Embracing the Failure

Unit testing with matchers

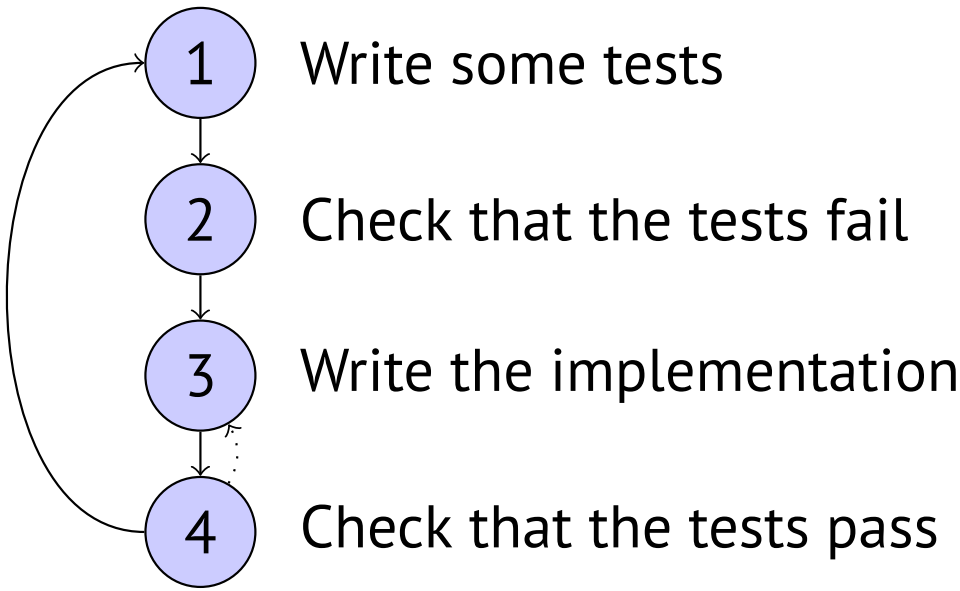
Roman Kashitsyn

HaskellerZ, October 2018

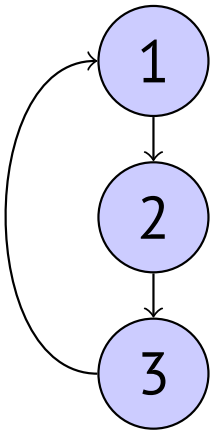
Disclaimer

- The library I'll be talking about is not an official Google product.
- All the code in this presentation is licensed under the Apache License, Version 2.0, with Google LLC being the copyright owner (2018).

Unit testing: theory



Unit testing: real life

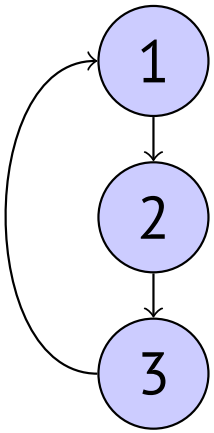


Write some code

Write some tests for the code

Check if the tests pass

Unit testing: real life



1. Write some code
2. Write some tests for the code
3. Check if the tests pass

This approach is **inherently broken**

Why?

Analogy: scientific method

a bug (in the bugtracker)	$\approx$	an observation log
debugging	$\approx$	inquiry
bugfix	$\approx$	hypothesis
unit test	$\approx$	observation/experiment

Analogy: drug test experiment

We took 50 people having cold and let them take the drug every day for 2 weeks. After 2 weeks, none of them had a cold anymore.

Does it sound right?

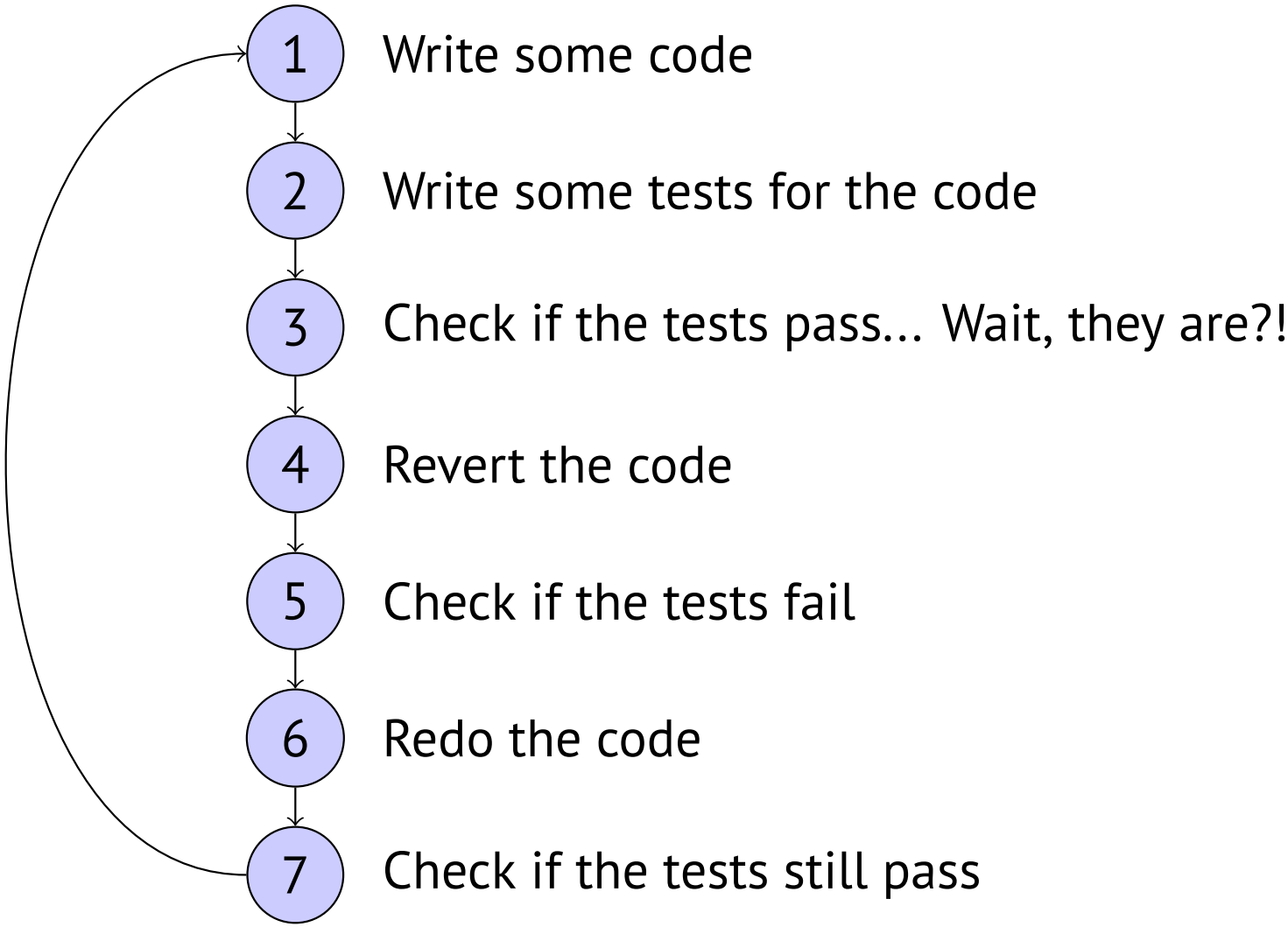
Analogy: drug test experiment

We took 50 people having cold and let them take the drug every day for 2 weeks. After 2 weeks, none of them had a cold anymore.

Does it sound right?

Hell no! We'll never know what would happen if they never took the drug. We need 2 groups of the same size, one taking the drug and one taking a placebo!

Unit testing: me



Why is this happening?

- I believe it's easier and more fun to write the implementation first.
- Confirmation bias. People don't really like to hear **"you're wrong"**. Check out this NY Times article!
- Error messages suck!

How do you like this error message?

```
test/Spec.hs:115:
1) Simple matchers can match pairs
     expected: MatchTree {mtValue = True, mtDescription = all of, mtMatchedValue =
           Just (3,4), mtSubnodes = [MatchTree {mtValue = True, mtDescription =
           projection fst", mtMatchedValue = Just (3,4), mtSubnodes = [MatchTree {
           mtValue = True, mtDescription = is a value equal to 3, mtMatchedValue =
           Just 3, mtSubnodes = []}]},MatchTree {mtValue = True, mtDescription =
           projection "snd", mtMatchedValue = Just (3,4), mtSubnodes = [MatchTree {
           mtValue = True, mtDescription = is a value > 1, mtMatchedValue = Just 4,
            mtSubnodes = []}]}]}
      but got: MatchTree {mtValue = True, mtDescription = all of, mtMatchedValue =
            Just (3,4), mtSubnodes = [MatchTree {mtValue = True, mtDescription =
            projection "fst", mtMatchedValue = Just (3,4), mtSubnodes = [MatchTree
            {mtValue = True, mtDescription = is a value equal to 3, mtMatchedValue
            = Just 3, mtSubnodes = []}]},MatchTree {mtValue = True, mtDescription =
             projection "snd", mtMatchedValue = Just (3,4), mtSubnodes = [MatchTree
             {mtValue = True, mtDescription = is a value > 1, mtMatchedValue = Just
             4, mtSubnodes = []}]}]}
```

Ehm, thank you, it was really... helpful.

How about this one?

```
1) README examples works for negative case
     predicate failed on expected exception: HUnitFailure (HUnitFailure (Just (
           SrcLoc {srcLocPackage = "test-matchers-0.1.0.0-7Ytou1BUlrG5HYe1sE7Q7b",
           srcLocModule = "Test.Matchers.HUnit", srcLocFile = "src/Test/Matchers/
           HUnit.hs", srcLocStartLine = 54, srcLocStartCol = 26, srcLocEndLine =
           55, srcLocEndCol = 71})) (Reason "\10008 prism \"Right\" \8592 <1>\n
           \10008 is a value equal to 0\nwhere:\n  <1> Left \"Division by zero\""))
```

Great, which predicate?

Alleviating the pain of bad messages

- Use differs like gdiff or tree-diff when comparing for equality.
- Use Hedgehog, it's pretty cool!
- Use *matchers*.

Ok, what ARE matchers?

Basically, they're predicates on steroids: they can describe what they do.

I use them at work almost every day (I write a lot of code in C++). Let me show you how it works.

Examples (C++) (1)

```
struct Service {
  StatusOr<User> GetUser(UserId user_id) const;
};

EXPECT_THAT(
  service.GetUser(user_id),
  IsOkAndHolds(AllOf(
    Property(&User::id,   Eq(user_id)),
    Property(&User::name, IsNotEmpty()))));
```

The error message for (1)

```
Value of: service.GetUser(user_id)

Expected: is OK and has a value that
  (is an object whose given property is equal to 5)
  and
  (is an object whose given property isn't empty)

  Actual: OK: User{id=5, name=''}
  (of type util::StatusOr<User>),
  which contains value User{id=5, name=''},
  whose given property is ""
```

Examples (C++) (2)

```
EXPECT_THAT(
  service.GetUser(kInvalidUserId),
  StatusIs(NOT_FOUND, HasSubstr(kInvalidUserId)));
```

The error message for (2)

```
Value of: service.GetUser(kInvalidUserId)

Expected: is in an error space that is <generic>,
  has a status code that is not_found,
  and has an error message that has substring "-1"

  Actual: generic::not_found: Unknown user
  (of type util::StatusOr<User>),
  whose error message is wrong
```

Writing an implementation in Haskell

Requirements:

1. Flexibility in the output format.
2. No Template Haskell.
3. Minimal dependencies.

The main idea

We can represent the result of the matcher execution as a tree reflecting the underlying structure of the matchers.

- ✗ all of
  - ✓ projection “id”
    - ✓ is equal to 5 (was: *5*)
  - ✗ projection “name”
    - ✗ is not empty (was: ””)

We can then use this structure to format the output in the most suitable way.

First attempt

```
data MatchTree
   = MatchTree
   { result      :: !Bool
   , description :: Message
   , value       :: Message
   , subtrees    :: [MatchTree]
   }

type Matcher a = a → MatchTree

shouldMatch :: a → Matcher a → Assertion

predicate
  :: (Show a)
  ⇒ (a → Bool)
  → Message
  → Matcher a
```

So far so good...

```
eq
  :: (Show a, Eq a)
  ⇒ a
  → Matcher a
eq x = predicate
       (== x)
       ("is a value equal to " ++ show x)
```

Now we can easily implement a lot of useful matchers: **gt**, **lt**, **floatApproxEq**,...

Projection combinator

We can already implement the **Property** combinator from the C++ example (to avoid confusion with **QuickCheck.property**, we'll call it **projection**).

```
projection
  :: (Show s)
  ⇒ Message  -- Name of the projection
  → (s → a) -- Projection from s to a
  → Matcher a
  → Matcher s
```

We can now implement matchers for product types and other sorts of projections:

```
userNameIs   nameM =
  projection "userName" userName nameM

reminderIs k modM  =
  projection ("mod " ++ show k) (`mod` k) modM
```

Projection is an outlaw

projection looks very much like **contramap**, but with an extra argument (name). Let's take a look at the **contramap** laws:

```
class Contravariant f where
  contramap :: (a → b) → f b → f a
```

$$\text{contramap id} = \text{id}$$

$$\text{contramap } f \circ \text{contramap } g = \text{contramap } (g \circ f)$$

$$\text{projection "id" id} \neq \text{id}$$

$$\text{projection "f" } f \circ \text{projection "g" } g \neq \text{projection "g ∘ f" } (g \circ f)$$

We don't obey the laws for a reason: we actually want to capture the exact projections applied, with all the intermediate values!

Enter **elementsAre**

```
[1, 2, 3] `shouldMatch`
  elementsAre [eq 1, gt 1, lt 5]
               ~^~~  ~^~~  ~^~~
                1     2     3
```

Enter **elementsAre**

```
[1, 2, 3] `shouldMatch`
  elementsAre [eq 1, gt 1, lt 5]
                ~^~~  ~^~~  ~^~~
                 1     2     3
```

The example above kind of works, but this one...

```
                                    Requires an a!
                                    ~v~~
 [ ] `shouldMatch` elementsAre [eq 1]
 ~^~
 Has no as to pass to the (eq 1)!
```

If values can be missing...

Just wrap them in **Maybe**!

```
type Matcher a = Maybe a → MatchTree
```

Great! **elementsAre** works now, as well as some other cool matchers.

Prism combinator

We can now implement another nice combinator:

```
prism
  :: (Show s)
  ⇒ Message          -- Name of the prism
  → (s → Maybe a) -- Selector of a from s
  → Matcher a
  → Matcher s
```

This combinator can be used to build matchers for sum types:

```
{-# LANGUAGE LambdaCase #-}
rightIs
  :: (Show l, Show r)
  ⇒ Matcher r
  → Matcher (Either l r)
rightIs = prism "Right" (\case { Right r → Just r
                               _ → Nothing })
```

How about exceptions?

Let's see. We would like to be able to write something like

```
myAction `shouldMatch` throws exceptionValueMatcher
```

Then the type of **throws** should be

```
throws
  :: (Exception e)
  ⇒ Matcher e
  → Maybe (IO a)
  → MatchTree
    ~~^~~~~~~
    Pure! Can't catch any exceptions with such a type.
```

Need to revise types again!

```
type MatcherF f a = Maybe (f a) → f MatchTree
```

```
shouldMatchIO :: IO a → MatcherF IO a → Assertion
```

This type is powerful enough for us to implement **throws**.

Quiz: why not

```
type MatcherF f a = f (Maybe a) → f MatchTree
```

?

Need to revise types again!

```
type MatcherF f a = Maybe (f a) → f MatchTree

shouldMatchIO :: IO a → MatcherF IO a → Assertion
```

This type is powerful enough for us to implement **throws**.

Quiz: why not

```
type MatcherF f a = f (Maybe a) → f MatchTree
```

?

Answer: consider

```
[action1] `shouldMatchIO`
  elementsAre [throws ex1, throws ex2]
```

Big surprise: **negationOf**

It'd be nice to have a combinator equivalent to the negation in the predicate world.

```
negationOf :: MatcherF f a → MatcherF f a
```

A naive implementation: construct a new `MatchTree` node for each negation:

Matcher	Tree
negationOf (eq 5)	▶ **not** ▶ is a value equal to *5*

Problems with the naive implementation

The naive implementation has major flaws:

- The messages look funny. When multiple negations involved, it's hard to understand what's going on!
- Unlike **not**, it's not an *involution*: **negationOf (negationOf m)** is not equivalent to **m**!

Solution: flipping matchers

```
data Direction = Positive | Negative

type MatcherF f a
  =  Direction
  → Maybe (f a)
  → f MatchTree

flipDirection :: Direction → Direction

negationOf m dir = m (flipDirection dir)
```

Flipping matchers

This is a major change: most of the matchers have to have 2 descriptions now: one for positive and one for negative direction.

```
eq x = predicate (== x)
       ("is a value equal to " ++ show x)
       ("is a value not equal to" ++ show x)
                    ~^~
                    "not" is in the right place now!
```

Tying it all together

```haskell
data User
   = User
   { user_id   :: Int
   , user_name :: String
   } deriving (Show)

getUser :: UserService → Int → IO (Either String User)

getUser service userId `shouldMatchIO` rightIs (allOf
  [ projection "user_id"   user_id   (eq userId)
  , projection "user_name" user_name isNotEmpty
  ])
```

Tying it all together: the error message

```
✗ prism "Right" ← <1>
  ✗ all of ← <2>
    ✓ projection "user_id" ← <2>
      ✓ is a value equal to 5 ← 5
    ✗ projection "user_name" ← <2>
      ✗ is not empty ← ""
Where:
  <1> Right (User {user_id = 5, user_name = ""})
  <2> User {user_id = 5, user_name = ""}
```

What's next?

- Publish on Hackage.
- Better error messages!
- More combinators!
- Extra info for debugging, e.g. structured diffs.
- **Control.Lens** integration path.
- Generating matchers for ADTs via **GHC.Generics**.

Summary

- **Make sure your test fails with the broken code and succeeds with the correct one.**
- It's certainly possible to make error messages better than they're now.
- `test-matchers` is coming soon, stay tuned! Any feedback is appreciated.