Hyperpragmatic pure FP testing with distage-testkit

Functional Scala 2019

Septimal Mind Ltd
team@7mind.io

Not all the tests are equal

Tests are important. We know that.
Tests are the simplest way to define and verify important contracts.
And contracts are crucial for project maintainability and viability.

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.

We spend more resources to write and maintain bad tests
than the value they bring us.

How may we make our tests better?

Let's ditch Unit/Functional/Integration trinity...

...and introduce some new terminology

Test Taxonomy: Encapsulation Axis

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

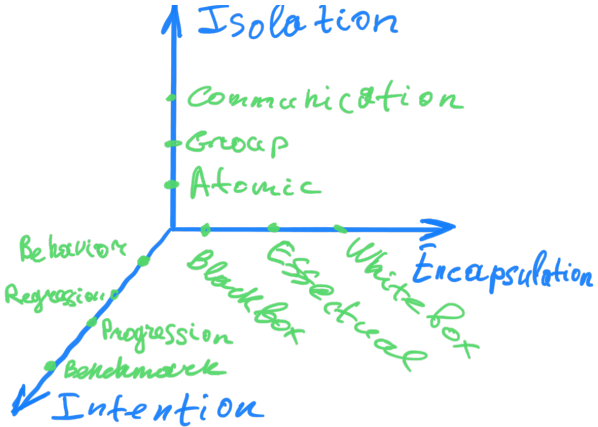
1. **Blackbox** tests check just interfaces not knowing anything about implementations behind them,
2. **Whitebox** tests may know about implementations and check them directly, sometimes even breaking into some internal state to verify if it conforms to test expectations.

Test Taxonomy: Isolation Axis

Let's say that every test falls 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.

This makes sense: such tests are fast and refactoring-resilient.

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
Communication tests
into
Group or **Atomic**[2]

[1] people also call them *"Fakes"*, *"in-memory implementations"* or *"Mocks"*
[2] this statement is valid for **Blackbox** tests only

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 business logic behaviour with "real" integrations,
3. **Dual Test Tactic** enforces us to follow **LSP** and design better.

Refactorings are crucial for long-term health of the project.
Blackbox tests enable refactorings.
Dual Test Tactic drastically reduces the price of refactorings.

Dual Test Tactic: ideas

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

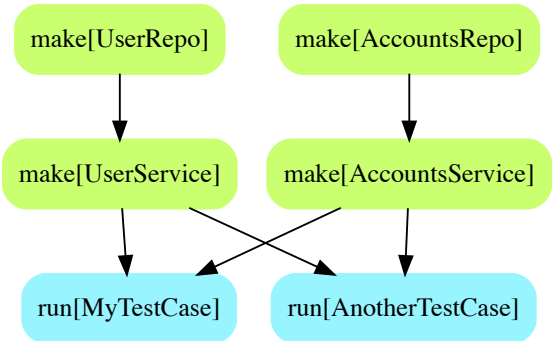
Dual Test Tactic: The Main Issue

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

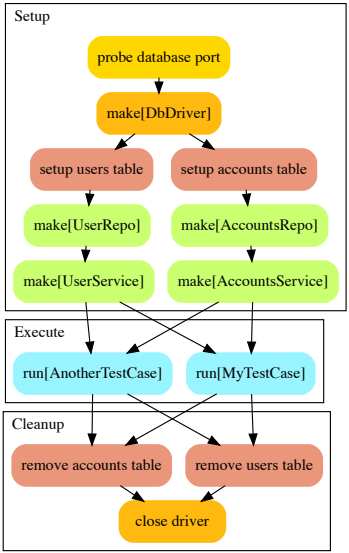
We have to take care of:

1. Resource acquisition and Cleanups,
2. Speed,
3. Memoization and resource reuse,
4. Interference,
5. ...

Dual Test Tactic: dummy repositories testcase



Dual Test Tactic: production repositories testcase



1. Order
2. How to avoid unneccessary job?
   - 2.1 Memoize resources?..
   - 2.2 In a singleton?..
   - 2.3 When we close resources?..
   - 2.4 On JVM shutdown?.. Oops, SBT...
   - 2.5 Disambiguation (same class, different parameters)?..
3. Resource deallocation
   - 3.1 ...even after a failure
   - 3.2 Order!
4. Other integrations, e.g. run Dockers
   - 4.1 ...and await until they open ports
   - 4.2 ...and stop them after the tests
5. Configs

Dual Test Tactic: Real testcase steps

- Integration points stack together
- . . . and make the problem notably hard
- Manual wiring is hard to maintain
- . . . and suffers from combinatoric explosion of possible code paths
- Cake pattern doesn't make much difference
- Conventional DI frameworks fail

It's hard to setup variable contexts.

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:

```
1  class JustATest[F[+_, +_]] {
2    "service" must {
3      "do something" in {
4        (users: UserService[F], accounts: AccountingService[F]) =>
5        for {
6          user  <- users.create()
7          balance <- accounts.getBalance(user)
8        } yield {
9          assert(balance == 0)
10       }
11     }
12   }
13 }
14
15 object JustATestZioProd extends JustATest[zio.IO] with Prod
16 object JustATestZioDummy extends JustATest[zio.IO] with Dummy
```

distage

is a Dependency Injection mechanism for Scala

But it lacks negative traits of a typical DI thingy...

...and has many unique properties.

...and it's blazing fast.

You may call it a module system for Scala...

...with automatic garbage-collecting solver

distage

1. Supports FP, wires Resources, Effects, ZIO Environments...
2. transparent and predictable
   - first plan then do
   - test the outcome without effect execution!
   - compile-time checks and tests
3. no `scala-reflect` nor Java reflection[1],
   - ScalaJS support is coming
4. is non-invasive.
   - You can add it to your project keeping business logic intact...
   - ...and remove it.
5. is not an ad-hoc thing, it has strong theory behind.

[1] Since version `0.10.0`, see https://blog.7mind.io/lightweight-reflection.html

Docker support in distage

While we develop and test we may need our integrations up and running...
...We have `docker-compose` for that...
But manual rituals are annoying and Compose is imperfect
(wait until service opens port? Hah)

Container orchestration is not easy.
But...
We made a `distage` extension allowing to get rid of Compose in most of the cases.

It can run test containers, reuse and shutdown them.
And it can **await until the service is ready**.

Let's look at a real example...

Source code: https://github.com/7mind/distage-example

`distage` performs optimal ahead of time planning.
It doesn't do any unnecessary job.

One More Thing: Roles

`distage` allow you to fuse microservices into "flexible monoliths".

We may:

1. Develop services (*Roles*) separately, even in a multirepo,
2. Build a single Docker image with multiple Roles in it,
3. Run several Roles within one process
   - We define roles we want to start as commandline parameters
4. $\Rightarrow$ higher computation density, savings on infrastructure,
5. $\Rightarrow$ *substantial* development simplification
   - full environment can be started on a low-end machine
   - with just one command

`distage-framework` is the most productive way to write maintainable pure functional applications with ZIO.
(And any other monad)

`distage-testkit` is the best way to make your tests performant and reliable.

`distage` is adopted by several different companies and tested by two years of production usage.

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`