**Principles of Unit testing**

The unit testing should follow the below principles

**Test Isolation:** Unit tests should be independent and isolated from external dependencies, such as databases, networks, or file systems. Dependencies should be mocked or stubbed to ensure that tests focus solely on the behavior of the unit under test. This allows for consistent and reliable test results.

**Test Focused on a Single Unit:** Each unit test should focus on testing a specific behavior or functionality of a single unit (e.g., a class or a method). This principle ensures that tests are targeted and provide clear feedback on the behavior of the unit being tested.

**Test Driven Development (TDD):** Following the TDD approach, unit tests are written before the corresponding production code. This helps in defining the desired behavior and acts as a guide during the implementation. The tests initially fail and then are incrementally made to pass by implementing the necessary functionality.

**Test Readability and Maintainability:** Unit tests should be easy to read, understand, and maintain. Writing clear and expressive test code with meaningful names, proper test structures, and minimal duplication improves the maintainability of the tests. This ensures that tests remain valuable and effective throughout the software development lifecycle.

**Fast Execution:** Unit tests should be lightweight and execute quickly. Fast-running tests encourage developers to run them frequently, enabling quick feedback on code changes. This rapid feedback loop supports agile development practices and allows for the early detection of defects.

**Assertions and Expected Outcomes:** Unit tests should include assertions to verify the expected behavior of the unit under test. Assertions check if the actual results match the expected outcomes. Well-designed assertions help validate that the unit is functioning correctly and provide clarity on what is being tested.

**Test Coverage:** Aim for high test coverage, striving to test as many code paths and scenarios as possible. Adequate coverage helps ensure that different branches, conditions, and edge cases are accounted for, reducing the risk of undiscovered defects.

**Continuous Integration and Continuous Testing:** Incorporating unit tests into a continuous integration and continuous testing process allows for automatic and frequent execution of tests. This ensures that tests are run consistently, providing continuous feedback on code quality and helping to catch issues early.

**Junit**

JUnit framework consists of

1. Fixtures
2. Test suites
3. Test runners
4. Junit classes

**Fixtures** is a fixed state of a set of objects used as a baseline for running tests. The purpose of a test fixture is to ensure that there is a well-known and fixed environment in which tests are run so that results are repeatable. It includes

* setUp() method, which runs before every test invocation.
* tearDown() method, which runs after every test method.

public class FixtureExampleTest extends FixtureExample {

protected int value1, value2;

// assigning the values

protected void setUp(){

value1 = 3;

value2 = 3;

}

// test method to add two values

public void testAdd(){

double result = value1 + value2;

assertTrue(result == 6);

}

}

**Test Suites**

A test suite is a collection of tests grouped together and run as a single unit. We use them to organize tests into logical groups, such as tests for a specific component or application feature.

@Suite

@SuiteDisplayName("My Test Suite")

public class JUnitTestSuite {

}

@RunWith(JUnitPlatform.class)

@SuiteDisplayName("My Test Suite")

public class JUnitTestSuite {

}

***Important Test Suites Annotations:***

**@SelectClasses**

The most common way to select tests for our test suites is to specify test classes using the @SelectClasses annotation:

@Suite

@SelectClasses({ClassOneUnitTest.class, ClassTwoUnitTest.class})

public class JUnitTestSuite {

}

The test suite now executes all @Test marked methods from both classes.

**@SelectPackages**

Instead of specifying the list of classes, we can use @SelectPackages to provide packages for test scanning:

@Suite

@SelectPackages({"com. example.testsuite", "com.example.testsuitetwo"})

public class JUnitTestSuite {

}

*Notably, this annotation also executes all classes from sub-packages.*

**@IncludePackages and @ExcludePackages**

We can use @IncludePackages and @ExcludePackages annotations, respectively to mention the packages to include and exclude

@Suite

@SelectPackages({"com. example.testsuite")

@IncludePackages("com. example.testsuite.subpackage")

public class JUnitTestSuite {

}

**@IncludeClassNamePatterns and @ExcludeClassNamePatterns**

If we don't want to specify inclusion rules using packages, we can use @IncludeClassNamePatterns and @ExcludeClassNamePatterns annotations and implement regex checking for class names:

@Suite

@SelectPackages("com.example.testsuite")

@IncludeClassNamePatterns("com.example.testsuite.Class.\*UnitTest")

@ExcludeClassNamePatterns("com.example.testsuite.ClassTwoUnitTest")

public class JUnitTestSuite {

}

Example-

@RunWith(Suite.class)

@SuiteClasses({ CalculatorTest.class, CalculatorTest1.class })

public class AllTests {

}

**Test Runners**

Test Runner is a class that runs a test or group of tests. This class is the class with main[ ] method which calls the JunitCore class of junit framework. JUnitCore class is responsible for running tests. This class contains a method called JUnitCore.RunClasses(Class Testclass)

public class TestMain {

public static void main(String[] args) {

// TODO Auto-generated method stub

Result result= JUnitCore.*runClasses*(TestSuite.class);

if(result.wasSuccessful()){

System.*out*.println("All tests passed successfully");

}

else{

System.*out*.println("Test Failures:");

}

for (Failure failure : result.getFailures()) {

System.*out*.println(failure.toString());

}

}

}

**Junit Test Cases**

Important annotations to be noted for Junit test cases are:

* **@Test :** This annotation is a replacement of org.junit.TestCase which indicates that public void method to which it is attached can be executed as a test Case.
* **@Before:** This annotation is used if you want to execute some statement such as preconditions before each test case.
* **@BeforeClass:** This annotation is used if you want to execute some statements before all the test cases for e.g. test connection must be executed before all the test cases.
* **@After:** This annotation can be used if you want to execute some statements after each Test Case for e.g resetting variables, deleting temporary files ,variables, etc.
* **@AfterClass**: This annotation can be used if you want to execute some statements after all test cases for e.g. Releasing resources after executing all test cases.
* **@Ignores:** This annotation can be used if you want to ignore some statements during test execution for e.g. disabling some test cases during test execution.
* **@Test(timeout=500):** This annotation can be used if you want to set some timeout during test execution for e.g. if you are working under some SLA (Service level agreement), and tests need to be completed within some specified time
* **@Test(expected=IllegalArgumentException.class)**: This annotation can be used if you want to handle some exception during test execution. For, e.g., if you want to check whether a particular method is throwing specified exception or not.
* **@Assert** : Assertions are utility methods to support asserting conditions in tests.

**Types of Assert annotations**:

1. void assertEquals(boolean expected, boolean actual) à Checks that two primitives/objects are equal.
2. void assertTrue(boolean condition) à Checks that a condition is true.
3. void assertFalse(boolean condition) à Checks that a condition is false.
4. void assertNotNull(Object object) à Checks that an object isn't null.
5. void assertNull(Object object) à Checks that an object is null.
6. void assertSame(object1, object2) à The assertSame() method tests if two object references point to the same object.
7. void assertNotSame(object1, object2) à The assertNotSame() method tests if two object references do not point to the same object.
8. void assertArrayEquals(expectedArray, resultArray) à The assertArrayEquals() method will test whether two arrays are equal to each other.

* **@AssertAll**

This is used to execute all the assert statements together.

Example-

Address address = unitUnderTest.methodUnderTest();

assertEquals("Pune", address.getCity());

assertEquals("Wakad Street", address.getStreet());

assertEquals("500065", address.getPincode());

*Please mind the below while comparing JSON response in JUNIT*

**JSONCompareMode.STRICT** : Extended fields are not allowed. Field ordering must match.

**JSONCompareMode.LENIENT** : Extended fields are allowed. Field ordering can differ.

**JSONCompareMode.NON\_EXTENSIBLE** : Extended fields are not allowed. Field ordering can differ.

**JSONCompareMode.STRICT\_ORDER** : Extended fields are allowed. Field ordering must match.

***Junit Naming Conventions***

Ensure the below while writing Junit test cases:

**Method Names**: Test methods should have descriptive names that indicate what is being tested. They often start with the word "test" followed by a brief description of the behavior being tested. For example: testAddition, testCalculateTotal, etc.

**Visibility:** Test methods should be declared as public to ensure they can be accessed by the JUnit test runner.

**Return Type:** Test methods should have a void return type. JUnit does not rely on the return value of test methods; it determines the success or failure based on assertions and exceptions thrown within the test method.

**Parameters:** Test methods do not typically take any parameters. However, if necessary, JUnit allows test methods to accept parameters as long as the test runner supports it (e.g., parameterized tests in JUnit 4).

**Annotations:** Test methods should be annotated with the appropriate JUnit annotations such as @Test, @Before, @After, etc. These annotations indicate the purpose of the method and how it should be executed by the test runner.