Test Schedule:**

- A timeline detailing the start and end dates for each testing phase, milestones, and deadlines for specific testing tasks.

**10. Resource Plan:**

- Information on the personnel involved in testing, their roles, and the skill sets required. It may also include any tools or software that will be used for automation or test management.

**11. Risk and Mitigation:**

- A list of potential risks that might affect testing (e.g., delays in development, lack of resources) and the strategies to mitigate or manage these risks.

**12. Test Case Design:**

- A description of how test cases will be designed, including the test case specification, techniques for writing test cases (e.g., boundary value analysis, equivalence partitioning), and how they will be organized.

**13. Defect Management Process:**

- The process for identifying, reporting, and managing defects, including the tools and methods for defect tracking, severity classification, and resolution process.

**14. Test Entry and Exit Criteria:**

- Specifies the conditions that must be met for testing to start (e.g., availability of test data, completion of development) and when testing will be considered complete (e.g., all high-priority defects fixed).

**15. Approval and Sign-off:**

- A section where stakeholders (such as project managers, developers, and test leads) approve the test plan, ensuring that it aligns with the project's objectives.

By having a comprehensive Test Plan, the testing process becomes more organized, transparent, and efficient, ensuring all relevant aspects of the software are tested and quality is maintained.

# How do you write a good test case?

To write a good test case, here are the key steps and best practices you should follow:

## 1. **Test Case ID:**

- Start with a unique identifier for each test case. This helps in tracking and referring to the test case in the future. Example: TC\_Login\_01, TC\_Registration\_02.

## 2. **Test Case Title:**

- Give your test case a descriptive title that clearly defines what the test case is verifying. Example: "Verify Login functionality with valid credentials."

## 3. **Test Objective:**

- Specify the purpose or the goal of the test. This provides context for what the test case is designed to accomplish. Example: "Ensure the user can log in successfully using valid username and password."

## 4. **Preconditions/Assumptions:**

- List any conditions that must be true or set up before the test case can be executed. Example: "User is already registered" or "The application is up and running."

## 5. **Test Data:**

- Provide the necessary data for the test. This could include valid/invalid credentials, input fields, expected values, etc. Example:  
Username: valid\_user  
Password: valid\_password123

## 6. **Test Steps:**

- List the steps that need to be followed to perform the test. The steps should be clear, concise, and easy to follow. Example:
  1. Open the login page.
  2. Enter the username in the username field.

3. Enter the password in the password field.

4. Click the "Login" button.

**7. Expected Result:**

- Define what you expect the outcome to be after performing the test steps. The expected result should be specific, measurable, and clear. Example: "User should be redirected to the homepage after successful login."

**8. Postconditions:**

- Specify any state that the system should be left in after the test execution. For example, "User should remain logged in" or "User should be on the homepage."

**9. Priority:**

- Define the priority of the test case based on the functionality's importance. Example: "High" for critical features and "Low" for less important features.

**10. Test Case Type:**

- Specify whether the test case is for Functional testing, Regression testing, Usability testing, etc.

**11. Status/Pass/Fail:**

- Include a column or section to track the outcome of the test case after execution: Pass, Fail, or Blocked.

**12. Additional Notes:**

- Include any special conditions, constraints, or observations relevant to the test case. This can be useful for testers to understand edge cases or any dependencies.

**Example:**

**Test Case ID**

**TC\_Login\_01**

**Test Case Title**    Verify successful login with valid credentials

**Test Objective**    Ensure the user can log in with valid credentials

**Preconditions**    User has a registered account, and the application is up and running

<b>Test Data</b>	Username: valid_user, Password: valid_password123
<b>Test Steps</b>	1. Open the login page 2. Enter the username in the username field 3. Enter the password in the password field 4. Click the "Login" button
<b>Expected Result</b>	User should be redirected to the homepage after a successful login
<b>Postconditions</b>	User is logged in and on the homepage
<b>Priority</b>	High
<b>Test Case Type</b>	Functional
<b>Status</b>	Pass

By following this structure, you'll ensure your test cases are well-defined, easy to understand, and testable, helping you achieve effective and efficient test execution.

## What is a Traceability Matrix (RTM)? Why is it used?

A **Traceability Matrix (RTM)** is a document used in software testing to ensure that all the requirements defined for a system are covered by test cases. It maps or traces the relationship between requirements and test cases, providing a way to track the progress and ensure full test coverage.

### Components of an RTM:

1. **Requirement ID:** Unique identifier for each requirement.
2. **Requirement Description:** A short description of the requirement.
3. **Test Case ID:** Unique identifier for each test case that corresponds to the requirement.
4. **Test Case Description:** Brief description of the test case.
5. **Test Status:** Indicates whether the test case has been executed and whether the result was pass or fail.
6. **Comments:** Any additional notes related to the test case or requirement.

### Types of Traceability Matrices:

1. **Forward Traceability:** Tracks test cases from requirements, ensuring each requirement is covered by a test case.
2. **Backward Traceability:** Tracks requirements from test cases to ensure that the implemented features match the initial requirements.

## Why is RTM Used?

1. **Ensures Test Coverage:** RTM guarantees that all requirements have corresponding test cases, thus ensuring full coverage.
2. **Helps in Gap Identification:** It highlights any gaps in test coverage by showing which requirements do not have test cases associated with them.
3. **Tracks Requirement Changes:** If requirements change during the development phase, the RTM helps assess which test cases are affected, ensuring testing is updated accordingly.
4. **Improves Communication:** It serves as a communication tool between the development, testing, and business teams, providing a clear map of how requirements are being tested.
5. **Facilitates Audits and Compliance:** RTM is important in regulated industries for audit purposes, as it provides a documented trail of how testing aligns with requirements.
6. **Provides Traceability for Regression Testing:** RTM is useful for regression testing by tracking which test cases need to be executed after changes are made to the system.

In summary, a Traceability Matrix is a critical document for ensuring that all requirements are covered by tests, and it helps in tracking and managing testing activities throughout the project lifecycle.

## What is Boundary Value Analysis (BVA)? Give an example.

Boundary Value Analysis (BVA) is a software testing technique used to identify errors at the boundaries rather than the center of input values. The main idea behind BVA is that errors often occur at the boundaries of input ranges rather than within the range itself. BVA is a part of black-box testing, and it focuses on testing the extreme values (boundaries) and values just inside and outside these boundaries to ensure that the system handles them correctly.



## Key points about BVA:

- It is based on testing the **boundaries** of input values.
- Boundary values are tested both **at the edges** and **just outside** the edges of input limits.
- This testing technique helps uncover issues that occur when the input values are on the boundary or are at the limit of valid input values.

## Example of Boundary Value Analysis:

Let's say there is a field in a form where users can input their age, and the valid age range is between 18 and 60 years.

The **boundary values** to test are:

- **At the lower boundary:** 18 (the minimum valid value)
- **Just below the lower boundary:** 17 (invalid)
- **Just above the lower boundary:** 19 (valid)
- **At the upper boundary:** 60 (the maximum valid value)
- **Just below the upper boundary:** 59 (valid)
- **Just above the upper boundary:** 61 (invalid)

In this case, the test cases would include the following values:

- 17 (invalid, below the boundary)
- 18 (valid, lower boundary)
- 19 (valid, above the lower boundary)
- 59 (valid, below the upper boundary)
- 60 (valid, upper boundary)
- 61 (invalid, above the boundary)

By testing these boundary values and values just outside the boundaries, you can ensure that the system properly handles edge cases where most errors are likely to occur.

## What is Equivalence Partitioning?

**Equivalence Partitioning** is a black-box testing technique used in manual testing that helps reduce the number of test cases to an optimal level while maintaining effective test coverage. It divides the input data of a program into partitions or classes, where the data in each class is expected to behave similarly. The idea is that testing one representative value from each partition is sufficient to test the entire set of data, as all values within that partition should trigger the same response in the system.

### Key Points:

- **Input Partitioning:** The input domain is divided into different groups or equivalence classes. Each class represents a set of inputs that are treated similarly by the system.
- **Valid and Invalid Partitions:** Equivalence classes can be classified as:
  - **Valid Equivalence Class:** Represents a valid input that the system should accept.
  - **Invalid Equivalence Class:** Represents an input that the system should reject (for example, out-of-range values).
- **Test Case Reduction:** Instead of testing every possible value in the input domain, you only need to test one value from each equivalence class, which reduces the number of test cases significantly.

### Example:

Let's say we are testing a function that accepts ages between 18 and 60 as valid input.

- **Valid Partition:** Age values from 18 to 60 (e.g., 20, 35, 50).
- **Invalid Partitions:**
  - Below 18 (e.g., 5, 10, 15).
  - Above 60 (e.g., 65, 70, 80).

Instead of testing every age in the valid or invalid range, you can test just one value from each partition (e.g., 20 for valid, 10 for invalid below 18, and 70 for invalid above 60). This ensures you are covering all potential scenarios with a reduced number of test cases.

### Benefits:

- **Efficiency:** Reduces the number of test cases, which helps in saving time and effort.
- **Improved Coverage:** Ensures that all possible input scenarios are considered without unnecessary repetition of test cases.

## What is Decision Table Testing?

**Decision Table Testing** is a systematic technique used in software testing that helps in evaluating complex business logic and decision-making scenarios. It is particularly useful when there are multiple conditions or combinations of inputs and you need to verify how the system behaves under each scenario.

A decision table is essentially a table that represents various combinations of inputs (conditions) and the corresponding system outputs (actions). It is a useful way to map the logic of the system based on different inputs and expected behavior.

Here's a simple breakdown of **Decision Table Testing**:

1. **Conditions:** These are the inputs or variables that can vary in the decision-making process.
2. **Actions:** These represent the outcomes or actions that are triggered based on the different combinations of conditions.
3. **Rules:** Each row in the decision table represents a different rule that maps a specific combination of conditions to an action.

### How it works:

1. **Create the Table:** Identify the different conditions and their possible outcomes. Each condition can have a binary value (true/false, yes/no, etc.).
2. **Fill in the Actions:** Based on the conditions, determine what actions or outcomes are expected.

3. **Test the Rules:** Each combination of conditions is tested, ensuring that the system behaves as expected for all possible inputs.

### Example:

Let's take an example of a login system:

- **Conditions:**
  - Is the username valid?
  - Is the password valid?
- **Actions:**
  - Login successful
  - Login failed

Rule No	Username Valid	Password Valid	Action
1	Yes	Yes	Login successful
2	Yes	No	Login failed
3	No	Yes	Login failed
4	No	No	Login failed

### Advantages of Decision Table Testing:

- Helps in testing complex logic with multiple combinations of inputs.
- Provides a clear, visual representation of all possible conditions and actions.
- Ensures that all combinations are tested, preventing any edge case from being missed.

### When to Use:

- When the application has complex business rules.

- When you need to cover multiple combinations of inputs and their corresponding outputs.

In interviews, you can highlight this structured approach as a way to ensure thorough testing of logic-based conditions, where other testing techniques might miss combinations or edge cases.

## What is State Transition Testing?

State Transition Testing is a type of software testing used to verify the behavior of a system or application based on different states it can be in and how it responds to various inputs or events. It focuses on testing the transitions between these states to ensure that the system behaves as expected under different conditions.

### Key Concepts:

1. **States:** These are specific conditions or configurations that a system can be in during its lifecycle. For example, an online banking system may have states like "Logged Out", "Logged In", "Funds Transferred", etc.
2. **Events/Inputs:** These are actions or events that trigger a change in the system's state. For example, in the case of the online banking system, an event might be "User clicks 'Login'", which causes a transition from "Logged Out" to "Logged In".
3. **State Transitions:** These represent the movement from one state to another as a result of an event. Transitions help ensure that the system responds correctly to inputs and moves between states as expected.
4. **Valid and Invalid Transitions:** Not all transitions are valid. For example, if the system is in the "Logged In" state, clicking "Login" again should not trigger a transition back to the "Logged In" state. Testing should verify that invalid transitions do not occur.

### Purpose of State Transition Testing:

- **Test for correctness:** Ensure that the system correctly handles state changes based on inputs and conditions.
- **Detect errors:** Identify issues such as incorrect state transitions, missing transitions, or invalid events causing incorrect behavior.

- **Cover all possible states and transitions:** Ensure that all possible states and transitions are tested to provide comprehensive test coverage.

## Example:

Consider a simple ATM system with the following states:

1. **Idle**
2. **Authenticated**
3. **Transaction in Progress**
4. **Transaction Completed**

The events might be:

- **Insert Card:** Moves from Idle → Authenticated.
- **Enter Pin:** Moves from Authenticated → Transaction in Progress.
- **Complete Transaction:** Moves from Transaction in Progress → Transaction Completed.
- **Timeout/Cancel:** Moves from any state back to Idle.

By creating test cases for each state and transition, the tester ensures that the system behaves correctly under different conditions.

## Types of State Transition Diagrams:

1. **Finite State Machine (FSM):** A model where the system can only be in one state at a time.
2. **Extended State Machines (ESM):** A more complex model where additional variables affect transitions.

## Example Test Scenario:

- **State:** Idle → **Event:** Insert Card → **Expected Transition:** Authenticated

- **State:** Authenticated → **Event:** Enter Pin → **Expected Transition:** Transaction in Progress
- **State:** Transaction in Progress → **Event:** Complete Transaction → **Expected Transition:** Transaction Completed

By following this approach, State Transition Testing ensures that all possible system states and the transitions between them are thoroughly tested.

## What is Exploratory Testing?

Absolutely! Here's a strong, interview-ready answer to the question "**What is Exploratory Testing?**", as if coming from an experienced manual tester:

---

**Exploratory Testing** is an informal, hands-on testing approach where test cases are not pre-defined. Instead, testers actively explore the application while testing it. The goal is to discover defects, understand system behavior, and evaluate the software from a user's perspective.

In this approach, the tester's experience, intuition, and domain knowledge play a major role. While exploring the application, testers simultaneously learn about the system, design tests, and execute them in real-time. It's particularly useful when:

- Requirements are incomplete or evolving,
- There's limited documentation,
- Or when we want to uncover unexpected behavior quickly.

Exploratory Testing is also time-boxed and can be structured using **Session-Based Test Management (SBTM)**, where each session has a mission, charters, and notes taken for later review.

This technique helps find real-world issues that scripted testing might miss, and it's especially effective in Agile environments.

---

Let me know if you want a shorter, one-line version or if you'd like to add a real-world example to impress the interviewer even more!

# Can you explain the bug life cycle?

Absolutely! Here's how you can confidently and clearly answer the question “**Can you explain the bug life cycle?**” in a **manual testing interview**, just like an experienced tester would:

---

## ✓ Bug Life Cycle (Defect Life Cycle)

The **Bug Life Cycle** refers to the **process a defect goes through during its lifetime**, from the moment it is identified until it is resolved and closed. Each bug moves through various stages, which help the team **track, manage, and resolve issues efficiently**.

---

## ↻ Stages of Bug Life Cycle:

### 1. New

- The tester identifies a new bug and logs it into the bug tracking tool (e.g., JIRA, Bugzilla).
- It is in the **New** state and awaits triage or review.

### 2. Assigned

- The **Test Lead** or **Project Manager** assigns the bug to a **developer** to fix it.

### 3. Open

- The developer starts working on the bug.
- The status is now **Open**, indicating it's being analyzed or fixed.

### 4. In Progress / Fixed

- The developer has **fixed the bug** and changes the status to **Fixed**.
- The fix is now ready to be tested again by the QA team.

### 5. Retest



- The bug is now in the **Retest** phase where testers verify if the fix works as expected.

#### 6. **Reopen (if failed)**

- If the bug still exists, the tester will **Reopen** it, and it goes back to the developer.

#### 7. **Verified / Closed (if passed)**

- If the bug no longer exists, the tester changes the status to:
  - **Verified** (optional stage depending on tool or process)
  - **Closed** – confirming that the defect is resolved successfully.

#### 8. **Deferred (Optional)**

- If the bug is valid but not fixed in the current release due to low priority or time constraints, it is marked as **Deferred**.

#### 9. **Rejected / Not a Bug / Duplicate (Optional)**

- If the developer or lead decides that:
  - It's **not an issue**, or
  - **Cannot be reproduced**, or
  - **Duplicate** of another bug  
Then it may be **Rejected** or closed with appropriate status.

---

### **Example Tools Involved:**

- **JIRA**
- **Bugzilla**
- **Redmine**
- **Mantis**

- **QC/ALM (Quality Center)**
- 

### **Summary Answer for Interview:**

*"The bug life cycle refers to the journey a defect takes from identification to closure. It starts from 'New' when a tester logs the bug, then goes to 'Assigned', 'Open', 'Fixed', 'Retest', and finally to 'Closed' if resolved. If the issue persists, it's 'Reopened'. In some cases, bugs may be 'Deferred', 'Rejected', or marked as 'Duplicate' depending on their relevance. The life cycle helps teams manage issues efficiently using tools like JIRA or Bugzilla."*

---

Would you like me to create a **flowchart or diagram** of this cycle for better understanding or for your notes/interview prep?

## How do you report a bug in Jira or Bugzilla?

Certainly! Here's how you can answer the interview question "**How do you report a bug in Jira or Bugzilla?**" in a professional and confident manner, based on real experience:

---

### **Answer:**

When reporting a bug in **Jira** or **Bugzilla**, I follow a clear and detailed process to ensure developers can reproduce and resolve the issue effectively. Here's how I handle it in both tools:

---

#### ♦ **In JIRA:**

1. **Navigate to the relevant project:**
  - I first open the Jira project associated with the application under test.
2. **Click on 'Create'** to open the bug creation form.
3. **Fill in the required fields:**

- **Issue Type:** I select **Bug**.
  - **Summary:** I write a concise title (e.g., “Login button not responding on Firefox”).
  - **Description:** I follow a standard format:
    - **Steps to Reproduce:**
      1. Launch Firefox browser.
      2. Navigate to the login page.
      3. Enter valid credentials.
      4. Click on the Login button.
    - **Expected Result:** User should be logged in.
    - **Actual Result:** Nothing happens after clicking the button.
  - **Severity and Priority:** I assign them based on the impact and urgency.
  - **Environment:** I mention the OS, browser, and version.
  - **Attachments:** I add screenshots, logs, or videos if necessary.
  - 4. **Assign or leave unassigned:**
    - Based on the team’s process, I either assign it to a dev or leave it for the triage team.
  - 5. **Click 'Create'** to submit the bug.
- 

#### ◆ **In Bugzilla:**

1. **Go to the appropriate product/project.**
2. **Click on 'File a Bug'.**
3. **Enter the following details:**

- **Component:** I choose the relevant module.
- **Severity and Priority:** Set based on the issue's criticality.
- **Summary:** A brief but informative title.
- **Description:**
  - Steps to reproduce.
  - Expected vs. actual results.
  - Additional details such as build number, browser, etc.
- **Attachments:** Screenshots, console logs, or crash reports.

#### 4. Submit the bug report.

---

#### ✅ **Best Practices I Follow:**

- I **verify the bug** before reporting, to rule out false positives.
  - I **check for duplicates** using keywords or IDs.
  - I keep the report **clear, concise, and reproducible**.
  - I **communicate with devs** through comments if more info is needed.
- 

Let me know if you'd like me to tailor this for a **fresher, experienced tester**, or based on your **project type** (web, mobile, API, etc.) – happy to help!

## What are the different statuses of a bug?

Absolutely! Here's a strong, professional-level answer you can give in a manual testing interview for the question:

---

## ✓ What are the different statuses of a bug?

In a typical bug life cycle, a bug goes through various **statuses** that indicate its current state in the testing and development process. The common **bug statuses** are:

1. **New:**  
The tester has just reported the bug. It hasn't been reviewed yet by the development team.
2. **Assigned:**  
The bug is assigned to a developer by the lead or manager for analysis and fixing.
3. **Open:**  
The developer has started working on the bug but hasn't fixed it yet.
4. **In Progress / In Development:**  
The bug is actively being fixed by the developer.
5. **Fixed / Resolved:**  
The developer has made code changes and claims the bug is resolved.
6. **Ready for Retest:**  
The fixed build is deployed, and the QA team can begin retesting the bug.
7. **Reopened:**  
The tester retested the bug and found that it's still not fixed or the issue persists partially. It is then reopened and sent back to the developer.
8. **Verified:**  
The tester has retested the bug and confirmed that the fix works correctly in the application.
9. **Closed:**  
The tester or test lead closes the bug after successful verification, indicating the bug no longer exists.
10. **Duplicate:**  
The bug is already reported and exists in the system.
11. **Rejected / Not a Bug:**  
The developer or lead decides the issue is not a defect or is based on a misunderstanding.

12. **Deferred / Postponed:**

The bug is valid but is decided to be fixed in a future release due to low priority or risk.

13. **Cannot Reproduce:**

The developer/tester is unable to replicate the issue in the given environment or with the provided steps.

---

Let me know if you'd like to practice answering it yourself or want a shorter or more fresher version for interviews.

## How do you handle conflicting bugs with developers?

Absolutely! Here's a strong and confident way to answer the interview question:

---

**Q: How do you handle conflicting bugs with developers?**

**Answer:**

Conflicting bugs between testers and developers are common in any project, and I approach them with a collaborative and evidence-based mindset.

First, I make sure the bug is well-documented with clear steps to reproduce, expected vs. actual results, relevant screenshots or logs, and the environment details. This ensures there's no ambiguity from the tester's side.

If the developer disagrees or marks it as *"Not a Bug"* or *"Working as Intended"*, I take the following steps:

1. **Clarify Requirements:** I refer to the requirement documents, user stories, or acceptance criteria. If needed, I involve the business analyst or product owner to confirm the expected behavior.
2. **Reproduce and Demonstrate:** I reproduce the issue in a screen-sharing session or provide a video recording to show the bug in action. This often helps clear up misunderstandings quickly.
3. **Collaborate, Not Confront:** I maintain a professional tone and focus on the quality of the product, not on assigning blame. I ask questions like, *"Could you help me*

*understand how this aligns with the requirement?"* This opens a healthy conversation rather than a debate.

4. **Escalate if Needed:** If the conflict isn't resolved and it impacts the user experience or functionality, I escalate it to the QA lead or product manager with all supporting evidence for a decision.

Ultimately, my goal is to ensure quality while maintaining a strong working relationship with developers. I treat conflicts as opportunities for better communication and shared understanding, rather than obstacles.

---

Would you like a shorter version too in case the interviewer wants a quick answer?

## What is Regression Testing? When is it done?

Sure! Here's an expert-level answer to the interview question:

---

### **Q: What is Regression Testing? When is it done?**

#### **Answer:**

Regression Testing is a type of software testing performed to ensure that recent changes or enhancements in the codebase have not adversely affected the existing functionality of the application. The primary goal is to detect bugs that may have been introduced inadvertently due to modifications like bug fixes, new features, or configuration changes.

It is crucial for maintaining the stability and reliability of the application over time.

#### **When is it done?**

Regression Testing is typically done in the following scenarios:

1. **After bug fixes** – To confirm that the reported issue is resolved and that the fix hasn't broken any other part of the application.
2. **After new feature implementation** – To verify that the newly added feature hasn't impacted existing functionalities.

3. **During integration of modules** – To ensure that newly integrated modules work well with existing components.
  4. **Before releases or deployments** – As a final check to ensure that the overall system is stable before it goes to production.
  5. **During continuous development** – In Agile or DevOps environments, regression testing is often automated and run frequently as part of CI/CD pipelines.
- 

Let me know if you'd like a fresher-level or more technical version too.

## What is Smoke Testing vs Sanity Testing?

Absolutely! Here's a **clear and confident** answer to the interview question on **Smoke Testing vs Sanity Testing**, as if you're an experienced manual tester:

---

**Interviewer Question: "What is the difference between Smoke Testing and Sanity Testing?"**

**Answer:**

Smoke Testing and Sanity Testing are both subsets of regression testing, but they serve different purposes and are performed at different stages of the software testing cycle.

### **Smoke Testing – "Build Verification Testing"**

- It is a **broad, shallow** testing approach.
- Performed **after a new build is deployed** to ensure the critical functionalities of the application are working.
- The goal is to **verify the stability of the build** before proceeding to more in-depth testing.
- Example: If it's a login page, we'll just check if the login button works, not the validation rules or user role access.
- **Usually automated** and executed quickly.



- If smoke test fails, the build is rejected.

"Think of it like checking if the car engine starts before a road trip."

---

## **Sanity Testing – "Narrow Regression Testing"**

- It is a **focused, deep** testing approach.
- Performed **after receiving a minor code change or bug fix**, to ensure that the specific functionality works as expected.
- The goal is to **verify the correctness of new changes** without going through the full regression suite.
- Example: If a bug is fixed in the cart module, we test that fix and its related functionality only.
- Often **done manually**, especially during tight deadlines.

"It's like checking if the specific repair in the car was successful, not testing the whole car again."

---

## **Key Differences at a Glance:**

Feature	Smoke Testing	Sanity Testing
Purpose	Check build stability	Verify bug fixes or minor changes
Test Coverage	Broad and shallow	Narrow and deep
Performed When	New build is received	After bug fixes or small changes
Automation	Often automated	Mostly manual
Build Acceptance	Rejects unstable builds early	Confirms recent changes work properly

---

So in summary:

**Smoke testing checks *whether to proceed further*. Sanity testing checks *whether the specific part is working fine*.**

---

Let me know if you'd like a one-liner version or a diagram to include in your notes!

## What is Ad hoc Testing?

Absolutely! Here's how you can confidently and professionally answer the interview question:

---

### Q: What is Ad hoc Testing?

#### Answer:

Ad hoc testing is an informal and unstructured type of software testing performed without any formal test planning or documentation. It is done with the intention of finding defects through random checking or exploring the application beyond predefined test cases.

In ad hoc testing, testers rely heavily on their intuition, experience, and understanding of the application to identify unexpected issues. This type of testing is particularly useful when there is limited time or when the tester wants to quickly validate a functionality after changes.

A common technique used during ad hoc testing is **Error Guessing**, where testers anticipate potential problem areas based on past experiences.

#### Example:

If a login page is being tested, the tester may deliberately enter special characters, empty fields, or SQL statements to see how the system reacts, even if such inputs were not part of the test cases.

Ad hoc testing is most effective when the tester has a strong understanding of the application and business logic.

---

Let me know if you want a version tailored to fresher-level or experienced-level interviews!

## What is Acceptance Testing?

Absolutely! Here's a professional and interview-ready answer to the question:

---

## Q: What is Acceptance Testing?

### Answer:

Acceptance Testing is a type of software testing performed to determine whether a system meets the business requirements and is ready for deployment. It is typically the final phase of testing and is conducted after system testing and before the product goes live.

The main goal of Acceptance Testing is to validate the end-to-end business flow and ensure that the software behaves as expected in real-world scenarios. It is usually carried out by the **QA team, product owners, or end-users**.

There are two common types of Acceptance Testing:

1. **User Acceptance Testing (UAT):** Performed by actual users or clients to verify that the system fulfills their needs and is ready for production.
2. **Business Acceptance Testing (BAT):** Focuses on validating whether the business requirements are met.

### Example:

Suppose a retail application includes a feature for users to place online orders. In UAT, a business user would test this feature by selecting products, adding them to the cart, and completing the checkout process to ensure the entire flow aligns with business expectations.

### Key Characteristics:

- Performed in a production-like environment.
- Involves real-world scenarios.
- Ensures software is fit for use from a business perspective.
- The outcome determines whether the software is accepted or rejected for release.

---

Let me know if you'd like a shorter version or one tailored for freshers vs experienced testers.

# How do you prioritize test cases for execution?

Absolutely! Here's a strong and expert-level response tailored for an interview setting:

---

**Question:** *How do you prioritize test cases for execution?*

**Answer:**

Prioritizing test cases for execution is a critical part of test planning to ensure maximum test coverage within limited time and resources. I follow a **risk-based and business-driven approach**, considering several key factors:

- 1. Business Impact:**  
Test cases that cover core business functionalities or high-priority modules (like payment, login, or data submission) are always given top priority. If these fail, it directly affects end-users.
- 2. Risk and Criticality:**  
I evaluate the potential impact and likelihood of failure. Features with high complexity, recent code changes, or past defects are tested earlier as they're more prone to issues.
- 3. Customer Requirements & SLAs:**  
Test cases related to critical customer requirements, compliance, or contractual SLAs are prioritized to meet quality expectations and avoid penalties.
- 4. Regression Scope:**  
For regression cycles, I prioritize smoke and sanity test cases first to ensure the build is stable, followed by high-risk and frequently used functionalities.
- 5. Defect History:**  
Modules or functionalities with a history of frequent or critical bugs are given preference since they are more likely to regress.
- 6. Test Dependencies:**  
I consider test dependencies and execution order. For example, if one test case sets up the environment or data for another, it is executed first.
- 7. Time & Resource Constraints:**  
In case of limited time, I apply **Pareto Principle (80/20 Rule)**—executing the top 20% of test cases that cover 80% of the risk and impact.

8. **Automation Availability:**

Automated test cases, especially for smoke or regression, are scheduled first to get quicker feedback, freeing up manual effort for exploratory or edge case testing.

9. **Feedback from Developers & Stakeholders:**

Collaboration with developers and product owners also helps in understanding hot spots in the application that might need focused testing.

---

**In summary**, I use a mix of **risk-based, business-driven, and data-driven strategies** to ensure that the most valuable and risk-prone areas are validated first, ensuring early detection of critical issues and efficient use of testing resources.

---

Would you like this answer customized for a fresher or an experienced candidate specifically?

## How do you ensure complete test coverage?

To ensure complete test coverage, a well-rounded approach is needed, combining different strategies and techniques to address all areas of the application. Here's how I would answer this question in an interview as a manual testing expert:

---

**1. Requirement Analysis:**

- Start by thoroughly understanding the business and technical requirements. This helps identify all the features and functionalities that need to be tested.
- Involve stakeholders (business analysts, developers, etc.) early to clarify any ambiguities in the requirements.

**2. Test Case Design:**

- **Test Case Mapping:** Map test cases to each requirement, ensuring each functionality is covered by at least one test case.
- **Boundary Value Analysis and Equivalence Partitioning:** Use these techniques to ensure that tests cover both valid and invalid inputs, edge cases, and boundary conditions.

- **Decision Tables and State Transition Testing:** For more complex systems, decision tables and state transition diagrams can help ensure that different input combinations and transitions are tested.

### 3. Test Coverage Techniques:

- **Functional Coverage:** Ensure that all functions, buttons, links, and interactions are tested.
- **Non-Functional Coverage:** Include performance, security, compatibility, and usability tests to address aspects like load times, security vulnerabilities, and user experience.
- **Code Coverage (for collaboration with developers):** If possible, get insight from developers on code coverage, ensuring that major code paths (including loops, branches, and exceptions) are tested.
- **Data Coverage:** Make sure to cover various data combinations, including boundary conditions, null values, and incorrect formats.

### 4. Test Types and Levels:

- Ensure that all levels of testing are covered, including unit, integration, system, and acceptance testing.
- **Positive and Negative Testing:** Test both expected use cases (positive) and error conditions (negative) to confirm the system handles all scenarios.

### 5. Traceability Matrix:

- Use a traceability matrix to map test cases to requirements, ensuring that every requirement has associated test cases, and all scenarios are tested.

### 6. Regression Testing:

- After each build or release, perform regression testing to verify that new changes don't break existing functionality.

### 7. Exploratory Testing:

- Conduct exploratory testing to identify unplanned defects, particularly focusing on user experience, workflows, and areas that may not have been explicitly covered by test

cases.

#### 8. Test Automation (for collaboration):

- Although focused on manual testing, collaborating with automation testers can help ensure that repetitive or critical tests are automated, providing a broader test coverage in the long run.

#### 9. Peer Reviews:

- Regularly review test cases with peers to ensure that all functionalities are covered and that no critical test cases are missed.

#### 10. Test Reporting and Defect Tracking:

- Continuously track defects, analyzing areas where defects are concentrated. This can help identify parts of the system that need more thorough testing.

---

By following these practices, I ensure that all aspects of the application, from functionality to performance, are thoroughly tested and that we achieve complete test coverage.

## What is Peer Review? What is Walkthrough?

Sure! Here's an expert-level response to the interview question:

---

**Peer Review:** Peer review is a process where colleagues or team members review each other's work to identify defects, errors, or areas for improvement. In manual testing, peer reviews are commonly used to check test cases, test scripts, or test plans before execution to ensure that all aspects of the testing process are thorough and well-designed. It's an informal, collaborative activity and doesn't require a formal checklist. The goal is to catch issues early, ensure the accuracy of test cases, and improve the overall quality of the testing process. Peer reviews foster team collaboration and knowledge sharing while also ensuring that testing deliverables meet the expected standards.

**Walkthrough:** A walkthrough is a formal or informal meeting where a team member presents a work product (like test cases, test plans, or code) to a group of peers for feedback. The objective is to understand the design or implementation, identify potential defects, and suggest

improvements. Walkthroughs are typically conducted in the early stages of the testing cycle, and they involve a detailed step-by-step explanation of the work. The presenter walks the group through the logic or steps behind the work, and everyone discusses it. This technique is helpful for gathering feedback, ensuring alignment, and sharing knowledge among team members. Walkthroughs are also useful in identifying areas that may need more focus or refinement before moving to the next phase.

---

Let me know if you'd like to dive deeper into any of the concepts!

## What is Agile Testing? What is your role in Agile?

Sure! Here's an expert-level response to the interview question:

---

**Peer Review:** Peer review is a process where colleagues or team members review each other's work to identify defects, errors, or areas for improvement. In manual testing, peer reviews are commonly used to check test cases, test scripts, or test plans before execution to ensure that all aspects of the testing process are thorough and well-designed. It's an informal, collaborative activity and doesn't require a formal checklist. The goal is to catch issues early, ensure the accuracy of test cases, and improve the overall quality of the testing process. Peer reviews foster team collaboration and knowledge sharing while also ensuring that testing deliverables meet the expected standards.

**Walkthrough:** A walkthrough is a formal or informal meeting where a team member presents a work product (like test cases, test plans, or code) to a group of peers for feedback. The objective is to understand the design or implementation, identify potential defects, and suggest improvements. Walkthroughs are typically conducted in the early stages of the testing cycle, and they involve a detailed step-by-step explanation of the work. The presenter walks the group through the logic or steps behind the work, and everyone discusses it. This technique is helpful for gathering feedback, ensuring alignment, and sharing knowledge among team members. Walkthroughs are also useful in identifying areas that may need more focus or refinement before moving to the next phase.

---

Let me know if you'd like to dive deeper into any of the concepts!



# What is a Sprint and what happens in a Sprint Review/Retrospective?

Sure! Here's how I would answer this interview question on **Sprint** and **Sprint Review/Retrospective**:

---

## What is a Sprint?

A **Sprint** is a time-boxed period, typically lasting between 1 to 4 weeks, during which a development team works on a set of predefined tasks or user stories. It is a fundamental concept in the **Scrum framework**, which is an Agile methodology used to manage software development. The goal of a Sprint is to create a potentially shippable product increment. At the end of each Sprint, the team should have a fully functional and tested feature or product increment ready for review.

During a Sprint, the team focuses on delivering the highest priority work from the product backlog, which is a list of features, bugs, and other deliverables needed for the product.

## What happens in a Sprint Review/Retrospective?

### 1. Sprint Review:

- The Sprint Review happens at the end of the Sprint and is an opportunity for the Scrum team (product owner, developers, and Scrum master) to demonstrate the work completed during the Sprint.
- The team presents the **product increment** (the working functionality developed during the Sprint) to stakeholders, including the product owner, business representatives, and sometimes customers.
- The primary goal is to get **feedback** from stakeholders about the completed work and to discuss the progress toward the overall product goals.
- During the Sprint Review, the **product backlog** is also discussed, and the next steps for the product are considered.

### 2. Sprint Retrospective:

- The Sprint Retrospective takes place after the Sprint Review and before the next Sprint Planning session.

- It is a meeting where the Scrum team reflects on the Sprint to identify what went well, what didn't go as planned, and what could be improved for the next Sprint.
  - The team discusses process improvements, team collaboration, tools, and any issues faced during the Sprint.
  - The goal is to **continuously improve** the way the team works by addressing challenges and incorporating lessons learned.
  - Action items or improvements are often identified and committed to be addressed in the next Sprint.
- 

Would you like further elaboration on any of these points, such as examples or deeper insights into Sprint activities?

## What is the role of a QA in Scrum?

In a Scrum framework, the role of a QA (Quality Assurance) is crucial to ensure that the product meets the required quality standards. While Scrum does not have a specific QA role defined, the QA's responsibilities are integrated into the Scrum process. Here's a detailed answer for an interview:

1. **Collaboration with the Scrum Team:** In Scrum, the QA works closely with the development team, Product Owner, and Scrum Master. They are part of the Scrum team and collaborate on defining acceptance criteria, reviewing user stories, and ensuring that the product's quality aligns with the business requirements.
2. **Test Planning and Strategy:** QA is involved in defining the testing strategy for the product. This includes deciding on the types of testing required (e.g., functional, regression, performance, security) and planning how to carry them out within each sprint.
3. **Test Case Creation:** The QA is responsible for designing and maintaining test cases based on the acceptance criteria of the user stories. These test cases ensure that the features developed meet the requirements and are working as expected.
4. **Testing During the Sprint:** QA is actively involved in testing during the sprint. They run tests, report defects, and collaborate with the developers to resolve issues. They also perform continuous testing to ensure the application remains stable as new features are added.

5. **Automation:** Depending on the project needs, the QA may work on automating repetitive tests to speed up the regression process. Automation is particularly important for regression and performance testing in iterative Scrum cycles.
6. **Participation in Scrum Ceremonies:** QA participates in key Scrum ceremonies like Sprint Planning, Daily Standups, Sprint Reviews, and Sprint Retrospectives:
  - **Sprint Planning:** They help define the "Definition of Done" (DoD) and clarify testing requirements.
  - **Daily Standups:** QA updates the team on testing progress, any blockers, and issues they are facing.
  - **Sprint Review:** They help in verifying that the deliverables meet the acceptance criteria and demonstrate the tested functionality.
  - **Sprint Retrospective:** QA provides feedback on the testing process and suggests improvements for future sprints.
7. **Defect Reporting and Tracking:** QA tracks and reports defects during the sprint. They work closely with the developers to prioritize bug fixes and ensure that issues are resolved before the product is released.
8. **Continuous Improvement:** QA focuses on improving the testing process continuously. This includes refining test cases, improving test automation coverage, and exploring new testing tools and techniques to ensure the product is of high quality.

In summary, the QA role in Scrum is to ensure that testing is an integral part of the development process, working closely with the team to deliver high-quality software in every sprint. They are advocates for quality at every stage, from planning to execution, and help the team achieve the "Definition of Done" for each user story.

## What is the difference between TDD and BDD?

**TDD (Test-Driven Development)** and **BDD (Behavior-Driven Development)** are both methodologies aimed at improving the quality of software development, but they differ in their focus, process, and approach to testing.

### 1. Focus:

- **TDD** focuses primarily on **unit testing**. It is centered around writing tests first for individual components or functions before writing the actual code to implement those

components.

- **BDD** focuses on the **behavior of the application** from the end-user's perspective. It aims to bridge the communication gap between developers, testers, and non-technical stakeholders by using plain language to describe how the system should behave.

## 2. Approach:

- **TDD**: Developers write **tests first**, which specify the behavior of a function or component. Afterward, they write the minimum code needed to pass the test and refactor if necessary. The cycle repeats.
  - *Example in TDD*: Write a test that checks if a function correctly adds two numbers. Then, write the code to make it pass.
- **BDD**: Developers write **user stories** or **scenarios** using natural language in a format like "Given, When, Then" to describe expected behavior. The tests are then automatically generated from these scenarios.
  - *Example in BDD*: "Given the user is on the login page, when they enter valid credentials and click 'Login,' then they should be redirected to the dashboard."

## 3. Test Writing Style:

- **TDD**: Tests are written in **technical terms**. These tests are usually written in a programming language and are often difficult for non-developers to understand.
  - *Example in TDD*: `assertEquals(add(2, 3), 5)`
- **BDD**: Tests are written in **natural language** and are more understandable by both technical and non-technical stakeholders (e.g., business analysts or product owners).
  - *Example in BDD*: "Given I am on the login page, When I submit correct credentials, Then I should see my dashboard."

## 4. Collaboration:

- **TDD** is mainly focused on developers. It doesn't specifically involve testers or non-technical stakeholders.

- **BDD** encourages collaboration among **developers, testers, and business stakeholders**. The user stories/scenarios are written in a language that all team members can understand, ensuring everyone is aligned.

## 5. Tools:

- **TDD**: Common tools include JUnit, NUnit, and TestNG.
- **BDD**: Common tools include Cucumber, SpecFlow, and Behave.

## 6. Goal:

- **TDD**: Ensures the code works as expected and meets the requirements by catching bugs early. It is more focused on ensuring the correctness of individual pieces of code.
- **BDD**: Ensures the software behaves as the stakeholders expect, and helps to clarify requirements. It is focused on delivering business value and ensuring that the product meets the user's needs.

## Summary:

- **TDD** is developer-centric and focuses on unit tests and technical correctness.
- **BDD** is collaborative, focuses on user behavior, and encourages writing tests in a language that is accessible to everyone involved.

In essence, TDD is about **"how"** you test, while BDD is about **"what"** you test and the behavior of the system from a user perspective.

# What is Shift Left Testing?

**Shift Left Testing** refers to the practice of performing testing activities earlier in the software development lifecycle (SDLC), rather than waiting until later stages such as the end of development or during the QA phase. The "shift left" concept originates from the traditional SDLC model, where testing activities typically occur towards the right side of the timeline. By shifting testing "left," it means starting tests as early as possible.

## Key Aspects of Shift Left Testing:

1. **Early Involvement in the SDLC:** Testers start their involvement from the requirements and design phase, rather than just starting testing after the development is completed. This includes activities like reviewing requirements, providing feedback, and collaborating with developers to identify potential risks.
2. **Continuous Testing:** Testing is integrated into the development process through continuous integration and continuous testing practices. This ensures that every code change is tested as soon as it is made, providing faster feedback to developers.
3. **Automated Testing:** The emphasis on automation is crucial for Shift Left Testing. Automated tests are created early, ensuring that they can be run frequently throughout the development lifecycle. This leads to faster detection of defects, reducing the cost and time spent on bug fixing later.
4. **Collaboration:** Shift Left promotes a collaborative approach between developers, testers, and other stakeholders. Developers and testers work together to ensure that the product is built with quality in mind from the start.

## Benefits:

- **Early Detection of Defects:** Testing early helps identify and fix defects at the initial stages, which is cheaper and faster compared to identifying them later in the development or post-production.
- **Improved Quality:** Since testing is conducted early and continuously, it leads to higher-quality software and fewer defects in the final product.
- **Faster Time-to-Market:** By finding and fixing bugs early, the time required for rework is reduced, leading to quicker releases.
- **Cost Efficiency:** Catching defects early prevents the costs associated with late-stage testing, debugging, or post-release fixes.

In summary, **Shift Left Testing** is a proactive approach to testing where testing activities are moved earlier in the SDLC to improve software quality, reduce costs, and speed up the delivery process.

# How do you perform Risk-Based Testing?

Risk-Based Testing (RBT) is an approach where testing efforts are focused on areas of the application that are at the highest risk of failure. The goal is to prioritize testing based on the likelihood of a defect occurring and the potential impact it would have if it did.

Here's how I would perform Risk-Based Testing:

## 1. Identify Risks:

- **Business Risks:** Understand the business functionality of the application and identify areas that are critical to the success of the application. For example, a payment system would be a high-risk area because a failure there could lead to significant financial losses.
- **Technical Risks:** Identify areas that might be prone to defects due to their complexity, unfamiliar technology, or previous issues. This could include legacy systems, third-party integrations, or complex algorithms.
- **User Risks:** Consider how end users will interact with the application. High-risk features are those that users will interact with frequently or that have a high chance of misuse.

## 2. Assess the Impact:

- Evaluate the potential **impact** of a defect in each area. Some areas of the application, if they fail, could cause catastrophic consequences (e.g., loss of data, system crashes), while others might be less critical (e.g., UI cosmetic issues).
- Impact can be categorized as high, medium, or low, depending on how it affects the overall application or business goals.

## 3. Evaluate Likelihood:

- Assess the **likelihood** of a defect occurring in each area. This could be based on past data, complexity of the feature, or historical trends of similar defects in the application.
- Likelihood can also be categorized as high, medium, or low.

## 4. Risk Prioritization:

- Once the risks have been identified, assess the **risk exposure** by combining the likelihood and impact.
  - High Likelihood & High Impact = Critical (Needs immediate and thorough testing)
  - Low Likelihood & High Impact = Important (Tested, but may require fewer test cases)
  - High Likelihood & Low Impact = High Priority (Focus more effort)
  - Low Likelihood & Low Impact = Low Priority (Minimal testing)

## 5. Create a Risk-Based Test Plan:

- Based on the prioritization, I would develop a **test strategy** that focuses on high-risk areas first. I'd decide on the number of test cases, the depth of testing (e.g., exploratory or scripted), and allocate resources accordingly.
- For high-risk areas, I would aim for a comprehensive test suite with multiple test scenarios, while low-risk areas could be covered with basic tests or checks.

## 6. Test Execution:

- During test execution, focus on the high-risk areas first. Ensure that the critical functionalities are tested thoroughly and any potential vulnerabilities are covered.
- After testing the high-risk areas, move on to the lower-risk areas if time and resources permit.

## 7. Monitor and Adjust:

- Continuously monitor the results of the testing process and adjust the testing priorities if needed. If new risks emerge or previously identified risks become more critical, the test plan should be updated accordingly.
- Regularly review risk assessments with stakeholders to ensure alignment with business priorities.

## 8. Post-Testing Review:



- After testing, I would perform a **post-test analysis** to evaluate whether the testing efforts were aligned with the most critical areas of risk.
- Document the risks, issues encountered, and the impact on the project. This documentation will help in future risk assessments and improve the risk-based testing process for subsequent projects.

By focusing on high-risk areas, Risk-Based Testing helps to ensure that testing is both effective and efficient, reducing the chance of high-impact issues slipping through undetected while managing resources more effectively.

## How do you handle testing in tight deadlines?

When handling testing in tight deadlines, it's essential to focus on efficiency and prioritization. Here's how I would approach it:

### 1. **Prioritize Testing Based on Risk and Impact:**

- First, I would collaborate with the development and product teams to understand the most critical areas of the application. I would focus on testing the most important features that have the highest impact on the users and business.
- Risk-based testing helps me identify the core functionalities that need to be tested thoroughly, and I can test other areas more lightly.

### 2. **Use Test Case Optimization:**

- I would analyze the test cases to ensure they are efficient and cover maximum functionality with minimum repetition. This could include reusing test cases from previous cycles if there haven't been significant changes.
- For repetitive tasks, I might consider creating a set of automated tests that can be run quickly to save time.

### 3. **Perform Smoke Testing and Regression Testing Strategically:**

- I would perform a *smoke test* (sanity check) to quickly verify that the build is stable enough for deeper testing.
- Instead of running a full regression suite, I would prioritize high-risk areas and frequently used functionalities.

#### 4. **Test in Parallel (if possible):**

- If the team is large enough, I would divide the testing workload into smaller portions and distribute them effectively among team members. This allows for parallel testing and faster execution.

#### 5. **Effective Communication:**

- Throughout the process, I would communicate regularly with the developers and product team to report any blockers, bugs, or issues that need attention, ensuring that fixes are provided promptly.

#### 6. **Timeboxing:**

- Given the deadline constraints, I would timebox testing efforts to ensure we can deliver the necessary results within the given time. For example, if a testing cycle is limited to 4 hours, I would focus only on executing critical tests.

#### 7. **Exploratory Testing:**

- While testing in parallel, I would use exploratory testing to uncover issues that might not have been anticipated in the test cases. This helps identify defects that might have been missed in scripted testing.

#### 8. **Post-Release Testing:**

- In case of an extremely tight deadline, I would suggest performing critical testing initially and then following up with post-release testing once the product is deployed. This can be done as hotfixes or patches if necessary.

This approach ensures that testing is thorough, focused, and efficient, even under tight deadlines.

## How do you estimate effort for testing?

To estimate effort for testing effectively, I consider several key factors that impact the testing process. Here's how I would answer this question in an interview:

---

**Effort estimation for testing** involves assessing the amount of time, resources, and effort required to complete the testing process for a given project. The estimation depends on various

factors, including the project scope, complexity, and available resources. Here's how I typically approach it:

## 1. Understanding the Scope of Testing

- **Requirements Analysis:** I start by reviewing the project requirements and understanding the scope of testing. This includes understanding what features need to be tested, their complexity, and the level of testing required (e.g., functional, regression, performance, security).
- **Test Plan Development:** Based on the scope, I prepare a detailed test plan, which includes identifying test scenarios, test cases, and resources required. This helps in laying out a clear roadmap for the testing process.

## 2. Test Case Design and Review

- **Test Case Design:** The number and complexity of test cases directly influence the effort estimation. I break down the requirements into individual test cases, considering different testing levels like unit, integration, system, and acceptance testing.
- **Test Case Review:** Reviewing the test cases with the team or stakeholders helps identify gaps or ambiguities that may require rework, affecting the estimation.

## 3. Complexity of the Application

- **Functional Complexity:** Applications with complex logic or features require more detailed test scenarios and more time for execution.
- **Technology Stack:** The technology stack (e.g., web, mobile, API, or desktop applications) influences the effort. For instance, testing a mobile application may require additional effort due to device fragmentation and compatibility testing.
- **Integration Points:** If the system interacts with external services or third-party applications, the complexity increases, and additional time must be factored for integration testing.

## 4. Test Environment and Tools

- **Test Environment Setup:** Setting up a test environment (e.g., configuring hardware, software, network configurations) and preparing test data may require significant time, especially if the environment is complex or needs to be replicated in multiple

configurations.

- **Test Tools:** If automation or performance testing tools are used, time should be included for scripting, tool setup, and maintenance. Additionally, tools like Selenium or JMeter require time to prepare and integrate with the application under test.

## 5. Team Skills and Experience

- **Team Experience:** The experience level of the testing team influences the estimation. A more experienced team can perform testing faster and more efficiently, while less experienced teams may require more time for learning and rework.
- **Collaboration with Development:** Close collaboration with developers can speed up the identification of issues and provide faster feedback, reducing the time needed for re-testing.

## 6. Risk Analysis

- **Risk-Based Testing:** If the project involves high-risk areas, I allocate more time for testing those areas to mitigate potential issues that could have a significant impact on the project's success.
- **Defect History:** Analyzing the defect history (if applicable) of similar features or past releases can help estimate effort based on previous challenges faced.

## 7. Test Execution Time

- **Test Execution:** I estimate the time required to execute each test case, considering factors like test case complexity, execution environment, and any dependencies that may delay execution (e.g., waiting for other components or systems).
- **Regression Testing:** If there are many changes or new features, regression testing may take up a significant portion of the effort, especially if a large number of test cases need to be re-executed.

## 8. Contingency and Buffer

- **Contingency Planning:** I always add a contingency buffer to the estimate, typically 10-20%, to account for unforeseen issues like test environment problems, unexpected

defects, or additional requirements that may arise during the testing phase.

## 9. Effort Breakdown

- **Break Down Effort into Phases:** The overall effort can be broken down into different phases, such as:
  - Test planning and preparation
  - Test case design and review
  - Test execution
  - Defect reporting and retesting
  - Test closure activities (e.g., reporting, documentation)

## 10. Use of Historical Data and Metrics

- **Historical Data:** If I have access to historical data from previous similar projects, I leverage it to make more accurate estimates. I consider metrics such as the number of test cases executed per day, the average defect discovery rate, and time taken for specific testing types.

---

By considering all these factors, I can provide a well-rounded and realistic effort estimate for testing, ensuring that both the team and stakeholders have a clear understanding of the time and resources required for successful test execution.

**Tell me about a critical bug you found and how you handled it.**

Sure! Here's an example answer that highlights a critical bug I found and how I handled it, which you could use in an interview:

---

**Answer:**

In one of the projects I was working on, I was testing a financial application that handled user transactions. During a regression test, I discovered a critical bug in the system that affected the transaction process. The issue was that when users tried to transfer funds between accounts, the application was deducting the correct amount from the sender's account but was failing to credit the recipient's account correctly under certain conditions (e.g., when a user had more than 100 pending transactions). This bug could lead to significant financial discrepancies, potentially causing serious issues for users and the business.

Here's how I handled it:

1. **Reproduction and Investigation:** I first verified the issue by carefully retracing the steps and observing the conditions under which it occurred. I repeated the process multiple times, under different scenarios, to ensure the bug was reproducible. After confirming it, I tried to isolate the exact cause by analyzing the logs, checking the code responsible for transaction handling, and collaborating with the development team to understand potential areas that could trigger such issues.
2. **Documenting the Bug:** I documented the bug in our bug tracking system, including a detailed description, steps to reproduce, screenshots of logs, and videos showing the issue in action. I also included the potential impact, emphasizing how this could severely damage the application's reputation and user trust, especially for users with high transaction volumes.
3. **Communicating with the Team:** I immediately raised the issue with the development team, providing all the necessary details for them to prioritize the fix. Since this bug was critical, I made sure to stress the urgency, and we arranged for a quick patch to address the issue. I also collaborated closely with the developers to understand the root cause, which was a race condition that occurred due to a lack of synchronization when processing large batches of transactions.
4. **Verification and Retesting:** After the fix was implemented, I retested the entire transaction process, ensuring that the bug was resolved and no other functionality had been impacted. I also conducted additional regression tests to confirm that other areas were not inadvertently affected by the patch.
5. **Post-Resolution Follow-up:** After verifying the fix, I ensured that proper logging and monitoring were in place to catch similar issues early in the future. I also contributed to the documentation for future testing to help identify such issues quickly, particularly in cases involving high-volume transactions.

This experience taught me the importance of thorough testing, clear communication with the team, and always prioritizing user-impacting issues.

---

This type of response not only demonstrates problem-solving skills but also highlights your attention to detail, collaboration, and ability to handle high-pressure situations.

## What would you do if the requirements keep changing?

If the requirements keep changing during the course of the project, here's how I would handle the situation:

1. **Maintain Clear Communication:** I would communicate with the stakeholders (product owners, business analysts, or project managers) to understand the exact nature of the changes. Clarifying the scope of each change is crucial to ensure we're aligned and to avoid any confusion.
2. **Reassess the Testing Strategy:** I would review the test plans, test cases, and test scripts to reflect the new requirements. If the changes are substantial, I may need to revise or re-write some of the test cases to ensure they are still relevant.
3. **Adapt the Test Prioritization:** In such cases, it's important to prioritize the testing efforts based on the most critical and high-risk features. I would collaborate with the team to ensure that any new or modified requirements are adequately tested, while also maintaining coverage of the existing functionality.
4. **Version Control of Test Cases:** Keeping track of the different versions of test cases becomes vital when requirements change. I would maintain proper version control to make sure we can trace which version of the requirements and test cases were valid for which builds.
5. **Regression Testing:** With changing requirements, I would place a strong emphasis on regression testing to ensure that new changes do not break existing functionality. Automated tests (if available) can be leveraged for quicker and more comprehensive regression testing.
6. **Incremental Testing:** I would focus on incremental testing, running tests for the new functionality as it's developed. This ensures that new requirements are thoroughly tested as soon as they are implemented.
7. **Feedback Loop:** I would provide frequent feedback to the development team about the impact of the changes on the testing process, highlighting areas of concern or potential issues. This ensures that everyone is aware of the risks and challenges introduced by the changes.

8. **Documentation:** I would ensure that all changes are well-documented, both for the test cases and the requirements themselves. Proper documentation helps in tracing the changes and maintaining clarity on what has been tested and what needs additional verification.

By being flexible, maintaining clear communication, and using strategic test planning, I can effectively manage the impact of changing requirements on the testing process while ensuring quality is upheld.

## What would you do if developers say "It's not a bug"?

When developers say, "It's not a bug," as a manual tester, here's how I would approach the situation:

1. **Understand Their Perspective:** First, I would try to understand the developer's reasoning behind their statement. It might be a case of a misunderstanding or a difference in interpretation of the requirements. I would ask clarifying questions to gain insight into their perspective and explain the issue from my end.
2. **Revisit the Requirements:** I would double-check the project's requirements, user stories, or any other documentation to ensure that the feature or functionality is being tested according to the correct specifications. If there's a discrepancy, I'd present this to the developer as evidence that the behavior is indeed unexpected.
3. **Provide Reproducible Steps:** I would share the detailed steps to reproduce the issue along with any screenshots, logs, or video recordings that demonstrate the problem. A clear and concise reproduction process can often clarify why it's being considered a bug.
4. **Explain the Impact:** I'd focus on the impact of the issue from a user's perspective. If the issue could lead to poor user experience or functionality failure, I would explain this clearly to the developer, emphasizing that it's not just a technical issue but one that could affect the end-users.
5. **Test Environment Consideration:** I would verify if the issue is specific to a particular environment or configuration (e.g., browser, operating system, or device). If it's environment-specific, I'd share those details and suggest testing across different environments.
6. **Refer to Similar Cases or Industry Standards:** If applicable, I'd refer to any similar bug cases or industry standards that support why the behavior should be considered a bug.



Sometimes, using real-world examples can help put things into perspective.

7. **Escalate If Necessary:** If the issue is critical, and after discussion, the developer still refuses to acknowledge it as a bug, I would escalate the matter to the project manager or the relevant team lead. I'd present the issue clearly, with evidence and an explanation of why it's important for the project.

The key is always to maintain open, constructive communication and ensure that the issue is addressed from both a technical and user experience standpoint.

## What test management tools have you used (e.g., Jira, TestRail)?

In response to the interview question "What test management tools have you used (e.g., Jira, TestRail)?", here's how you can answer, assuming you're experienced in manual testing:

---

"I have worked with several test management tools during my experience in manual testing. Some of the key tools I have used include:

1. **Jira:** Primarily for test case management, tracking issues, and integrating with other tools like TestRail for test execution. I've used Jira for creating, updating, and managing test cases and defects in both agile and waterfall environments. Its powerful integration capabilities with tools like Confluence and Bitbucket have been quite useful in managing the overall project lifecycle.
2. **TestRail:** I've used TestRail for organizing and executing test cases, as well as tracking test execution results. It's beneficial for managing large test suites, creating detailed reports, and ensuring comprehensive coverage across test cases. TestRail's seamless integration with Jira also allows for linking test cases and defects, making the defect management process smoother.
3. **ALM (Application Lifecycle Management):** In some projects, I've worked with HP ALM (formerly Quality Center) for test case management, defect tracking, and test execution. It's a comprehensive tool that supports the entire testing lifecycle from planning to execution and reporting.
4. **Zephyr:** A Jira plugin that I have used in some projects to manage test cases and integrate with Jira's issue tracking system. Zephyr offers a more integrated experience within the Jira ecosystem and simplifies the management of both manual and automated

tests.

5. **TestLink:** I have also worked with TestLink in some projects for managing test cases and requirements traceability. It's a popular open-source tool that allowed me to plan, execute, and report test cases, and it also offered useful features for managing requirements.
6. **QTest:** In some agile environments, I have used QTest for test case management, reporting, and collaboration among testing teams. It's easy to track test progress and integration with CI/CD pipelines.

Each tool has its strengths, and depending on the project requirements, I've been able to choose the one that best fits the team's workflow and test management needs."

---

This answer demonstrates not only your experience with various test management tools but also highlights your understanding of when and how to use them based on the project requirements.

## What is the difference between Jira and Bugzilla?

Sure! Here's how I would answer this question based on my experience with manual testing:

---

### Jira vs Bugzilla: Key Differences

#### 1. Purpose and Functionality:

- **Jira:** Jira is a comprehensive project management tool developed by Atlassian. While it is widely used for bug tracking, it also provides features for project management, issue tracking, agile planning (scrum and kanban boards), and integration with various development tools. It is highly customizable and supports workflows, user stories, sprints, and more.
- **Bugzilla:** Bugzilla is an open-source bug tracking tool developed by Mozilla. It is primarily focused on tracking and managing bugs. It offers basic features for bug reporting, tracking, and resolution, but lacks some of the more advanced features related to project management and agile workflows that Jira provides.

#### 2. Ease of Use:

- **Jira:** Known for its user-friendly interface and powerful reporting capabilities, Jira provides a smoother experience for both testers and developers. It allows easy navigation through different boards, backlog management, and is highly customizable to fit specific project needs.
- **Bugzilla:** Bugzilla's interface is more utilitarian and can feel somewhat outdated. While it's effective for bug tracking, it may require a bit more effort to configure and manage compared to Jira.

### 3. Customization:

- **Jira:** Offers extensive customization options such as custom fields, issue types, workflows, and permissions. This makes it adaptable to a variety of project management needs beyond just bug tracking.
- **Bugzilla:** Bugzilla allows customization but is more limited compared to Jira. Custom fields, workflow management, and permissions are more basic, making it better suited for simpler bug tracking.

### 4. Integration:

- **Jira:** Jira integrates seamlessly with a wide range of development and testing tools, including Confluence (documentation), Bitbucket (version control), Trello (task management), and many others. It also supports various plugins to enhance functionality.
- **Bugzilla:** Bugzilla also integrates with other tools, but the number of integrations and the ease of setup are not as extensive as Jira's. It can integrate with version control systems and CI/CD tools but may require additional configuration.

### 5. Agile Support:

- **Jira:** Jira is designed with agile teams in mind. It has built-in support for Scrum and Kanban boards, sprint management, and backlog prioritization. It allows teams to manage their entire agile workflow in one place.
- **Bugzilla:** Bugzilla does not have built-in agile support. While it can be used in an agile environment, it lacks native features like sprint planning, Kanban boards, and agile reports.

### 6. Cost:

- **Jira:** Jira is a commercial product with subscription-based pricing. It offers a free tier with limited features for small teams, but larger teams and enterprises need

to pay for advanced features.

- **Bugzilla:** Bugzilla is open-source and free to use. There are no licensing costs, making it an attractive option for organizations with budget constraints.

## 7. Reporting:

- **Jira:** Jira provides advanced reporting features, including detailed metrics like burndown charts, velocity charts, and more, which are highly useful for agile project management.
- **Bugzilla:** Bugzilla provides basic reporting functionality, focusing mainly on bug-related reports, but does not offer the wide range of project and agile reports that Jira does.

## Summary:

- **Jira** is a versatile tool with advanced project management and agile capabilities, making it suitable for organizations needing robust issue tracking and full project management integration.
- **Bugzilla** is simpler, focusing primarily on bug tracking, and is a cost-effective choice for teams that just need a straightforward bug management solution without the complexity of project management features.

---

This distinction is important in interviews, especially when discussing the context in which these tools are used and their advantages depending on team needs.

## How do you write test cases in Jira?

To write test cases in Jira, you typically use a plugin called **Xray** or **Zephyr** (most commonly used test management tools in Jira) or Jira's native issue types if you're using a custom workflow. Here's a detailed approach for writing test cases in Jira using these tools:

### 1. Log in to Jira

- Open your Jira instance and log in with your credentials.

### 2. Navigate to the Project

- Go to the project where you want to create test cases. Ensure you have the appropriate permissions to add issues.

### 3. Choose the Test Case Issue Type

- If you're using a tool like **Zephyr** or **Xray**, you will find an issue type for **Test**.
- Click on the **Create** button, and from the issue type dropdown, select **Test** (if using Xray or Zephyr).

### 4. Fill in the Test Case Details

- **Summary:** Write a short, descriptive title for the test case. E.g., "Verify user login functionality with valid credentials."
- **Description:** Include the purpose and detailed information about the test case. Describe what is being tested, any preconditions, and expected behavior.
- **Test Steps** (in Zephyr or Xray):
  - List the specific steps needed to execute the test case.
  - For example:
    - Step 1: Open the application.
    - Step 2: Enter valid username and password.
    - Step 3: Click on the Login button.
    - Step 4: Verify user is directed to the dashboard.
- **Expected Results:** State the expected outcome for each step or for the test as a whole. For instance, "The user should be logged in and redirected to the dashboard."

(In Xray or Zephyr, this might be in the form of **Test Steps** and **Expected Results** fields.)

### 5. Set the Priority and Assignee

- **Priority:** Set the priority of the test case (e.g., High, Medium, Low).

- **Assignee:** Assign the test case to the appropriate tester or team.

## 6. Link Test Cases (if applicable)

- Link the test case to the corresponding **User Story**, **Epic**, or **Bug** if applicable. This ensures that your test case is related to a specific feature or requirement.
- In **Xray**, you can link the test to a **Test Execution**.

## 7. Add Test Case Components and Labels (Optional)

- **Components:** Categorize the test case (e.g., Login, Registration).
- **Labels:** Add any labels to help in filtering or categorizing the test case, like "Regression," "Smoke," or "Sanity."

## 8. Attach Files/Attachments (if applicable)

- Attach any relevant files or screenshots that could help in understanding the test case or are necessary for its execution.

## 9. Save and Review

- After filling in all the necessary details, click on the **Create** button to save the test case.
- Review the test case to ensure everything is captured correctly.

## 10. Run the Test (if using Xray/Zephyr)

- Once the test case is created, it can be included in a **Test Execution** cycle, where testers can execute the test, log results, and track progress.

## Additional Tips:

- **Test Case Structure:** Test cases should be simple, clear, and easy to understand. Follow a format like:

- **Test Case ID:** A unique identifier for the test case.
- **Test Objective:** What the test aims to achieve.
- **Preconditions:** Conditions that must be met before starting the test.
- **Test Steps:** Clear instructions for executing the test.
- **Expected Result:** What the tester should expect to happen after executing the test.
- **Actual Result:** To be filled out after the test is executed.
- **Status:** Pass/Fail or any other applicable status.

By following these steps, you'll ensure that your test cases are properly written and easy to manage within Jira.

## Do you have experience with any test case management tools?

Yes, I have experience with several test case management tools. Some of the tools I've worked with include:

1. **TestRail:** This is one of the most widely used test case management tools. It allows you to create, manage, and track test cases, as well as report on test execution. I've used TestRail to organize test suites, track progress during test cycles, and integrate it with other tools like JIRA for bug tracking.
2. **Quality Center (ALM):** I've worked with HP ALM (now known as Micro Focus ALM), which provides comprehensive test management capabilities. It allows for test planning, test execution, defect tracking, and reporting, and integrates well with other development tools.
3. **JIRA + Zephyr:** I have experience using JIRA in combination with Zephyr, a test management plugin. Zephyr allows for managing test cases, test execution, and reporting directly within JIRA, making it a convenient solution when integrated with agile project management.
4. **TestLink:** TestLink is an open-source test case management tool I've used to organize test cases, plan test execution, and generate reports. It also integrates with bug tracking

systems like Bugzilla, JIRA, and others.

5. **PractiTest:** I've used PractiTest for managing test cases, tracking issues, and reporting on test execution. It allows for customization of workflows and integrates with other tools like JIRA, GitHub, and Jenkins.

These tools have helped me in test case organization, execution tracking, defect reporting, and overall project management. I've used them in both manual testing and automated testing contexts, ensuring that the testing process is streamlined and that all stakeholders can access up-to-date information on test status.

## Why do you want to be a QA?

As a QA professional, I want to contribute to creating high-quality products that meet both functional and user expectations. The role of a QA is crucial because it ensures that the end product is not only reliable but also seamless in its performance, usability, and overall user experience.

Personally, I find the process of identifying issues, thinking critically about how users will interact with a product, and collaborating with development teams to resolve them extremely fulfilling. It's also rewarding to see the direct impact of the work I do, as it helps prevent defects from reaching users and improves the overall satisfaction of customers.

Additionally, QA gives me the opportunity to continuously learn and grow within the field. I enjoy using various testing techniques, automation tools, and methodologies, which keep me engaged and allow me to adapt to ever-evolving industry trends.

In summary, I want to be a QA because it aligns with my passion for problem-solving, quality assurance, and the continuous improvement of software products.

## What challenges have you faced as a tester?

As an experienced tester, I've encountered several challenges throughout my career, and each one has helped me grow professionally. Here are some common challenges I've faced:

1. **Lack of Clear Requirements:** One of the major challenges is working with unclear or incomplete requirements. This can lead to ambiguity in test cases, which makes it difficult to ensure that all aspects of the application are being tested effectively. To overcome this, I collaborate closely with business analysts, developers, and stakeholders to clarify requirements and ensure a solid understanding of what needs to be tested.



2. **Time Constraints and Deadlines:** Tight deadlines can often mean having to test under pressure. In these situations, prioritization becomes crucial. I rely on risk-based testing to focus on the most critical parts of the application and work closely with the development team to ensure the key features are thoroughly tested within the given timeframe.
3. **Environment and Setup Issues:** A challenging aspect of testing is often the setup of the testing environment. Discrepancies between different environments (e.g., development, staging, production) can lead to tests behaving differently in each. To address this, I work on making sure the testing environment is stable, and I frequently communicate with the infrastructure or DevOps teams to mitigate issues related to the environment.
4. **Unstable Builds and Frequent Code Changes:** Working with unstable builds or frequent code changes can disrupt the testing process. It becomes difficult to maintain test cases when the product is constantly changing. To combat this, I practice continuous integration and regression testing, and I ensure communication with the development team to stay updated on code changes.
5. **Automation and Manual Testing Balance:** Deciding which tests should be automated and which should remain manual can be challenging. Automation is great for repetitive tasks, but some scenarios require a human touch. I've learned to identify areas where automation brings the most value, and where manual testing can provide insights that automated tests may miss.
6. **Cross-Browser and Cross-Device Testing:** With multiple devices, browsers, and OS configurations, testing for compatibility can be overwhelming. I use tools like BrowserStack and Sauce Labs for cross-browser and cross-device testing, but it requires thorough planning and constant updates as new versions of browsers and devices emerge.
7. **Test Data Management:** Finding or creating the right test data for different test cases can be a challenge. Sensitive or confidential data may not always be available for testing, and generating data that mimics real-world scenarios is time-consuming. To address this, I create data sets and work with the development team to generate realistic data while adhering to security standards.
8. **Handling Communication and Collaboration:** Effective communication with developers, product managers, and other team members is essential. Miscommunication or lack of coordination can lead to misunderstandings about test coverage or defect severity. I strive to maintain clear and transparent communication channels and ensure everyone is aligned on priorities.
9. **Dealing with Repetitive Test Scenarios:** Sometimes, testing the same functionality repeatedly becomes tedious, especially with regression testing. To alleviate this, I focus

on automating repetitive tasks and creating reusable test scripts that can save time and effort in the long run.

10. **Defect Rejection and Misinterpretation:** In some cases, the defects I report may be rejected by developers due to a lack of understanding or misinterpretation. This can be frustrating, but I ensure that I provide detailed steps, screenshots, and logs to support my findings. I also maintain a collaborative approach with developers to discuss and resolve issues effectively.

Despite these challenges, my experience has allowed me to develop strategies to overcome them and continue providing high-quality testing and valuable insights to the development process.

## How do you stay updated with testing trends?

As an experienced manual tester, staying updated with the latest trends in testing is crucial to maintaining expertise and ensuring the best practices are applied. Here's how I stay updated:

1. **Continuous Learning and Online Courses:** I regularly enroll in advanced courses on platforms like Udemy, Coursera, or LinkedIn Learning to deepen my knowledge of testing tools, methodologies, and frameworks. This helps me stay ahead in both manual and automated testing.
2. **Industry Blogs and Websites:** I follow well-established testing blogs and websites like Ministry of Testing, Software Testing Help, and TestAutomationU to read about the latest trends, case studies, and best practices in testing. These platforms also have a community of testers where discussions on new trends and challenges happen frequently.
3. **Webinars and Conferences:** Attending webinars and conferences like STAREAST, TestBash, or Agile Testing Days gives me insights into current trends, tools, and the future of testing. They also provide networking opportunities with other professionals in the field.
4. **Networking with Peers:** I actively participate in testing communities, online forums (like Stack Overflow and Reddit), and social media groups (on LinkedIn and Twitter). These communities help me stay updated with industry trends, tools, and methodologies being adopted globally.
5. **Books and Research Papers:** I read books from experts in the field of software testing. Books like "The Art of Software Testing" by Glenford Myers and "Foundations of Software Testing" by Rex Black are great resources. Additionally, I keep an eye on

research papers in software testing and quality assurance.

6. **Follow Influencers and Thought Leaders:** I follow recognized leaders in the software testing field on social media and blogs. These thought leaders often share new tools, testing trends, and innovations. For example, I follow people like Michael Bolton, James Bach, and Lisa Crispin.
7. **Hands-on Practice:** To truly understand the latest trends, I apply them in my daily work or on side projects. Experimenting with new tools, testing methods, and techniques ensures that I stay ahead of the curve.
8. **Participate in Open Source Projects:** I contribute to open-source testing projects on GitHub. It allows me to learn about new tools and technologies, and gain insights into industry standards and practices that are shaping the future of testing.

By using a combination of these strategies, I ensure I'm always up-to-date with the latest trends in the testing world.

## How do you manage stress when testing under pressure?

When answering this question in an interview, it's important to showcase not only your stress management skills but also how you maintain productivity and focus while under pressure. Here's how you can structure your answer:

---

### Managing Stress While Testing Under Pressure:

"I understand that manual testing often involves tight deadlines, high expectations, and sometimes dealing with unexpected issues, all of which can contribute to stress. Over the years, I've developed a few key strategies to manage stress effectively during these situations:

1. **Prioritizing Tasks:** When I'm under pressure, I prioritize tasks based on the severity and impact of issues. I first focus on high-priority tests that will have the most significant impact on the product's quality. This helps me avoid feeling overwhelmed by the volume of work and ensures that I'm addressing the most critical aspects first.
2. **Time Management:** I break down my tasks into manageable chunks and set realistic deadlines for each. This helps me stay organized and reduces the feeling of being rushed. I also use test case management tools to keep track of my progress and ensure

nothing falls through the cracks.

3. **Clear Communication:** I believe in maintaining open communication with my team. If I feel the pressure is mounting, I'll reach out to discuss challenges or share progress updates. This ensures that the stakeholders are aware of where things stand and helps in adjusting timelines or expectations if needed.
4. **Staying Calm and Focused:** When I start to feel stressed, I take a step back and focus on one task at a time. I remind myself to stay calm, breathe deeply, and approach each test methodically. Stress often arises when we try to tackle everything at once, so staying focused on one thing at a time really helps.
5. **Problem Solving and Flexibility:** Testing under pressure can sometimes bring up unexpected challenges. I remain flexible and embrace a problem-solving mindset, trying to find solutions quickly without panicking. I also collaborate with the development team to resolve blockers efficiently.
6. **Self-Care and Taking Breaks:** It's also essential to take short breaks when needed. A few minutes of walking around or stretching can help me reset and regain focus, which ultimately reduces stress and increases productivity.

By using these techniques, I've been able to maintain a high level of performance even under stressful conditions. I also view challenges as opportunities to grow and improve my testing process, which helps me manage pressure more effectively."

---

This answer demonstrates your ability to handle stress in a productive and calm manner while showcasing skills that are valuable in manual testing, such as time management, communication, and problem-solving.

## Are you open to learning automation?

If asked "Are you open to learning automation?" in a manual testing interview, here's how you might respond:

"Yes, absolutely. While I have a strong foundation in manual testing, I understand that the industry is moving towards automation to improve efficiency and scalability. I'm eager to expand my skill set and learn automation testing. In fact, I see it as an opportunity to complement my manual testing experience, allowing me to work more effectively on repetitive tasks and focus more on complex test scenarios. I've already started exploring automation tools like Selenium and Cypress, and I'm excited to gain more hands-on experience with them."

This response demonstrates your openness to learning, willingness to adapt, and proactive approach to expanding your skill set.

## What's your long-term career goal in testing?

As an experienced manual tester, I would respond to the question "What's your long-term career goal in testing?" with something like this:

"My long-term career goal in testing is to become an expert in both manual and automation testing, with a strong focus on optimizing testing processes and ensuring high-quality software delivery. I aim to evolve into a testing leader who can mentor and guide junior testers, share best practices, and help shape testing strategies within an organization. Over time, I also see myself playing a key role in implementing advanced testing techniques, such as performance and security testing, and leading teams in adopting cutting-edge testing tools and frameworks. Ultimately, I aspire to contribute to the overall improvement of software quality by integrating testing throughout the development lifecycle and helping organizations deliver products that meet the highest quality standards."

This response shows your dedication to growth in the field and highlights a vision for both technical and leadership development in testing.

Assume you know what interview questions will be asked for manual testing and have been experienced in this interview process then what are all the possible questions will be asked in interview from beginner to advance

That's a great mindset to prepare for interviews like a pro! Here's a **comprehensive list of manual testing interview questions**—structured from **beginner** to **advanced** levels—based on what interviewers typically ask in real scenarios. I'll also group them by topic for better clarity.

---

### Beginner Level (0–1 year experience)

#### Manual Testing Basics

1. What is Software Testing?
2. What is the difference between Verification and Validation?
3. What are the different levels of testing?
4. What are the types of testing (Manual vs Automation)?
5. What is the difference between Functional and Non-functional testing?
6. What is SDLC and STLC?
7. What is a Test Case and Test Scenario?
8. What is the difference between Test Plan and Test Strategy?
9. What is a Defect/Bug? How do you report it?
10. What is the difference between Severity and Priority?

### **Test Artifacts**

11. What is a Test Plan and what does it include?
  12. How do you write a good test case?
  13. What is a Traceability Matrix (RTM)? Why is it used?
- 

## **Intermediate Level (1–3 years experience)**

### **Test Design Techniques**

14. What is Boundary Value Analysis (BVA)? Give an example.
15. What is Equivalence Partitioning?
16. What is Decision Table Testing?
17. What is State Transition Testing?

18. What is Exploratory Testing?

### ✓ Bug Life Cycle & Reporting

19. Can you explain the bug life cycle?

20. How do you report a bug in Jira or Bugzilla?

21. What are the different statuses of a bug?

22. How do you handle conflicting bugs with developers?

### ✓ Testing Types

23. What is Regression Testing? When is it done?

24. What is Smoke Testing vs Sanity Testing?

25. What is Ad hoc Testing?

26. What is Acceptance Testing?

### ✓ Test Execution & Reviews

27. How do you prioritize test cases for execution?

28. How do you ensure complete test coverage?

29. What is Peer Review? What is Walkthrough?



## Advanced Level (3+ years experience)

### ✓ Agile & Modern Testing Practices

30. What is Agile Testing? What is your role in Agile?

31. What is a Sprint and what happens in a Sprint Review/Retrospective?

- 32. What is the role of a QA in Scrum?
- 33. What is the difference between TDD and BDD?
- 34. What is Shift Left Testing?

### **Risk-Based Testing & Strategy**

- 35. How do you perform Risk-Based Testing?
- 36. How do you handle testing in tight deadlines?
- 37. How do you estimate effort for testing?

### **Real-Time Situational Questions**

- 38. Tell me about a critical bug you found and how you handled it.
- 39. What would you do if the requirements keep changing?
- 40. What would you do if developers say "It's not a bug"?

### **Tools Awareness**

- 41. What test management tools have you used (e.g., Jira, TestRail)?
- 42. What is the difference between Jira and Bugzilla?
- 43. How do you write test cases in Jira?
- 44. Do you have experience with any test case management tools?



### **Bonus: Behavioral & Miscellaneous**

- 45. Why do you want to be a QA?
- 46. What challenges have you faced as a tester?



- 47. How do you stay updated with testing trends?
- 48. How do you manage stress when testing under pressure?
- 49. Are you open to learning automation?
- 50. What's your long-term career goal in testing?

---

Would you like me to turn these into a **study sheet**, or provide **answers with examples** for each? Or maybe a **mock interview simulation** to test your readiness?