# **Software Testing - Detailed Notes**

## ****Unit I: Quality Concepts****

### ****1. Quality****

* Refers to the degree to which a product, service, or process meets customer or user expectations and requirements.
* Focuses on delivering defect-free products and ensuring customer satisfaction.
* Measured by factors like performance, reliability, usability, and efficiency.
* **Example:** A smartphone with smooth performance, no crashes, and a user-friendly interface represents high quality.

### ****2. Software Quality****

* Involves ensuring that software products meet specified requirements and are free of defects.
* Includes functional quality (correctness, completeness) and non-functional quality (usability, maintainability, security).
* Standards like ISO/IEC 25010 define various quality attributes.
* **Example:** A mobile banking app should correctly process transactions (functional quality) and offer a smooth interface (non-functional quality).

### ****3. Software Quality Factors****

* **Product Operation Factors:** Reliability, usability, efficiency, integrity, and correctness.
* **Product Revision Factors:** Maintainability, flexibility, and testability.
* **Product Transition Factors:** Portability, reusability, and interoperability.
* **Example:** A web application should be reliable (operate smoothly), maintainable (easy to update), and portable (run on different browsers).

### ****4. The Cost of Quality****

* **Prevention Costs:** Expenses for activities to prevent defects, such as training and process improvement.
* **Appraisal Costs:** Costs of testing and inspection to identify defects early.
* **Failure Costs:**
  + **Internal Failure:** Costs of defects found before release, like rework and bug fixing.
  + **External Failure:** Costs of defects found after release, such as customer support and patches.
* **Key Insight:** Investing in prevention significantly reduces overall costs.
* **Example:** Automated testing during development catches bugs early, saving the cost of post-release patches.

### ****5. Quality and Security****

* **Security as a Quality Attribute:** Ensures that software resists threats and unauthorized access.
* Focuses on data confidentiality, integrity, and availability.
* Involves security testing methods like penetration testing and vulnerability scanning.
* **Example:** An e-commerce site must protect customer data with SSL/TLS encryption.

### ****6. Quality Control****

* Involves identifying defects in the final product through testing and reviews.
* A reactive approach focused on defect detection and correction.
* **Example:** Conducting manual and automated tests to find bugs in a software application.

### ****7. Software Quality Assurance (SQA)****

* A proactive process ensuring quality throughout the software development lifecycle.
* Involves standards, guidelines, process audits, and preventive actions.
* **Example:** SQA activities include code reviews, test planning, and compliance audits.

### ****8. Software Reviews****

* Systematic examination of software by one or more individuals to find defects.
* **Types:** Peer reviews, walkthroughs, and inspections.
* **Example:** Peer reviewing code for adherence to coding standards.

### ****9. Formal Technical Reviews (FTRs)****

* Structured reviews conducted by a team to detect defects early in the development process.
* Involves roles like moderator, presenter, and reviewers.
* **Example:** An FTR conducted for a module before integration.

### ****10. Software Reliability****

* The probability of software operating without failure under specified conditions.
* Measured by metrics like Mean Time Between Failures (MTBF) and failure rate.
* **Techniques:**
  + **Fault Avoidance:** Preventing defects during development.
  + **Fault Tolerance:** Ensuring functionality despite defects.
* **Example:** Redundant systems in medical software to ensure reliability.

### ****11. The SQA Plan****

* **Purpose:** To define standards, procedures, and tools for ensuring software quality.
* **Key Components:**
  + **Management:** Roles and responsibilities for quality activities.
  + **Documentation:** Standards for documents and records.
  + **Standards and Practices:** Guidelines for coding, testing, and reviews.
  + **Testing Requirements:** Types of tests (unit, integration, system) and schedules.
* **Example:** An SQA plan outlining compliance requirements for a healthcare app.

**Summary:**

* Effective SQA and quality control practices reduce defects and enhance reliability.
* Understanding and managing software quality is crucial for successful software projects.

**Unit II: SOFTWARE TESTING STRATEGIES**

**1. A Strategic Approach to Software Testing**

* **Definition:** A systematic process to ensure that software meets specified requirements and is free of defects.
* **Key Principles:**
  + Testing begins at the module level and progresses to the system level.
  + Emphasis on early defect detection to reduce costs.
  + Uses a mix of manual and automated testing techniques.
* **Phases of Testing Strategy:**
  + **Planning:** Define objectives, scope, and resources.
  + **Design:** Create test cases and scenarios based on requirements.
  + **Execution:** Run tests, record results, and identify defects.
  + **Evaluation:** Assess results against acceptance criteria.
* **Example:** A mobile app development project where testing starts with individual features (unit testing) and progresses to overall app testing (system testing).

**2. Unit Testing**

* **Definition:** Testing individual components or modules of software to verify they function as intended.
* **Performed by:** Typically conducted by developers during the coding phase.
* **Tools Used:** JUnit for Java, pytest for Python, NUnit for .NET.
* **Key Focus:**
  + Testing functions, methods, or classes in isolation.
  + Ensuring inputs produce expected outputs.
* **Example:**
  + **Scenario:** Testing a function that calculates the total price of items in a shopping cart.
  + **Test Case:** Provide different item quantities and prices, and verify the correct total is calculated.

**3. Integration Testing**

* **Definition:** Testing combined parts of an application to ensure they work together as intended.
* **Approaches:**
  + **Top-Down:** Testing starts from the topmost module and progresses downward.
  + **Bottom-Up:** Testing starts from the lowest level modules and moves upward.
  + **Sandwich:** A mix of top-down and bottom-up approaches.
* **Key Objectives:**
  + Identify interface defects between modules.
  + Validate data flow and control flow between integrated components.
* **Example:**
  + **Scenario:** Testing the integration between a login module and a dashboard module.
  + **Test Case:** Verify if a successful login redirects to the correct dashboard.

**4. Validation Testing**

* **Definition:** Ensures that the software meets the user requirements and performs as expected in a real-world scenario.
* **Methods Used:**
  + **Alpha Testing:** Conducted in a controlled environment by internal teams.
  + **Beta Testing:** Conducted by end-users in a real-world environment.
* **Key Focus:**
  + Functional validation to ensure features work as specified.
  + Non-functional validation for performance, security, and usability.
* **Example:**
  + **Scenario:** Testing a banking app’s ability to handle transactions.
  + **Test Case:** Check if a user can transfer funds successfully and securely.

**5. System Testing**

* **Definition:** Testing the complete and integrated software system to verify it meets requirements.
* **Key Aspects:**
  + **Functional Testing:** Validates all functionalities as per requirements.
  + **Non-Functional Testing:** Tests performance, security, and reliability.
* **Types of System Testing:**
  + **Performance Testing:** Ensures the system performs under expected load.
  + **Security Testing:** Identifies vulnerabilities to prevent data breaches.
  + **Usability Testing:** Assesses user-friendliness and navigation.
* **Example:**
  + **Scenario:** Testing an e-commerce website.
  + **Test Case:** Verify if users can search, add to cart, and complete payments smoothly.

**6. The Art of Debugging**

* **Definition:** The process of identifying, analyzing, and fixing defects or bugs in software.
* **Key Techniques:**
  + **Brute Force:** Examining all possible areas of code manually.
  + **Backtracking:** Reviewing code step-by-step from the point of failure.
  + **Cause Elimination:** Using hypothesis and testing to isolate issues.
* **Tools Used:** Debuggers in IDEs like Eclipse, Visual Studio, or GDB for C/C++.
* **Key Principles:**
  + Reproduce the bug consistently to understand its cause.
  + Analyze error messages and logs for clues.
  + Fix the root cause, not just the symptoms.
* **Example:**
  + **Scenario:** Fixing a null pointer exception in a Java program.
  + **Approach:** Identify the line causing the issue, inspect object initialization, and add null checks.

**Summary of Unit II:**

* A strategic approach to testing involves multiple levels: unit, integration, validation, and system testing.
* Effective debugging is crucial to identify and resolve defects efficiently.
* Using a mix of testing strategies ensures comprehensive coverage of both functional and non-functional requirements.

**Unit III: TESTING Tactics**

**1. Software Testing Fundamentals**

* **Definition:** Ensures that software behaves as expected and meets requirements by identifying and fixing defects.
* **Key Principles:**
  + Testing shows the presence of defects, not their absence.
  + Early testing saves time and cost.
  + Exhaustive testing is impossible; risk-based testing is essential.
* **Types of Testing:**
  + **Static Testing:** Involves code reviews and walkthroughs without executing the program.
  + **Dynamic Testing:** Involves executing code with test cases.
* **Example:**
  + **Static Testing:** Reviewing a piece of code for compliance with coding standards.
  + **Dynamic Testing:** Running test cases on a login module to check if correct and incorrect credentials work as expected.

**2. Black Box Testing**

* **Definition:** Focuses on testing software functionality without knowing internal code or structure.
* **Techniques:**
  1. **Equivalence Partitioning:** Divides input data into valid and invalid partitions to reduce test cases.
     + **Example:** For an age input field (1–100), test cases for 0 (invalid), 50 (valid), and 101 (invalid).
  2. **Boundary Value Analysis:** Tests at the boundaries of input ranges.
     + **Example:** For an age field (1–100), test cases for 0, 1, 100, and 101.
  3. **Decision Table Testing:** Uses tables to represent combinations of inputs and their expected outcomes.
     + **Example:** Testing a loan approval system with conditions like credit score and income.
  4. **State Transition Testing:** Focuses on testing different states of an application based on events.
     + **Example:** Testing an ATM's state transitions for card insertion, PIN entry, and cash withdrawal.
* **Advantages:** Detects missing functions and user interface issues.
* **Disadvantages:** Cannot uncover internal errors in the code.

**3. White-Box Testing**

* **Definition:** Focuses on testing internal logic, code paths, and structure of the software.
* **Techniques:**
  1. **Statement Coverage:** Ensures every line of code is executed at least once.
     + **Example:** Testing all if-else branches in a calculator program.
  2. **Branch Coverage:** Ensures all branches (true/false) of decision points are executed.
     + **Example:** Testing both paths of an if-else condition.
  3. **Condition Coverage:** Focuses on testing all logical conditions independently.
     + **Example:** For a condition (A && B), test cases for A=true, B=false, and vice versa.
  4. **Path Coverage:** Ensures all possible paths through a program are tested.
     + **Example:** Testing all execution paths in a function with multiple loops and conditions.
* **Advantages:** Helps find hidden errors and optimize code.
* **Disadvantages:** Not effective for large and complex systems due to the high number of possible paths.

**4. Basic Path Testing**

* **Definition:** A type of white-box testing that focuses on executing all possible paths in a program at least once.
* **Key Concepts:**
  + **Control Flow Graph (CFG):** Represents the flow of control in the program with nodes (statements) and edges (branches).
  + **Cyclomatic Complexity:** Measures the complexity of a program using the formula: V(G)=E−N+2PV(G) = E - N + 2PV(G)=E−N+2P Where:
    - EEE = Number of edges
    - NNN = Number of nodes
    - PPP = Number of disconnected parts
  + **Basis Path Testing:** Ensures execution of all independent paths derived from cyclomatic complexity.
* **Example:**
  + **Scenario:** Testing a function with an if-else and a loop.
  + **Cyclomatic Complexity:** If there are 5 edges and 4 nodes, the complexity is 5−4+2=35 - 4 + 2 = 35−4+2=3.
  + **Test Cases:** Create three independent paths to ensure all scenarios are tested.

**5. Control Structural Testing**

* **Definition:** Focuses on testing control structures in code like loops, conditions, and branches.
* **Key Techniques:**
  1. **Condition Testing:** Tests individual conditions in decision statements.
     + **Example:** Testing if (A > B || C < D) works correctly for all combinations of A, B, C, and D.
  2. **Loop Testing:** Focuses on different types of loops—simple, nested, concatenated.
     + **Types of Tests:**
       - **Zero Iterations:** Ensures loop handles no execution scenario.
       - **One Iteration:** Tests loop execution exactly once.
       - **Multiple Iterations:** Tests typical execution scenarios.
       - **Boundary Conditions:** Tests upper and lower limits.
     + **Example:** Testing a for-loop iterating over an array for zero, one, and multiple elements.
  3. **Data Flow Testing:** Focuses on variable usage to identify anomalies like uninitialized variables.
     + **Example:** Checking if all variables in a function are properly initialized before use.
* **Benefits:** Enhances test coverage by focusing on different control structures.
* **Drawbacks:** Requires understanding of internal code structure.

### ****Unit IV: O-O Testing Methods****

### ****1. Object-Oriented (O-O) Testing Methods****

* **Definition:** Focuses on testing object-oriented software by considering classes, objects, inheritance, polymorphism, and other O-O concepts.
* **Challenges:**
  + Complex interactions between objects.
  + Inheritance and polymorphism introduce hidden dependencies.
  + Requires testing both state (attributes) and behavior (methods) of objects.
* **Approach:**
  + **Class Testing:** Testing individual classes as standalone units.
  + **Integration Testing:** Ensuring interactions between classes function correctly.
  + **System Testing:** Testing the complete O-O system for functional and non-functional requirements.
* **Key Insight:** Testing should focus on objects' states and transitions, ensuring that methods produce correct outcomes for all possible states.
* **Example:** Testing a BankAccount class for methods like deposit(), withdraw(), and getBalance().

### ****2. Testing Methods Applicable on the Class Level****

* **Goal:** Ensure individual classes perform correctly in isolation by verifying methods, states, and data encapsulation.

#### **a. Method Testing**

* Focuses on testing individual methods within a class for correctness.
* **Key Techniques:**
  + **Black Box Testing:** Tests method inputs and outputs without considering internal code.
  + **White Box Testing:** Tests internal logic of methods using techniques like branch and path testing.
* **Example:** Testing a calculateInterest() method for correct interest calculation.

#### **b. Attribute Testing**

* Ensures that class attributes (instance variables) are correctly initialized, updated, and accessed.
* **Example:** Testing if a balance attribute in BankAccount is correctly updated after each transaction.

#### **c. State-Based Testing**

* Focuses on testing object states and transitions based on state diagrams.
* **Key Techniques:**
  + **State Transition Testing:** Creates test cases for every valid and invalid state transition.
  + **Guard Condition Testing:** Ensures conditions for state transitions are correctly evaluated.
* **Example:** Testing a User class with states like Active, Suspended, and Deactivated.

#### **d. Lifecycle Testing**

* Tests the entire lifecycle of an object, from creation to destruction.
* **Phases:**
  1. **Instantiation:** Testing constructor methods.
  2. **Operation:** Testing normal method usage.
  3. **Destruction:** Ensuring resources are properly released.
* **Example:** Testing a FileHandler class for proper file opening, reading, and closing.

### ****3. Inter-Class Test Case Design****

* **Goal:** Ensure that interactions between different classes work correctly.
* **Key Techniques:**
  1. **Collaboration Testing:** Focuses on method calls between objects and their responses.
     + **Example:** Testing if a Payment class correctly interacts with a BankAccount class.
  2. **Integration Testing Strategies:**
     + **Top-Down:** Starts testing from higher-level classes down to lower-level ones.
     + **Bottom-Up:** Begins with testing lower-level classes and moves upward.
     + **Sandwich Approach:** Combines both top-down and bottom-up approaches.
  3. **Message Passing Testing:** Ensures objects can correctly send and receive messages.
     + **Example:** Testing if a Cart object can correctly call addItem() on a Product object.
* **Challenges:**
  1. Complex inter-object communications.
  2. Inheritance and polymorphism introduce additional test cases.

### ****4. Testing for Specialized Environments, Architectures, and Applications****

#### **a. Client-Server and Web Applications**

* **Focus:**
  + Testing request-response mechanisms.
  + Ensuring data integrity between client and server.
* **Techniques:**
  + **Load Testing:** Measures performance under heavy traffic.
  + **Security Testing:** Focuses on authentication, authorization, and data protection.
  + **Compatibility Testing:** Ensures compatibility with different browsers and devices.
* **Example:** Testing a shopping cart system for high traffic during a sale.

#### **b. Distributed and Cloud Applications**

* **Focus:**
  + Testing data synchronization across nodes.
  + Ensuring fault tolerance and scalability.
* **Techniques:**
  + **Latency Testing:** Measures delays in data transmission.
  + **Resilience Testing:** Tests recovery from network failures.
* **Example:** Testing a cloud-based file storage system for data consistency.

#### **c. Mobile Applications**

* **Focus:**
  + Testing on different devices, screen sizes, and operating systems.
* **Techniques:**
  + **Usability Testing:** Ensures intuitive user interfaces.
  + **Battery Consumption Testing:** Measures app’s impact on battery life.
  + **Network Testing:** Ensures app works offline and handles slow networks gracefully.
* **Example:** Testing a fitness app for accuracy of GPS tracking and offline capabilities.

#### **d. Real-Time Systems**

* **Focus:**
  + Ensuring tasks complete within specified time limits.
* **Techniques:**
  + **Response Time Testing:** Measures time taken for a system to respond.
  + **Concurrency Testing:** Ensures proper handling of multiple tasks simultaneously.
* **Example:** Testing an automated traffic control system for timely signal changes.

### ****5. Testing Patterns****

* **Definition:** Reusable solutions to common testing challenges in O-O systems.
* **Key Testing Patterns:**
  1. **Factory Testing Pattern:** Creates mock objects for testing classes in isolation.
     + **Example:** Using mock Database objects to test a UserService class.
  2. **Observer Testing Pattern:** Focuses on testing observer-subject relationships in event-driven systems.
     + **Example:** Testing if a StockObserver is notified when stock prices change.
  3. **State-Based Testing Pattern:** Uses state diagrams to derive test cases.
     + **Example:** Testing a VendingMachine class based on its state transitions.
  4. **Behavioral Testing Pattern:** Focuses on testing object behavior through interfaces.
     + **Example:** Testing if a PaymentProcessor correctly processes payments through different methods (credit, debit, etc.).
* **Benefits:**
  1. Promotes reuse of test cases and strategies.
  2. Simplifies testing complex O-O systems.
* **Challenges:** Requires understanding of design patterns and their testing implications

### ****Unit V: Testing Concepts for WebApps****

### ****1. The Testing Process – An Overview****

* **Definition:** A systematic approach to identify defects in web applications to ensure they function correctly, securely, and efficiently.
* **Goals:**
  + Ensure functionality meets requirements.
  + Validate performance under different conditions.
  + Identify and fix security vulnerabilities.
* **Phases in Web Application Testing:**
  + **Requirement Analysis:** Understanding what the web app should do.
  + **Test Planning:** Creating a strategy for testing, including types of tests and tools.
  + **Test Case Design:** Writing specific scenarios to test different parts of the web app.
  + **Test Environment Setup:** Configuring servers, databases, and networks to mirror production.
  + **Test Execution:** Running test cases and logging defects.
  + **Test Reporting:** Documenting test results and suggesting improvements.
  + **Test Closure:** Analyzing test coverage and preparing a closure report.

#### **Key Challenges:**

* Compatibility across different browsers and devices.
* Handling high traffic and complex user interactions.
* Ensuring security and data privacy.

#### **Key Techniques:**

* **Automated Testing:** Using tools like Selenium, JMeter, and Cypress for efficiency.
* **Manual Testing:** For exploratory and usability testing.
* **Regression Testing:** Ensures new changes don’t break existing functionality.

### ****2. Content Testing****

* **Definition:** Ensures that text, images, videos, and other content in the web app are accurate, appropriate, and accessible.
* **Key Objectives:**
  + Verify correctness and relevancy of information.
  + Ensure consistent formatting and style.
  + Check for broken links and missing media.
  + Ensure SEO compliance (meta tags, keywords).
* **Testing Techniques:**
  + **Spelling and Grammar Check:** Automated tools like Grammarly or manual proofreading.
  + **Link Testing:** Ensures all internal and external links work properly.
  + **Multimedia Testing:** Verifies that images, videos, and animations load correctly.
  + **Localization Testing:** Ensures content is culturally appropriate for different regions.
* **Tools:** Screaming Frog, Grammarly, Broken Link Checker.
* **Example:** Testing a blog page to ensure all articles have correct grammar, working links, and appropriate tags.

### ****3. User Interface (UI) Testing****

* **Definition:** Ensures that the web app's interface is user-friendly, consistent, and visually appealing.
* **Key Objectives:**
  + Test layout, color schemes, fonts, and icons.
  + Ensure interactive elements (buttons, forms) work correctly.
  + Verify responsiveness on different devices and screen sizes.
* **Testing Techniques:**
  + **Visual Testing:** Compares UI against design mockups.
  + **Responsive Testing:** Checks layout and elements on various devices (mobile, tablet, desktop).
  + **Cross-Browser Testing:** Ensures compatibility with Chrome, Firefox, Safari, Edge, etc.
  + **Accessibility Testing:** Checks compliance with standards like WCAG (Web Content Accessibility Guidelines).
* **Tools:** Selenium, BrowserStack, LambdaTest, Axe.
* **Example:** Testing an e-commerce site to ensure the checkout button is visible and clickable on mobile devices.

#### **Key Challenges in UI Testing:**

* Handling dynamic and personalized content.
* Ensuring consistency across different platforms.
* Managing frequent UI changes.

### ****4. Navigation Testing****

* **Definition:** Ensures that users can navigate through the web app seamlessly and intuitively.
* **Key Objectives:**
  + Verify that menus, links, and navigation bars work correctly.
  + Ensure breadcrumb trails and site maps are accurate.
  + Test navigation flow for all possible paths.
* **Testing Techniques:**
  + **Link Testing:** Ensures all navigation links redirect to the correct pages.
  + **Breadcrumb Testing:** Verifies the accuracy of hierarchical paths.
  + **Menu Testing:** Ensures dropdowns and collapsible menus function correctly.
  + **Redirection Testing:** Checks for proper redirection of outdated or removed URLs.
* **Tools:** Screaming Frog, Selenium, Katalon Studio.
* **Example:** Testing an online learning platform to ensure users can easily navigate between courses, dashboards, and profiles.

#### **Key Challenges in Navigation Testing:**

* Managing deep and complex navigation hierarchies.
* Handling user-specific navigation paths (authenticated vs. guest users).
* Ensuring proper handling of broken or redirected links.

### ****5. Security Testing****

* **Definition:** Identifies vulnerabilities and ensures the web app protects data and prevents unauthorized access.
* **Key Objectives:**
  + Detect and fix security vulnerabilities (SQL injection, XSS).
  + Ensure secure data transmission using HTTPS.
  + Verify user authentication and authorization mechanisms.
* **Testing Techniques:**
  + **Vulnerability Scanning:** Uses automated tools to find security loopholes.
  + **Penetration Testing:** Ethical hacking to identify potential exploits.
  + **Authentication Testing:** Verifies login mechanisms, password policies, and session management.
  + **Authorization Testing:** Ensures users can access only permitted resources.
  + **Cookie Testing:** Checks for secure and HttpOnly flags in cookies.
* **Tools:** OWASP ZAP, Burp Suite, Nessus, Acunetix.
* **Example:** Testing an online banking application for SQL injection vulnerabilities and secure login mechanisms.

#### **Common Security Threats:**

* **SQL Injection:** Malicious SQL code execution through input fields.
* **Cross-Site Scripting (XSS):** Injecting scripts to steal session cookies or user data.
* **Cross-Site Request Forgery (CSRF):** Forcing a logged-in user to perform unwanted actions.
* **Man-in-the-Middle (MITM) Attacks:** Intercepting and altering communication between users and the server.

#### **Best Practices for Security Testing:**

* Implement HTTPS and secure headers.
* Validate and sanitize user inputs.
* Use security testing tools regularly.

### ****Summary of Key Testing Types for WebApps****

| **Testing Type** | **Focus** | **Tools** |
| --- | --- | --- |
| **Content Testing** | Accuracy, links, multimedia | Screaming Frog, Grammarly |
| **UI Testing** | Layout, consistency, responsiveness | Selenium, BrowserStack |
| **Navigation Testing** | Link accuracy, flow, menus | Screaming Frog, Selenium |
| **Security Testing** | Vulnerabilities, authentication | OWASP ZAP, Burp Suite |

**Unit VI: PRODUCT METRICS**

**1. A Framework for Product Metrics**

* **Definition:**  
  Product metrics are quantitative measures used to assess the quality, complexity, and efficiency of a software product. They help in identifying potential issues, optimizing processes, and ensuring that the final product meets the desired standards.
* **Purpose of Product Metrics:**
  + Evaluate software quality and performance.
  + Assist in risk assessment and management.
  + Support decision-making throughout the software lifecycle.
  + Facilitate process improvement and adherence to standards.
* **Types of Product Metrics:**
  + **Internal Metrics:** Measure internal attributes like code complexity and maintainability.
  + **External Metrics:** Assess the product’s behavior in a live environment (e.g., reliability and usability).
  + **Hybrid Metrics:** Combine internal and external measures.
* **Key Characteristics of Effective Metrics (SMART):**
  + **S**pecific: Measures a specific attribute.
  + **M**easurable: Quantifiable and objective.
  + **A**ctionable: Provides insights for improvement.
  + **R**elevant: Related to project goals.
  + **T**imely: Available when needed for decision-making.

**2. Metrics for the Requirements Model**

* **Purpose:** Evaluate the completeness, clarity, and quality of requirements to minimize misunderstandings and rework in later stages.
* **Key Metrics:**
  1. **Requirements Stability Index (RSI):**  
     Measures how stable the requirements are over time. RSI=Number of requirements unchangedTotal number of requirements×100RSI = \frac{\text{Number of requirements unchanged}}{\text{Total number of requirements}} \times 100
  2. **Requirements Completeness:**  
     Assesses if all functionalities are defined and covered.
  3. **Requirements Ambiguity:**  
     Identifies vague or unclear requirements to reduce misinterpretations.
  4. **Traceability Metrics:**  
     Ensures each requirement can be traced through design, implementation, and testing phases.
* **Example:**  
  If 80 out of 100 requirements remain unchanged, the RSI would be 80%.

**3. Metrics for the Design Model**

* **Purpose:** Evaluate the efficiency, modularity, and maintainability of software design.
* **Key Metrics:**
  1. **Structural Complexity (SC):**  
     Measures interconnections between modules.
  2. **Cohesion Metrics:**  
     Assesses the degree to which elements within a module work together.
     + High cohesion is desirable for maintainability.
  3. **Coupling Metrics:**  
     Evaluates interdependencies between modules.
     + Low coupling is preferable.
  4. **Design Size Metrics:**
     + Measures the number of classes, interfaces, and packages.
  5. **Fan-in and Fan-out:**
     + **Fan-in:** Number of functions calling a module.
     + **Fan-out:** Number of functions called by a module.
* **Example:**  
  If a module has high fan-out and low cohesion, it indicates a need for design improvement.

**4. Metrics for Source Code**

* **Purpose:** Assess the complexity, quality, and maintainability of the source code.
* **Key Metrics:**
  1. **Lines of Code (LOC):**  
     Simple measure of code size but lacks insight into complexity.
  2. **Cyclomatic Complexity (CC):**  
     Measures decision points in the code to assess complexity. CC=E−N+2PCC = E - N + 2P Where:
     + **E:** Number of edges in the control flow graph.
     + **N:** Number of nodes.
     + **P:** Number of connected components.
  3. **Halstead Metrics:**  
     Evaluates code complexity based on operators and operands.
  4. **Comment Density:**  
     Ratio of comments to code to assess code readability.
  5. **Code Coverage:**  
     Measures percentage of code executed during testing.
* **Example:**  
  If a function has a cyclomatic complexity of 15, it might be too complex and require refactoring.

**5. Metrics for Testing**

* **Purpose:** Evaluate the effectiveness, efficiency, and coverage of testing activities.
* **Key Metrics:**
  1. **Defect Density:**  
     Number of defects per thousand lines of code (KLOC). Defect Density=Total defects foundKLOC\text{Defect Density} = \frac{\text{Total defects found}}{\text{KLOC}}
  2. **Test Coverage:**  
     Measures the extent to which the source code is tested.
     + **Statement Coverage:** Tests all executable statements.
     + **Branch Coverage:** Tests all decision paths.
  3. **Defect Removal Efficiency (DRE):**  
     Measures the effectiveness of the testing process. DRE=Defects found during testingDefects found during testing + after release×100DRE = \frac{\text{Defects found during testing}}{\text{Defects found during testing + after release}} \times 100
  4. **Test Case Effectiveness:**  
     Percentage of test cases that detect defects.
  5. **Mean Time to Detect (MTTD):**  
     Average time taken to discover a defect.
* **Example:**  
  A DRE of 90% indicates effective testing, as most defects were caught before release.

**6. Metrics for Maintenance**

* **Purpose:** Assess the maintainability, reliability, and stability of the software after deployment.
* **Key Metrics:**
  1. **Mean Time to Repair (MTTR):**  
     Average time required to fix a defect after it is reported.
  2. **Mean Time Between Failures (MTBF):**  
     Average time between two consecutive failures. MTBF=Total operational timeNumber of failuresMTBF = \frac{\text{Total operational time}}{\text{Number of failures}}
  3. **Maintenance Effort:**  
     Measures time and resources spent on maintenance activities.
  4. **Code Change Rate:**  
     Frequency and volume of code changes.
  5. **Corrective, Adaptive, and Perfective Maintenance Metrics:**
     + **Corrective:** Fixing defects.
     + **Adaptive:** Adapting to environment changes.
     + **Perfective:** Enhancements and optimizations.
* **Example:**  
  An MTTR of 2 hours means that on average, it takes 2 hours to fix a defect.

**Summary of Key Product Metrics**

| **Category** | **Key Metrics** | **Purpose** |
| --- | --- | --- |
| **Requirements Model** | RSI, Completeness, Ambiguity | Validate requirement clarity and stability |
| **Design Model** | Cohesion, Coupling, Fan-in, Fan-out | Evaluate design efficiency and maintainability |
| **Source Code** | LOC, Cyclomatic Complexity, Code Coverage | Assess complexity and maintainability |
| **Testing** | Defect Density, DRE, Test Coverage | Ensure testing effectiveness and defect detection |
| **Maintenance** | MTTR, MTBF, Maintenance Effort | Measure reliability and maintainability |