

|  |
| --- |
|  |

**Toolname Guidelines and Best Practices**

# Table of Contents

1. INTRODUCTION……………………………………………………………………………………………………………………………1

2. BEST PRACTICE S OF juNIT…………………………………………………………………………………………………………..2

# 

# Introduction

This document provides Best Practices of Junit for performing isolated and atomic functional testing.

# BEST PRACTICES OF JUNIT

Following are the guidelines for developers:

1. **Test only one code unit at a time :**

Unit can have multiple use cases. We should always test each use case in separate test case. For example, if we are writing test case for a function which is supposed to take two parameters and should return a value after doing some processing, then different use cases might be

* First parameter can be null. It should throw Invalid parameter exception.
* Second parameter can be null. It should throw Invalid parameter exception.
* Both can be null. It should throw Invalid parameter exception*.*
* Finally, test the valid output of function. It should return valid pre-determined output.

This helps when you do some code changes or do refactoring then to test that functionality has not broken, running the test cases should be enough. Also, if you change any behavior then you need to change single or least number of test cases

1. **Do not make unnecessary assertion and use strong assertion :**

* unit tests are a design specification of how a certain behavior should work, not a list of observations of everything the code happens to do
* Do not try to Assert everything just focus on what we have to test otherwise it will end up having multiple testcases failures for a single reason, which does not help in achieving anything
* Without strong assertions, unit tests provide nothing more than coverage. In other words, unit tests that lack strong assertions ensure that the production code does not throw an exception but when is the lack of an exception sufficient to determine that a method behaves correctly? Coverage is a worthy outcome of unit tests but we can do much better! In particular, we want our unit tests to ensure that our production code works correctly. Without strong assertions, our unit tests only ensure that our production code doesn't blow up in our face.

In order of decreasing strength, assertions fall into the falling category

* **strongest assertions** - asserting on the return value of a method
* **strong assertions** - verifying that vital dependent mock objects were interacted with correctly
* **weak assertions** - verifying that non-vital dependent mock objects (such as a logger) were interacted with correctly
* **non-existent assertions**

1. **Make each test independent to all other :**

Do not make chain of unit test cases. It will prevent y to identify the root cause of test case failures and will have to debug the code. Also, it creates dependency, means if we have to change one test case then we need to make changes in multiple testcases unnecessarily.

Try to use @Before and @After to set up per-requisites if any for all your test cases. If you need to multiple things to support different test cases in @Before or @After, then consider creating new test cases.

**// Don't do this!**

public final class TestFoo

{

private FooDependency fooDependency = new FooDependency();

// rest of test class removed for brevity

}

**// Do this instead**

public final class TestFoo

{

private FooDependency fooDependency;

@Before

public void setUp()

{

fooDependency = new FooDependency();

}

}

1. **Mock out all external services and state :**

**B**ehavior in external services overlaps multiple tests, and state data means that different unit tests can influence each other’s outcome. Mock out all external services like Database and network connection this make test cases independent and isolated, All external services are unpredictable in nature like(how much time it will take to connect, connection will success or not) this will make test cases run slow, thus mocking all external services is best practice.

1. **Don’t test configuration settings :**

Configuration settings aren’t part of any unit of code.

1. **Name your unit tests clearly and consistently( follow convention while naming test cases) :**

Name the test case TestClassUnderTest. For example, the test case for the class MessageLog should be TestMessageLog. That make it simple to work out what class a test case tests. Test method’s names within the test case should describe what they test

* testLoggingEmptyMessage()
* testLoggingNullMessage()
* testLoggingWarningMessage()
* testLoggingErrorMessage()

Proper naming helps code readers understand each test’s purpose.

1. **Document tests in Javadoc**

Test plans document in a word processor tent to be error-prone and tedious to create. Also, word-processor-based document must be kept synchronized with the unit tests, adding another layer of complexity to the process. If possible, a better solution would be to include the test plans in the test’s Javadoc, ensuring that all test plan data reside in one place.

1. **Keep test cases small and fast**

Executing every test for the entire system shouldn’t take hours. Indeed, developers will more consistently run tests that execute quickly. Without regularly running the full set of tests, it will be difficult to validate the entire system when changes are made. Errors will start to creep back in, and the benefit of unit test will be lost. This means stress tests and load tests for single classes or small frameworks of classes shouldn’t be run as part of the unit test suite; they should be executed separately.

1. **Write test for methods that have the fewest dependencies first**

Unit test cases should be atomic and isolated.

1. **Put assertion method in proper order**

The parameter to Junit assertion are :

* Expected
* Actual

For example, use assertEqual(expected, actual) rather assertEqual(actual, expected). Ordering the parameter

Correctly ensures that Junit messages are accurate.

1. **Use Exact matching using a mocking framework**

For example, avoid using Mockito’s method that start with any; prefer configuring and verifying exact

Parameter values.

1. **Do not print anything out in unit tests**

Unit tests are intended for consumption by the computer. Developers may add print statements to their

Private working copy for debugging purposes, but there is typical no need to submit a unit test that prints.

For a specific best practice not printing stack traces.

1. **Do not use static members in a test class**

Static members make unit test methods dependent. Instead write test methods that are completely

Independent.

1. **Ensure that test code is separated from production code**

In your build script, ensure that code is not deployed with actual source code. It’s a wastage of resource.

1. **Do not verify how mock logger was interacted with unless logging is critical to the correctness of the method being tested**

Consider writing unit tests for the following production method:

Final class FOO

{

//Members and other methods removed for brevity.

Public int bar(final int i)

{

If(shouldReturnDefaultValue(i))

{

Logger.debug(”Return default value for: ” +i );

Return DEFAULT\_VALUE;

}

Else

{

Logger.debug(“Returning calculated value for:” +i);

Return calculateValue(i);

}

}

}

How should one test this method? A native approach might be to test method as follows:

**// Don't do this... the test is too brittle**

@Test

public void foo\_three()

{

final Logger logger = mock(Logger.class);

public Foo foo = new Foo(logger);

assertEquals(Foo.DEFAULT\_VALUE, foo.bar(3));

verify(logger).debug("Returning default value for: 3");

}

However, this test is too brittle because it fails if the production code is changed in a non-functional way. For example, the test fails if the production code is changed so that log statements occur at a different level (say info instead of debug) or with a different message (say "Returning default value for 3" without the colon). You can eliminate the latter problem using a less strict assertion (verify(logger).debug(anyString())) but the test is still too brittle. You're better off not verifying any logger behaviour unless logging is critical to the correctness of a production method.

1. **Do not rely on indirect testing**

Do not assume that particular test case tests another scenario also. This adds ambiguity. Instead, write another test case for each scenario