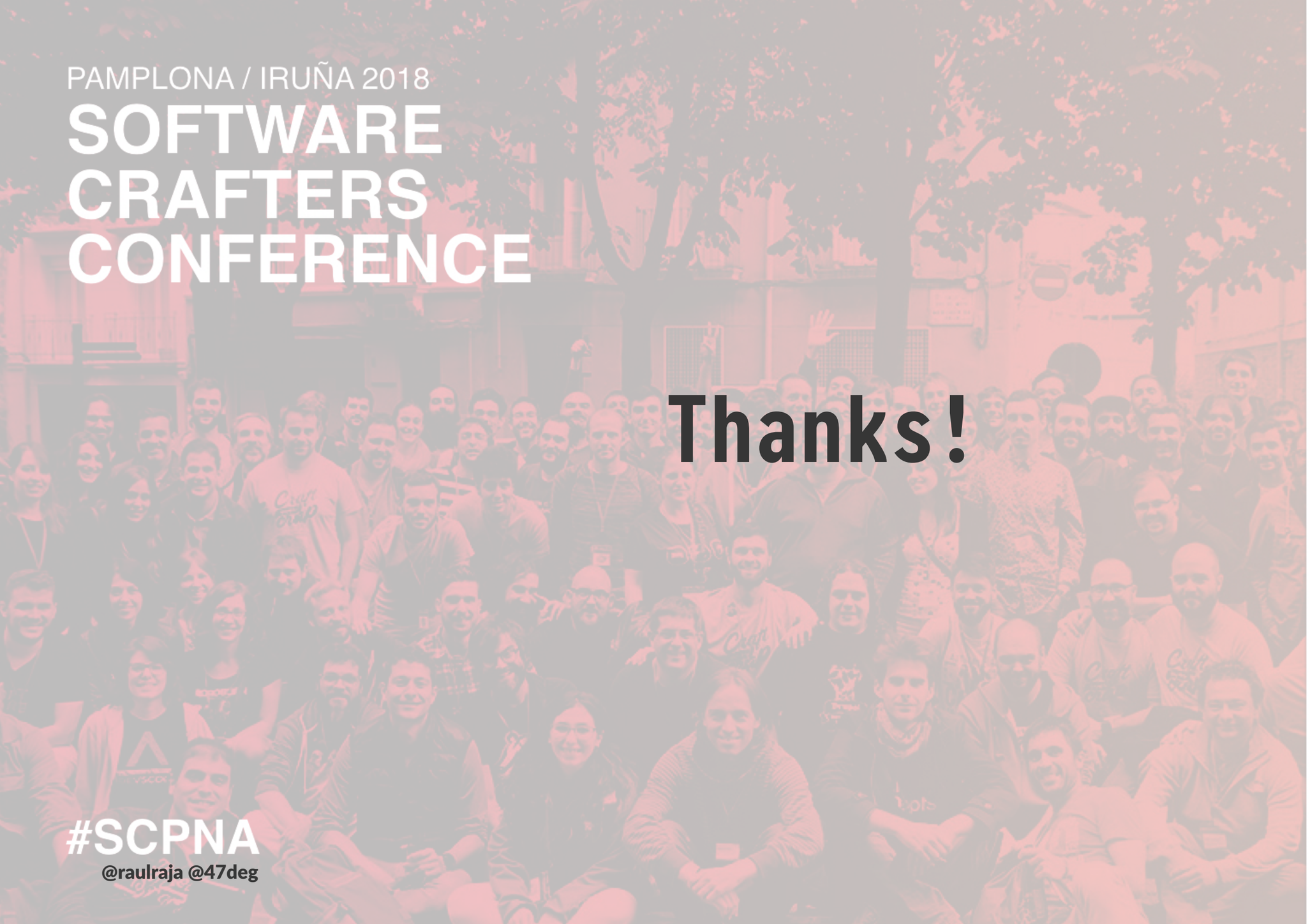
Better Types ⇒ Fewer Tests

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>



Thanks!

#SCPNA

@raulraja @47deg

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

```
import scala.concurrent._
import scala.concurrent.ExecutionContext.Implicits.global
import java.util.concurrent.atomic.AtomicInteger

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

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>:22)
```

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. etc...)

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?

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

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

Stronger refinement for 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 Amount = Int Refined GreaterEqual[W.`0`.T]
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 Amount = Int Refined GreaterEqual[W.`0`.T]
+ 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

```
class CounterSpec extends BaseTest {
  // Testing an invariant xD
  test("Can't compile with literal negative number") {
    "new Counter(-1)" shouldNot compile
  }
}
// defined class CounterSpec

(new CounterSpec).execute
// CounterSpec:
// - Can't compile with literal negative number
```

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(): Counter = // Every operation returns an immutable copy
    Amount.validate(amount.value + 1).fold( // Amount.validate does not need to be tested
     { _ => new Counter(amount) }, // potential failures are also contemplated
     { a => 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
+     { _ => new Counter(amount) }, // potential failures are also contemplated
+     { a => 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().amount shouldBe 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 made us realize our component needed to support also async computations

```
class FutureCounter(val amount: Amount) { // values are immutable
  def increase(): Future[Counter] = // Every operation returns an immutable copy
    Future {
      Amount.validate(amount.value + 1).fold( // Amount.validate does not need to be tested
       { _ => new Counter(amount) }, // potential failures are also contemplated
       { a => 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") { // The actual test case
    val asyncResult = new FutureCounter(Amount(0)).increase()
    Await.result(asyncResult, 10.seconds).amount shouldBe Amount(1)
  }
}
// defined class CounterSpec

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

What are we testing? ⇒ Runtime

Most specialized implementations denote insufficient polymorphism

```
class FutureCounter(val amount: Amount) {
  def increase: Future[Counter] = // increase returns immediately and starts its computation async
    Future {
      Amount.validate(amount.value + 1).fold( // Amount.validate does not need to be tested
       { _ => new Counter(amount) }, // potential failures are also contemplated
       { a => new Counter(a) }
      )
    }
}
```

What are we testing? ⇒ Runtime

We reduce the possibility of bugs and increase flexibility by working with abstractions such as type classes

```
import cats.effect.Sync

class Counter[F[_]: Sync](val amount: Amount) { // F[_] can be any box for which `Sync` instance is available
  def increase(): F[Counter[F]] = // A counter is returned in a generic box
    Sync[F].delay { //delay defers execution of the contained block
      Amount.validate(amount.value + 1).fold(
       { _ => new Counter(amount) },
       { a => new Counter(a) }
      )
    }
}
```

What are we testing? ⇒ Runtime

Our program is now polymorphic and the same code supports many different data types.

```
import monix.eval.Task
import cats.effect.IO
import cats.effect.implicits._

new Counter[IO](Amount(0)).increase
// res14: cats.effect.IO[Counter[cats.effect.IO]] = IO$2007768453

new Counter[Task](Amount(0)).increase
// res15: monix.eval.Task[Counter[monix.eval.Task]] = Task.Eval$1773896745
```

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

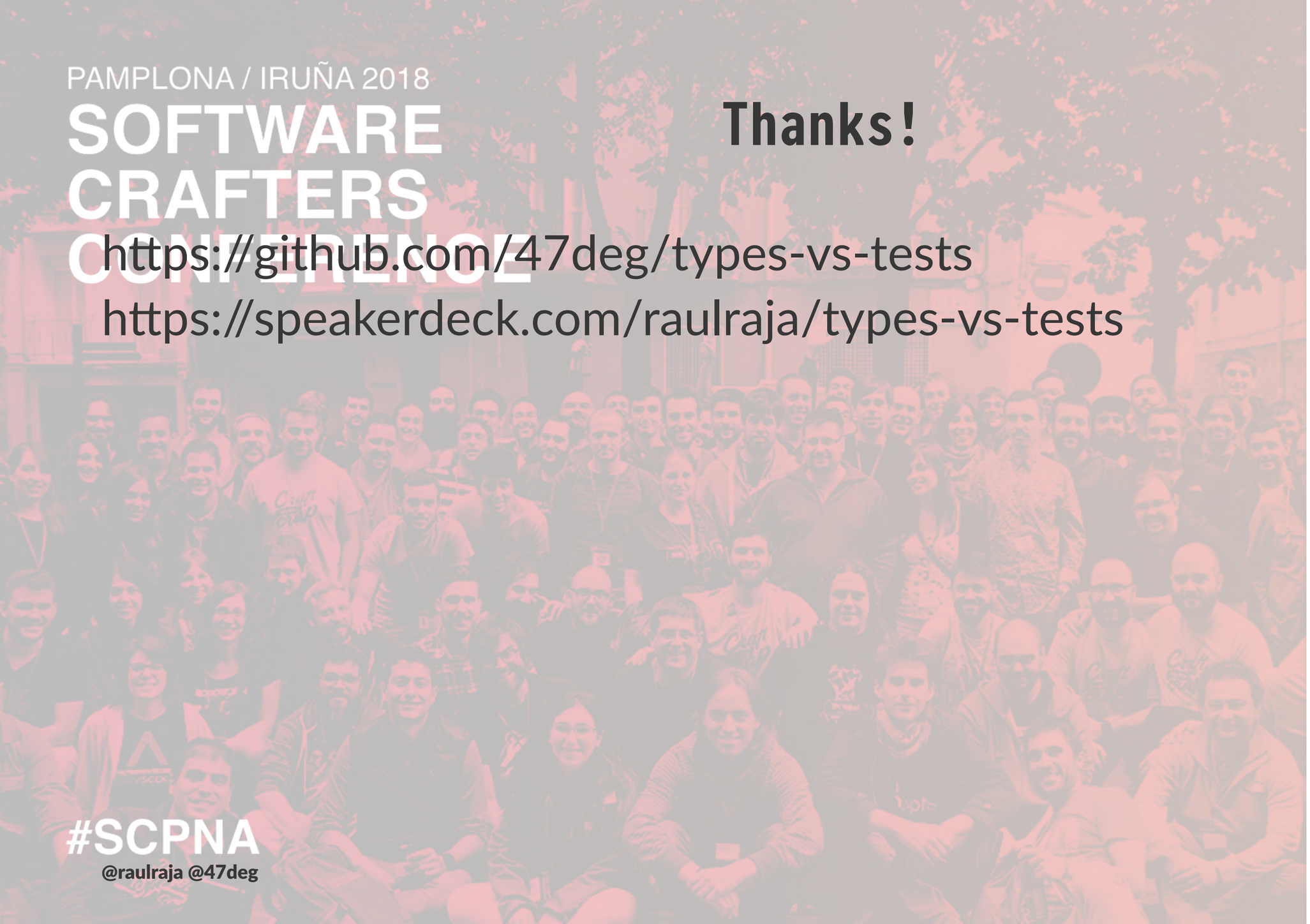
And what is it that programmers are supposed to do to prevent defects?
I'll give you one guess. Here are some hints.
It's a verb. It starts with a "T". Yeah.
You got it. ~~TEST!~~ **TYPES!**,

-- 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.



Thanks!

https://github.com/47deg/types-vs-tests

https://speakerdeck.com/raulraja/types-vs-tests

#SCPNA

@raulraja @47deg

etermax Berlin Studio

flywire

autentia

bikos

CODESAI

idealista

aleasoluciones

CodelyTV

StubHub

540