An experience of unit testing with the Arrange, Act, Assert (AAA) pattern: Part I

Nelson Wright
May 18 · 5 min read ★

For the last 6 years I've been part of an Android team doing a lot of unit testing, usually with TDD. This is how we've been using the "Arrange, Act, Assert" (AAA) pattern as part of our daily routine, and what worked for us.

. . .

The AAA pattern for structuring unit tests has been around for a while now. This part (Part I) sets out what our team did in practice, to apply this pattern. Part II explores some nuances, including how at times it highlighted issues in our code.

Why do it at all?

Jeff Griggs enumerates the benefits of AAA in this succinct wiki article:

> *Clearly separates what is being tested from the setup and verification steps.*
>
> *Clarifies and focuses attention on a historically successful and generally necessary set of test steps.*
>
> ***Makes some Test Smells more obvious:***
>
> *Assertions intermixed with "Act" code.*
>
> *Test methods that try to test too many different things at once.*

here on Medium, and elsewhere.

So, everyone must know about this by now, right?

I had thought so. But then a Tech Lead friend at a global company told me that when he joined, most of the developers there had not heard of the pattern. And partly as a result of that, the tests were hard to read and understand. Then I recalled that every developer who'd joined our team hadn't heard of it either.

It's a very simple concept to understand, but there are some interesting questions around its use. Overall, we found it useful in our unit tests, in just the way Jeff Griggs describes above. We've retained the practice since it's inception over six years ago.

The Rules

This is what our team adopted as the "rules" when formatting tests, as outlined by Mark Seemann:

1. If the test is 3 lines or less, there's no need for blank lines between the phases.
2. If the test is more than 3 lines, then have a blank line between the phases, i.e. one blank line either side of the "Act" line.
3. If the test is sufficiently complex that having a blank line within any of the three phases is beneficial, then a comment line labelling each phase should also be used.

That's all fine and well, but it does give rise to some questions. Lets take a look at those next, with some examples.

. . .

3 lines or less

If we take a Kotlin version of Jeff' Grigg's example test, we can have:

```
@Test
fun shouldReverseStringInput() {
    val input = "abc"
    val result = input.reversed()
```

(*Note: we wouldn't normally test the Kotlin language feature* **reversed()***, but this is just for illustration purposes. I've also used assertj for the assertions, as I find it more readable.*)

That's all fine, we've got 3 lines in the test method, matching the three phases of Arrange, Act, and Assert. What happens though, if we inline the `input` val?

```
@Test
fun shouldReverseStringInput() {
    val result = "abc".reversed()
    assertThat(result).isEqualTo("cba")
}
```

How do we now know what are the stages in the test? The thinking is that as it's a very simple test (2 lines), it should be fairly self-evident. And, looking at the test, there doesn't seem to be any ambiguity that

```
val result = “abc”.reversed()
```

is the "Act" line. There isn't a separate "arrange" phase now, as we're performing an **Act**ion directly on the `abc` string literal.

What would happen if we then inlined the `result` val?

```
@Test
fun shouldReverseStringInput() {
    assertThat("abc".reversed()).isEqualTo("cba")
}
```

We're left with a test with just an "Assert" phase, with the `assertThat` line. The "Act" part is embedded. That can be OK, as long as it's still readable. Whether it is or not, is up to you and your team. The goal is readable, understandable tests, after all. We want to avoid ultra-dense tests that take time to understand if they fail in say, 3 months time.

More than 3 lines

```
@Test
fun shouldReverseStringInput() {
    val input = "abc"
    val expectedReversedString = "cba"

    val result = input.reversed()

    assertThat(result).isEqualTo(expectedReversedString)
}
```

As we've now got more than 3 lines, the second rule comes into effect, so we now have a blank line to separate the phases. This makes the "Act" stand out well, and it's clear where the other phases start and end. So far, so good.

What happens if more lines are added?

Say we want to do some more setup in the "Arrange" phase. We want to join three strings, and reverse that compound string:

```
@Test
fun shouldReverseStringInput() {
    val exclaim = "!!"
    val firstInput = "cba"
    val secondInput = "gfed"
    val compoundInput = "$exclaim$secondInput$firstInput"
    val expectedReversedString = "abcdefg!!"

    val result = compoundInput.reversed()

    assertThat(result).isEqualTo(expectedReversedString)
}
```

We can still see the phases, but perhaps it would aid understanding if there was a blank line before the `expectedReversedString` line, to separate the setting up of the input and the result variables:

```
@Test
fun shouldReverseStringInput() {
    val exclaim = "!!"
    val expectedReversedString = "abcdefg!!"

    val result = compoundInput.reversed()

    assertThat(result).isEqualTo(expectedReversedString)
}
```

Now it's not quite so easy to scan the test quickly and identify the phases. This would likely only increase if the test became more complex. We should use Rule 3, and label the three phases with a comment:

```
@Test
fun shouldReverseStringInput() {
    // arrange
    val exclaim = "!!"
    val firstInput = "cba"
    val secondInput = "gfed"
    val compoundInput = "$exclaim$secondInput$firstInput"

    val expectedReversedString = "abcdefg!!"

    //act
    val result = compoundInput.reversed()

    //assert
    assertThat(result).isEqualTo(expectedReversedString)
}
```

There's a slight downside in that it increases the size of the test in terms of the total number of lines. However, adding the comments doesn't increase the cognitive load in trying to understand it.

Those are the basic "rules". It's probably becoming clear that if we have a more complex test that requires the labelling above, it isn't always easy to read. Even with the labels.

So, with that in mind, let's take a look next at what using this pattern can tell us about our code.

Continue Reading

Testing Unit Testing Experience Coding