KMock

a hitchhike to (more) convenient Kotlin tests.

About me

Before we start...

Disclamer

KMock 0.2.0 release is coming soon
so I will speak based on that even
if the docs will tell you otherwise.

Note

For all the things I talk about I have written my own tools (and they will be published somewhen).

Expectation management

Goals:

- you have a feeling of how to make your daily work in KMM/KMP testing more convenient
- you have an idea of how you could pull off your own mock library
- you have an overview on what is there and how they work in principle (expect MockingBird)

But:

- this talk is not about how to test certain things and not a deep dive in how to use certain tools

Warmup - Assertions

Why? - To make the code more readable!

Tools - kotest has some, but you can always write your own!

Warmup - Annotations

Why? - To make platform specific tools available (like Robolectric) or fine grain test in shared sources.

Warmup - Fixtures/Test values

Why? - To gain certain degrees of fuzziness for values and to avoid writing a gazillion times 42.

How? - Random, which can build upon. But be cautious that you still can reproduce values of test runs!

Warmup - to be continued

....

Let’s build a mocking library - Why even build such a thing?

Why not classical state testing?

➡ coupling

What’s about pure functions?

➡ are you certain it is pure?

Why not just test on JVM?

➡ you certainly miss something

Let’s build a mocking library - Why even build such a thing?

…most importantly writing mocks/stubs by hand is an avoidable cost.

And how hard can it be?™

Let’s build a mocking library -
What are the goals/constraints of this library?

- will do it only for interfaces
- it must compile in reasonable time (the golden 10s)
- easy to use
- reduction of mental overload
- maximize convenience
- providing a handle to assert/verify relations between objects

Let’s build a mocking library - Terminology*

Dummy - objects are passed around but never actually used. Usually they are just used to fill parameter lists.

Fake - objects actually have working implementations, but usually take some shortcut which makes them not suitable for production.

Stubs - provide canned answers to calls made during the test.

Spies - are stubs that also record some information based on how they were called.

Mocks - are pre-programmed objects with expectations which form a specification of the calls they are expected to receive.

* https://martinfowler.com/articles/mocksArentStubs.html

Let’s build a mocking library - Terminology*

Intrusive behaviour - you have to do something if the mocks parent changes (eg. adjusting the signature of a function).

Non Intrusive behaviour - you plug it in and it simply works (like relaxing in mockk).

Let’s build a mocking library - AAA

```
@Test
fun `Given a well descriptive test name`() {
    // Arrange/Assign
    val dummy = 42
    val sut = Thing()

    // Act
    val actual = sut.doSomething(dummy)

    // Assert
    assertEquals(
        actual = actual,
        expected = 23,
    )
}
```

Let’s build a mocking library - RRV

```
@Test
fun `Given a well descriptive test name` () {
    // Record
    val mock: Something = mockk()
    every { mock.doSomethingElse(any()) } returns 23
    val sut = Thing(mock)
    // Replay
    val actual = sut.doSomething(dummy)
    // Verify
    assertEquals(
        actual = actual,
        expected = 23
    )
    verify(exactly = 1) {
        mock.doSomethingElse(dummy)
    }
}
```

Let’s build a mocking library - Where can we learn from?

- mockk
- jest/mocha (JS)
- MagicMock! (Python)
- mockall (Rust)
- ...

Let’s build a mocking library - Lessons learned from MocKMP (Kodein)

- follows RRV (it imitates mockk)
- fastest approach (and easiest to begin with from a develop view)

Tradeoffs:

- ArgumentMatchers are the hard part
- merging Verification and Recording is not always a good idea
- Receivers are difficult
- it is completely intrusive

Let’s build a mocking library - Lessons learned from Mockactive (Nicklas Jensen)

- uses its own mind-model which is strongly BDD oriented (aka given-when-then)
- it is less intrusive (you can use it as Stubs)
- almost no dependencies
- super fast in code generation, but more compile time intensive

Tradeoffs:

- overloading is hard to come by
- it can be cumbersome
- strongly opinionated

Let’s build a mocking library - Tools - Kotlin Symbol Processing (Google)

- Gamechanger
- it is actually a Kotlin Compiler Plugin, which means it will only see platform code (for tests)

Let’s build a mocking library - Tools - Kotlin Poet (Square)

- well established code generation tool for Kotlin
- has (experimental) interop with KSP
- simply a bunch of Builders which are super convenient

Let’s build a mocking library - Test Tools - Kotlin Compiler Tests (Thilo Schuchort)

We simply want to check if what we generate is what we think we generate.

- snapshotesting

Let’s build a mocking library - Runtime - Proxies

What we want is a handle which allows us to have some sort of boxed values or behaviour for functions or properties.

Hence Proxies! (like do it by hand)

Let’s build a mocking library - Runtime - Proxies

Pros:

- autocompletion works just fine and we do not need any auxiliary methods to interact with them
- we can utilize behaviour and simple return values alike

Tradeoffs:

- we have to deal with generics, overloading and Receivers as well when generating code :(
- we are putting an overhead on the compile time

Let’s build a mocking library - Runtime - Assertions & Verification

What we want is somehow to capture relation between Proxies.

Hence Asserter/Verifier (there are actually the same thing)!

Let’s build a mocking library - Codegeneration

What we want is to generate shared/meta sources directly without any hassle of moving generated sources around.

Hence we simply make our “own” implementation of the CodeGenerator and propagate the targets via Annotations.

Let’s build a mocking library - Gradle

What we want is convenience! A consumer should not need to interact with KSP directly to configure our library.

Hence we create a Gradle Plugin which does the job and let a consumer “simply” configure our library via its extension.

Let’s build a mocking library - Can we make it more pleasant?

What we want is to minimize the intrusive behaviour of our lib.

Hence we make some sort of relaxation (remember fixtures) and spying possible.

Tradeoff:

- consumers have to write a certain degree of code to ensure it works correctly

Let’s build a mocking library - Can we mock all the things?

On JVM certainly - we can do what mockk does and use ByteBuddy to manipulate the ByteCode.

KMP - well...that’s complicated...even for JS (since Kotlin makes it typesafe at runtime)

We have to go through the compiler and make something similar to *all-open* or bend all references to right mocks.

Let’s build a mocking library - Can we mock all the things?

Tradeoffs:

- it does not fit into the JVM world since we have to merge Test and Production code together
- it is dangerous to a certain degree
- it will contribute a lot to the compile time (since we will effectively compile 2x production code)

Let’s build a mocking library - Done

Congrats you just have build KMock!

...Questions, Remarks, Pizza?

Let’s build a mocking library - Done

Thanks for listening!