Better Types ⇒ Fewer Tests

Who am I?

@raulraja
@47deg

- Co-Founder and CTO at 47 Degrees
- Scala advisory board member
- FP advocate
- Electric Guitar @ <Ben Montoya & the Free Monads>

More tests ≒ Better Software?

What are we testing?

Testing : Programs

Programs

```
class Counter(var amount: Int) {
  require(amount ⩾ 0, s"($amount seed value) must be a positive integer")
  def increase(): Unit =
    amount += 1
}
```

What are we testing? ⇒ Input values

```
class CounterSpec extends BaseTest {
  test("Can't be constructed with negative numbers") {
    the [IllegalArgumentException] thrownBy {
      new Counter(-1)
    } should have message "requirement failed: (-1 seed value) must be a positive integer"
  }
}
// defined class CounterSpec

(new CounterSpec).execute
// CounterSpec:
// - Can't be constructed with negative numbers
```

What are we testing? ⇒ Side effects

```
class CounterSpec extends BaseTest {
  test("`Counter#amount` is mutated after `Counter#increase` is invoked") {
    val counter = new Counter(0)
    counter.increase()
    counter.amount shouldBe 1
  }
}
// defined class CounterSpec

(new CounterSpec).execute
// CounterSpec:
// - `Counter#amount` is mutated after `Counter#increase` is invoked
```

What are we testing? ⇒ Output values

```
class CounterSpec extends BaseTest {
  test("`Counter#amount` is properly initialized") {
    new Counter(0).amount shouldBe 0
  }
}
// defined class CounterSpec

(new CounterSpec).execute
// CounterSpec:
// - `Counter#amount` is properly initialized
```

What are we testing? ⇒ Runtime

Our component is now distributed and may fail

```
import scala.concurrent._
import cats.data.NonEmptyList
import scala.concurrent.ExecutionContext.Implicits.global
import java.util.concurrent.atomic.AtomicInteger
import scala.util.control._

sealed abstract class KnownError extends Throwable with NoStackTrace
case object ServiceUnavailable extends KnownError
case class CounterOutOfRange(msg: NonEmptyList[String]) extends KnownError

class FutureCounter(val amount: AtomicInteger) {
  require(amount.get ⩾ 0, s"($amount seed value) must be a positive atomic integer")
  def increase(): Future[Either[KnownError, Int]] =
    Future(Right(amount.incrementAndGet)) // mocked for demo purposes
}
```

What are we testing? ⇒ Runtime

Changes in requirements

```
class FutureCounterSpec extends BaseTest {
  test("`FutureCounter#amount` is mutated after `FutureCounter#increase` is invoked") {
    val counter = new FutureCounter(new AtomicInteger(0))
    counter.increase()
    counter.amount.get shouldBe 1
  }
}
// defined class FutureCounterSpec

(new FutureCounterSpec).execute
// FutureCounterSpec:
// - `FutureCounter#amount` is mutated after `FutureCounter#increase` is invoked *** FAILED ***
//   0 was not equal to 1 (<console>:26)
```

What are we testing? ⇒ Runtime

Changes in requirements

```
import scala.concurrent.duration._
// import scala.concurrent.duration._

class FutureCounterSpec extends BaseTest {
  test("`FutureCounter#amount` is mutated after `FutureCounter#increase` is invoked") {
    val counter = new FutureCounter(new AtomicInteger(0))
    val result = counter.increase() map { _ => counter.amount.get shouldBe 1 }
    Await.result(result, 10.seconds)
  }
}
// defined class FutureCounterSpec

(new FutureCounterSpec).execute
// FutureCounterSpec:
// - `FutureCounter#amount` is mutated after `FutureCounter#increase` is invoked
```

What are we testing?

- **Input values** are in range of acceptance
  (-N is not)
- **Side effects** caused by programs
  (counter is mutated in the outter scope)
- Programs produce expected **output values** given correct input values.
  (counter value is consistent with our biz logic)
- **Runtime** machinery
  (The program may work sync/async and it may fail)

What are we NOT testing?

We don't test for: Invariants

We don't test for: Invariants

In computer science, an invariant is a condition that can be relied upon to be true during execution of a program

— Wikipedia Invariant*(computer*science)

We don't test for: Invariants

In computer science, an invariant is a condition that can be relied upon to be true during execution of a program

- Compilation: We trust the compiler says our values will be constrained by properties
- Math Laws: (identity, associativity, commutativity, ...)
- 3rd party dependencies

The Dark Path

Now, **ask yourself why these defects happen too often**.
If your answer is that our languages don't prevent them,
then I strongly suggest that you quit your job and never
think about being a programmer again;
because defects are never the fault of our languages.
Defects are the fault of programmers.
It is programmers who create defects – not languages.

— Robert C. Martin (Uncle Bob) `The Dark Path`

The Dark Path

And **what is it that programmers are supposed to do to prevent defects?**
I'll give you one guess. Here are some hints.
It's a verb. It starts with a "T". Yeah.
You got it. **TEST!**

— Robert C. Martin (Uncle Bob) `The Dark Path`

The Dark Path

And **what is it that programmers are supposed to do to prevent defects?**
I'll give you one guess. Here are some hints.
It's a verb. It starts with a "T". Yeah.
You got it. **TEST!**

— Robert C. Martin (Uncle Bob) `The Dark Path`

Does our programming style affect the way we test?

What is Functional Programming?

In computer science, functional programming
is a programming paradigm.
A style of building the structure and elements
of computer programs that treats computation
as the evaluation of mathematical functions
and avoids changing-state and mutable data.

-- Wikipedia

Common traits of Functional Programming

- Higher-order functions
- Immutable data
- Referential transparency
- Lazy evaluation
- Recursion
- Abstractions

What are we testing?

Back to our original concerns

- **Input values are in range of acceptance**
- Programs produce an expected output value given an accepted input value.
- Side effects caused by programs
- Changes in requirements

What are we testing? ⇒ Input values

`counter: Int` is a poorly chosen type. Let's fix that!

```
class CounterSpec extends BaseTest {
  test("Can't be constructed with negative numbers") {
    the [IllegalArgumentException] thrownBy {
      new Counter(-1)
    } should have message "requirement failed: (-1 seed value) must be a positive integer"
  }
}
// defined class CounterSpec

(new CounterSpec).execute
// CounterSpec:
// - Can't be constructed with negative numbers
```

What are we testing? ⇒ Input values

Refining Int constrains our values at compile and runtime

```
import eu.timepit.refined.W
import eu.timepit.refined.cats.syntax._
import eu.timepit.refined.api.{Refined, RefinedTypeOps}
import eu.timepit.refined.numeric._

type Zero = W.`0`.T
type Ten = W.`10`.T
type Amount = Int Refined Interval.Closed[Zero, Ten]
object Amount extends RefinedTypeOps[Amount, Int]

class Counter(var amount: Amount) {
  def increase(): Unit =
    Amount.from(amount.value + 1).foreach(v => amount = v)
}
```

What are we testing? ⇒ Input values

The compiler can verify the range and we can properly type amount

```
+ import eu.timepit.refined.api.{Refined, RefinedTypeOps}
+ import eu.timepit.refined.numeric._

+ type Zero = W.`0`.T
+ type Ten = W.`10`.T
+ type Amount = Int Refined Interval.Closed[Zero, Ten]
+ object Amount extends RefinedTypeOps[Amount, Int]

- class Counter(var amount: Int) {
+ class Counter(var amount: Amount) {
-   require(amount ⩾ 0, s"($amount seed value) must be a positive integer")
    def increase(): Unit =
-     amount += 1
+     Amount.from(amount.value + 1).foreach(v => amount = v)
  }
```

What are we testing? ⇒ Input values

We can still test this but this test proves nothing

```
import eu.timepit.refined.scalacheck.numeric._
// import eu.timepit.refined.scalacheck.numeric._

class CounterSpec extends BaseTest {
  test("`Amount` values are within range") {
    check((amount: Amount) => amount.value ⩾ 0 && amount.value ⩽ 10)
  }
}
// defined class CounterSpec

(new CounterSpec).execute
// CounterSpec:
// - `Amount` values are within range
```

What are we testing? ⇒ Input values

The compiler can verify the range and we can properly type amount

```
class CounterSpec extends BaseTest {
-  test("Can't be constructed with negative numbers") {
-    the [IllegalArgumentException] thrownBy {
-      new Counter(-1)
-    } should have message "requirement failed: (-1 seed value) must be a positive integer"
-  }
}
```

What are we testing?

Back to our original concerns

- ~~Input values are in range of acceptance~~
- **Side effects caused by programs**
- Programs produce an expected output value given an accepted input value.
- Changes in requirements

What are we testing? ⇒ Side effects

```
class CounterSpec extends BaseTest {
  test("`Counter#amount` is mutated after `Counter#increase` is invoked") {
    val counter = new Counter(Amount(0))
    counter.increase()
    counter.amount.value shouldBe 1
  }
}
// defined class CounterSpec

(new CounterSpec).execute
// CounterSpec:
// - `Counter#amount` is mutated after `Counter#increase` is invoked
```

What are we testing? ⇒ Side effects

```
class Counter(var amount: Amount) { // mutable
  def increase(): Unit = // Unit does not return anything useful
    Amount.from(amount.value + 1).foreach(v => amount = v) // mutates the external scope
}
```

What are we testing? ⇒ Side effects

No need to test side effects if functions are **PURE!**

```
class Counter(val amount: Amount) { // values are immutable
  def increase(): Either[KnownError, Counter] = // Every operation returns an immutable copy
    Amount.validate(amount.value + 1).fold( // Amount.validate does not need to be tested
     { errors => Left(CounterOutOfRange(errors)) }, // No Exceptions are thrown
     { a => Right(new Counter(a)) }
    )
}
```

What are we testing? ⇒ Side effects

Side effects caused by programs.

```
- class Counter(var amount: Amount) { // mutable
+ class Counter(val amount: Amount) { // values are immutable
-  def increase(): Unit = // Unit does not return anything useful
+  def increase(): Counter = // Every operation returns an immutable copy
-    Amount.from(amount.value + 1).foreach(v => amount = v) // mutates the external scope
+    Amount.validate(amount.value + 1).fold( // Amount.validate does not need to be tested
+      { errors => Left(CounterOutOfRange(errors)) }, // No Exceptions are thrown
+      { a => Right(new Counter(a)) }
+    )
}
```

What are we testing?

Back to our original concerns

- ~~Input values are in range of acceptance~~
- ~~Side effects caused by programs~~
- **Programs produce an expected output value given an accepted input value**
- Changes in requirements

What are we testing? ⇒ Output values

Programs produce an expected output value given an accepted input

```
class CounterSpec extends BaseTest {
  test("`Counter#amount` is immutable and pure") {
    new Counter(Amount(0)).increase().map(_.amount) shouldBe Right(Amount(1))
  }
}
// defined class CounterSpec

(new CounterSpec).execute
// CounterSpec:
// - `Counter#amount` is immutable and pure
```

What are we testing?

Back to our original concerns

- ~~Input values are in range of acceptance~~
- ~~Side effects caused by programs~~
- Programs produce an expected output value given an accepted input value
- **Runtime requirements**

What are we testing? ⇒ Runtime

Changes in runtime requirements force us to consider other effects (async, failures,...)

```
class FutureCounter(val amount: Amount) { // values are immutable
  def increase(): Future[Either[KnownError, Counter]] = // Every operation returns an immutable copy
    Future {
      Amount.validate(amount.value + 1).fold( // Amount.validate does not need to be tested
       { error => Left(CounterOutOfRange(error)) }, // potential failures are also contemplated
       { a => Right(new Counter(a)) }
      )
    }
}
```

What are we testing? ⇒ Runtime

We are forcing call sites to block even those that did not want to be async

```
class CounterSpec extends BaseTest {
  test("`FutureCounter#amount` is immutable and pure") {
    val asyncResult = new FutureCounter(Amount(0)).increase()
    Await.result(asyncResult, 10.seconds).map(_.amount) shouldBe Right(Amount(1))
  }
}
// defined class CounterSpec

(new CounterSpec).execute
// CounterSpec:
// - `FutureCounter#amount` is immutable and pure
```

What are we testing? ⇒ Runtime

Concrete data types lack flexibility and increase the chance of bugs

```
class FutureCounter(val amount: Amount) {
  def increase: Future[Either[KnownError, Counter]] =
    Future {
      Amount.validate(amount.value + 1).fold(
       { error => Left(CounterOutOfRange(error)) },
       { a => Right(new Counter(a)) }
      )
    }
}
```

What are we testing? ⇒ Runtime

Everything we need to describe computation

Type class	Combinator
`Functor`	`map`, `lift`
`Applicative`	`pure`, `ap`
`ApplicativeError`	`raiseError`, `catch`
`Monad`	`flatMap`, `flatten`
`MonadError`	`ensure`, `rethrow`
`Sync`	`delay`, `suspend`
`Async`	`async`
`Effect`	`toIO`

What are we testing? ⇒ Runtime

Everything we need to describe combination

Type class	Combinator
`Semigroup`	`combine`
`Monoid`	`empty`
`Foldable`	`foldLeft`, `foldRight`
`Traverse`	`traverse`, `sequence`

What are we testing? ⇒ Runtime

Abstract type classes increase flexibility and decrease the chance of bugs

```
import cats.effect.Sync

// F[_] can be any box for which `Sync` instance is available
class Counter[F[_]](val amount: Amount)(implicit F: Sync[F]) {
  // A counter is returned in a generic box
  def increase(): F[Counter[F]] = {
    F.suspend { // Effects in the block are deferred
      Amount.validate(amount.value + 1).fold(
       { error => F.raiseError(CounterOutOfRange(error)) },
       { a => F.pure(new Counter(a)) }
      )
    }
  }
}
```

What are we testing? ⇒ Runtime

Our tests may also be polymorphic

```
import cats.effect._
import cats.effect.implicits._

class CounterSpec[F[_]](implicit F: Effect[F]) extends BaseTest {
  test("`Counter#amount` is immutable and pure") {
    val result: F[Counter[F]] = new Counter[F](Amount(0)).increase()
    F.toIO(result).unsafeRunSync().amount shouldBe Amount(1)
  }
}
```

What are we testing? ⇒ Runtime

A la carte

```
(new CounterSpec[IO]).execute
// CounterSpec:
// - `Counter#amount` is immutable and pure
```

What are we testing? ⇒ Runtime

A la carte

```
import monix.eval.Task
// import monix.eval.Task

import monix.execution.Scheduler.Implicits.global
// import monix.execution.Scheduler.Implicits.global

(new CounterSpec[Task]).execute
// CounterSpec:
// - `Counter#amount` is immutable and pure
```

What are we testing? ⇒ Runtime

A la carte (only for the well behaved)

```
(new CounterSpec[Future]).execute // fails to compile, because Future can't suspend effects
// <console>:46: error: Cannot find implicit value for Effect[scala.concurrent.Future].
// Building this implicit value might depend on having an implicit
// s.c.ExecutionContext in scope, a Scheduler or some equivalent type.
//        (new CounterSpec[Future]).execute // fails to compile, because Future can't suspend effects
//         ^
```

What are we testing?

Back to our original concerns

- ~~Input values are in range of acceptance~~
- ~~Side effects caused by programs~~
- Programs produce an expected output value given an accepted input value
- ~~Runtime requirements~~

The Dark Path

I disagree & so does the Compiler

And **what is it that programmers are supposed to do to prevent defects?**

I'll give you one guess. Here are some hints.

It's a verb. It starts with a "T". Yeah.

You got it. ~~TEST!~~.,

The Bright Path

And what is it that programmers are supposed to do to prevent defects?
I'll give you one guess. Here are some hints.
It's a verb. It starts with a "T". Yeah.
You got it. ~~TEST!~~ **TYPES!**,

-- Compiler.

The Bright Path

And what is it that programmers are supposed to do to prevent defects?
I'll give you one guess. Here are some hints.
It's a verb. It starts with a "T". Yeah.
You got it. ~~TEST!~~ **TYPED FP!**,

-- Compiler.

Better Types ⇒ Fewer Tests

Better Types ⇒ Better Programs

Thanks!

Better Types ⇒ Better Programs