06. Software QA & Testing Tools - Comprehensive Textbook Reference

> **Module Focus**: Software Testing Fundamentals, Defect Lifecycle, Black-Box (EP, BVA, Decision Tables), White-Box (Coverage Criteria, CFG, Cyclomatic Complexity $V(G)$), Test Automation Frameworks (Selenium WebDriver, JUnit 5, Postman, Apache JMeter) with Worked Examples & Numerical Calculations.

---

1. Software Testing Fundamentals & Testing Levels

1.1 Verification vs Validation & Static vs Dynamic Testing

Software Quality Assurance (QA) relies on two complementary activities: **Verification** and **Validation** (the V&V principle).

Attribute	Verification	Validation
Core Question	*"Are we building the product right?"*	*"Are we building the right product?"*
Nature	Static testing (code execution NOT required).	Dynamic testing (code execution required).
Activities	Reviews, walkthroughs, inspections, static code analysis.	Functional testing, system testing, load testing, UAT.
Artifacts Evaluated	SRS, architecture diagrams, source code reviews, design docs.	Compiled binaries, running application web/ mobile interfaces.
Defect Detection Stage	Early SDLC (Requirements/Design phase).	Late SDLC (Testing/Deployment phase).

Static vs Dynamic Testing Mechanics:

- **Static Testing**: Evaluates software work products without running code. Includes Desk Checking, Code Walkthroughs (informal), Code Inspections (formal with checklists and roles like Moderator, Author, Reader, Inspector), and Automated Static Code Analysis (e.g., SonarQube, SpotBugs, ESLint).
- **Dynamic Testing**: Executes code against specific inputs to observe runtime behavior, output accuracy, memory usage, and execution speed.

1.2 The 4 Levels of Software Testing

Software testing is structured hierarchically corresponding to the software architecture and SDLC phases:

```
+-------------------------------------------------------------------------+
| Level 4: Acceptance Testing (UAT, Alpha, Beta, Operational)             |
```

```
+-------------------------------------------------------------------------+
| Level 3: System Testing (Functional, Performance, Security, Usability)  |
+-------------------------------------------------------------------------+
| Level 2: Integration Testing (Top-Down, Bottom-Up, Big-Bang, Sandwich)  |
+-------------------------------------------------------------------------+
| Level 1: Unit Testing (Methods / Classes / Functions - Isolated)        |
+-------------------------------------------------------------------------+
```

1. Unit Testing:

- Focuses on individual methods, functions, or modules in isolation.
- Written and executed by developers using frameworks like **JUnit 5, TestNG, PyTest, Jest**.
- Employs **Test Doubles** (Mocks, Stubs, Fakes, Spies) to isolate the unit under test from database, network, or external dependencies.

2. Integration Testing:

Verifies interactions and data exchange between integrated modules.

- **Top-Down Integration**:
- Testing starts from top-level control modules down to low-level utility modules.
- Lower-level missing modules are replaced by **Stubs** (dummy code that returns hardcoded responses to higher-level modules).
- **Bottom-Up Integration**:
- Testing starts from low-level utility modules up to high-level control modules.
- Higher-level missing modules are replaced by **Drivers** (test scripts/programs that call and pass parameters to low-level modules).
- **Sandwich / Hybrid Integration**: Combines top-down for higher layers and bottom-up for lower layers, testing toward a central target layer.
- **Big-Bang Integration**: All modules are integrated simultaneously and tested at once. *Disadvantage*: Extremely difficult to isolate the root cause of failures.

```
STUB vs DRIVER COMPARISON:
--------------------------
Top-Down Testing:    [ High-Level Module ] ---> Calls ---> [ STUB (Simulates Low-Level) ]
Bottom-Up Testing:   [ DRIVER (Simulates High-Level) ] ---> Calls ---> [ Low-Level
Module ]
```

3. System Testing:

- End-to-end testing of the fully integrated software application.
- Verifies compliance against the **Software Requirement Specification (SRS)**.
- Validates both **Functional Requirements** (business logic, workflows) and **Non-Functional Requirements** (Performance, Load, Security, Usability, Reliability).

4. Acceptance Testing:

Validates whether the product meets business requirements and customer readiness for production.

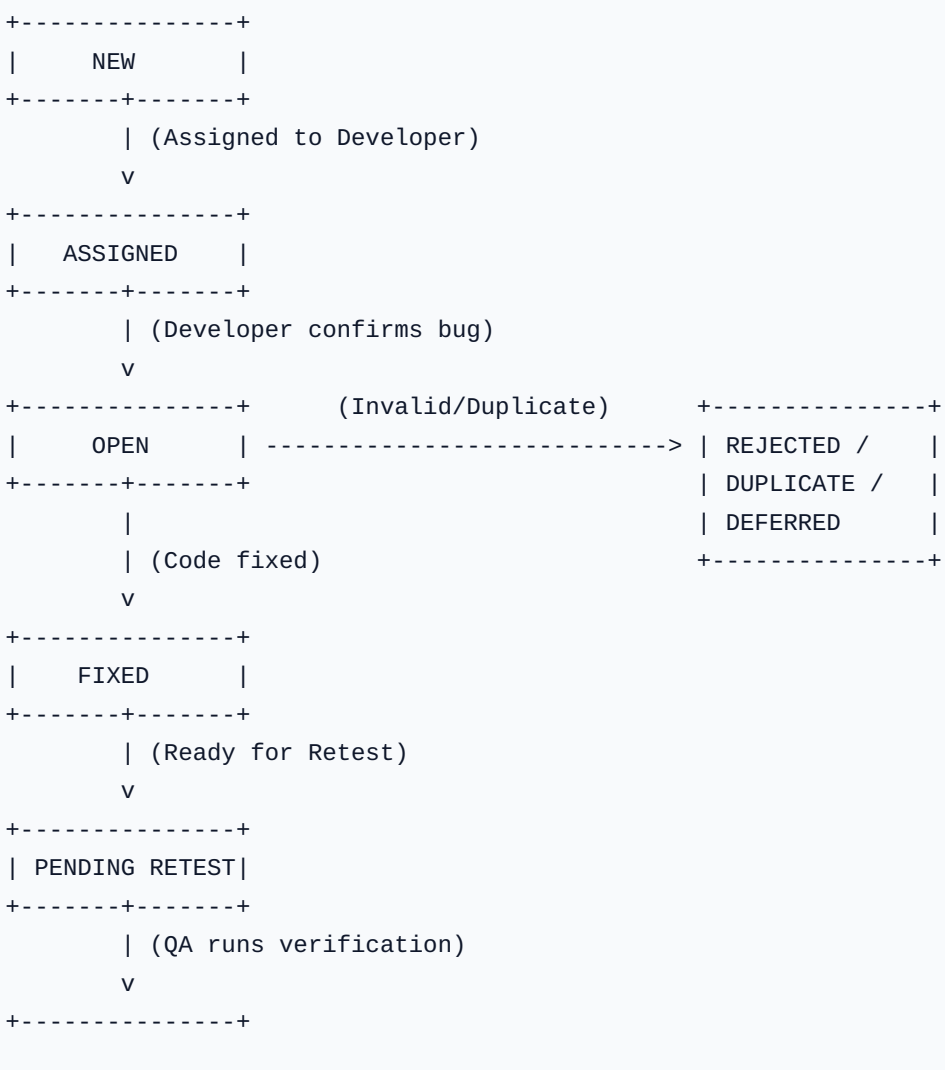
- **Alpha Testing**: Conducted by internal test teams or select internal users at the developer's site in a controlled environment.
- **Beta Testing**: Conducted by external end users/customers in their own real-world operational environments prior to final release.
- **User Acceptance Testing (UAT)**: Business users verify that the software fulfills real-world business process scenarios.
- **Operational Acceptance Testing (OAT)**: Verifies operational readiness (backup/recovery, maintenance tasks, failover checks).

---

2. Defect / Bug Lifecycle & Defect Metrics

2.1 Defect Workflow State Diagram

A **Defect (Bug)** is a deviation between expected result and actual result. It progresses through defined states during its lifecycle:



```
        +----> |    RETEST     |
        |      +-------+-------+
        |              |
(Retest Failed)        +-----------------------+
        |              | (Retest Passed)       |
        |              v                       v
+-------+--------+  +---------------+  +---------------+
|   REOPENED     |  |   VERIFIED    |  |    CLOSED     |
+----------------+  +---------------+  +---------------+
```

Key State Definitions:

- **New**: Defect logged by tester with steps to reproduce, logs, and screenshots.
- **Assigned**: Lead assigns bug to developer for root cause analysis.
- **Open**: Developer actively investigating or fixing code.
- **Deferred**: Fix postponed to a future release (e.g., low priority, close to release date).
- **Rejected / Duplicate / Not a Bug**: Defect invalid, duplicate of an existing bug, or working as designed.
- **Fixed**: Developer completes code fix and deploys to QA environment.
- **Pending Retest**: Assigned back to QA team for verification.
- **Retest / Verified**: QA executes test case; if fixed, marked **Verified** and then **Closed**.
- **Reopened**: If retest fails, defect is set to **Reopened** and sent back to developer.

2.2 Severity vs Priority Matrix

- **Severity**: Measures the **technical impact** of the bug on application functionality or system stability (Engineered aspect).
- **Priority**: Measures the **business urgency** of fixing the bug based on user impact and release timelines (Business aspect).

```
                  HIGH SEVERITY                       LOW SEVERITY
         +-----------------------------------+-----------------------------------+
 HIGH    | QUADRANT 1: High Sev / High Prio  | QUADRANT 2: Low Sev / High Prio   |
PRIORITY | System crash on checkout page;    | Company logo upside down; typo in |
         | payment gateway throwing 500 error| brand name on homepage.           |
         +-----------------------------------+-----------------------------------+
  LOW    | QUADRANT 3: High Sev / Low Prio   | QUADRANT 4: Low Sev / Low Prio    |
PRIORITY | System crash on legacy browser    | Minor alignment issue on obscure  |
         | (IE 11) used by < 0.01% users.    | Help page footer.                 |
         +-----------------------------------+-----------------------------------+
```

2.3 Key Defect Metrics & Formulas

1. Defect Density:

$$\text{Defect Density} = \frac{\text{Total Defects Found}}{\text{Size of Module (in KLOC or Story Points)}}$$

2. Defect Removal Efficiency (DRE):

$$\text{DRE} = \left(\frac{D_{\text{pre}}}{D_{\text{pre}} + D_{\text{post}}} \right) \times 100$$ *(Where $D_{\text{pre}} = \text{Defects found before release (during testing)}$, $D_{\text{post}} = \text{Defects found post-release by customers}$)*

✏️ **Worked Numerical Example:**

A software system has 15,000 lines of code (15 KLOC). During testing, 45 defects were identified. After release, customers reported 5 defects in production.

1. **Calculate Defect Density during testing**: $$\text{Defect Density} = \frac{45}{15} = \mathbf{3.0 \text{ Defects / KLOC}}$$
2. **Calculate Defect Removal Efficiency (DRE)**: $$\text{DRE} = \left(\frac{45}{45 + 5} \right) \times 100 = \left(\frac{45}{50} \right) \times 100 = \mathbf{90\%}$$

---

3. Black-Box Test Design Techniques

Black-box testing focuses on inputs and expected outputs without examining source code logic.

3.1 Equivalence Partitioning (EP)

Divides the continuous input data domain into discrete valid and invalid partitions. Assumptions: all values within a partition are processed identically by the system. Select **one representative value** from each partition.

✏️ **Worked Example (Single-Variable EP):**

Requirement: An online bank application accepts loan duration in months between 12 and 60 (inclusive). - **Partition 1 (Invalid - Below Range)**: $\text{Duration} < 12 \implies \text{Test Value: } \mathbf{6}$ - **Partition 2 (Valid Range)**: $12 \le \text{Duration} \le 60 \implies \text{Test Value: } \mathbf{36}$ - **Partition 3 (Invalid - Above Range)**: $\text{Duration} > 60 \implies \text{Test Value: } \mathbf{72}$

3.2 Boundary Value Analysis (BVA)

Defects occur overwhelmingly at boundaries. BVA tests values at boundaries of input partitions.

Boundary Testing Paradigms:

1. **2-Value Boundary Analysis (Standard)**: For boundary range $[A, B]$, test points are $\{A-1, A, B, B+1\}$ (Invalid Below, Minimum Valid, Maximum Valid, Invalid Above).
2. **3-Value Boundary Analysis (Extended)**: Tests $\{A-1, A, A+1, \text{Nominal}, B-1, B, B+1\}$ ($Min-, Min, Min+, Nom, Max-, Max, Max+$).

✏️ **Worked Example (BVA Comparison):**

Requirement: Input integer percentage score $S \in [0, 100]$.

- **2-Value BVA Test Cases (4 Boundary Values)**: $$\text{Test Set} = \{-1, 0, 100, 101\}$$
- **3-Value BVA Test Cases (7 Boundary Values)**: $$\text{Test Set} = \{-1, 0, 1, 50, 99, 100, 101\}$$

✏️ **Multi-Variable Boundary Value Analysis Count Formula:**

For n input variables with valid range constraints: - **Boundary Value Analysis (Single fault assumption)**: $4n + 1$ test cases. - **Robust Boundary Value Analysis**: $6n + 1$ test cases. - **Worst-Case Boundary Value Analysis**: 5^n test cases. - **Robust Worst-Case Boundary Value Analysis**: 7^n test cases.

Example: For $n = 3$ input variables, standard BVA requires $(4 \times 3) + 1 = \mathbf{13 \text{ test cases}}$, while Robust BVA requires $(6 \times 3) + 1 = \mathbf{19 \text{ test cases}}$.

3.3 Decision Table Testing

Used for complex business rules involving multiple input conditions producing combinations of actions.

✏️ **Worked Example (E-Commerce Discount Rules):**

Business Rules: - Rule 1: If user is Premium Member AND Purchase > $100, apply 20% discount. - Rule 2: If user is Premium Member AND Purchase $\le \$100$, apply 10% discount. - Rule 3: If user is Standard Member AND has Coupon Code, apply 10% discount. - Rule 4: Otherwise, 0% discount.

Conditions & Actions	**Rule 1**	**Rule 2**	**Rule 3**	**Rule 4**
Is Premium Member?	**Y**	**Y**	N	N
Purchase > $100?	**Y**	N	-	-
Has Coupon Code?	-	-	**Y**	N
Action: 20% Discount	**X**			
Action: 10% Discount		**X**	**X**	
Action: 0% Discount				**X**

---

4. White-Box Test Design Techniques & Cyclomatic Complexity $V(G)$

White-box testing inspects code logic, decision branches, execution paths, and control structures.

4.1 Structural Coverage Criteria

1. **Statement Coverage**: Percentage of executable source code statements executed by test cases. $$ \text{Statement Coverage \%} = \left(\frac{\text{Executed Statements}}{\text{Total Statements}} \right) \times 100$$

2. **Branch / Decision Coverage**: Percentage of decision outcomes (True/False evaluation of `if`, `switch`, `while`) executed. $$\text{Branch Coverage \%} = \left(\frac{\text{Executed Decision Outcomes}}{\text{Total Decision Outcomes (2 } \times \text{ Predicate Nodes)}} \right) \times 100$$
3. **Condition Coverage**: Evaluates every Boolean sub-condition within compound decisions (e.g., in `if (A && B)`, both `A=True/False` and `B=True/False` must be evaluated).
4. **Path Coverage**: Ensures every linearly independent path through the Control Flow Graph (CFG) is executed. (100% Path Coverage $\implies$ 100% Branch Coverage $\implies$ 100% Statement Coverage).

4.2 Cyclomatic Complexity $V(G)$ Theory & Formulas

Cyclomatic complexity $V(G)$ (developed by Thomas J. McCabe) measures the structural complexity of a program graph G and defines the upper bound on the number of test cases required to achieve 100% branch and path coverage.

3 Mathematical Methods to Calculate $V(G)$:

1. **Edges and Nodes Formula**: $$V(G) = E - N + 2P$$ *(Where $E = \text{Number of Edges}, N = \text{Number of Nodes}, P = \text{Number of Connected Components / Procedures, typically } 1)*
2. **Predicate Nodes Formula**: $$V(G) = P_{\text{nodes}} + 1$$ *(Where $P_{\text{nodes}} = \text{Number of Decision/Predicate Nodes such as } \texttt{if}, \texttt{while}, \texttt{for}, \texttt{case})*
3. **Bounded Regions Formula**: $$V(G) = R = \text{Number of Closed Bounded Regions in planar CFG} + 1$$

4.3 Worked Step-by-Step Code Example: CFG & $V(G)$ Calculation

Consider the following Java function that evaluates discount eligibility and updates account balance:

```
public static int processOrder(int age, boolean isStudent, int amount) {
    int discount = 0;                  // Node 1
    if (age < 18 || isStudent) {       // Node 2 & 3 (Predicate Nodes)
        discount = 15;                 // Node 4
    } else {
        discount = 5;                  // Node 5
    }
    if (amount > 1000) {               // Node 6 (Predicate Node)
        discount += 10;                // Node 7
    }                                  // Node 8 (Join Node)
    return discount;                   // Node 9
}
```

Step 1: Control Flow Graph (CFG) Representation

Step 2: $V(G)$ Calculation Using All 3 Methods

- **Method 1: Edge-Node Formula**:
- Nodes (N) = 9 (Nodes 1 through 9)

- Edges (E) = 11 directed edges: $(1 \to 2), (2 \to 4), (2 \to 3), (3 \to 4), (3 \to 5), (4 \to 6), (5 \to 6), (6 \to 7), (6 \to 8), (7 \to 8), (8 \to 9)$
- Connected Components (P) = 1 $$V(G) = E - N + 2P = 11 - 9 + 2(1) = \mathbf{4}$$
- **Method 2: Predicate Node Formula**:
- Decision conditions (P_{nodes}):
    1. `age < 18` (Node 2)
    2. `isStudent` (Node 3)
    3. `amount > 1000` (Node 6) $$V(G) = P_{\text{nodes}} + 1 = 3 + 1 = \mathbf{4}$$
- **Method 3: Regions Formula**:
- Number of enclosed closed regions in graph $= 3$
- Enclosed regions + 1 unbounded outer region $= 3 + 1 = \mathbf{4}$

Step 3: Basis Paths Identification

Since $V(G) = 4$, there are **4 linearly independent basis paths**:

1. **Path 1**: $1 \to 2(\text{True}) \to 4 \to 6(\text{False}) \to 8 \to 9$
2. *Test Input*: `age = 15, isStudent = false, amount = 500` $\implies$ Output: `15`
3. **Path 2**: $1 \to 2(\text{False}) \to 3(\text{True}) \to 4 \to 6(\text{True}) \to 7 \to 8 \to 9$
4. *Test Input*: `age = 25, isStudent = true, amount = 1500` $\implies$ Output: `25`
5. **Path 3**: $1 \to 2(\text{False}) \to 3(\text{False}) \to 5 \to 6(\text{False}) \to 8 \to 9$
6. *Test Input*: `age = 30, isStudent = false, amount = 800` $\implies$ Output: `5`
7. **Path 4**: $1 \to 2(\text{False}) \to 3(\text{False}) \to 5 \to 6(\text{True}) \to 7 \to 8 \to 9$
8. *Test Input*: `age = 40, isStudent = false, amount = 2000` $\implies$ Output: `15`

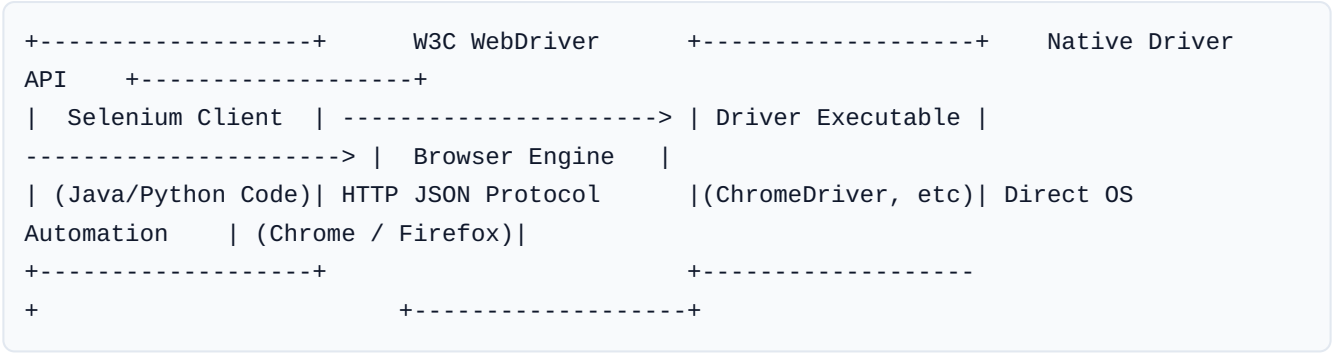
Executing these 4 test cases achieves 100% Statement, 100% Decision, and 100% Path Coverage.

---

5. Selenium WebDriver Automation Framework

5.1 Architecture & W3C JSON Wire Protocol

Selenium WebDriver facilitates browser automation by issuing commands directly to browser engines via native browser drivers.



5.2 Locators Hierarchy & Syntax

Web elements are located on the DOM using the `By` class strategies.

Locator Strategy	Syntax Example (Java)	Speed / Priority	Best Use Case
`By.id`	`driver.findElement(By.id("username"))`	**1 (Fastest)**	Unique element I in well-designed DOM.
`By.name`	`driver.findElement(By.name("email"))`	2	Form inpu fields.
`By.className`	`driver.findElement(By.className("btn-primary"))`	3	Styling-based element lookup.
`By.cssSelector`	`driver.findElement(By.cssSelector("input#user[type='text']"))`	4 (Fast)	High performa alternativ to XPath.
`By.xpath`	`driver.findElement(By.xpath("//div[@class='card']//button[text()='Submit']"))`	5 (Flexible)	Complex relative navigatio and dynamic DOMs.

Advanced XPath Syntax & Axes:

- **Relative XPath**: `//input[@id='email' and @type='text']`
- **Text Match**: `//button[text()='Login']` or `//a[contains(text(),'Forgot')]`
- **XPath Axes**:
- `//input[@id='username']/parent::div` (Parent node)
- `//label[text()='Password']/following-sibling::input` (Sibling element)
- `//td[text()='John Doe']/ancestor::tr` (Table row lookup)

5.3 Selenium Synchronization & Waits Mechanics

Unsynchronized script execution leads to `NoSuchElementException` or `ElementNotInteractableException`.

```
              +-------------------------------------------------------------+
              |                    SELENIUM WAIT TYPES                      |
              +-------------------------------------------------------------+
              | 1. IMPLICIT WAIT: Global timeout for DOM element polling.   |
              | 2. EXPLICIT WAIT: Conditional wait for specific condition.  |
              | 3. FLUENT WAIT: Explicit wait with custom polling interval.  |
              +-------------------------------------------------------------+
```

Worked Java Code Implementation (Locators & Waits):

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
import org.openqa.selenium.support.ui.FluentWait;
import java.time.Duration;
import java.util.NoSuchElementException;

public class SeleniumWaitDemo {
    public static void main(String[] args) {
        WebDriver driver = new ChromeDriver();

        // 1. IMPLICIT WAIT (Global setting applied to all findElement calls)
        driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));

        driver.get("https://example.com/login");

        // 2. EXPLICIT WAIT (Waits up to 15 seconds until element is clickable)
        WebDriverWait wait = new WebDriverWait(driver, Duration.ofSeconds(15));
        WebElement loginButton = wait.until(
            ExpectedConditions.elementToBeClickable(By.xpath("//button[@id='submit-
btn']"))
        );
        loginButton.click();

        // 3. FLUENT WAIT (Waits 30s max, polls every 500ms, ignores
NoSuchElementException)
        FluentWait<WebDriver> fluentWait = new FluentWait<>(driver)
                .withTimeout(Duration.ofSeconds(30))
                .pollingEvery(Duration.ofMillis(500))
                .ignoring(NoSuchElementException.class);

        WebElement successMessage = fluentWait.until(d ->
            d.findElement(By.cssSelector("div.alert-success"))
        );
```

```
        System.out.println("Message: " + successMessage.getText());
        driver.quit();
    }
}
```

---

6. JUnit 5 Testing Framework & Mocking

6.1 Core Annotations & Lifecycle

JUnit 5 (JUnit Jupiter) structures unit testing in Java.

Annotation	Description	Lifecycle Execution
`@Test`	Denotes a test method.	Runs once per test invocation.
`@BeforeEach`	Executes setup code before *each* test method.	Pre-test setup.
`@AfterEach`	Executes cleanup code after *each* test method.	Post-test teardown.
`@BeforeAll`	Executes once before *all* tests (must be `static`).	One-time expensive setup (e.g., DB connection).
`@AfterAll`	Executes once after *all* tests complete (must be `static`).	One-time cleanup.
`@Disabled`	Disables test method or class execution.	Skipped during test run.
`@ParameterizedTest`	Executes test multiple times with different data arguments.	Data-driven execution.

6.2 Worked Java Code Example (JUnit 5 + Mockito)

```
import org.junit.jupiter.api.*;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.ValueSource;
import org.mockito.Mockito;
import static org.junit.jupiter.api.Assertions.*;
import static org.mockito.Mockito.*;

class BankAccount {
    private double balance;
    public BankAccount(double initialBalance) { this.balance = initialBalance; }

    public double withdraw(double amount) {
        if (amount > balance) throw new IllegalArgumentException("Insufficient funds");
        balance -= amount;
        return balance;
    }
    public double getBalance() { return balance; }
```

```
}

public class BankAccountTest {
    private BankAccount account;

    @BeforeEach
    void setUp() {
        account = new BankAccount(500.0); // Fresh instance before each test
    }

    @Test
    @DisplayName("Valid Withdrawal Test")
    void testValidWithdrawal() {
        double remaining = account.withdraw(200.0);
        assertEquals(300.0, remaining, 0.001, "Balance should reflect withdrawal");
    }

    @Test
    @DisplayName("Overdraft Exception Verification")
    void testOverdraftThrowsException() {
        Exception exception = assertThrows(IllegalArgumentException.class, () -> {
            account.withdraw(600.0);
        });
        assertEquals("Insufficient funds", exception.getMessage());
    }

    @ParameterizedTest
    @ValueSource(doubles = {50.0, 100.0, 250.0})
    @DisplayName("Parameterized Valid Deposits")
    void testMultipleWithdrawals(double amount) {
        double initial = account.getBalance();
        account.withdraw(amount);
        assertTrue(account.getBalance() < initial);
    }

    @Test
    @DisplayName("Mockito Stubbing Example")
    void testDependencyWithMock() {
        // Create mock object for external payment service
        PaymentGateway mockGateway = Mockito.mock(PaymentGateway.class);
        when(mockGateway.processPayment(100.0)).thenReturn(true);

        assertTrue(mockGateway.processPayment(100.0));
        verify(mockGateway, times(1)).processPayment(100.0);
    }
}

interface PaymentGateway {
    boolean processPayment(double amount);
}
```

7. Postman API Testing Framework

Postman enables API test automation against HTTP REST endpoints using JavaScript test scripts built on the Chai assertion library.

7.1 Postman Execution Sandbox & API Mechanics

```
[ Pre-Request Script ] ---> [ Send HTTP Request ] ---> [ Response Received ] ---> [ Test Script Assertions ]
```

7.2 Worked Postman JavaScript Test Scripts

```
// 1. ASSERT STATUS CODE IS 200 OK
pm.test("Status code is 200 OK", function () {
    pm.response.to.have.status(200);
});

// 2. ASSERT RESPONSE TIME IS WITHIN SLA (< 500ms)
pm.test("Response time is less than 500ms", function () {
    pm.expect(pm.response.responseTime).to.be.below(500);
});

// 3. ASSERT JSON RESPONSE STRUCTURE AND VALUE
pm.test("Validate User Payload Attributes", function () {
    var jsonData = pm.response.json();

    // Assert root fields
    pm.expect(jsonData.success).to.eql(true);
    pm.expect(jsonData.data.userId).to.equal(1042);
    pm.expect(jsonData.data.email).to.include("@example.com");

    // Extract Authentication Token and save to Environment Variable for chaining
    pm.environment.set("authToken", jsonData.data.token);
});

// 4. HEADER & CONTENT-TYPE ASSERTION
pm.test("Content-Type is application/json", function () {
    pm.response.to.have.header("Content-Type", "application/json; charset=utf-8");
});
```

8. Apache JMeter Performance & Load Testing

8.1 Key JMeter Building Blocks

1. **Test Plan**: Root container configuring performance test scenario.
2. **Thread Group**: Simulates virtual concurrent users. Properties:
3. **Number of Threads (N)**: Total concurrent users simulated.
4. **Ramp-Up Period (T)**: Time taken to start all virtual users.
5. **Loop Count**: Iteration count per virtual user.
6. **Samplers**: Requests sent to target server (e.g., HTTP Request, JDBC Request, FTP Request).
7. **Listeners**: Collects and visualizes performance metrics (e.g., View Results Tree, Summary Report, Aggregate Report).
8. **Timers**: Introduces realistic user think time between requests (e.g., Constant Timer, Gaussian Random Timer).
9. **Assertions**: Validates server response (e.g., Response Assertion for status 200, Duration Assertion).

8.2 Performance Testing Metrics & Formulas

- **Throughput (TPS / RPS)**: Requests processed per second. $$\text{Throughput} = \frac{\text{Total Completed Requests}}{\text{Total Elapsed Execution Time}}$$
- **Latency**: Time from sending request to receiving *first byte* of response.
- **Response Time (Elapsed Time)**: Total time from sending request to receiving *last byte* of response.
- **Percentiles ($90^{\text{th}}, 95^{\text{th}}, 99^{\text{th}}$)**: Value below which $X\%$ of response time samples fall (eliminates outlier skew).
- **Little's Law Formula (Queueing Theory)**: $$L = \lambda \times W$$ *(Where $L = \text{Average active virtual users in system}, \lambda = \text{Throughput (RPS)}, W = \text{Average User Response Time + Think Time})*

8.3 ✏️ Worked Performance Workload Calculation Example

Scenario: A web system requires a performance test simulating **3,600 transactions per hour**. The average response time is **1.5 seconds**, and user think time is **3.5 seconds**.

1. **Calculate Target Throughput (λ) in Requests per Second**: $$\lambda = \frac{3600 \text{ transactions}}{3600 \text{ seconds}} = \mathbf{1.0 \text{ request/sec (RPS)}}$$
2. **Calculate Total User Residence Time (W)**: $$W = \text{Response Time} + \text{Think Time} = 1.5\text{s} + 3.5\text{s} = \mathbf{5.0 \text{ seconds}}$$
3. **Calculate Required Concurrent Threads (L) using Little's Law**: $$L = \lambda \times W = 1.0 \times 5.0 = \mathbf{5 \text{ Concurrent Virtual User Threads}}$$

---

9. Technical MCQs & Problem-Solving Bank

Q1: Boundary Value Analysis

Question: An input field accepts integers in the range $[20, 80]$. Using 2-value Boundary Value Analysis, what are the exact test cases required?
A) $\{19, 20, 80, 81\}$
B) $\{20, 21, 79, 80\}$
C) $\{19, 20, 21, 50, 79, 80, 81\}$
D) $\{18, 19, 20, 80, 81, 82\}$

Answer: **A**
Explanation: Standard 2-value BVA tests the minimum value (20), maximum value (80), one below minimum (19), and one above maximum (81). Option C corresponds to 3-value BVA.

Q2: Cyclomatic Complexity

Question: A program control flow graph has 14 edges (E) and 10 nodes (N), with a single main module ($P=1$). What is its Cyclomatic Complexity $V(G)$?
A) 4
B) 6
C) 5
D) 8

Answer: **B**
Explanation: Apply McCabe's formula: $V(G) = E - N + 2P = 14 - 10 + 2(1) = 6$.

Q3: Integration Testing Strategy

Question: During integration testing, a tester writes a temporary program component to simulate a high-level module that calls the module under test. What is this component called?
A) Stub
B) Driver
C) Mock
D) Spy

Answer: **B**
Explanation: Higher-level calling components used in bottom-up integration are called **Drivers**. Dummy low-level called components used in top-down integration are called **Stubs**.

Q4: Postman API Assertions

Question: Which JavaScript snippet correctly asserts in Postman that the HTTP response code is 201 Created?
A) `assert.statusCode(201);`
B) `pm.response.to.have.status(201);`

C) `pm.expect(response.code).is(201);`

D) `http.getStatus() == 201;`

Answer: **B**

Explanation: Postman uses the `pm.response.to.have.status(code)` syntax built on Chai HTTP assertions.