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.

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:

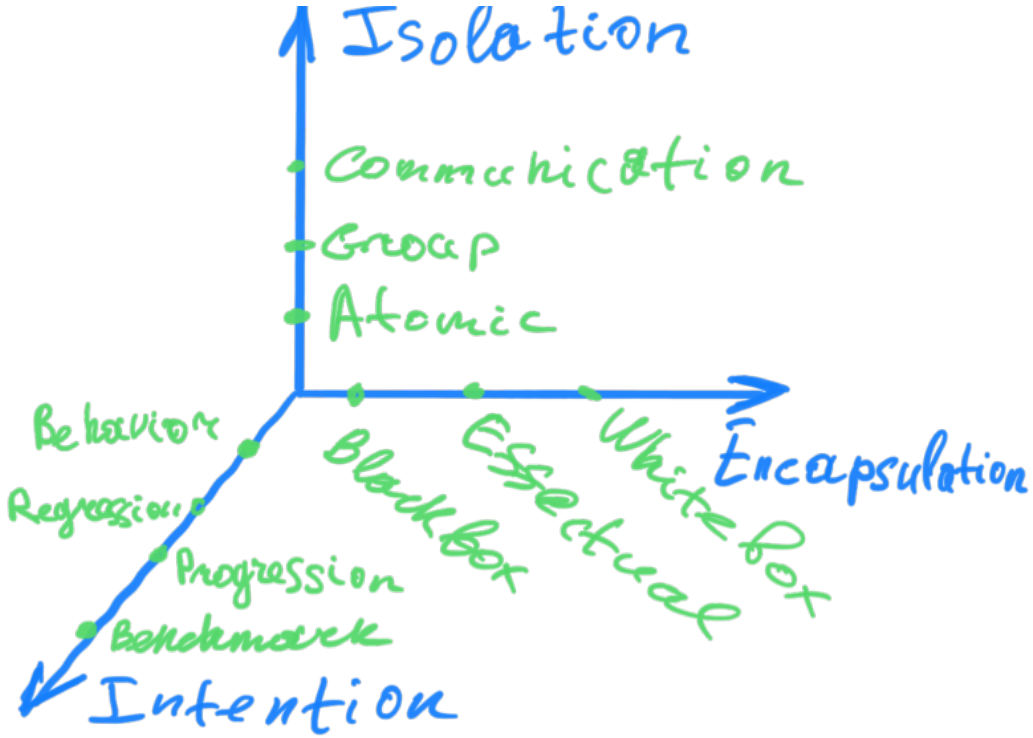
1. **Blackbox** tests check just interfaces not knowing anything about implementations behind them,
2. **Whitebox** tests may know about implementations and check them directly, sometimes even breaking into some internal state to verify if it conforms to test expectations.

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. 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**).

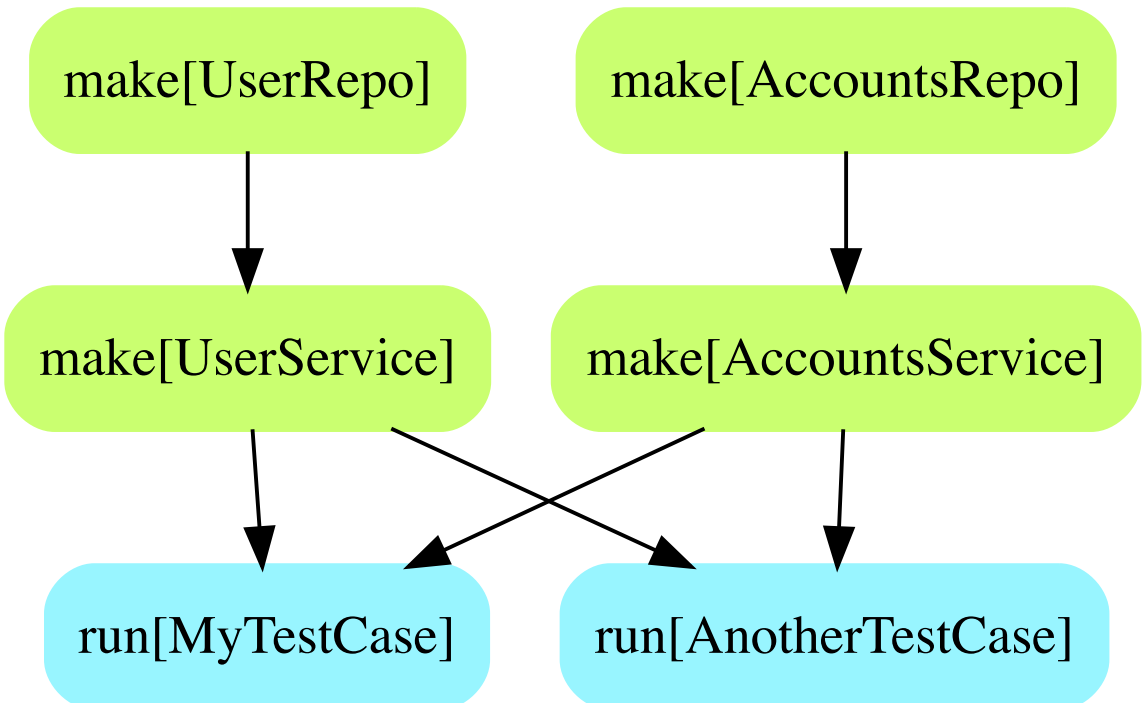
Dual Test Tactic: The Main Issue

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

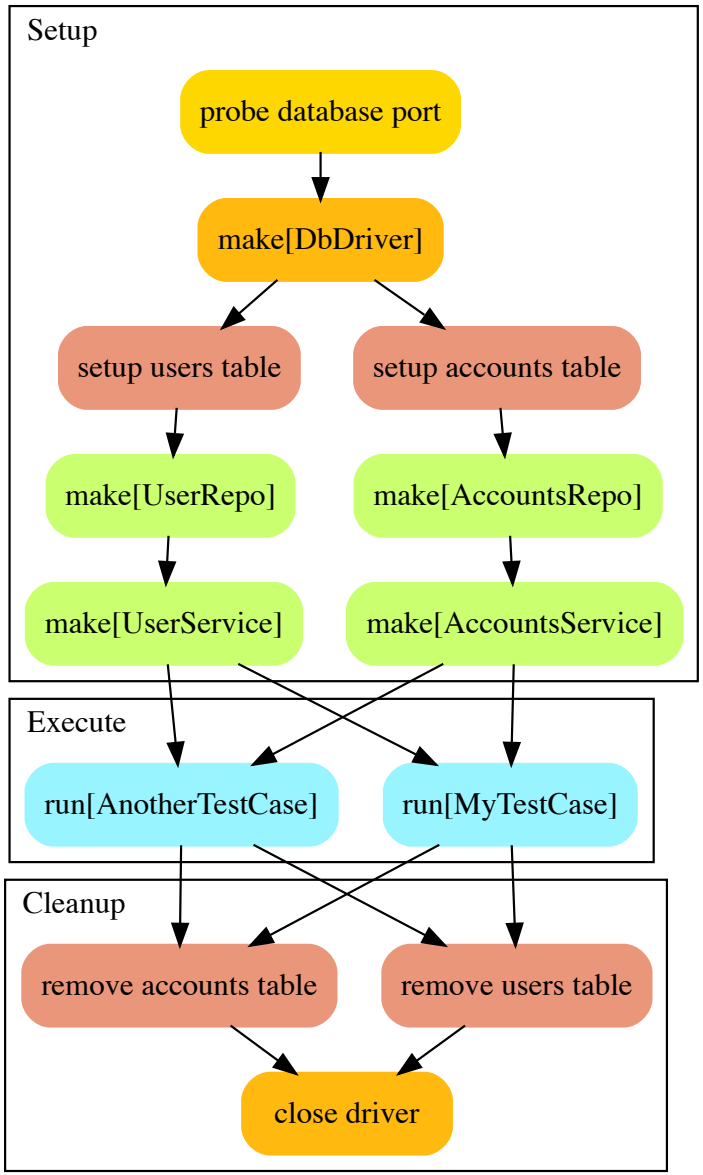
We have to take care of:

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

Dual Test Tactic: Dummy testcase steps



Dual Test Tactic: Real testcase steps



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 module system for Scala. . .

. . . with automatic garbage-collecting solver

You may call it a "Dependency Injection" mechanism

But it lacks many traits of a typical DI thingy. . .
And has many unique properties

distage

1. is non-invasive.
   - You can add it to your project keeping business logic intact...
   - ...and remove it.
2. does most of the job in compile-time,
3. does not depend on `scala-reflect`[1],
4. replaces `scala-reflect`[2] with own lightweight reflection,
5. has virtually complete support of Scala typesystem,
6. is not an ad-hoc thing, it has strong theory behind.

[1] Since version `0.10.0`

[2] https://blog.7mind.io/lightweight-reflection.html

Let's look at a real example...

How it works: Plans

`distage` takes your bindings and then:

1. translates bindings into simple Turing-incomplete DSL (like `make`, `reference`, etc.),
2. represents the DSL statements as Directed Acyclic Graph using dependecy information and breaking circular dependencies if any,
3. resolves conflicts (one DAG node with several associated operations),
4. performs Garbage Collection (yeah, real one),
5. applies other transformations (like config reference resolution),
6. turns the DAG back into sequential form — a Plan — with topological sorting.
7. $\Rightarrow$ the Plan may be introspected, printed, executed in compile-time by a code generator or executed in runtime.

`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*[1]) separately, even in multirepo,
2. Build a single Docker image with multiple Roles in it,
3. Pass Roles we want to start as commandline parameters (and run several Roles within one process),
4. ⇒ higher computation density, savings on infrastructure,
5. ⇒ *substantial* development simplification: full environment can be started on a low-end machine with 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`