Hyperpragmatic pure FP testing with
distage-testkit

Functional Scala 2019

Septimal Mind Ltd
team@7mind.io

7mind

Not all the tests are equal

Tests are important. We know that.
Tests are the simplest way to define and verify important contracts.

We write tests. A lot of.

But

. . .

some tests prove themselves useful and some do not.

Not all the tests are equal

Which tests are good and which are bad?

Bad test criterias

Bad tests are:

- ▶ **Slow**,
- ▶ **Unstable**: they fail randomly,
- ▶ **Nonviable**: they don't survive refactorings,
- ▶ **Demanding**: they require complex preconditions to be met: external services up and running, fixtures loaded, etc, etc,
- ▶ **Incomprehensible**: they signal about a problem but don't help us to localize the cause.

Bad tests bring us less value than the resources we spend on them

How may we make our tests better?

Let's introduce some terminology

Test Taxonomy: Encapsulation Axis

Let's say that every test would fall under one of the following
categories:

1. **Blackbox** tests check just interfaces not knowing anything about
   implementations behind them,
2. **Whitebox** tests may know about implementations and check
   them directly, sometimes even breaking into some internal state to
   verify if it conforms to test expectations.

Test Taxonomy: Isolation Axis

Let's say that every test would fall under one of the following
categories:

- ▶ **Atomic** tests check just one "unsplittable" software component,
- ▶ **Group** tests check multiple software components,
- ▶ **Communication** tests communicate with outer world (databases,
  API providers, etc, etc).

You may extend and modify this Test Taxonomy as it would be convenient for you.
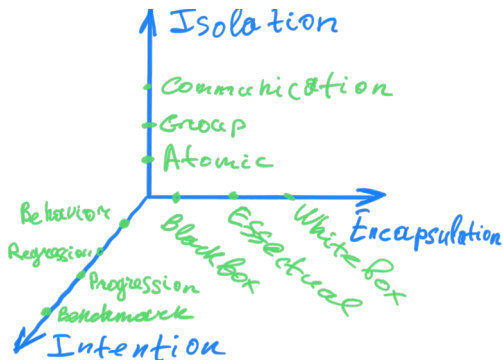
More about test taxonomies:

`https://blog.7mind.io/constructive-test-taxonomy.html`

Test Taxonomy: Why So?

According to our experience the best tests are
Blackbox tests with **Atomic** or **Group** Isolation Level.

And it's obvious.

Communication tests

But in real projects most of the tests fall under **Communication** category ("Integration" tests).

Why?

1. Engineers want to test The Whole Thing,

2. It's hard to separate components,

3. etc, etc. . .

Communication tests can be more useful

We may replace *integration components* with *Dummies*[1].

This way we may turn
Blackbox Communication tests
into
Group or **Atomic**

_____

[1]people also call them *"Fakes"*, *"in-memory implementations"* or *"Mocks"*

Dual Test Tactic

The same test scenario executed with both Production and Dummy implementations of *integration components* is beneficial:

1. We can test business logic quickly, without any interference,

2. We still able to verify component behaviour in "real" circumstances,

3. We have to follow **LSP** and design better to make code compatible with **Dual Test Tactic**.

Dual Test Tactic: ideas

1. <u>Ignore</u> **Communication** tests in case their dependencies are unsatisfied (service unavailable), don't fail,
2. <u>Avoid</u> automatic *"Mocks"*, they prevent encapsulation,
3. <u>Never run</u> heavy dependencies in-process (don't bring whole Kafka or Cassandra into the classpath. **PLEASE**).
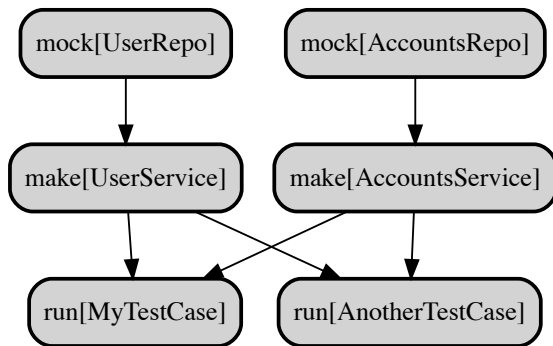
Dual Test Tactic: The Main Issue

It's easy to setup test dependencies while working with *Dummies*
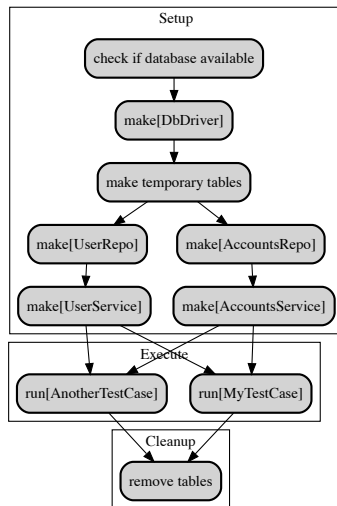But you have to do things differently for *Production* dependencies.

We have to take care of:

1. Resource acquisition and Cleanups,

2. Speed,

3. Memoization and resource reuse,

4. Interference,

5. . . .

Dual Test Tactic: Dummy testcase steps

Dual Test Tactic: Real testcase steps

Dual Test Tactic: Real testcase steps

And that's not everything!

We may need to add more aspects, like run Docker containers and await until they open ports. And stop them after the tests. Etc, etc. . .

It's hard to setup variable contexts. Manual wiring is hard to maintain and suffers from combinatoric explosion of possible code paths. Cake pattern doesn't make much difference. Even heavy machinery like conventional DI frameworks fails.

Dual Test Tactic
may be very pricy under usual circumstances.

Can we make it cheap?

Yes.

The Goal

We want to write code like this and never care about setting things up:

```scala
class JustATest[F[+_, +_]] {
  "service" must {
    "do something" in {
      (users: UserService[F], accounts: AccountingService[F]) =>
      for {
        user  <- users.create()
        balance <- accounts.getBalance(user)
      } yield {
        assert(balance == 0)
      }
    }
  }
}
```

Let's look at a <u>real</u> example. . .

Thank you for your attention

distage website: `https://izumi.7mind.io`

Septimal Mind is an Irish software consultancy and R&D Company

We're looking for clients, contributors, adopters and colleagues ;)

Contact: team@7mind.io

Github: `https://github.com/7mind`

Slides: `https://github.com/7mind/slides`

Follow us on Twitter

`https://twitter.com/shirshovp`

`https://twitter.com/kai_nyasha`