Unit Testing

Introduction

Unit testing is a level of software testing where individual units or components of a software code are tested. The purpose is to validate that each unit of the software performs as designed. A unit is the smallest testable part of any software. It usually has one or a few inputs and usually a single output. In procedural programming, a unit may be an individual program, function, procedure, etc. In object-oriented programming, the smallest unit is a method, which may belong to a base/super class, abstract class or derived/child class.

Java Unit Testing with JUnit and Mockito

JUnit is an open source Java testing framework which is used to create and run automated tests to ensure that code is working as expected. JUnit is widely used in the industry and can be used as both a stand alone Java program or within an IDE such as Eclipse.

Mockito is an open source Java mocking framework that is used, normally in conjunction with JUnit, to mock interfaces and external dependencies in order to allow for more granular, isolated unit testing.

Javascript Unit Testing with Jasmine and Karma

Jasmine is one of the more popular JavaScript behavior-driven development frameworks for unit testing JavaScript applications. It provides utilities that can be used to run automated tests for both synchronous and asynchronous code.

Karma is essentially a tool which spawns a web server that executes source code against test code for each of the browsers connected. The results of each test against each browser are examined and displayed via the command line to the developer such that they can see which browsers and tests passed or failed.

Unit Testing Best Practices

- Always write isolated test cases
- The order of execution has to be independent between test cases. This gives you the chance to rearrange the test cases in clusters (e.g. short-, long-running) and retest single test cases.
- Test one thing only in one test case

Each test should focus on one aspect of an isolated method and/or class. For integration test this goal may be difficult to achieve, but also here trying to focus on single aspects is helping. This means that you usually have just one assert per test

case. More than one asserts are always a hit that your test doesn't focus on one aspect or functionality.

Use a single assert methode per test case

Don't test too many different things at once! If you use just one Assert it's easier. This increases the number of test cases and the single test better to maintain. If one or more tests fail, you may find the root cause in a easier way.

Use a naming convention for test cases

A clear naming convention like: "Method-Name Under Test"_"Scenario"_"Expected-Outcome". This helps to avoid comments and increases the maintainability and in the case a test fails you know faster what functionality has been broken.

Use the Arrange-Act-Assert style or Given-When-Then style

Use separated blocks ("ARRANGE", "ACT" and "ASSERT") help to improve maintainability and use comments to mark these blocks. An alternative is the ("given", "when" and "then") style. This gives more structure to your unit tests.

Avoid the expected exception tests

Maybe @Test(expected = ArithmeticException.class) doesn't test what it claims to test. Sometimes it happens that the exception is thrown some were else in your code (this happened to me some time ago).

Structure all test cases

Like Short-/Long-Running, Integration-/Unit-tests, but don't use test suits to control order of execution. When you have hundreds of test cases you don't like to wait several minutes till all tests are ready. Especially integration test may be slow, so separate them from the unit tests.

Use descriptive messages in assert methods

Describe the WHY and not the WHAT, not like Assert.assertEquals(a, b, "must be equal"). This helps to avoid to much comments in the test cases and increases the maintainability.

Measure code coverage to find missing test cases

Best indicator to find out what is not tested, but don't be to sure that the code works.

Don't forget to refactor the test code

Also maintain your test code (especially when after refactoring the code under test).

Limit use of mocks

In some cases absolutely necessary, but with better design stubs should be enough.

Use parameterized tests

They can help avoid code duplication and the best is the business gives you the data.