Best Practices for Unit Testing in Java

Last updated: May 11, 2024

Written by: Anshul Bansal
Reviewed by: Greg Martin

Testing
JUnit

1. Overview

Unit Testing is a crucial step in software design and implementation.

It not only improves the efficiency and effectiveness of the code, but it also makes the code more robust and reduces the regressions in future development and maintenance.

In this tutorial, we'll discuss a few best practices for unit testing in Java.

2. What Is Unit Testing?

Unit Testing is a methodology of testing source code for its fitment of use in production.

We start out writing unit tests by creating various test cases to verify the behaviors of an individual unit of source code.

Then **the complete test suite executes to catch the regressions, either in the implementation phase or while building packages for various stages of deployments** such as staging and production.

Let's take a look at a simple scenario.

To start with, let's create the *Circle* class and implement the *calculateArea* method in it:

```
public class Circle {

    public static double calculateArea(double radius) {
        return Math.PI * radius * radius;
    }
}
```

Then we'll create unit tests for the *Circle* class to make sure the *calculateArea* method works as expected.

Let's create the *CalculatorTest* class in the *src/main/test* directory:

```
public class CircleTest {

    @Test
    public void testCalculateArea() {
        //...
    }
}
```

In this case, we're using JUnit's *@Test* annotation along with build tools such as Maven or Gradle to run the test.

3. Best Practices

3.1. Source Code

It's a good idea to keep the test classes separate from the main source code. So, **they are developed, executed and maintained separately from the production code.**

Also, it avoids any possibility of running test code in the production environment.

We can follow the steps of the **build tools such as Maven and Gradle that look for *src/main/test* directory for test implementations.**

3.2. Package Naming Convention

We should create a similar package structure in the *src/main/test* directory for test classes, this way improving the readability and maintainability of the test code.

Simply put, **the package of the test class should match the package of the source class** whose unit of source code it'll test.

For instance, if our *Circle* class exists in the *com.baeldung.math* package, the *CircleTest* class should also exist in the *com.baeldung.math* package under the *src/main/test* directory structure.

3.3. Test Case Naming Convention

The **test names should be insightful**, and users should understand the behavior and expectation of the test by just glancing at the name itself.

For example, the name of our unit test was *testCalculateArea*, which is vague on any meaningful information about the test scenario and expectation.

Therefore, we should name a test with the action and expectation such as *testCalculateAreaWithGeneralDoubleValueRadiusThatReturnsAreaInDouble*, *testCalculateAreaWithLargeDoubleValueRadiusThatReturnsAreaAsInfinity*.

However, we can still improve the names for better readability.

It's often helpful to name the test cases in *given_, when_ then* to elaborate on the purpose of a unit test:

```
public class CircleTest {

    //...

    @Test
    public void givenRadius_whenCalculateArea_thenReturnArea() {
        //...
    }

    @Test
    public void givenDoubleMaxValueAsRadius_whenCalculateArea_thenReturnAreaAsInfinity() {
        //...
    }
}
```

We should also **describe code blocks in the *Given*, *When* and *Then* format.** In addition, it **helps to differentiate the test into three parts: input, action and output.**

First, the code block corresponding to the *given* section creates the test objects, mocks the data and arranges input.

Next, the code block for the *when* section represents a specific action or test scenario.

Likewise, the *then* section points out the output of the code, which is verified against the expected result using assertions.

3.4. Expected vs Actual

A **test case should have an assertion between expected and actual values.**

To corroborate the idea of the expected vs actual values, we can look at the definition of the *assertEquals* method of JUnit's *Assert* class:

```
public static void assertEquals(Object expected, Object actual)
```

Let's use the assertion in one of our test cases:

```
@Test
public void givenRadius_whenCalculateArea_thenReturnArea() {
    double actualArea = Circle.calculateArea(10);
    double expectedArea = 3.14159265358979;
    Assert.assertEquals(expectedArea, actualArea);
}
```

It's suggested to prefix the variable names with the actual and expected keyword to improve the readability of the test code.

3.5. Prefer Simple Test Case

In the previous test case, we can see that the expected value was hard-coded. This is done to avoid rewriting or reusing actual code implementation in the test case to get the expected value.

It's not encouraged to calculate the area of the circle to match against the return value of the *calculateArea* method:

```
@Test
public void givenRadius_whenCalculateArea_thenReturnArea() {
    double actualArea = Circle.calculateArea(10);
    double expectedArea = 3.14159265358979 * 10 * 2;
    Assert.assertEquals(expectedArea, actualArea);
}
```

In this assertion, we're calculating both expected and actual values using similar logic, resulting in similar results forever. So, our test case won't have any value added to the unit testing of code.

Therefore, we should **create a simple test case that asserts hard-coded expected value against the actual one.**

Although it's sometimes required to write the logic in the test case, we shouldn't overdo it. Also, as commonly seen, **we should never implement production logic in a test case to pass the assertions.**

3.6. Appropriate Assertions

Always **use proper assertions to verify the expected vs. actual results.** We should use various methods available in the *Assert* class of JUnit or similar frameworks such as *AssertJ*.

For instance, we've already used the *Assert.assertEquals* method for value assertion. Similarly, we can use *assertNotEquals* to check if the expected and actual values are not equal.

Other methods such as *assertNotNull*, *assertTrue* and *assertNotSame* are beneficial in distinct assertions.

3.7. Specific Unit Tests

Instead of adding multiple assertions to the same unit test, we should create separate test cases.

Of course, it's sometimes tempting to verify multiple scenarios in the same test, but it's a good idea to keep them separate. Then, in the case of test failures, it'll be easier to determine which specific scenario failed and, likewise, simpler to fix the code.

Therefore, always **write a unit test to test a single specific scenario.**

A unit test won't get overly complicated to understand. Moreover, it'll be easier to debug and maintain unit tests later.

3.8. Test Production Scenarios

Unit testing is more rewarding when we **write tests considering real scenarios in mind.**

Principally, it helps to make unit tests more relatable. Also, it proves essential in understanding the behavior of the code in certain production cases.

3.9. Mock External Services

Although unit tests concentrate on specific and smaller pieces of code, there is a chance that the code is dependent on external services for some logic.

Therefore, we should **mock the external services and merely test the logic and execution of our code for varying scenarios.**

We can use various frameworks such as Mockito, EasyMock and JMockit for mocking external services.

3.10. Avoid Code Redundancy

Create more and more **helper functions to generate the commonly used objects and mock the data or external services** for similar unit tests.

As with other recommendations, this enhances the readability and maintainability of the test code.

3.11. Annotations

Often, testing frameworks provide annotations for various purposes, for example, performing setup, executing code before and tearing down after running a test.

Various annotations such as JUnit's *@Before*, *@BeforeClass* and *@After* and from other test frameworks such as TestNG are at our disposal.

We should **leverage annotations to prepare the system for tests** by creating data, arranging objects and dropping all of it after every test to keep test cases isolated from each other.

3.12. 80% Test Coverage

More test coverage for the source code is always beneficial. However, it's not the only goal to achieve. We should make a well-informed decision and choose a better trade-off that works for our implementation, deadlines and the team.

As a rule of thumb, we should **try to cover 80% of the code by unit tests.**

Additionally, we can use tools such as JaCoCo and Cobertura along with Maven or Gradle to generate code coverage reports.

3.13. TDD Approach

Test-Driven Development (TDD) is the methodology where we create test cases before and in ongoing implementation. The approach couples with the process of designing and implementing the source code.

The benefit includes **testable production code from the start, robust implementation with easy refactorings and fewer regressions.**

3.14. Automation

We can **improve the reliability of the code by automating the execution of the entire test suite** while creating new builds.

Primarily, this helps to avoid unfortunate regressions in various release environments. It also ensures rapid feedback before a broken code is released.

Therefore, **unit test execution should be part of CI-CD pipelines** and alert the stakeholders in case of malfunctions.

4. Conclusion

In this article, we explored some best practices of Unit Testing in Java. Following best practices can help in many aspects of software development.