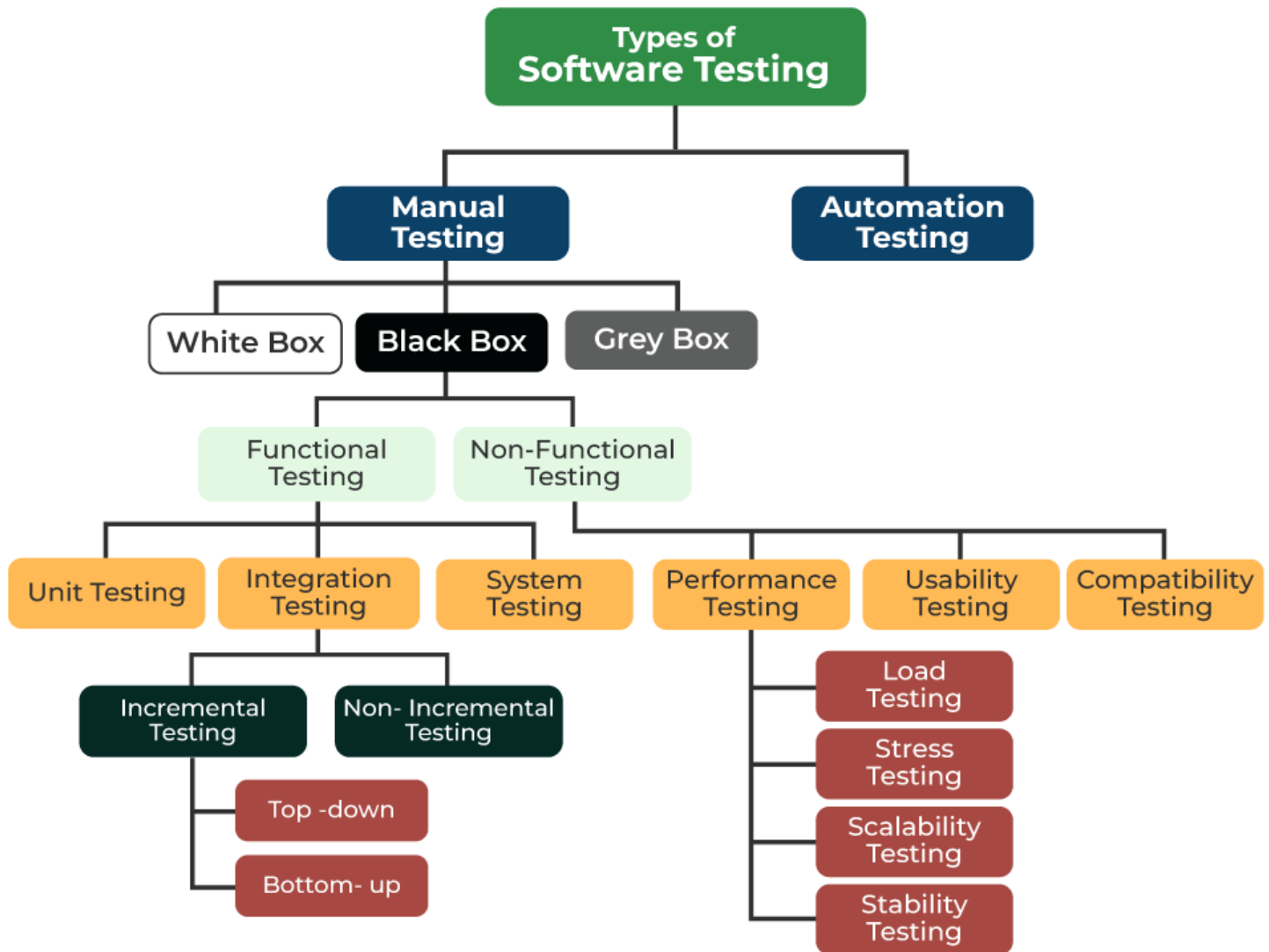


Chapter 9 : Testing software



1. Introduction to Software Testing

Software Testing is the process of evaluating and verifying that a software product or application does what it is supposed to do. It helps identify bugs or errors in the software, and ensures that the product is reliable and meets the requirements.

2. Why is Software Testing Important?

- To find defects before release.
 - To ensure the product works as expected.
 - To improve product quality.
 - To ensure user satisfaction.
 - To avoid costly post-release bugs.
-

3. Benefits of Software Testing

- Detect bugs early.
 - Saves time and cost in the long term.
 - Increases customer confidence.
 - Improves security.
 - Ensures compatibility across platforms and devices.
-

4. Limitations and Challenges of Software Testing

Limitations:

- It cannot guarantee a 100% bug-free product.
- It requires time and resources.
- Some complex scenarios are hard to test.
- Not all defects can be found.

Challenges:

- Changing requirements during development.
 - Time constraints.
 - Lack of skilled testers.
 - Incomplete requirements.
 - Testing on multiple environments and platforms.
-

5. Types of Software Testing

1. Manual Testing

- Performed by human testers without any tools.
- Suitable for exploratory testing, usability, and ad-hoc testing.

Example:

- Tester opens a website and manually tries logging in with valid/invalid credentials.

2. Automated Testing

- Performed using tools and scripts.
- Useful for regression testing, load testing, and repetitive tasks.

Java Example (Using JUnit):

```
import static org.junit.Assert.assertEquals;
```

```
import org.junit.Test;
```

```
public class CalculatorTest {  
    public int add(int a, int b) {  
        return a + b;  
    }  
}
```

```
@Test
```

```
public void testAdd() {  
    CalculatorTest calc = new CalculatorTest();  
    assertEquals(5, calc.add(2, 3));  
}  
}
```

6. Testing Levels

1. Unit Testing

- Tests individual components (functions, methods).
- Written by developers.

Java Example:

```
public class MathUtils {  
    public static int square(int x) {  
        return x * x;  
    }  
}
```

```
}

import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class MathUtilsTest {
    @Test
    public void testSquare() {
        assertEquals(25, MathUtils.square(5));
    }
}
```

2. Integration Testing

- Tests the interaction between modules or components.

3. System Testing

- Tests the complete integrated application.

4. Acceptance Testing

- Validates the system against business requirements.
- Usually done by clients or end-users.

7. Testing Methods

Black Box Testing

- Tester does not need to know the internal code.
- Focus is on inputs and outputs.

White Box Testing

- Tester understands the internal code and logic.
- Focus is on paths, conditions, and branches.

Gray Box Testing

- Partial knowledge of internal logic.
-

8. Functional vs Non-Functional Testing

Functional Testing

- Verifies specific features or functions.
- Example: login, registration, file upload.

Non-Functional Testing

- Tests performance, usability, reliability, etc.
- Examples:
 - Performance Testing
 - Security Testing
 - Compatibility Testing

- Usability Testing
-

9. Automated Testing Tools

- JUnit – Java Unit Testing
- Selenium – Web UI Testing
- TestNG – Advanced Java testing
- Postman – API Testing

Java Example using Junit :

JUnit Testing in Java

1. Assertion Methods in JUnit

JUnit provides a set of assertion methods that help verify that the actual output of a program matches the expected result.


Method	Description
<code>assertTrue(test)</code>	Fails if test is false.
<code>assertFalse(test)</code>	Fails if test is true.
<code>assertEquals(expected, actual)</code>	Fails if the values are not equal using <code>.equals()</code> .
<code>assertSame(expected, actual)</code>	Fails if the two references are not the same (<code>==</code>).

<code>assertNotSame(expected, actual)</code>	Fails if the two references are the same.
<code>assertNull(value)</code>	Fails if value is not null.
<code>assertNotNull(value)</code>	Fails if value is null.
<code>fail()</code>	Immediately causes the test to fail.

You can also include a custom failure message in any assertion:
`assertEquals("Expected value after 1 day", expected, actual);`

2. Structuring Assertions Clearly

Well-structured assertions improve the readability and usefulness of test failures.

 **Bad Example:** `assertEquals("should have gotten " + expected + "\n but instead got " + actual + "\n", expected, actual);`

This message is hard to read and not useful in test output.

 **Good Example:** `assertEquals("adding one day to 2050/2/15", expected, actual);`

3. Testing for Timeouts

JUnit allows you to specify a timeout for test methods. The test fails if it doesn't complete in time: `@Test(timeout = 5000)` // milliseconds

```
public void testSomethingFast() {  
    // This test must complete in 5 seconds  
}
```

```
private static final int TIMEOUT = 2000;
```

```
@Test(timeout = TIMEOUT)  
public void testQuickOperation() {  
    // ...  
}
```

4. Testing for Exceptions

You can verify that a method throws an expected exception using:

```
@Test(expected = ArrayIndexOutOfBoundsException.class)
```

```
public void testBadIndex() {  
    ArrayList list = new ArrayList();  
    list.get(4); // Should throw  
}
```

If the exception is not thrown, the test fails automatically.

5. Setup and Teardown Methods

JUnit allows you to run setup or cleanup code before or after each test or once per test class.

Per-Test Methods: @Before

```
public void setUp() {  
    // Runs before each @Test  
}
```

@After

```
public void tearDown() {  
    // Runs after each @Test  
}
```








Per-Class Methods (run once): @BeforeClass

```
public static void initAll() {  
    // Runs once before all tests  
}
```

@AfterClass

```
public static void cleanAll() {  
    // Runs once after all tests  
}
```

6. Tips for Effective Testing

-  **Test one thing at a time: Avoid combining many assertions in one test.**
 -  **Use descriptive test names:**
`test_addDays_addJustOneDay_returnsCorrectDate()`.
 -  **Avoid logic inside tests: If/else or loops in test methods are discouraged.**
 -  **Always use timeouts: Prevent infinite loops or hangs.**
 -  **Include edge cases: Test for null, empty arrays, max/min values.**
 -  **Use helper methods or @Before to reduce duplication.**
 -  **Avoid catching exceptions manually unless needed.**
-

7. Example: JUnit Test with Assertions and Messages

```
public class DateTest {  
    @Test  
    public void test_addDays_addJustOneDay_1() {  
        Date actual = new Date(2050, 2, 15);  
        actual.addDays(1);  
        Date expected = new Date(2050, 2, 16);  
        assertEquals("adding one day to 2050/2/15", expected, actual);  
    }  
}
```

```
@Test(timeout = 2000)
public void test_addDays_timeout() {
    Date d = new Date(2050, 2, 15);
    d.addDays(14);
    assertEquals("month after +14 days", 3, d.getMonth());
}
```

```
@Test(expected = IllegalArgumentException.class)
public void test_invalidDateThrowsException() {
    new Date(2050, -5, 32); // Invalid date, should throw
}
}
```

9. Summary of Best Practices

- Write many small test methods, not one long test.
- Every test should have a clear assertion message.
- Use `assertEquals`, `assertTrue`, etc., not just `assertTrue(x == y)`.
- Use `@Before`, `@After` to remove repetitive code.
- Use `@Test(timeout = ...)` to prevent hanging tests.
- Use `@Test(expected = ...)` to test error handling.
- Test edge cases: null, zero, empty, boundary indices.
- Avoid logic in tests — tests are not code under test.
- Use meaningful names like `test_addDays_crossesMonth()`.

What's wrong with this?

```
public class DateTest {
    @Test
    public void test_addDays_addJustOneDay_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(1);
        Date expected = new Date(2050, 2, 16);
        assertEquals(
            "should have gotten " + expected + "\n" +
            " but instead got " + actual + "\n",
            expected, actual);
    }
    ...
}
```

What's wrong with this?

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        assertEquals(d.getYear(), 2050);
        assertEquals(d.getMonth(), 2);
        assertEquals(d.getDay(), 19);
    }

    @Test
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        assertEquals(d.getYear(), 2050);
        assertEquals(d.getMonth(), 3);
        assertEquals(d.getDay(), 1);
    }
}
```

10. Test-Driven Development (TDD)

Test-Driven Development is a methodology where you write the test before writing the actual code.

TDD Cycle:

1. Write a failing test.
2. Write code to pass the test.
3. Refactor the code.
4. Repeat.

Java Example:

```
public class Calculator {  
    public int multiply(int a, int b) {  
        return a * b;  
    }  
}
```

```
import static org.junit.Assert.assertEquals;  
import org.junit.Test;
```

```
public class CalculatorTest {  
    @Test  
    public void testMultiply() {  
        Calculator calc = new Calculator();  
        assertEquals(12, calc.multiply(3, 4));  
    }  
}
```

11. Bug Life Cycle

1. New – A bug is found.

2. Assigned – Assigned to a developer.
 3. Open – Developer starts working on it.
 4. Fixed – Code is fixed.
 5. Retested – Tester verifies the fix.
 6. Closed – Bug is resolved.
 7. Reopened – Bug still exists.
-

12. Example Test Cases for Login Feature

Test Case 1: Valid Login

- Input: Correct username and password
- Expected Output: Redirect to dashboard

Test Case 2: Invalid Login

- Input: Incorrect password
- Expected Output: "Invalid password" message

Test Case 3: Empty Fields

- Input: Empty username and password
 - Expected Output: "Fields cannot be empty" error
-

13. Best Practices in Software Testing

- Start testing early in the development cycle.
- Automate repetitive tests.
- Use clear and detailed test cases.
- Perform both positive and negative testing.
- Regularly update test scripts.
- Collaborate with developers and stakeholders.

15. Manual Testing vs Automated Testing

Manual Testing

Manual Testing is the process of manually executing test cases without the use of any automated tools. It is performed by human testers who interact with the software, simulate user behavior, and compare actual results with expected results.

Advantages of Manual Testing:

- Useful for exploratory testing, where human intuition and experience are important.
- Ideal for usability testing, to evaluate the user interface and user experience.
- No need for programming skills or automation tools.
- Easy to adapt to changing requirements on the fly.

Disadvantages of Manual Testing:

- Time-consuming and repetitive for large test cases.

- Error-prone due to human fatigue or oversight.
- Not suitable for regression testing or performance testing.
- Difficult to maintain consistency across test runs.

Example:

A QA engineer manually opens the login page, enters username and password, and verifies if the user is logged in correctly.

Automated Testing

Automated Testing involves writing scripts or using tools to execute test cases automatically. Once the tests are written, they can be run repeatedly and quickly across different environments.

Advantages of Automated Testing:

- Faster execution, especially for regression and load testing.
- Repeatable and consistent — same test behaves the same every time.
- Saves time and cost in the long term.
- Supports data-driven testing and parallel execution.
- Ideal for Continuous Integration/Continuous Deployment (CI/CD) environments.

Disadvantages of Automated Testing:

- High initial setup cost (writing scripts, tools, environment).
- Requires programming skills.
- Not suitable for UI/UX testing or situations requiring human judgment.
- Maintenance overhead when application changes frequently.

Conclusion

- Use Manual Testing when:
- You are testing new features.
- The UI is rapidly changing.
- Human observation is essential (e.g., UX, layout).
- Use Automated Testing when:
- You need to test the same flows frequently (regression).
- You work in an Agile/DevOps environment.
- You want to test large-scale data or performance.

In real-world projects, both types are combined for maximum coverage and efficiency — a practice known as Hybrid Testing.

16. Manual Testing: Advantages and Disadvantages

Advantages of Manual Testing

- Human perspective: Useful for UI/UX and visual layout testing.
- No tools required: Quick to start, no need for automation tools.
- Flexible: Easy to adapt to last-minute requirement changes.

- Exploratory testing: Ideal when detailed test cases are not available.

Disadvantages of Manual Testing

- Time-consuming: Especially with regression testing.
 - Prone to errors: Human mistakes can happen in repetitive tests.
 - Not scalable: Difficult to test large applications regularly.
 - No reusability: Test cases must be re-executed manually every time.
-

17. Automated Testing: Advantages and Disadvantages

Advantages of Automated Testing

- Speed: Tests can run very fast, even thousands at once.
- Repeatability: Same scripts can be reused across builds.
- Coverage: Can run many test cases across environments.
- Best for regression: Ensures that updates do not break existing functionality.
- Cost-efficient over time: Saves effort in the long run.

Disadvantages of Automated Testing

- High setup cost: Time, tools, and expertise required to start.
- Maintenance: Scripts must be updated when the app changes.
- Not ideal for UI/UX: Can't judge colors, spacing, or user feelings.

- False positives/negatives: Poor scripts can give wrong test results.

17. QA Testing Overview (Quality Assurance Testing)

What is QA Testing?

Quality Assurance (QA) Testing is the practice of monitoring the software development process to ensure that the final product meets the required quality standards. QA focuses on preventing defects by improving the process, while testing focuses on identifying defects after they have been introduced.

Key Goals of QA Testing:

- Prevent bugs and issues.
 - Ensure that the software meets functional and non-functional requirements.
 - Deliver a stable, usable, secure, and high-performance product.
 - Ensure compliance with business needs, contracts, or regulations.
-

Responsibilities of a QA Tester

A QA engineer or tester is responsible for:

- Understanding business requirements.

- Writing test plans and test cases.
 - Executing manual and automated tests.
 - Reporting and verifying bugs.
 - Ensuring that software works correctly on all platforms/environments.
 - Working closely with developers and product managers.
 - Participating in Agile/Scrum ceremonies (if Agile methodology is followed).
-

Types of QA Testing

1. Functional Testing

- Focuses on checking that the system does what it's supposed to do.
- Examples: login, sign-up, shopping cart.

2. Non-Functional Testing

- Tests performance, usability, reliability, scalability, etc.
- Examples:
 - Performance Testing
 - Security Testing
 - Compatibility Testing
 - Load Testing

3. Regression Testing

- Re-testing old features to ensure they work after new updates.

4. Smoke Testing

- Basic tests to verify that the major functionalities of the system are working.

5. Acceptance Testing

- Final level of testing before release, usually done by the customer or client.

QA Roles in a Software Team

- QA Analyst: Writes test plans and test cases based on requirements.
 - QA Engineer: May automate test cases and ensure processes are followed.
 - Test Engineer: Executes manual or automated tests.
 - QA Lead/Manager: Oversees QA activities, metrics, and reporting.
-

Typical QA Process

1. Requirement Analysis
 - Understand and clarify business and functional requirements.
 - Collaborate with stakeholders.
2. Test Planning

- Define the testing strategy, scope, environment, tools, and schedule.
3. Test Case Design
 - Create detailed test cases for different scenarios (positive, negative, edge cases).
 4. Test Environment Setup
 - Prepare the hardware/software/testing environment.
 5. Test Execution
 - Run manual or automated tests, document actual vs expected results.
 6. Bug Reporting & Tracking
 - Use tools like Jira or Bugzilla to log and track bugs.
 7. Regression Testing
 - Re-run tests after bug fixes or new features.
 8. Test Closure
 - Final reporting, lessons learned, process improvement.
-

Types of QA Documentation

- Test Plan
- Test Cases
- Test Summary Report
- Bug Reports
- Traceability Matrix (mapping test cases to requirements)

Types of QA Testing Approaches

1. Static Testing (QA-focused)
 - Code reviews
 - Requirements reviews
 - Design walkthroughs
 2. Dynamic Testing (Tester-focused)
 - Manual and automated testing while the software is running
-

QA in Agile and DevOps

In modern environments, QA is integrated into the Agile and DevOps workflows.

In Agile:

- QA is part of the Scrum team.
- Testing happens in sprints.
- QA contributes to User Story acceptance criteria.
- Emphasis on Test-Driven Development (TDD) and Behavior-Driven Development (BDD).

In DevOps:

- QA ensures tests are integrated into the CI/CD pipelines.

- Focus on Shift-Left testing — testing earlier in the development cycle.
- Supports automated smoke, unit, and regression tests on every build.

Challenges in QA

- Changing or unclear requirements
 - Tight deadlines and lack of time for testing
 - Insufficient test data or environment
 - Poor communication between devs and testers
 - Lack of test automation knowledge or resources
-

Best Practices in QA

- Start testing early (Shift Left)
- Write clear and maintainable test cases
- Automate where possible but don't skip manual where necessary
- Use version control for test scripts
- Continuously review and improve processes
- Prioritize risk-based testing