Unit Testing with JUnit

1. The purpose of software tests

1.1. What are software tests?

A software test is a piece of software, which executes another piece of software. It validates if that code results in the expected state (state testing) or executes the expected sequence of events (behavior testing).

1.2. Why are software tests helpful?

Software unit tests help the developer to verify that the logic of a piece of the program is correct.

Running tests automatically helps to identify software regressions introduced by changes in the source code. Having a high test coverage of your code allows you to continue developing features without having to perform lots of manual tests.

2. Testing terminology

2.1. Code (or application) under test

The code which is tested is typically called the *code under test*. If you are testing an application, this is called the *application under test*.

2.2. Test fixture

A *test fixture* is a fixed state in code which is tested used as input for a test. Another way to describe this is a test precondition.

For example, a test fixture might be a a fixed string, which is used as input for a method. The test would validate if the method behaves correctly with this input.

2.3. Unit tests and unit testing

A *unit test* is a piece of code written by a developer that executes a specific functionality in the code to be tested and asserts a certain behavior or state.

The percentage of code which is tested by unit tests is typically called *test coverage*.

A unit test targets a small unit of code, e.g., a method or a class. External dependencies should be removed from unit tests, e.g., by replacing the dependency with a test implementation or a (mock) object created by a test framework.

Unit tests are not suitable for testing complex user interface or component interaction. For this, you should develop integration tests.

2.4. Integration tests

An *integration test* aims to test the behavior of a component or the integration between a set of components. The term *functional test* is sometimes used as synonym for integration test. Integration tests check that the whole system works as intended, therefore they are reducing the need for intensive manual tests.

These kinds of tests allow you to translate your user stories into a test suite. The test would resemble an expected user interaction with the application.

2.5. Performance tests

Performance tests are used to benchmark software components repeatedly. Their purpose is to ensure that the code under test runs fast enough even if it’s under high load.

2.6. Behavior vs. state testing

A test is a behavior test (also called interaction test) if it checks if certain methods were called with the correct input parameters. A behavior test does not validate the result of a method call.

State testing is about validating the result. Behavior testing is about testing the behavior of the application under test.

If you are testing algorithms or system functionality, in most cases you may want to test state and not interactions. A typical test setup uses mocks or stubs of related classes to abstract the interactions with these other classes away Afterwards you test the state or the behavior depending on your need.

2.7. Testing frameworks for Java

There are several testing frameworks available for Java. The most popular ones are JUnit and TestNG

This description focuses on JUnit. It covers both JUnit 4.x and JUnit 5.

2.8. Where should the test be located?

Typical, unit tests are created in a separate project or separate source folder to keep the test code separate from the real code. The standard convention from the Maven and Gradle build tools is to use:

- src/main/java - for Java classes
- src/test/java - for test classes

3. Using JUnit

3.1. The JUnit framework

JUnit is a test framework which uses annotations to identify methods that specify a test. JUnit is an open source project hosted at Github.

3.2. How to define a test in JUnit?

A JUnit *test* is a method contained in a class which is only used for testing. This is called a *Test class*. To define that a certain method is a test method, annotate it with the `@Test` annotation.

This method executes the code under test. You use an *assert* method, provided by JUnit or another assert framework, to check an expected result versus the actual result. These method calls are typically called *asserts* or *assert statements*.

You should provide meaningful messages in assert statements. That makes it easier for the user to identify and fix the problem. This is especially true if someone looks at the problem, who did not write the code under test or the test code.

3.3. Example JUnit test

The following code shows a JUnit test using the JUnit 5 version. This test assumes that the `MyClass` class exists and has a `multiply(int, int)` method.

```
import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

public class MyTests {

    @Test
    public void multiplicationOfZeroIntegersShouldReturnZero() {
```

```
        MyClass tester = new MyClass(); // MyClass is tested

        // assert statements
        assertEquals(0, tester.multiply(10, 0), "10 x 0 must be
0");
        assertEquals(0, tester.multiply(0, 10), "0 x 10 must be
0");
        assertEquals(0, tester.multiply(0, 0), "0 x 0 must be
0");
    }
}
```

3.4. JUnit naming conventions

There are several potential naming conventions for JUnit tests. A widely-used solution for classes is to use the "Test" suffix at the end of test classes names.

As a general rule, a test name should explain what the test does. If that is done correctly, reading the actual implementation can be avoided.

One possible convention is to use the "should" in the test method name. For example, "ordersShouldBeCreated" or "menuShouldGetActive". This gives a hint what should happen if the test method is executed.

Another approach is to use "Given[ExplainYourInput]When[WhatIsDone]Then[ExpectedResult]" for the display name of the test method.

4. Using JUnit 4

4.1. Defining test methods

JUnit uses annotations to mark methods as test methods and to configure them. The following table gives an overview of the most important annotations in JUnit for the 4.x and 5.x versions. All these annotations can be used on methods.

Table 1. Annotations

JUnit 4	Description
`import org.junit.*`	Import statement for using the following annotations.

Table 1. Annotations

JUnit 4	Description
`@Test`	Identifies a method as a test method.
`@Before`	Executed before each test. It is used to prepare the test environment (e. class).
`@After`	Executed after each test. It is used to cleanup the test environment (e.g. defaults). It can also save memory by cleaning up expensive memory s
`@BeforeClass`	Executed once, before the start of all tests. It is used to perform time in connect to a database. Methods marked with this annotation need to be JUnit.
`@AfterClass`	Executed once, after all tests have been finished. It is used to perform c disconnect from a database. Methods annotated with this annotation ne with JUnit.
`@Ignore` or `@Ignore("Why disabled")`	Marks that the test should be disabled. This is useful when the underly test case has not yet been adapted. Or if the execution time of this test practice to provide the optional description, why the test is disabled.
`@Test (expected = Exception.class)`	Fails if the method does not throw the named exception.
`@Test(timeout=100)`	Fails if the method takes longer than 100 milliseconds.

4.2. Assert statements

JUnit provides static methods to test for certain conditions via the `Assert` class. These *assert statements* typically start with `assert`. They allow you to specify the error message, the expected and the actual result. An *assertion method* compares the actual value returned by a test to the expected value. It throws an `AssertionException` if the comparison fails.

The following table gives an overview of these methods. Parameters in [] brackets are optional and of type String.

Table 2. Methods to assert test results

Statement	Description
fail([message])	Let the method fail. Might be used to check that a certain part of the failing test before the test code is implemented. The message param
assertTrue([message,] boolean condition)	Checks that the boolean condition is true.
assertFalse([message,] boolean condition)	Checks that the boolean condition is false.
assertEquals([message,] expected, actual)	Tests that two values are the same. Note: for arrays the reference is
assertEquals([message,] expected, actual, tolerance)	Test that float or double values match. The tolerance is the number
assertNull([message,] object)	Checks that the object is null.
assertNotNull([message,] object)	Checks that the object is not null.
assertSame([message,] expected, actual)	Checks that both variables refer to the same object.
assertNotSame([message,] expected, actual)	Checks that both variables refer to different objects.

4.3. JUnit test suites

If you have several test classes, you can combine them into a test suite. Running a test suite executes all test classes in that suite in the specified order. A test suite can also contain other test suites.

The following example code demonstrates the usage of a test suite. It contains two test classes (MyClassTest and MySecondClassTest). If you want to add another test class, you can add it to the `@Suite.SuiteClasses` statement.

```
package com.vogella.junit.first;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
```

```
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({
        MyClassTest.class,
        MySecondClassTest.class })

public class AllTests {

}
```