Black Box Testing

Testing data …environment … techniques

White box testing

Quality

Automation (UI, Api, performance, unit)

**Black Box Testing**

**Understand Requirements**: Break down inputs, outputs, and constraints.

**Apply Techniques**:

* ECP: Divide inputs into valid/invalid classes.
* BVA: Identify boundary values and test OFF, ON, IN.
* Decision Table: Map all combinations of conditions.
* State Transition: Identify states and transitions.

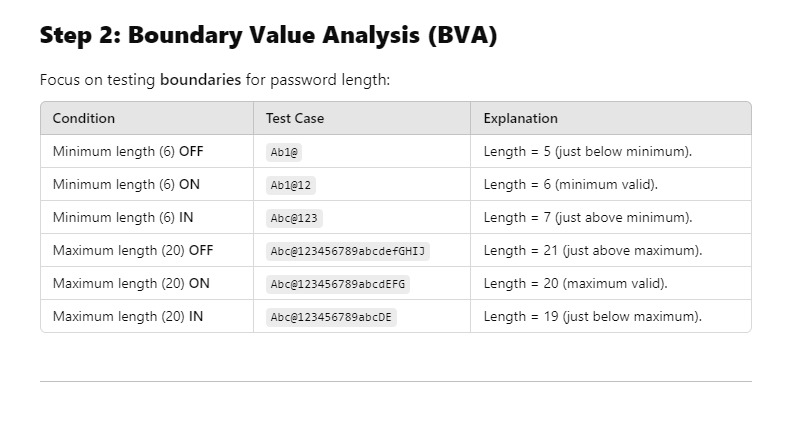
**Write Test Cases**: Use Standard Format or Gherkin Syntax.

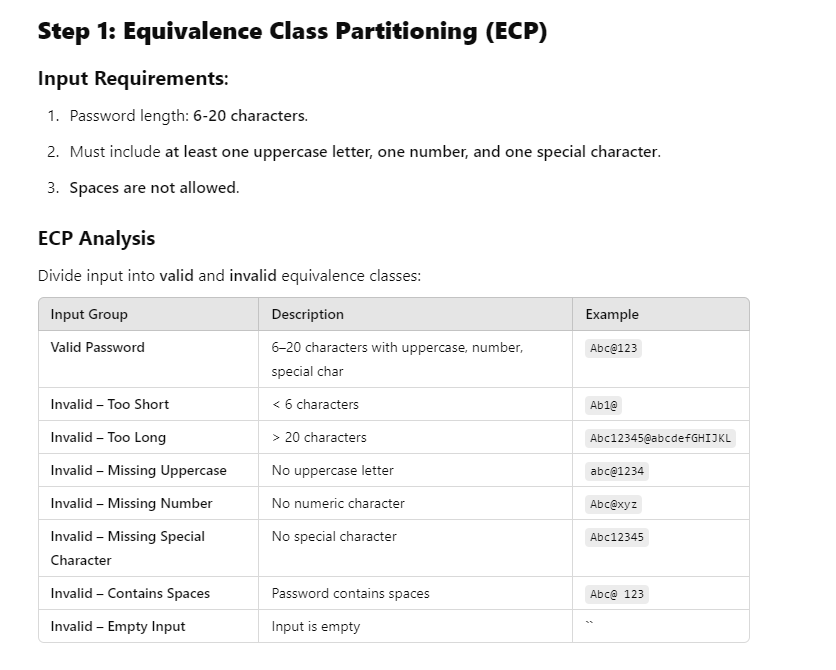
Example

 Password length: **6-20 characters**.

 Must include **at least one uppercase letter, one number, and one special character**.

 **Spaces are not allowed**.





**Gherkin** syntax that includes all key Gherkin keywords: Feature, Scenario, Scenario Outline, Given, When, Then, And, But, and Examples

**Feature:** Password Validation

Description: The system validates user passwords based on predefined rules.

**Background:**

Given the user is on the password entry screen

Scenario: Valid password input

Given the user is on the password entry screen

When the user enters a password "Abc@123"

Then the system should accept the password

And display a success message "Password accepted successfully."

Scenario: Password is too short

Given the user is on the password entry screen

When the user enters a password "Ab1@"

Then the system should reject the password

And display an error message "Password too short. Minimum length is 6."

Scenario Outline: Password fails specific validation criteria

Given the user is on the password entry screen

When the user enters a password "<password>"

Then the system should display an error message "<error message>"

But the system should not accept invalid passwords

Examples:

| password | error message |

| Ab1@ | Password too short. |

| Abc@123456789abcdefGHIJ | Password too long. |

| abc@1234 | Password must include uppercase. |

| Abc@xyz | Password must include a number. |

| Abc12345 | Password must include a special char.|

| Abc@ 123 | Password cannot contain spaces. |

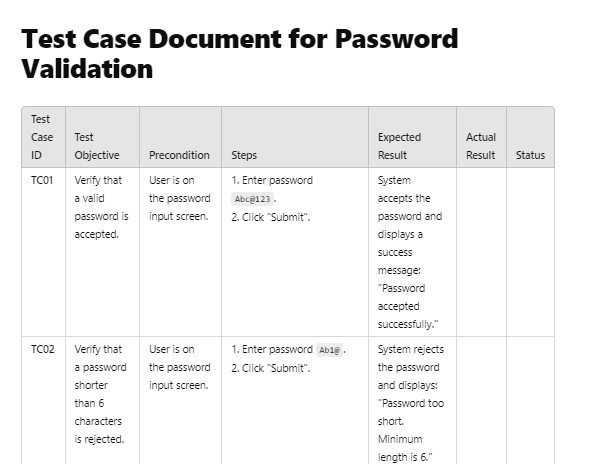
Scenario: Password contains spaces

Given the user is on the password entry screen

When the user enters a password "Abc@ 123"

Then the system should reject the password

But display an error message "Password cannot contain spaces."

****

**How to select Testing Techniques:**

**1. When Requirements are Unclear or Missing**

When requirements are incomplete or absent:

* **Approach**: Use **Experience-Based Testing Techniques**.
* **Techniques**:
  + **Error Guessing**:
    - Predict common issues based on past experience or known software failures.
    - Example: Test common areas where issues arise, like boundary conditions, null inputs, invalid combinations, or integration errors.
  + **Checklist-Based Testing**:
    - Create a checklist of all functional and non-functional areas.
    - Example: Browser compatibility, mobile responsiveness, security.
  + **Exploratory Testing**:
    - Learn and test simultaneously by identifying areas of focus.
    - Example: Use **Session-Based Testing**: Time-boxed sessions (e.g., 60-90 minutes) with a specific testing objective (e.g., login functionality).

**2. Systematic Approach with Known Requirements**

If the requirements are clear, documented, or explained:

* Apply a **combination of Black-Box Testing Techniques** to optimize test cases:

**A. Equivalence Class Partitioning (ECP)**

* **Purpose**: Divide input data into groups (valid/invalid partitions).
* **Approach**: Test one value from each group.
* **Example**:
  + Input range 1–100 → Test cases: 0 (invalid), 50 (valid), 101 (invalid).

**B. Boundary Value Analysis (BVA)**

* **Purpose**: Test boundary conditions where bugs are most likely to occur.
* **Approach**: Test **ON**, **OFF**, and **IN** boundaries.
* **Example**:
  + Minimum: 6 → Test 5 (OFF), 6 (ON), 7 (IN).
  + Maximum: 20 → Test 19 (IN), 20 (ON), 21 (OFF).

**C. Decision Table Testing**

* **Purpose**: Handle multiple input combinations and conditions.
* **Approach**: Create a decision table for possible combinations and expected outcomes.
* **Example**:  
  | Condition 1 | Condition 2 | Condition 3 | Outcome |  
  |-------------|-------------|-------------|---------|  
  | Yes | No | No | A |  
  | Yes | Yes | No | B |

**D. State Transition Testing**

* **Purpose**: Test systems with multiple states and transitions.
* **Approach**: Identify valid and invalid state changes.
* **Example**: Login → Incorrect attempts (Lock account) → Reset → Retry.

**3. Test Strategy Based on Scenarios**

**Scenario 1: No Requirements, Testing Existing Software**

* **Approach**: Experience-Based Testing:
  + Use **Error Guessing** for common failure areas.
  + Perform **Exploratory Testing** to discover unknown issues.
  + Create a **Checklist** for functional and non-functional testing.

**Scenario 2: Agile/Scrum Environment**

* **Approach**:
  + Collaborate with developers during requirement discussions.
  + Apply **Behavior-Driven Development (BDD)** using Gherkin syntax.
  + Use techniques like **ECP**, **BVA**, and **State Transition** testing to design optimized test cases during each sprint.

**Scenario 3: Detailed Requirements Document (V-Model)**

* **Approach**:
  + Design systematic test cases using **ECP, BVA, Decision Tables**, and **State Diagrams**.
  + Use detailed requirement analysis to ensure coverage of all inputs/outputs.
  + Prioritize high-risk and critical functionalities for testing.

**4. Consolidated Steps to Solve Any Logical Question**

Follow these steps to approach any test case scenario logically:

1. **Understand Requirements**:
   * Identify the inputs, outputs, constraints, and system behavior.
2. **Choose the Right Technique**:
   * **If inputs have clear ranges**: Use **ECP and BVA**.
   * **If multiple conditions exist**: Use **Decision Tables**.
   * **If states and transitions exist**: Use **State Transition Testing**.
   * **If requirements are unclear**: Use **Error Guessing** or **Exploratory Testing**.
3. **Design Test Cases**:
   * Use a systematic format:  
     | **Test Case ID** | **Objective** | **Steps** | **Expected Result** |
4. **Optimize Test Coverage**:
   * Ensure all partitions (valid/invalid), boundaries, and combinations are tested.
5. **Review and Refine**:
   * Review test cases to remove redundancies and ensure completeness.

**5. Behavior-Driven Development (BDD) as a Tool**

Use Gherkin syntax to write structured and logical test cases:

1. **Feature**: High-level functionality description.
2. **Background**: Shared preconditions for all scenarios.
3. **Scenario/Scenario Outline**: Detailed test case steps.
4. **Steps**:
   * **Given**: Setup or precondition.
   * **When**: Action taken by the user.
   * **Then**: Expected outcome.
   * **And/But**: Extend steps or define contradictions.

**Example**:

**6. Combining Techniques in Practice**

To ensure comprehensive test coverage:

* Start with **ECP** to identify partitions.
* Apply **BVA** to test boundary values.
* Use **Decision Tables** for multi-condition inputs.
* Use **State Transition Testing** if workflows/states are involved.
* Perform **Exploratory Testing** for areas without documentation.

**1. Test Data Creation**

**What is Test Data?**

* All the data created or used during testing.
* It can be **Input Data** (e.g., passwords entered) or **Pre-Test Data** (e.g., database records for testing login).

**Categories of Test Data:**

To handle any logical scenario, include the following test data types:

1. **No Data**: System behavior when no input is provided.
2. **Normal Data**: Valid and expected input.
3. **Erroneous Data**: Invalid inputs that break rules.
4. **Illegal Data Format**: Test invalid formats (e.g., letters in a numeric field).
5. **Boundary Condition Data**: Focus on limits using BVA (e.g., min/max lengths).
6. **Equivalence Partition Data**: Valid/Invalid partitions using ECP.
7. **Decision Table Data**: Combinations of conditions.
8. **State Transition Data**: Data triggering state changes.
9. **Use Case Data**: Test data based on user scenarios.

**Functional Testing - Test Data, Test Env, and Reporting**

**How to Create Test Data**

1. **Manual Generation**:
   * Best for exploratory testing.
   * Time-consuming and bottlenecks CI/CD pipelines.
2. **From Production Data**:
   * Use real production data after **masking sensitive data**.
   * Add synthetic data for edge cases.
3. **Synthetic Test Data**:
   * Use tools like **Mockaroo** or **Generatedata.com** for robust and quick generation.

**2. Test Environment**

**Definition:**

A **test environment** is a setup of hardware, software, and data where testing occurs.

**Types of Test Environments:**

1. **Development Environment**: For developer testing (no client data).
2. **QA/Test Environment**: Used by testers with test data.
3. **Staging Environment**: Pre-production testing with limited production data.
4. **Production Environment**: Live environment with real client data.

**3. Bug Reporting**

**Bug Report Components**

A well-written bug report must include:

1. **Identifier**: Unique ID for the bug.
2. **Title/Summary**: A concise summary of the bug.
3. **Steps to Reproduce**: Actions to recreate the bug.
4. **Test Data Used**: Input data that caused the issue.
5. **Expected Result**: What should happen.
6. **Actual Result**: What actually happened.
7. **Severity & Priority**:
   * **Severity**: Impact of the defect (e.g., Critical, Major, Minor).
   * **Priority**: How soon it should be fixed.
8. **Screenshots/Logs**: Visual proof or system logs.

**Bug Life Cycle**

1. **New** → Reported bug.
2. **Assigned** → Sent to developer for fixing.
3. **In Progress** → Developer is working on the fix.
4. **Fixed** → Developer marks it resolved.
5. **Retest** → QA retests the issue.
6. **Closed** → Bug verified as fixed.
7. **Reopened** → If the issue still persists.

**4. Logical Approach to Solve Any Testing Scenario**

Follow these steps systematically to approach logical questions:

**Step 1: Analyze the Scenario**

* Identify the system behavior and requirements (e.g., inputs, expected outputs, constraints).

**Step 2: Create Test Data**

* Include all **categories** of test data: No Data, Normal Data, Erroneous Data, Boundary Data, etc.
* Use **ECP** and **BVA** for systematic testing.
* For complex conditions, create **Decision Tables** or **State Transition Diagrams**.

**Step 3: Set Up the Test Environment**

* Identify which environment (Development, QA, Staging) fits the scenario.
* Ensure all configurations, data, and tools are ready.

**Step 4: Execute Tests**

* Run test cases using:
  + **Manual Testing**: For exploratory or ad-hoc scenarios.
  + **Automated Testing**: For repetitive or regression scenarios.
* Track test execution status: Passed, Failed, Blocked, In-Progress.

**Step 5: Report Bugs**

* Log bugs with detailed information:
  + Steps, test data, screenshots, severity, priority, and logs.

**Step 6: Prioritize Tests**

* Execute **Sanity/Smoke Tests** first.
* Prioritize test cases based on risk and requirement changes.
* Run regression tests to ensure no new issues arise.

**Key Takeaways (Crux)**

1. **Understand the Problem**: Extract requirements, inputs, and outputs.
2. **Choose Test Techniques**:
   * Use **ECP** and **BVA** for input/output validation.
   * Use **Decision Tables** for condition-based scenarios.
   * Use **State Transition Testing** for workflows and state-based systems.
3. **Prepare Test Data**: Ensure coverage of all data categories (Normal, Boundary, Erroneous, etc.).
4. **Set Up Test Environment**: Configure software, tools, and hardware for testing.
5. **Report and Track Bugs**: Use a systematic approach to log and communicate defects clearly.
6. **Prioritize Testing**: Execute critical and high-risk scenarios first.

**Shift Left, Test Levels and Test Plan**

**1. Shift Left Testing – Test Early to Reduce Costs**

* **Concept**: Testing starts as early as possible in the **Software Development Life Cycle (SDLC)** to identify defects early and reduce costs.
* **Why?**
  + Cost of fixing defects increases exponentially in later phases.
  + Bugs in requirements or design can propagate to production and cause critical issues.
* **Approach**:
  + Use **Static Testing** in the early phases (e.g., reviews, inspections).
  + Combine it with **Dynamic Testing** in later phases (e.g., functional execution).

**2. Types and Levels of Testing**

Testing happens at **multiple levels** to ensure quality.

| **Test Level** | **Purpose** | **Example** |
| --- | --- | --- |
| **Unit Testing** | Test individual components/functions. | Test a “login function” logic. |
| **Integration Testing** | Verify interaction between components/modules. | Ensure API calls are successful. |
| **System Testing** | Validate the entire system works as expected. | Test an end-to-end application. |
| **Acceptance Testing** | Ensure system meets business requirements. | User validates final functionality. |

* **Key Idea**: Each level verifies functionality at different granularity and coverage.

**3. Test Plan – A Blueprint for Testing**

The **Test Plan** is the roadmap for the testing process and helps solve any testing question logically.

**Components of a Test Plan**

1. **High-Level Expectations**: Objectives, goals, and scope of testing.
2. **In-Scope/Out-of-Scope**: What features will and won’t be tested.
3. **Test Strategy**:
   * **What to test**: Features, modules, use cases.
   * **How to test**: Techniques like ECP, BVA, Decision Table, State Transition.
   * **When to test**: Test Phases – Sanity, Smoke, Functional, Regression.
4. **Resource Requirements**:
   * **People**: QA leads, testers.
   * **Tools**: Testing tools (e.g., Selenium, JIRA, Postman).
   * **Environments**: Dev, QA, Staging setups.
5. **Test Schedule**: Define testing activities with timelines. Example:

| **Phase** | **Start Date** | **End Date** | **Resources** |
| --- | --- | --- | --- |
| Test Plan Finalization | Week 1 | Week 2 | QA Lead |
| Test Case Writing | Week 2 | Week 4 | Testers (50-100%) |
| Execution – Phase 1 | Week 5 | Week 6 | Testers (100%), Lead (20%) |

1. **Bug Reporting**:
   * Steps to reproduce, expected vs. actual results, screenshots/logs.
   * Severity (Critical, Major, Minor) and Priority (High, Medium, Low).
2. **Risks and Issues**: Identify risks to testing (e.g., delayed environments, unavailable data).

**4. Logical Steps to Solve Any Testing Question**

Follow these structured steps for any logical or scenario-based testing problem:

**Step 1: Analyze the Scenario**

* Break down the problem into:
  + Inputs
  + Outputs
  + Constraints
  + System behavior

**Step 2: Choose Test Levels**

* Identify what to test: Unit, Integration, System, or Acceptance.
* Example: For a login feature, start with **Unit Testing** (function logic), then **Integration** (API response), and **System Testing** (full user flow).

**Step 3: Choose the Right Test Design Techniques**

* Use Black-Box Testing Techniques:
  1. **Equivalence Class Partitioning (ECP)**: Group inputs into valid/invalid partitions.
  2. **Boundary Value Analysis (BVA)**: Test edge cases for min/max boundaries.
  3. **Decision Table**: For scenarios with multiple input combinations.
  4. **State Transition**: For workflows with state changes.

**Step 4: Create Test Plan**

1. **Define Scope**: What to test, what not to test.
2. **Test Data**: Use real or synthetic data covering all test scenarios (Valid, Invalid, Boundary, etc.).
3. **Environment Setup**: Prepare test environments (QA, Dev, Staging).
4. **Schedule Testing Activities**: Break testing into phases.
5. **Assign Resources**: QA lead, testers, tools, and responsibilities.

**Step 5: Execute Tests**

* Run tests in phases:
  + **Sanity/Smoke**: Basic testing to ensure stability.
  + **Functional Tests**: Validate features meet requirements.
  + **Regression Tests**: Verify existing features are unaffected by changes.
* Log results for **Pass, Fail, Blocked, or In Progress** status.

**Step 6: Report Bugs**

* Create detailed bug reports:
  + Steps to reproduce.
  + Test data used.
  + Expected vs. actual results.
  + Severity and priority.

**Example Scenario: Solving a Logical Question**

**Scenario**: Test a new login feature where the user enters a username and password.

**Solution Steps:**

1. **Analyze Requirements**:
   * Username: 4-20 characters, alphanumeric.
   * Password: 6-12 characters, must include numbers.
2. **Test Levels**:
   * **Unit Testing**: Test validation logic for username and password.
   * **Integration Testing**: Verify login API integration.
   * **System Testing**: Test the entire login process.
3. **Test Techniques**:
   * **ECP**: Divide input classes (Valid, Invalid – empty, too short, too long).
   * **BVA**: Test edge cases for username (3, 4, 20, 21 characters).
   * **Decision Table**: Create combinations for valid/invalid username-password pairs.
4. **Test Data**:
   * **Valid**: user123, Pass123.
   * **Invalid**: "", abc, 12345.
5. **Environment Setup**: QA environment with a database of test users.
6. **Test Execution**:
   * Run functional tests for login.
   * Verify API responses and error messages.
7. **Bug Reporting**: Report failures with steps, expected results, and severity.

**Key Takeaways**

To solve any logical testing problem:

1. **Understand the Scenario**: Break it down logically.
2. **Use Shift Left**: Start testing early to reduce cost and time.
3. **Test Levels**: Select appropriate levels – Unit, Integration, System, Acceptance.
4. **Test Techniques**: Apply ECP, BVA, Decision Table, or State Transition Testing.
5. **Test Plan**: Define scope, strategy, schedule, and resources.
6. **Report Issues Clearly**: Use structured bug reporting to communicate defects.

**1. Test Monitoring and Test Control**

* **Test Monitoring**: Collect data to assess test progress and measure completion against goals (e.g., coverage, defect count).
* **Test Control**: Use monitoring results to **adjust testing** as needed:
  + Reprioritize tests based on risks.
  + Adjust schedules due to delays.
  + Allocate resources dynamically.

**2. Metrics – What to Measure and Why?**

Metrics help quantify testing activities, monitor progress, and ensure quality. Use these categories:

| **Metric Category** | **What It Measures** | **Examples** |
| --- | --- | --- |
| **Project Metrics** | Progress of tasks against schedule/budget. | % Test Cases Executed, Resource Utilization. |
| **Product Metrics** | Quality attributes of the product. | Defect Density, Mean Time to Failure (MTTF). |
| **Process Metrics** | Efficiency of the testing process. | % Defects Detected, Test Execution Time. |
| **People Metrics** | Tester performance and efficiency. | Test Cases Designed per Hour. |
| **Defect Metrics** | Defect discovery, resolution, and trends. | Number of Critical Defects, Defect Arrival Rate. |
| **Risk Metrics** | Risk coverage and mitigation. | % Risks Tested, Residual Risk. |
| **Coverage Metrics** | Degree to which testing covers requirements/code. | Requirements Coverage, Code Coverage. |

**Testing Quality Measuring & Control**

**3. Key Metrics to Solve Logical Questions**

**A. Progress Metrics**

* **Question**: How much testing is left, and when will it complete?
  + **Metrics**:
    - % Test Cases Executed.
    - Average Time per Test Case.
    - Test Execution Progress (Planned vs Actual).

**B. Quality Metrics**

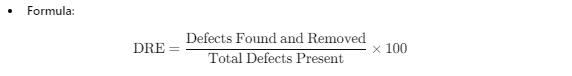
* **Question**: Should the software be deployed to production?
  + **Metrics**:
    - % Critical and High-Severity Defects.
    - % Test Coverage.
    - Defect Fix Rate vs Arrival Rate.

**C. Cost and Effort Metrics**

* **Question**: How can testing effort be estimated?
  + **Metrics**:
    - Number of Test Cases.
    - Average Test Execution Time.
    - Cost of Resources and Infrastructure.

**D. Defect Metrics**

* **Defect Density**: Number of defects per KLOC (Kilo Lines of Code).
* **Defect Arrival Rate**: Trend of defect detection over time.
* **Defect Containment**: Effectiveness of defect detection per phase.

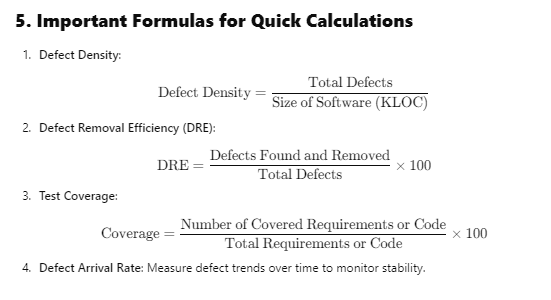
** E. Coverage Metrics**

* **Question**: Is testing thorough and complete?
  + **Metrics**:
    - Requirements Coverage.
    - Code Coverage (% lines executed).
    - Risk Coverage (tested vs untested risks).

**4. Practical Steps to Solve Logical Questions**

Follow this structured approach:

1. **Understand the Problem**:
   * Identify the key question (e.g., progress, quality, cost).
2. **Choose the Relevant Metrics**:
   * For **progress**: Use Execution Progress, % Completed Tests.
   * For **quality**: Use Defect Density, % Critical Defects, Coverage.
   * For **cost**: Estimate based on time, resources, and infrastructure.
3. **Apply Test Monitoring and Control**:
   * Track metrics continuously.
   * Adjust testing schedules, priorities, and resources dynamically.
4. **Calculate Key Metrics**:
   * Use formulas for defect density, DRE, and coverage.
   * Visualize progress using charts (e.g., defect trends, completion rates).
5. **Analyze Results and Take Action**:
   * Decide whether to continue testing, release the product, or optimize further.



**6. Key Takeaways**

* **Monitor and Control Testing**: Use **metrics** to guide decisions.
* **Focus on Relevant Metrics**: Pick the right metrics based on the logical question:
  + Progress → Execution Metrics.
  + Quality → Defect and Coverage Metrics.
  + Cost → Effort Estimation Metrics.
* **Defect Management**: Use Defect Density, Arrival Rate, and Containment Effectiveness.
* **Continuous Monitoring**: Track metrics over time for better test control.

**Crux to Solve Any Logical Question:**

1. Identify the **question**: Progress, Quality, or Cost?
2. Select relevant **metrics**: Execution Progress, Defect Rates, Test Coverage, etc.
3. Measure and track metrics using tools (e.g., JIRA, TestRail).
4. Apply **Test Monitoring and Control** to adjust testing priorities.
5. Use formulas like **Defect Density** and **DRE** for quick analysis.

When **Defect Removal Efficiency (DRE)** is **high**, it means that **a large percentage of defects have been detected and removed during the testing process**. This indicates **good testing effectiveness** and **better product quality**.

**White-Box Testing**: Test internal logic using techniques like **Statement Coverage**, **Branch Coverage**, and **Path Coverage**.

**Coverage Metrics**: Measure testing thoroughness to identify gaps.

**Systematic Approach**:

* Analyze code structure.
* Write tests to cover all conditions and paths.
* Measure and optimize coverage.

**Coverage Measurement**

Use coverage tools to identify untested parts of the code:

1. **Statement Coverage**: % of statements executed.
2. **Branch Coverage**: % of branches (if-else) executed.
3. **Function Coverage**: % of functions called/tested.
4. **Path Coverage**: % of logical paths executed.

**Cyclomatic complexity**

* V(G) = E - N + 2
  + E is the number of edges in Control Flow Graph
  + N is the number of nodes in the control flow graph
* V(G) = P + 1
  + P is the number of predicates nodes in control flow graph
* V(G) = Number of In-bound Regions + 1

**Static Testing** identifies defects **without executing code** through manual reviews and automated tools. It is cost-effective and detects early-stage issues in requirements, design, and code.

**Key Techniques**

1. **Reviews**:
   * **Informal Review**: Casual review.
   * **Walkthrough**: Author-led; builds understanding.
   * **Technical Review**: Experts evaluate technical correctness.
   * **Inspection**: Formal, structured, defect-focused.
2. **Static Analysis**:
   * **Control Flow Analysis**: Finds unreachable code, infinite loops.
   * **Data Flow Analysis**: Detects undefined/unused variables, memory leaks.
   * **Tools**: SonarQube, Checkstyle, ESLint, SAST tools (Semgrep, Coverity).

**Quality Models**

**2. ISO 25010 Model**

ISO 25010 evaluates software across **8 characteristics**, with sub-characteristics.

**Key Characteristics**

1. **Functional Suitability**:
   * **Completeness**: Does the software cover all tasks?  
     *Example*: A learning management system supporting grading and attendance.
   * **Correctness**: Does it produce precise results?  
     *Example*: A payment system deducting exact amounts.
   * **Appropriateness**: Is it useful for its intended tasks?  
     *Example*: Attendance tracking in a learning platform.
2. **Performance Efficiency**:
   * **Time Behavior**: Response/processing time meets expectations.  
     *Example*: <2s response for 95% of API calls.
   * **Resource Utilization**: Efficient resource consumption.  
     *Example*: Using minimal CPU and memory during operation.
   * **Capacity**: Supports the maximum expected load.  
     *Example*: 10,000 concurrent users with consistent performance.
3. **Compatibility**:
   * **Co-existence**: Runs alongside other systems without conflict.  
     *Example*: MySQL server and .NET framework on the same machine.
   * **Interoperability**: Can exchange and use information with other systems.  
     *Example*: Slack integrates with GitHub.
4. **Usability**:
   * **Learnability**: Easy for users to learn.  
     *Example*: A banking app with onboarding tutorials.
   * **Operability**: Easy to operate and control.  
     *Example*: Email client with intuitive settings.
   * **Accessibility**: Usable by people with disabilities.  
     *Example*: Alternative text for images and keyboard navigation.
5. **Reliability**:
   * **Maturity**: Works consistently without disruptions.  
     *Example*: Operating system running 48 hours without failure.
   * **Availability**: Accessible when required.  
     *Example*: A website accessible 98% of the time.
   * **Fault Tolerance**: Operates even with faults.  
     *Example*: System remains operational despite server failure.
   * **Recoverability**: Restores after failure.  
     *Example*: Recovering hospital data from backups.
6. **Security**:
   * **Confidentiality**: Protects sensitive data.  
     *Example*: Email encryption.
   * **Integrity**: Prevents unauthorized changes.  
     *Example*: Financial transactions with strict access control.
   * **Accountability**: Tracks user actions.  
     *Example*: Audit trails in healthcare systems.
   * **Authenticity**: Verifies identities.  
     *Example*: Secure voting systems.
7. **Maintainability**:
   * **Modularity**: Minimal impact of changes.  
     *Example*: CMS with separate modules for content types.
   * **Reusability**: Components reused in multiple systems.  
     *Example*: Libraries for input validation.
   * **Testability**: Easy to test.  
     *Example*: Apps with built-in testing frameworks.
8. **Portability**:
   * **Adaptability**: Works in different environments.  
     *Example*: Proxy design patterns for APIs.
   * **Installability**: Easy to install.  
     *Example*: Software with installation wizards.
   * **Replaceability**: Replaces old systems easily.  
     *Example*: Switching accounting software without disrupting operations.

**3. FURPS Model**

The **FURPS** model evaluates software based on:

* **Functionality**: Completeness, correctness, security.
* **Usability**: Ease of learning and use.
* **Reliability**: Handles failures, consistent behavior.
* **Performance**: Response times, resource usage.
* **Supportability**: Maintainability, testability.

**4. GQM and OKRs**

**Goal-Question-Metric (GQM)**:

1. Define **Goals** (e.g., improve API response time).
2. Formulate **Questions** to measure the goals (e.g., "Is response <2s?").
3. Define **Metrics** to validate questions (e.g., average response time).

**OKRs (Objectives and Key Results)**:

* **Objective**: Improve overall software quality.
* **Key Results**: Reduce bugs by 20%, maintain 95% test coverage.

**5. Capability Maturity Model (CMM)**

CMM focuses on improving software processes across 5 levels:

1. **Initial**: Ad hoc processes.
2. **Managed**: Basic project management.
3. **Defined**: Organization-level standardization.
4. **Quantitatively Managed**: Measured and controlled processes.
5. **Optimizing**: Continuous improvement through feedback.

**Steps to Solve Scenario-Based Questions**

1. **Understand the Scenario**: Identify the quality aspect (e.g., usability, reliability).
2. **Select the Appropriate Model**:
   * **ISO 25010**: For detailed quality analysis.
   * **FURPS**: For quick high-level evaluation.
   * **GQM**: For measurable goals and outcomes.
   * **CMM**: For process improvement.
3. **Map Scenario to Sub-Characteristics**:
   * Example: Poor UI → **Usability → Learnability, Operability**.
   * Example: Integration issues → **Compatibility → Interoperability**.
4. **Provide Recommendations**:
   * Use examples to explain practical solutions (e.g., adding error protection for usability).
   * Suggest improvements (e.g., modular design for maintainability).

**2. ISO 9001:2015 Standard**

ISO 9001:2015 is a globally recognized standard for QMS implementation.

**Key Principles:**

1. **Customer Focus**: Enhance customer satisfaction and loyalty.
2. **Leadership**: Establish vision and operational direction.
3. **Engagement of People**: Empower employees for better involvement.
4. **Process Approach**: Use the **PDCA (Plan-Do-Check-Act)** cycle for consistent results.
5. **Continuous Improvement**: Ongoing enhancement of processes.
6. **Evidence-Based Decision Making**: Use data to make informed decisions.
7. **Relationship Management**: Manage relationships with suppliers, partners, and stakeholders.

**Example**: A software team adopts ISO 9001 to improve processes, measure customer satisfaction, and promote continual improvement.

**3. Quality Maturity Models**

Maturity models assess the organization's ability to **continuously improve**.

**Key Models:**

1. **Capability Maturity Model (CMM)**:
   * **Levels**: Initial → Managed → Defined → Quantitatively Managed → Optimizing.
   * Helps evaluate process maturity and areas for improvement.
2. **Test Maturity Model**:
   * Evaluates software testing processes across maturity levels.
3. **Agile ISO Maturity Model**:
   * Aligns Agile processes with ISO standards.

**4. Continual Improvement**

Continual Improvement is central to QMS and involves:

* Identifying root causes of defects.
* Implementing corrective and preventive actions.
* Encouraging innovation to meet changing customer needs.

**Example**: A team identifies recurring bugs through defect analysis and optimizes the coding review process to prevent similar issues.

**5. Logical Steps to Solve Scenario-Based Questions**

**Step 1: Identify the Problem**

* Analyze the scenario to determine which quality aspect is in focus:
  + **Customer Satisfaction** → Customer Focus (ISO Principle).
  + **Defect Prevention** → Continuous Improvement.
  + **Process Standardization** → QMS Implementation.

**1. Reliability in Software Systems**

* **Definition**: The degree to which a system performs its intended function under specific conditions for a specified period.
* **Key Attributes** (ISO 25010):
  + **Maturity**: Stability under normal operations (e.g., Mean Time Between Failures - MTBF).
  + **Availability**: Proportion of operational time (e.g., 99.99% uptime).
  + **Fault Tolerance**: System operates even with faults.
  + **Recoverability**: Ability to restore the system after a failure.

**Metrics**:

* **MTBF**: Mean time between system failures.
* **MTTR**: Mean time to repair a failure.
* **Failure Rate**: Number of failures per unit of time.
* **Availability**: A=MTBFMTBF+MTTRA = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}A=MTBF+MTTRMTBF​

**2. Fault Tolerance**

Fault tolerance ensures a system continues functioning despite **faults** in hardware or software.

**Key Fault-Tolerant Strategies:**

1. **Redundancy**: Duplicate critical components or systems to ensure backup.
   * **Component Redundancy**: E.g., multiple servers.
   * **Data Redundancy**: Backup copies of data.
2. **Failover Mechanisms**: Switch to backup systems when the primary system fails.
3. **Recovery Blocks**: Use a sequence of alternative blocks if the primary block fails acceptance tests.
4. **Checkpoints**: Save states periodically to enable restoration after failure.

**Example**: A server cluster with failover ensures continued operation even if one server crashes.

**3. Recoverability**

* **Definition**: Ability of the system to recover from a fault or failure and restore service.
* **Key Mechanisms**:
  + **Failover Capability**: Switch to backup servers automatically.
  + **Backup and Restore**: Periodic backups to restore lost data.
  + **Simulation of Failures**: Test systems for recovery using failure scenarios.

**4. Reliability Growth Models**

* Used to predict and improve reliability over time.
* **Key Models**:
  + **Static Models**: No expected changes (e.g., legacy systems).
  + **Basic Growth Models**: Failures decrease over time as software matures.
  + **Logarithmic Poisson Models**: Reliability increases exponentially with fixes.

**5. Fault-Tolerant Design Patterns**

**Key Patterns:**

1. **Redundancy**: Duplicate systems to provide backups.
2. **Circuit Breaker**: Stop repeated operations when a failure is detected.
3. **Retries and Timeouts**: Automatically retry failed operations.
4. **Rollback Mechanisms**: Restore the system to a previous stable state.
5. **Health Checks (Heartbeat)**: Monitor system health for proactive recovery.
6. **Minimize Human Intervention**: Automatic fault detection and recovery.
7. **Escalation**: Alert human operators when automated recovery is not possible.

**6. Logical Steps to Solve Scenario-Based Questions**

**Step 1: Identify the Problem**

* Understand if the scenario focuses on **reliability**, **fault tolerance**, or **recoverability**.

**Step 2: Select Key Metrics and Strategies**

* **Reliability**: Focus on MTBF, MTTR, Availability, and Failure Rates.
* **Fault Tolerance**: Apply redundancy, failover mechanisms, and recovery blocks.
* **Recoverability**: Implement backup, restore, and checkpoint mechanisms.

**Step 3: Use Design Patterns**

* Choose suitable **fault-tolerant patterns** based on the scenario:
  + **Failover** for system crashes.
  + **Circuit Breaker** for repetitive failures.
  + **Rollback** for data inconsistencies.

**Performance Characteristics**

Performance is evaluated through:

* **Time Behavior**: Response and processing time.
* **Resource Utilization**: CPU, memory, and bandwidth usage.
* **Capacity**: Maximum system load handling.

**2. Principles of Performance Testing**

1. Align tests with stakeholder expectations.
2. Ensure tests are reproducible and statistically identical.
3. Test in environments close to production systems.
4. Results should be understandable and comparable.

**3. Types of Performance Testing**

| **Type** | **Purpose** |
| --- | --- |
| **Load Testing** | Check system performance under expected user loads. |
| **Stress Testing** | Test behavior under extreme load (beyond limits). |
| **Spike Testing** | Assess system response to sudden traffic spikes. |
| **Endurance Testing** | Measure system stability over a long period (e.g., 24-hour load). |
| **Scalability Testing** | Check how well the system scales as load increases. |
| **Capacity Testing** | Find maximum load the system can handle without failing. |
| **Concurrency Testing** | Evaluate system performance with multiple users simultaneously accessing it. |

**4. Performance Failures**

Common issues include:

1. **Slow Response Times**: Delays under all or heavy load conditions.
2. **Crashes**: System failure under sudden spikes.
3. **Degraded Performance Over Time**: System slows during prolonged use.
4. **Inefficient Resource Usage**: Excessive CPU, memory, or bandwidth consumption.
5. **Error Handling Issues**: System fails to handle errors gracefully under load.

**5. Common Performance Metrics**

**Technical Metrics:**

* **Response Time**: Time taken to process a request.
* **Throughput**: Number of transactions per second.
* **Resource Utilization**: CPU, memory, disk usage, and network bandwidth.
* **Latency**: Delays in network communication.

**6. Tools for Performance Testing**

* **Load Generation Tools**: JMeter, K6, Gatling.
* **Monitoring Tools**: Datadog, New Relic, Prometheus.

**7. Steps to Solve Scenario-Based Questions**

1. **Understand the Problem**:
   * Identify the performance concern (e.g., slow response, crash, latency).
   * Determine the type of test required (Load, Stress, Spike, etc.).
2. **Choose the Appropriate Metrics**:
   * Use **response time**, **throughput**, or **resource utilization** for analysis.
3. **Identify Tools**:
   * Use **JMeter** or **K6** for testing, and monitoring tools for system health.
4. **Design Load Profiles**:
   * Simulate user load based on expected traffic patterns.
5. **Execute Tests and Analyze Results**:
   * Compare test results against defined performance benchmarks.
6. **Provide Solutions**:
   * Apply **Performance Design Patterns**:
     + **Caching**: Store frequently accessed data temporarily.
     + **Batching**: Combine multiple requests for efficiency.
     + **Lazy Loading**: Load critical data first, delay non-essential parts.
     + **Failover Mechanisms**: Manage failures gracefully.

**Performance Testing**

**Directory Structure**

Ensure your file structure looks like this:

bash

Copy code

project-folder/

│-- script.js # k6 performance script

│-- users.csv # CSV file with test data

**1. Basic Structure of k6 Script**

javascript

Copy code

import http from 'k6/http'; // For HTTP requests

import { check, sleep } from 'k6'; // Check results, delays

import { Trend, Rate, Counter } from 'k6/metrics'; // Custom metrics

// Load options: Test configuration

export let options = {

vus: 10, // Virtual users

duration: '1m', // Total test duration

thresholds: {

http\_req\_duration: ['p(95)<500'], // 95% requests < 500ms

http\_req\_failed: ['rate<0.01'], // < 1% request failure rate

},

};

export default function () {

const url = 'https://example.com/api';

const params = { headers: { 'Content-Type': 'application/json' } };

// Send POST request

let response = http.post(url, JSON.stringify({ key: 'value' }), params);

// Perform checks

check(response, {

'status is 200': (r) => r.status === 200,

'response time < 500ms': (r) => r.timings.duration < 500,

});

sleep(1); // Wait for 1 second between iterations

}

**2. Load Testing Options**

**A. Basic Test Configuration**

javascript

Copy code

export let options = {

vus: 10, // Number of virtual users

duration: '1m', // Duration of the test

};

**B. Ramp-Up and Ramp-Down (Stages)**

Stages simulate traffic gradually increasing or decreasing.

javascript

Copy code

export let options = {

stages: [

{ duration: '30s', target: 20 }, // Ramp up to 20 VUs in 30 seconds

{ duration: '1m', target: 50 }, // Hold steady at 50 VUs for 1 minute

{ duration: '30s', target: 0 }, // Ramp down to 0 VUs

],

};

**3. HTTP Requests**

**GET Request**

javascript

Copy code

let response = http.get('https://example.com/api');

**POST Request**

javascript

Copy code

let payload = JSON.stringify({ key: 'value' });

let params = { headers: { 'Content-Type': 'application/json' } };

let response = http.post('https://example.com/api', payload, params);

**PUT Request**

javascript

Copy code

let payload = JSON.stringify({ key: 'updatedValue' });

let response = http.put('https://example.com/api/1', payload, params);

**DELETE Request**

javascript

Copy code

let response = http.del('https://example.com/api/1');

**4. Checks and Validations**

**Basic Check**

javascript

Copy code

check(response, {

'status is 200': (r) => r.status === 200,

});

**Multiple Checks**

javascript

Copy code

check(response, {

'status is 200': (r) => r.status === 200,

'response time < 500ms': (r) => r.timings.duration < 500,

'response body contains key': (r) => JSON.parse(r.body).key !== undefined,

});

**5. Thresholds**

Thresholds define pass/fail conditions for performance metrics.

javascript

Copy code

export let options = {

thresholds: {

http\_req\_duration: ['p(95)<500'], // 95% of requests must complete under 500ms

http\_req\_failed: ['rate<0.01'], // < 1% failure rate

},

};

**6. Data-Driven Testing**

**Using CSV Files**

javascript

Copy code

import { SharedArray } from 'k6/data';

import http from 'k6/http';

import { check } from 'k6';

const data = new SharedArray('userData', function () {

return open('./users.csv').split('\n').map(line => {

const [username, password] = line.split(',');

return { username, password };

});

});

export default function () {

const user = data[Math.floor(Math.random() \* data.length)];

let payload = JSON.stringify({ username: user.username, password: user.password });

let response = http.post('https://example.com/login', payload, { headers: { 'Content-Type': 'application/json' } });

check(response, { 'status is 200': (r) => r.status === 200 });

}

**7. Scenarios**

Scenarios allow you to simulate multiple test configurations.

javascript

Copy code

export let options = {

scenarios: {

constant\_load: {

executor: 'constant-vus',

vus: 10,

duration: '1m',

},

ramping\_load: {

executor: 'ramping-vus',

startVUs: 0,

stages: [

{ duration: '1m', target: 50 },

{ duration: '1m', target: 0 },

],

},

},

};

**8. Custom Metrics**

Track custom performance metrics.

javascript

Copy code

import { Trend, Counter } from 'k6/metrics';

let durationTrend = new Trend('http\_req\_duration');

let failureCount = new Counter('failed\_requests');

export default function () {

let response = http.get('https://example.com/api');

durationTrend.add(response.timings.duration);

if (response.status !== 200) {

failureCount.add(1);

}

}