Better Unit Testing — Part 1

Reading this without any prior knowledge of unit testing? That's even better, let's get started

Dhananjay Trivedi
Nov 28, 2019 · 5 min read ★

Photo by Stanley Dai on Unsplash

Many projects, especially Android ones, don't have unit tests. Well, even the official documentation at the beginning of Android development didn't encourage/had any support for unit tests. If there are no unit tests, we developers don't care about having the correct unit tests.

Many projects that have unit tests are mostly written after the production code was written and the developers were asked to write unit tests following some "policy".

These are bad test cases that don't give anybody any confidence about the integrity of the full platform.

No unit tests are better than having bad unit tests.

. . .

What Does Unit Testing Mean?

A unit is a container of a specific functionality that has clearly defined boundaries which, in Android, can be wrapped up in a `Class`.

Testing means ensuring something works perfectly. In unit tests, we can say that we make sure that our class does what it is supposed to do.

Unit tests ensure that the system under test behaves according to the requirements.

JUnit is a popular framework that we use to write unit tests to test our production code.

. . .

How To Write Better Unit Tests

Typically, we see the class whose unit test we have to write, go to the required method. See how it's written, understand some high-level logic, and then start to write its unit tests. This is a big no-no.

We should see the requirements and then write unit tests to fulfill those requirements. If you have followed critical thinking then you should have a document you can refer to which has all the agreed-upon edge test cases that you need to write.

This way, you are testing your code to see if it actually does a good job at fulfilling the specified requirements.

A simple structure of a unit test looks like:

```
import org.junit.Assert.*
import org.junit.Before
import org.junit.Test
```

```
class ProductionCodeClassTest {

@Before
    fun setup() {
        // Setup the class dependencies here
        // Which are required to run a successful test
    }

@Test
    fun testSomeMethod() {
        // Call the required logic
        // Assert the returned result
    }

}
```

`@Before` tells the JVM to run this particular method before every test function which is annotated with `@Test`.

. . .

How To Better Name Unit Tests

When working on bigger projects where you have reports of thousands of unit tests, and you get a message:

```
Test testFunctionFinal() failed
```

Or:

```
Test test_user_click_redirects_to_home() failed
```

Which one is easier to debug?

Unit tests' names don't necessarily have to be short, take as long as you need to write a descriptive unit test name. One quick tip is to include these three things:

```
UnitOfWork_StateUnderTest_ExpectdBehavior
```

So, following this, if you want to write the name of a method that reverses a string, it would be:

```
reverse_emptyString_emptyStringReturned
```

. . .

Better Approach for Unit Tests

Generally, we write unit tests after we have written our production code that we are satisfied with and it works. Hence, just write some casual unit tests that pass for something we already know is working.

The problem with these tests is that they don't really certify anything. If they pass, we don't care because we know we have written them casually, causal unit test passing doesn't mean anything. If they fail, yeah, they can help with something for sure.

Solution: If we write unit tests *before* we write our actual production code, and when those unit tests fail (compile error/runtime error/assertion error) we write some production code to make them pass.

This approach is called test-driven development.

Is this not making much sense? Let's write a simple test, shall we?

. . .

Writing Unit Test Cases Following TDD

We will be writing the code for string reversal, following test-driven development.

1. Create a class called `StringReverser`, once done, use the auto-generate option to generate a unit test file for the same class.

```
package com.sample.example

class StringReverser {
```

```
}
```

Generated unit test class:

```
import org.junit.Assert.*

class StringReverserTest {

}
```

2. Let's start writing our first unit test case.

Let's write that production code now to resolve the error.

Now that our error is resolved, we are not allowed to write any further production code according to the second law of TDD.

3. Finish our first unit test case.

Since our test passes, we can continue to write our second unit test.

4. Write the second unit test case.

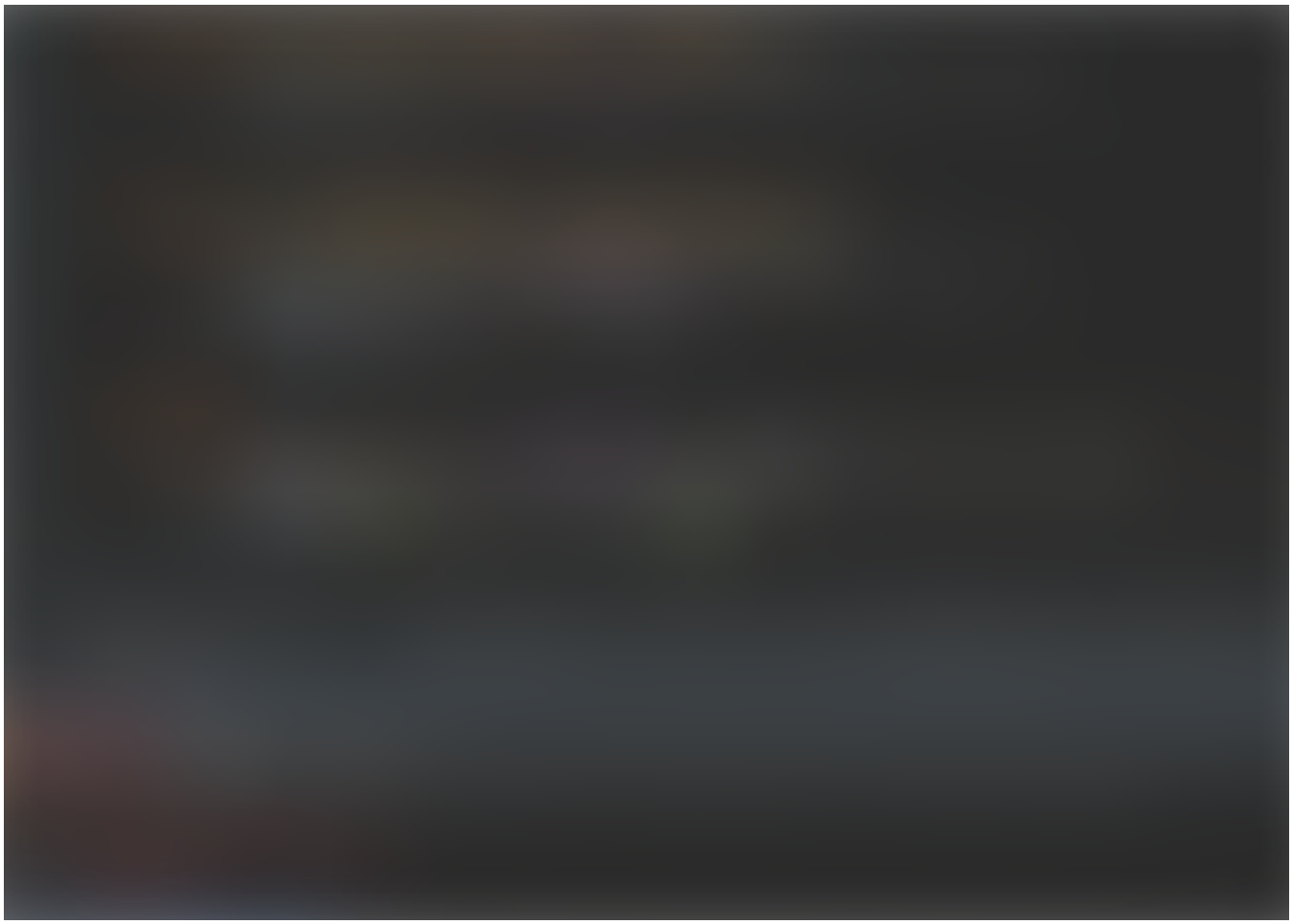
```
@Test
fun reverse_singleCharacter_sameStringReturned() {
    val result = stringReverser.reverse("x")
    assertEquals("x", result)
}
```

So, we just add another unit test case, and it also successfully passes. Let's continue to write another test case.

5. Write the third unit test case.

```
@Test
fun
reverse_multipleCharacterString_reversedMultipleCharacterStringRetur
ned() {
    val result = stringReverser.reverse("xyz")
    assertEquals("zyx", result)
}
```

Since we get no compilation errors, we just go ahead and run our unit test case and we see that it fails.



Now we are again allowed to write production code which just passes this failing unit test case. So, we go ahead and modify the `StringReverser` class.

Now, if we go back to our unit tests and run them all, all of them pass.

Feel free to add any more tests as you like, all of them should pass.

After following TDD, we are quite confident of our `StringReverser` class and are sure that there are no bugs. It is production-ready and the unit tests supporting our `StringReverser` class are always there to certify the `StringReverser` class.

In case somebody wants to make any changes in the future, they won't be afraid of breaking things as they just have to run the unit tests to see if the changes work with all the test cases, instead of them having to manually check.

. . .

What About Code Coverage?

How much code coverage should your unit tests have? 70%?

- A simple rule of thumb is: you only have to cover the modules of the code in your unit tests which you want to work in production, now that means 100%. We cannot have 80% of modules working while the other 20% are not tested in production. Hence, always try to get 100% code coverage.
- By 100%, we don't mean just cover all lines of code. There can be multiple logics in play in our system, hence, testing all the possible combinations of logic within our system boundaries also comes under 100% code coverage.
- There can also be multiple inputs or a combination of inputs coming into your system, 100% code coverage also covers that.

. . .

Conclusion

We have covered the fundamentals and we have learned a lot. If this topic interests you further, here are some more stories on the same topic worth your time.

Tests are your silver bullet for Clean Production Code

TDD is cool, but know how to do write the Tests cases professionally

medium.com

Getting Started With Test-Driven Development

Learn to build rock-solid solutions with TDD

medium.com

Thanks for reading!
DevDeejay

Hey hey hey! Part 2 is here 😇
https://medium.com/better-programming/better-unit-testing-part-2-80d29fc0f4b3

Testing Software Engineering Software Development Development Programming

About Help Legal