**Software testing**

**Software testing** is the process of evaluating and verifying that a software application or system works as expected. The main goals of software testing are to:

1. **Ensure the software is functional** – It performs the tasks it's supposed to do.
2. **Identify defects (bugs)** – Find errors in the code or behaviour.
3. **Ensure reliability and performance** – Confirm it runs smoothly under various conditions.
4. **Validate user requirements** – Make sure it meets the needs and expectations of users or clients.

**Types of Software Testing**

Software testing is broadly classified into two main types:

**1. Manual Testing**

* Testers perform tests manually without using any automation tools.
* Examples: exploratory testing, usability testing.

**2. Automated Testing**

* Uses software tools to run tests automatically.
* Examples: unit testing, regression testing using tools like Selenium, JUnit, etc.

**Common Testing Methods**

* **Unit Testing** – Tests individual units or components of the software.
* **Integration Testing** – Tests combined parts of an application to ensure they work together.
* **System Testing** – Tests the complete system as a whole.
* **Acceptance Testing** – Checks if the system meets business requirements; often done by the client or end-user.
* **Regression Testing** – Ensures that new code changes haven’t broken existing functionality.
* **Performance Testing** – Measures responsiveness, speed, and stability.
* **Security Testing** – Checks for vulnerabilities and ensures data protection.

**Why Is Software Testing Important?**

* Prevents costly bugs in production.
* Improves software quality.
* Enhances user satisfaction.
* Reduces maintenance costs.
* Ensures compliance with standards or regulations.

**Levels of Testing**

In software testing, **"levels of testing"** refer to the different stages at which testing is performed during the software development lifecycle (SDLC). Each level focuses on a specific part of the software system to ensure overall quality.

Here are the **four main levels of software testing**:

**1. Unit Testing**

* **What it tests:** Individual components or functions (units) of the code.
* **Who performs it:** Usually developers.
* **Tools used:** JUnit, NUnit, PyTest, etc.
* **Goal:** Ensure that each unit of the software performs as expected.

*Example: Testing a function that calculates the total price of a shopping cart.*

**2. Integration Testing**

* **What it tests:** Interaction between integrated units/modules.
* **Who performs it:** Developers or testers.
* **Types:** Top-down, bottom-up, sandwich, and big bang.
* **Tools used:** JUnit, TestNG, Postman (for APIs), etc.
* **Goal:** Verify that modules or services work together correctly.

*Example: Testing the interaction between a login module and a user dashboard module.*

**3. System Testing**

* **What it tests:** The complete and integrated software system.
* **Who performs it:** Independent testing team (QA).
* **Types:** Functional testing, usability testing, performance testing, etc.
* **Goal:** Validate the overall behaviour and compliance with the specified requirements.

*Example: Testing the whole e-commerce platform — login, browse, cart, payment, and order confirmation.*

**4. Acceptance Testing**

* **What it tests:** The system’s readiness for delivery/use.
* **Who performs it:** End users, clients, or testers.
* **Types:** Alpha testing (internal), Beta testing (external).
* **Goal:** Confirm that the software meets business needs and is acceptable to the user.

*Example: A client tests the software in a real-world scenario to decide whether to accept it.*

**Summary Table:**

| **Level** | **Focus Area** | **Performed By** | **Purpose** |
| --- | --- | --- | --- |
| **Unit Testing** | Individual functions/components | Developers | Ensure code correctness |
| **Integration Testing** | Interactions between units | Developers/Testers | Validate module cooperation |
| **System Testing** | Whole application/system | QA/Testers | Check system functionality |
| **Acceptance Testing** | Final product vs. requirements | End Users/Clients | Approve for release or production |

                ------------------------ ----------------------------------------------------------------------------------------

**Functional**, **Non-functional Testing Types**

Software testing can be categorized into several **types** based on the purpose, approach, and scope of testing. Here’s a comprehensive list of the **main types of software testing**, grouped into **functional**, **non-functional**, **maintenance**, and **other specialized testing** types:

**A. Functional Testing Types**

These tests verify that the software behaves as expected based on requirements.

1. **Unit Testing**
   * Tests individual components or functions.
   * Usually done by developers.
2. **Integration Testing**
   * Tests the interaction between integrated modules.
   * Types: Top-down, Bottom-up, Big bang, Incremental.
3. **System Testing**
   * Tests the complete system as a whole.
4. **Acceptance Testing**
   * Verifies the system meets business requirements.
   * Types:
     + **Alpha Testing**: By internal team.
     + **Beta Testing**: By end users.
5. **Smoke Testing**: - A basic test to check major functionalities after a build.
6. **Sanity Testing**: - A narrow regression test focusing on specific bug fixes or functionalities.
7. **Regression Testing**: - Ensures new changes haven’t broken existing functionality.
8. **Interface Testing**: - Tests the interaction between systems or software components.
9. **End-to-End Testing**: - Simulates real user scenarios to validate the system flow.

**B. Non-Functional Testing Types**

These tests validate aspects unrelated to specific behaviours or functions.

1. **Performance Testing**
   * Measures responsiveness, stability, and scalability.
   * Includes:
     + **Load Testing**: Under expected load.
     + **Stress Testing**: Beyond normal load.
     + **Spike Testing**: Sudden spikes in load.
     + **Soak (Endurance) Testing**: Long-duration load testing.
2. **Security Testing**
   * Checks for vulnerabilities and data protection.
   * Includes: penetration testing, ethical hacking, vulnerability scanning.
3. **Usability Testing**: - Evaluates user experience, interface design, and navigation.
4. **Compatibility Testing**: - Checks performance across different browsers, OS, devices, etc.
5. **Reliability/Recovery Testing**: - Ensures software recovers from crashes, hardware failures, or network issues.
6. **Maintainability Testing**: - Measures how easy it is to maintain and update the software.
7. **Portability Testing**: - Verifies software works across different environments or platforms.
8. **Compliance Testing**: - Ensures the software adheres to industry or legal standards.

**Key Differences**

| **Aspect** | **Functional Testing** | **Non-Functional Testing** |
| --- | --- | --- |
| Purpose | Verify software features work correctly | Verify software quality attributes |
| Focus | *What* the system does | *How* the system performs |
| Examples | Unit, Integration, System, UAT | Performance, Load, Stress, Security |
| Nature | Usually black box testing | Can be black box or specialized testing |
| Outcome | Pass/fail based on requirements | Metrics (e.g., response time, throughput, usability scores) |

**C. Maintenance Testing Types**

Used after deployment or during upgrades.

1. **Regression Testing** (again): - Re-run of existing test cases after updates.
2. **Maintenance Testing**: - Includes impact analysis and testing for enhancements or bug fixes.

**D. Specialized/Advanced Testing Types**

1. **Exploratory Testing**: - Tester explores the application freely without predefined test cases.
2. **Ad-hoc Testing**: - Informal testing to find defects randomly.

* Ad-hoc testing: It is an informal testing method that is performed without any pre-defined test cases.
* It is basically rely on tester's experience and also the knowledge he possess to explore the functionalities of the application that are being missed during formal  
  testing methods.

1. **Mutation Testing**: - Deliberate errors are introduced to evaluate the effectiveness of test cases.
2. **A/B Testing**: - Compares two versions of a webpage or an app to see which performs better.
3. **Static Testing**: - Code is reviewed without executing it (e.g., code review, static analysis).
4. **Dynamic Testing**: - Code is executed during testing.

**Various Testing Techniques**

**Black Box Testing**

* **Focus**: Functionality of the application without knowledge of internal code.
* **Tester’s Knowledge**: No access to internal code or architecture.
* **Main Goal**: Validate outputs against inputs.
* **Used for**: Functional testing, system testing, acceptance testing.
* **Examples**:
  + Input validation
  + UI testing
  + Functional test cases

**White Box Testing (aka Clear Box or Glass Box Testing)**

* **Focus**: Internal structure and logic of the code.
* **Tester’s Knowledge**: Full access to source code and design.
* **Main Goal**: Verify internal operations and logic.
* **Used for**: Unit testing, security testing, code coverage.
* **Examples**:
  + Statement coverage
  + Branch coverage
  + Path testing

**Grey Box Testing**

* **Focus**: Combines elements of both black box and white box testing.
* **Tester’s Knowledge**: Partial knowledge of internal code/architecture.
* **Main Goal**: Test both user-level functionality and internal logic.
* **Used for**: Integration testing, penetration testing, end-to-end testing with insight.
* **Examples**:
  + Session management testing
  + API security with known endpoints

**Similar or Related Testing Techniques:**

**Glass Box Testing**

* Another name for White Box Testing.

**Opaque Box Testing**

* Another name for Black Box Testing.

**Clear Box Testing**

* Synonym of White Box Testing.

**Transparent Box Testing**

* Also refers to White Box Testing.

**Extended Concepts & Related Testing Types**

| **Testing Type** | **Description** | **Relation** |
| --- | --- | --- |
| **Unit Testing** | Testing individual units of code | White Box |
| **Functional Testing** | Validating application functionality against requirements | Black Box |
| **Integration Testing** | Testing how modules work together | Grey Box / White Box |
| **System Testing** | Testing the complete integrated system | Black Box |
| **Acceptance Testing (UAT)** | Validating the system against business needs | Black Box |
| **Regression Testing** | Ensuring new changes don’t affect existing features | Black/Grey Box |
| **Security Testing** | Checking for vulnerabilities | Grey Box (esp. in penetration testing) |
| **Penetration Testing** | Simulated attacks on system | Grey Box |
| **Code Coverage Analysis** | Measuring how much code is tested | White Box |
| **Mutation Testing** | Introducing small changes in code to test the test cases | White Box |

**Summary Table**

| **Feature** | **Black Box** | **White Box** | **Grey Box** |
| --- | --- | --- | --- |
| Knowledge of Code | No | Yes | Partial |
| Testing Type | Functional | Structural | Combination |
| Example Use Case | UI Testing | Unit Testing | Security Testing |
| Tester Role | End-user/tester | Developer/Tester | Developer/Test Analyst |
| Speed to Execute | Faster | Slower (in-depth) | Moderate |

**What is JUnit in Java?**

JUnit is a **unit testing framework** for the Java programming language. It is used to write and run repeatable tests to ensure that individual parts (units) of your program (usually methods in classes) work as expected.

JUnit is part of the xUnit family of testing frameworks and is one of the most popular testing tools in Java development. It is widely used for Test-Driven Development (TDD).

**Key Features of JUnit:**

* Annotations to identify test methods
* Assertions to test expected vs actual results
* Test runners to execute tests
* Integration with build tools (e.g., Maven, Gradle) and IDEs (e.g., IntelliJ, Eclipse)

**Common JUnit Annotations**

| **Annotation** | **Description** |
| --- | --- |
| @Test | Marks a method as a test method |
| @BeforeEach | Runs before each test method |
| @AfterEach | Runs after each test method |
| @BeforeAll | Runs once before all tests in the class |
| @AfterAll | Runs once after all tests in the class |
| @Disabled | Disables a test method |
| @DisplayName | Sets a custom name for the test method |

**Common JUnit Methods (Assertions)**

*JUnit provides a set of assertion methods to check values.*

| **Method** | **Purpose** |
| --- | --- |
| assertEquals(expected, actual) | Checks if two values are equal |
| assertNotEquals(expected, actual) | Checks if two values are not equal |
| assertTrue(condition) | Checks if the condition is true |
| assertFalse(condition) | Checks if the condition is false |
| assertNull(value) | Checks if the value is null |
| assertNotNull(value) | Checks if the value is not null |
| assertThrows(Exception.class, () -> ...) | Expects an exception to be thrown |
| assertAll(...) | Group multiple assertions |
| fail() | Forces a test to fail |

**Example JUnit Test**

import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.\*;

public class CalculatorTest {

   @Test

   void testAddition() {

       int result = 2 + 3;

       assertEquals(5, result);

   }

   @Test

   void testIsEven() {

       boolean isEven = (4 % 2 == 0);

      assertTrue(isEven);

   }

   @Test

   void testNullValue() {

       String str = null;

       assertNull(str);

   }

}

| **Case** | **Suggested Delta** |
| --- | --- |
| High-precision math | 0.000001 or smaller |
| Typical calculations | 0.0001 |
| Approximate comparisons | 0.01 or larger |
| Percentages or estimations | 0.1 or 1.0 |

Throws:-

Calculator.java  
==============  
public int divide(int a,int b){  
  if(b==0) throw new IllegalArgumentException("Cannot divide by zero");  
  return a/b;    
}

CalculatorTest.java  
===================  
@Test  
public void testDivide(){  
  Exception ex = assertThrows  
  (IllegalArgumentException.class,  
     ()->calculator.divide(5,0);  
  assertEquals("cannot Divide By Zero",ex.getMessage());  
  });

**package** main.java;

**import** java.sql.\*;

**public** **class** Assessment {

**public** Connection con = **null**;

**public** **void** connect() **throws** SQLException {

      con = DriverManager.*getConnection*("jdbc:mysql://localhost:3306/junit\_test", "root", "Wasteoftime@3110");

    }

**public** **int** insertStudent(**int** id, String name) **throws** SQLException {

        PreparedStatement stmt = con.prepareStatement("INSERT INTO student VALUES (?, ?)");

        stmt.setInt(1, id);

        stmt.setString(2, name);

**return** stmt.executeUpdate();

    }

**public** **int** updateStudent(**int** id, String newName) **throws** SQLException {

        PreparedStatement stmt = con.prepareStatement("UPDATE student SET name=? WHERE id=?");

        stmt.setString(1, newName);

        stmt.setInt(2, id);

**return** stmt.executeUpdate();

    }

**public** **int** deleteStudent(**int** id) **throws** SQLException {

        PreparedStatement stmt = con.prepareStatement("DELETE FROM student WHERE id=?");

        stmt.setInt(1, id);

**return** stmt.executeUpdate();

    }

**public** **void** closing() **throws** SQLException {

**if** (con != **null**) con.close();

    }

}

**package** test.java;

**import** main.java.Assessment;

**import** org.junit.jupiter.api.\*;

**import** **static** org.junit.jupiter.api.Assertions.\*;

**import** java.sql.Connection;

**public** **class** AssessmentTest {

**static** Assessment *assess*;

**static** Connection *con*;

    @BeforeEach

**public** **void** setUp() **throws** Exception {

*assess* = **new** Assessment();

*assess*.connect();

        System.***out***.println("Connection created....");

    }

    @Test

**public** **void** testInsertStudent() **throws** Exception {

**int** result = *assess*.insertStudent(1, "Sai");

*assertEquals*(1, result);

    }

    @Test

**public** **void** testUpdateStudent() **throws** Exception {

*assess*.insertStudent(2, "Bharath");

**int** result = *assess*.updateStudent(2, "Divya");

*assertEquals*(1, result);

    }

    @Test

**public** **void** testDeleteStudent() **throws** Exception {

*assess*.insertStudent(3, "Meghu");

**int** result = *assess*.deleteStudent(3);

*assertEquals*(1, result);

    }

    @AfterEach

**public** **void** closing() **throws** Exception {

*assess*.closing();

    }

}

**Test Planning and Design**

*Test Planning and Test Design are two critical stages in the software testing lifecycle.*

***They help ensure that testing is thorough, organized, and aligned with project goals.***

**1. Test Planning**

**What is Test Planning?**

Test planning is the process of defining the scope, objectives, approach, and resources required for testing activities. It results in a Test Plan document, which acts as a blueprint for all testing efforts.

**Key Components of a Test Plan:**

| **Section** | **Description** |
| --- | --- |
| **Test Objectives** | What is being tested and why. |
| **Test Scope** | What **will** and **won’t** be tested. |
| **Test Strategy** | Overall testing approach (manual, automated, black box, etc.). |
| **Test Environment** | Hardware, software, and network setups required. |
| **Test Schedule** | Timeline for each test activity. |
| **Resources & Roles** | Who will perform what tasks (testers, developers, tools). |
| **Entry & Exit Criteria** | Conditions to start and stop testing. |
| **Risk Management** | Potential issues and mitigation strategies. |
| **Deliverables** | What will be produced (test cases, bug reports, metrics). |

**Example:**

*In a* ***banking app*** *project, the test plan may state:*  
**"Functional testing will be performed on the login, account summary, and fund transfer modules using a black-box approach. Testing will begin on Sept 1 and end by Sept 20. Selenium will be used for automation."**

**2. Test Design**

**What is Test Design?**

Test design is the process of creating detailed test cases and test data based on the requirements and specifications defined in the planning phase.

**Key Activities in Test Design:**

| **Activity** | **Description** |
| --- | --- |
| **Requirement Analysis** | Understand what needs to be tested. |
| **Test Scenario Identification** | High-level descriptions of what to test. |
| **Test Case Design** | Detailed steps, inputs, expected outcomes. |
| **Test Data Creation** | Input values and environmental data required for tests. |
| **Traceability Matrix** | Mapping test cases to requirements for full coverage. |

**Example of a Test Case:**

| **Field** | **Example** |
| --- | --- |
| **Test Case ID** | TC\_Login\_001 |
| **Description** | Verify login with valid credentials |
| **Steps** | 1. Open login page 2. Enter valid username and password 3. Click login |
| **Test Data** | Username: user123 Password: pwd123 |
| **Expected Result** | User is redirected to the dashboard |
| **Actual Result** | (Filled during testing) |
| **Status** | Pass(Green Bar)/Fail(Red Bar) |

**Summary: Test Planning vs. Test Design**

| **Aspect** | **Test Planning** | **Test Design** |
| --- | --- | --- |
| **Goal** | Define *what* and *how* to test | Create *detailed* test cases |
| **Focus** | Strategy, resources, scheduling | Test scenarios, steps, test data |
| **Output** | Test Plan document | Test cases, test scripts, test data |
| **Performed By** | Test Manager / Lead | QA Engineers / Testers |

**Test Execution in Software Testing**

**Test Execution** is the phase where the actual testing takes place — test cases are run, results are recorded, and any deviations (bugs or defects) are logged for fixing.

**What is Test Execution?**

Test execution is the process of running the test cases designed during the test design phase and comparing actual results with expected results to determine if the software behaves as intended.

**Key Activities in Test Execution**

| **Step** | **Description** |
| --- | --- |
| **1. Test Environment Setup** | Make sure the hardware/software environment is ready (e.g., test server, database, network, external jar files to be imported). |
| **2. Test Case Execution** | Run manual or automated test cases. |
| **3. Record Results** | Log actual results and compare with expected outcomes. |
| **4. Defect Reporting** | If the test fails, log a **defect/bug** in a tracking tool (e.g., Jira, Bugzilla). |
| **5. Retesting and Regression Testing** | Once a defect is fixed, re-run test cases and check related functionality. |
| **6. Test Status Reporting** | Track progress (e.g., % passed, failed, blocked) and report to stakeholders. |

**Example:**

| **Test Case ID** | **Description** | **Expected Result** | **Actual Result** | **Status** | **Defect ID** |
| --- | --- | --- | --- | --- | --- |
| TC\_Login\_001 | Valid login credentials | Redirect to dashboard | Redirect to dashboard | Pass | – |
| TC\_Login\_002 | Invalid password | Show error message | Login page reloads | Fail | BUG\_105 |
| TC\_Logout\_003 | Click logout button | Redirect to login screen | Redirect to login screen | Pass | – |

**Tools Commonly Used in Test Execution**

| **Type** | **Tools** |
| --- | --- |
| **Test Management** | TestRail, Zephyr, Xray, HP ALM |
| **Bug Tracking** | Jira, Bugzilla, Mantis |
| **Automation** | Selenium, Cypress, JUnit, TestNG |
| **CI/CD Execution** | Jenkins, GitLab CI/CD |

**Test Execution Status Terms**

| **Status** | **Meaning** |
| --- | --- |
| **Pass** | Test ran successfully and matched expectations. |
| **Fail** | Test ran but didn’t meet the expected result. |
| **Blocked** | Test couldn’t be executed due to an external issue (e.g., network down). |
| **Not Run** | Test case wasn’t executed yet. |
| **Skipped** | Test was intentionally skipped (e.g., not relevant for this cycle). |

**Best Practices for Test Execution**

* Prioritize critical test cases first.
* Log defects clearly with steps, screenshots, and logs.
* Communicate regularly with the development team.
* Continuously update test status and reports.
* Track defect fixes and retest promptly.

**Defect Management in Software Testing**

**Defect Management** is the structured process of identifying, documenting, prioritizing, tracking, and resolving defects (bugs) found during the software development and testing life cycle.

**What Is a Defect (Bug)?**

A **defect** is a flaw or error in a software product that causes it to behave incorrectly or unexpectedly, not meeting the requirements or specifications.

**Defect Management Process Steps**

Defect Life Cycle (also called the Bug Life Cycle):

| **Step** | **Description** |
| --- | --- |
| **1. Defect Detection** | A tester or user finds unexpected behaviour in the software. |
| **2. Defect Logging** | The issue is logged in a **bug tracking tool** (e.g., Jira, Bugzilla) with details. |
| **3. Defect Triage** | Team analyzes, validates, and assigns priority and severity. |
| **4. Defect Assignment** | The bug is assigned to a developer to fix. |
| **5. Defect Fixing** | The developer investigates and resolves the issue. |
| **6. Retesting** | The tester re-executes the failed test cases to verify the fix. |
| **7. Closure** | If the fix works, the defect is marked **Closed**. If not, it's **Reopened**. |

**Defect Life Cycle (Status Flow)**

New → Assigned → Open → Fixed → Retest → Verified → Closed

                                  ↘ Reopen ↖

**Key Defect Attributes**

| **Attribute** | **Description** |
| --- | --- |
| **Defect ID** | Unique identifier |
| **Title** | Short summary |
| **Description** | Detailed explanation with steps to reproduce |
| **Severity** | Impact on system (e.g., Critical, Major, Minor) |
| **Priority** | Urgency to fix (e.g., High, Medium, Low) |
| **Status** | Current state (e.g., New, Open, Closed) |
| **Environment** | Where it occurred (e.g., Browser, OS) |
| **Attachments** | Screenshots, logs, video evidence |

                                            -----

**A simple Java integration testing example.**

We’ll test the integration of two components:

**· A UserRepository that stores users in a file:-**

* A UserRepository that stores users in a file.
* Handles data persistence. In this example, it stores and retrieves user data from a file.
* Think of it as the “data access” layer.
* It knows where and how to save data, but it doesn’t decide *why* or *when* to do so.

**· A UserService that uses this repository:-**

* A UserService that uses this repository.

**UserService –** Business logic layer that depends on UserRepository.

* It contains methods like addUser() or getUserByName() that internally call the repository methods.
* It focuses on *what to do* with user data, not *how to store it*.

**The integration test will check that UserService and UserRepository work correctly together (integration testing = testing the interaction between real components).**

--------------------------------------------------------------------------------------------

Project Structure

/project

├── User.java

├── UserRepository.java

├── UserService.java

├── IntegrationTest.java

**1. User.java**

public class User {

private String id;

private String name;

public User(String id, String name) {

this.id = id;

this.name = name;

}

// Getters

public String getId() { return id; }

public String getName() { return name; }

}

----------------------------------------------------------------------------------------------------------

**2. UserRepository.java**

import java.io.\*;

import java.util.\*;

public class UserRepository {

private File storageFile;

public UserRepository(String filePath) {

this.storageFile = new File(filePath);

}

public void save(User user) throws IOException {

try (FileWriter writer = new FileWriter(storageFile, true)) {

writer.write(user.getId() + "," + user.getName() + "\n");

}

}

public List<User> findAll() throws IOException {

List<User> users = new ArrayList<>();

if (!storageFile.exists()) return users;

try (BufferedReader reader = new BufferedReader(new FileReader(storageFile))) {

String line;

while ((line = reader.readLine()) != null) {

String[] parts = line.split(",");

users.add(new User(parts[0], parts[1]));

}

}

return users;

}

public void clear() throws IOException {

new FileWriter(storageFile).close(); // clear contents

}

}

----------------------------------------------------------------------------------------------------------------

**3. UserService.java**

import java.io.IOException;

import java.util.List;

public class UserService {

private UserRepository repository;

public UserService(UserRepository repository) {

this.repository = repository;

}

public void registerUser(String id, String name) throws IOException {

User user = new User(id, name);

repository.save(user);

}

public List<User> getAllUsers() throws IOException {

return repository.findAll();

}

}

-------------------------------------------------------------------------------------------------------------------------------------

**4. IntegrationTest.java (INTEGRATION TEST)**

import java.io.IOException;

import java.util.List;

public class IntegrationTest {

public static void main(String[] args) throws IOException {

// Setup

String testFilePath = "test\_users.txt";

UserRepository repo = new UserRepository(testFilePath);

repo.clear(); // make sure the file is empty before testing

UserService service = new UserService(repo);

// Test

service.registerUser("1", "ABC");

service.registerUser("2", "DEF");

List<User> users = service.getAllUsers();

assert users.size() == 2 : "Expected 2 users";

assert users.get(0).getName().equals("ABC");

assert users.get(1).getName().equals("DEF");

System.out.println("Integration test passed!");

// Clean up

repo.clear();

}

}

------------------------------------------------------------------

**How to Run**

Compile and run with:

javac \*.java

java IntegrationTest