JUnit Best Practices Guide

howtodoinjava.com/best-practices/unit-testing-best-practices-junit-reference-guide

I am assuming that you know the basics of JUnit. If you do not have the basic knowledge, first read JUnit tutorial (Updated for JUnit 5). Now we will go through the **junit best practices** you must consider while writing your test cases.

> It's overwhelmingly easy to write bad unit tests that add very little value to a project while inflating the cost of code changes astronomically.

Table of Contents

Unit testing is not about finding bugs
Tips for writing great unit tests
- Test only one code unit at a time
- Don't make unnecessary assertions
- Make each test independent to all the others
- Mock out all external services and state
- Don't unit-test configuration settings
- Name your unit tests clearly and consistently
- Write tests for methods that have the fewest dependencies first, and work your way up
- All methods, regardless of visibility, should have appropriate unit tests
- Aim for each unit test method to perform exactly one assertion
- Create unit tests that target exceptions
- Use the most appropriate assertion methods.
- Put assertion parameters in the proper order
- Ensure that test code is separated from production code
- Do not print anything out in unit tests
- Do not use static members in a test class
- Do not write your own catch blocks that exist only to fail a test
- Do not rely on indirect testing
- Integrate Testcases with build script
- Do not skip unit tests
- Capture results using the XML formatter

Summary

In programming, "**Unit testing** is a method by which individual units of source code are tested to determine if they are fit for use." Now, this unit of source code can very on different scenarios.

For example: in procedural programming a unit could be an entire module but is more commonly an individual function or procedure. In object-oriented programming a unit is often an entire interface, such as a class, but could be an individual method. Intuitively, one should view a unit as the smallest testable part of an application.

Unit testing is not about finding regression bugs

Well, it's important to understand the motive behind unit testing. **Unit tests are not an effective way to find application wide bugs or detect regressions defects. Unit tests, by definition, examine each unit of your code separately.** But when your application runs for real, all those units have to work together, and the whole is more complex and subtle than the sum of its independently-tested parts. Proving that components X and Y both work independently doesn't prove that they're compatible with one another or configured correctly.

So, if you're trying to find regression bugs, it's far more effective to actually run the whole application together as it will run in production, just like you naturally do when testing manually. If you automate this sort of testing in order to detect breakages when they happen in the future, it's called integration testing and typically uses different techniques and technologies than unit testing.

> "Essentially, Unit testing should be seen as part of design process, as it is in TDD (**Test Driven Development**)". This should be used to support the design process such that designer can identify each smallest module in the system and test it separately.

Tips for writing great unit tests

1. Test only one code unit at a time

First of all and perhaps most important. When we try to test a unit of code, this unit can have multiple use cases. We should always test each use case in a separate test case. For example, if we are writing the 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:

1. *First parameter can be null. It should throw Invalid parameter exception.*
2. *Second parameter can be null. It should throw Invalid parameter exception.*
3. *Both can be null. It should throw Invalid parameter exception.*
4. *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.

2. Don't make unnecessary assertions

Remember, 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 you are testing otherwise you will end up having multiple test cases failures for a single reason, which does not help in achieving anything.

3. Make each test independent of all the others

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

Try to use **@Before** and **@After** methods to setup 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 class.

4. Mock out all external services and state

Otherwise, behavior in those external services overlaps multiple tests, and state data means that different unit tests can influence each other's outcome. You've definitely taken a wrong turn if you have to run your tests in a specific order, or if they only work when your database or network connection is active.

Also, this is important because you would not love to debug the test cases which are actually failing due to bugs in some external system.

(By the way, sometimes your architecture might mean your code touches static variables during unit tests. Avoid this if you can, but if you can't, at least make sure each test resets the relevant statics to a known state before it runs.)

5. Don't unit-test configuration settings

By definition, your configuration settings aren't part of any unit of code (that's why you extracted the setting out in some properties file). Even if you could write a unit test that inspects your configuration, then write only single or two test cases for verifying that configuration loading code is working and that's all.

Testing all your configuration settings in each separate test cases proves only one thing: "**You know how to copy and paste**."

6. Name your unit tests clearly and consistently

Well, this is perhaps most important point to keep remember and keep following. You must name your test cases on what they actually do and test. Testcase naming convention which uses class names and method names for test cases name is never a good idea. Every time you change the method name or class name, you will end up updating a lot of test cases as well.

But, if your test cases names are logical i.e. based on operations then you will need almost no modification because most possibly application logic will remain the same.

E.g. Test case names should be like:

```
1 ) TestCreateEmployee_NullId_ShouldThrowException
2 ) TestCreateEmployee_NegativeId_ShouldThrowException
3 ) TestCreateEmployee_DuplicateId_ShouldThrowException
4 ) TestCreateEmployee_ValidId_ShouldPass
```

7. Write tests for methods that have the fewest dependencies first, and work your way up

This principle says that if you are testing Employee module than you should first test Create Employee module as it has the minimum dependency on external test cases. Once they are done, start writing Modify Employee test cases as they need some employee to be in the database.

To have some employee in the database, your create employee test cases must pass before moving forward. In this way, if there is some error in employee creation logic, it will be detected much earlier.

8. All methods, regardless of visibility, should have appropriate unit tests

Well, this is controversial indeed. You need to look for most critical portions of your code and you should test them without worrying if they are even private. These methods can have certain critical algorithm called from one or two classes, but they play an important part. You would like to be sure that they work as intended.

9. Aim for each unit test method to perform exactly one assertion

Even this is not a thumb rule then also you should try to test only one thing in one test case. Do not test multiple things using assertions in the single test case. This way, if some test case fails, you know exactly what went wrong.

10. Create unit tests that target exceptions

If some of your test cases, which expect the **exceptions** to be thrown from application, use "**expected**" attribute like this. try avoiding catching exception in catch block and using fail/ or asset method to conclude the test.

```
@Test (expected=SomeDomainSpecificException.SubException. class )
```

11. Use the most appropriate assertion methods

There will be many assert methods you can work with each test case. Use the most appropriate with proper reasoning and thought. They are there for a purpose. Honor them.

12. Put assertion parameters in the proper order

Assert methods takes usually two parameters. One is the expected value and the second is the original value. Pass them in sequence as they are needed. This will help in correct message parsing if something goes wrong.

13. Ensure that test code is separated from production code

In your build script, ensure that test code is not deployed with actual source code. Its a wastage of resource.

14. Do not print anything out in unit tests

If you are correctly following all the guidelines, then you will never need to add any print statement in your test cases. If you feel like having one, revisit your test case(s), you have done something wrong.

15. Do not use static members in a test class. If you have used then re-initialize for each test case

We already have stated that each test case should be independent of each other, so there shall never be a need to have static data members. But, if you need any for the critical situation, then remember to re-initialize to its initial value before executing each test case.

16. Do not write your own catch blocks that exist only to fail a test

If any method in test code throws some exception then do not write a catch block only to catch the exception and fail the test case. Instead, use throws Exception statement in test case declaration itself. I will recommend using Exception class and do not use specific subclasses of Exception. This will increase the test coverage also.

17. Do not rely on indirect testing

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

18. Integrate Testcases with build script

It's better if you can integrate your test cases with build script so that they will get executed in your production environment automatically. This increases the reliability of the application as well as test setup.

19. Do not skip unit tests

If some test cases are not valid now then remove them from your source code. Do not use **@Ignore** or

svn.test.skip to skip their execution. Having invalid test cases in source code will not help anyone.

20. Capture results using the XML formatter

It is for feel-good factor. It will definitely not bring direct benefit but can make running unit tests interesting and entertaining. You can integrate JUnit with ant build script and generate test cases run a report in XML format using some color coding also. It is also a good practice to get followed.

Summary

Without a doubt, unit testing can significantly increase the quality of your project. Many scholars in our industry claim that any unit tests are better than none, but I disagree: a test suite can be a great asset, but bad ones can become the equally great burden that contributes little. It depends on the quality of those tests, which seems to be determined by how well its developers have understood the goals and principles of unit testing.

If you understood above guidelines and will try to implement most of them in your next set of test cases, you will certainly feel the difference.

Please let me know of your thoughts.

Happy Learning !!