Unit testing in Java: Mockist vs classical style

presented by Kristiyan Stoyanov

About me

- 5 years professional experience in Software Engineering
- currently work in Paysafe
- main focus is backend microservices written in Java (using Spring Boot)

linkedIn: https://www.linkedin.com/in/kristiyan-stoyanov-395310184/

email: kristiyan.stoyanov99@icloud.com

Table of contents

What is a unit test?

A piece of code that is used to verify that a “unit” of the program behaves as intended.

Unit tests are most commonly a method/function, which can be executed as part of an automated testing suite.

Example

Why do we need unit tests?

- spend less time to manually test the behaviour of software
- prevents breaking existing functionality with new changes
- fast and accessible feedback during development
- makes you write better (testable) code
- can be used as documentation

Characteristics of a good unit test

- small (no IO, no multithreading)
- readable (you should know what the test is checking in a matter of seconds)
- fast (you will be running the unit test suite a LOT of times, unit tests which run slow will cause developers to run them less often)
- reliable (tests shouldn't fail at random, if a test is flaky you might be better off without a test)

Best practices for writing unit tests

- one assertion per test
- avoid magic values
- write tests during development
- avoid exposing/duplicating implementation logic
- have a consistent naming convention for all tests
- have a consistent formatting convention for all tests (e.g. **A**rrange **A**ct **A**ssert)

What is mocking?

Mocking is a mechanism of creating a “fake” object, which can be used to replace a dependency of the class, which is under test.

Example Mocking

Advantages of Mocking

- helps keep tests small and isolated
- makes tests deterministic
- allows you to verify interactions with the “fake” object
- is fast to write thanks to very well supported frameworks (e.g. Mockito)*

The disadvantages of using Mocks (1)

The tests “know” implementation details of the class they are testing.

This is caused by the fact that mocks are described in the class that uses the dependencies.

Which leads to high coupling of the class, it’s dependencies and it’s tests.

When a change happens in one of the dependencies of the class the test for that class needs also to change.

Allowing to verify interaction with dependencies also promotes having methods with side effects and/or void return type.

The disadvantages of using Mocks (2)

As mocks are easy to write, this may cause their overuse.

When overusing mocks, your tests become less reliable as they are not testing true production code.

Classical style unit tests

- use real objects as much as possible (unless using the real object causes IO, multithreading or non-determinism)
- verify state instead of behavior (pure functions, no side effects)
- highly decoupled tests from the classes (and their dependencies) which are under test
- makes TDD easier

Example Classical Style

Mockist

- use fake objects
- test behavior and state
- need to know implementation details when writing tests
- a bug in one class causes only tests for this class to fail
- lower upfront cost of writing tests, but more changes might be needed in the future

vs.

Classical

- use real objects as much as possible
- test state only
- don’t care for implementation details when writing tests
- a bug in one class causes a ripple in all of its consumers
- higher upfront cost of writing unit tests, less changes in the future

Questions?