Scala for all

property-based testing met ScalaCheck

Exception Twente - 16 februari 2017

scala>

Philippus Baalman

Scala en Akka developer bij Wehkamp

scala>

An order with order lines


```
scala>
```

```
case class Order(
    id: String,
    orderTime: DateTime,
    orderState: String,
    customerType: Option[CustomerType],
    orderLines: List[OrderLine]) {
  def addOrderLine(orderLine: OrderLine): Order = {
    this.copy(orderLines = orderLine :: orderLines)
  }
  def total: Int = orderLines.map(x => x.total).sum
}

sealed abstract class CustomerType(override val toString: String)
case object NewCustomer extends CustomerType("new")
case object ExistingCustomer extends CustomerType("existing")

case class OrderLine(
    id: Option[String],
    productId: String,
    productCategory: String,
    quantity: Int,
    netValue: Int) {
  def total: Int = quantity * netValue
}
```



```
scala>
```

```
val testOrderLine = OrderLine(Option("a"), "b", "000", 1, 2)

val testOrder = Order(
  "a", DateTimeSupport.stringToDateTime("20161209T10:50:24.091Z"), "ordered",
  Some(NewCustomer), List(testOrderLine))

"Order" should {
  "be able to add an order line to an order" in {
    testOrder.addOrderLine(testOrderLine).orderLines.length
    shouldEqual testOrder.orderLines.length + 1
  }
}
```


scala>

```
"Order" should {
  "calculate total correctly" in {
    testOrder.total shouldEqual testOrderLine.total
  }
}
```


scala>

```
val testOrderWithoutOrderLines = Order(
  "a", DateTimeSupport.stringToDateTime("20161209T10:50:24.091Z"), "ordered",
  Some(NewCustomer), List())
```


scala>

```
"Order" should {
  "be able to add an order line to an order" in {
    testOrder.addOrderLine(testOrderLine).orderLines.length
    shouldEqual testOrder.orderLines.length + 1
  }

  "be able to add an order line to an order without order lines" in {
    testOrderWithoutOrderLines.addOrderLine(testOrderLine).orderLines.length
    shouldEqual 1
  }

  "be able to add an order line to an order" in {
    testOrder.addOrderLine(testOrderLine).orderLines.length
    shouldEqual testOrder.orderLines.length + 1
  }

  "calculate total correctly for orders without order lines" in {
    testOrderWithoutOrderLines.total shouldEqual 0
  }
}
```


scala>

```
val testOrderLine = OrderLine(Option("a"), "b", "000", 1, 2)

val jsonOrderLine =
  """
    {"id":"a","productId":"b","productCategory":"000","quantity":1,"netValue":2}
  """.parseJson

"OrderLineMarshalling" should {
  "correctly serialize an order line" in {
    jsonOrderLine shouldEqual testOrderLine.toJson
  }
  "correctly deserialize an order line" in {
    jsonOrderLine.convertTo[OrderLine] shouldEqual testOrderLine
  }
}
```


scala>

```
*****************************************************************************************
*                                                                                       *
*                                                                                       *
*                                                                                       *
*         val testOrder = Order(                                                        *
*           "a", DateTimeSupport.stringToDateTime("20161209T10:50:24.091Z"), "ordered",  *
*           Some(NewCustomer), List(testOrderLine))                                     *
*                                                                                       *
*         val jsonOrder =                                                               *
*           """                                                                         *
*             {"orderLines":[{"id":"a","productId":"b","productCategory":"000","quantity":1,  *
*             "netValue":2}],"id":"a","orderTime":"20161209T10:50:24.091Z",             *
*             "orderState":"ordered","customerType":"new"}                              *
*           """.parseJson                                                               *
*                                                                                       *
*         "OrderMarshalling" should {                                                   *
*           "correctly serialize an order" in {                                         *
*             jsonOrder shouldEqual testOrder.toJson                                    *
*           }                                                                           *
*           "correctly deserialize an order" in {                                       *
*             jsonOrder.convertTo[Order] shouldEqual testOrder                          *
*           }                                                                           *
*         }                                                                             *
*                                                                                       *
*                                                                                       *
*                                                                                 10/90 *
*****************************************************************************************
scala>
```

```
val testOrder = Order(
  "a", DateTimeSupport.stringToDateTime("20161209T10:50:24.091Z"), "ordered",
  Some(NewCustomer), List(testOrderLine))

val jsonOrder =
  """
    {"orderLines":[{"id":"a","productId":"b","productCategory":"000","quantity":1,
    "netValue":2}],"id":"a","orderTime":"20161209T10:50:24.091Z",
    "orderState":"ordered","customerType":"new"}
  """.parseJson

val testOrderWithoutOrderLines = Order(
  "a", DateTimeSupport.stringToDateTime("20161209T10:50:24.091Z"), "ordered",
  Some(NewCustomer), List())

val jsonOrderWithoutOrderLines =
  """
    {"orderLines":[],"id":"a","orderTime":"20161209T10:50:24.091Z",
    "orderState":"ordered","customerType":"new"}
  """.parseJson
```


scala>

```
"OrderMarshalling" should {
  "correctly serialize an order" in {
    jsonOrder shouldEqual testOrder.toJson
    }
  "correctly deserialize an order" in {
    jsonOrder.convertTo[Order] shouldEqual testOrder
  }
  "correctly serialize an order without order lines" in {
    jsonOrderWithOrderLines shouldEqual testOrderWithoutOrderLines.toJson
  }
  "correctly deserialize an order without order lines" in {
    jsonOrderWithOrderLines.convertTo[Order] shouldEqual testOrderWithoutOrderLines
  }
}
```


scala>

Bij property-based testing definieer je eigenschappen die gelden voor de output van je code, gebaseerd op de input.

Properties

scala>

```scala
import org.scalacheck.Prop._

val propInt = forAll {
  i: Int =>
    i + 0 == i
}
```


scala>

```
propInt.check
```



```
scala>
```

```scala
val propEndsWith = forAll {
  (a: String, b: String) =>
    (a + b).endsWith(b)
}
```


scala>

```
propEndsWith.check
```



```
scala>
```

```
val propList = forAll {
  l: List[Int] =>
    l.reverse == l
}
```


scala>

```
propList.check
```



```
scala>
```

```
val propReverseList = forAll {
  l: List[String] =>
    l.reverse.reverse == l
}
```


scala>

```
propReverseList.check
```



```
scala>
```

```scala
val propSumInt = forAll {
  (i: Int, j: Int) =>
    i + j == j + i
}
```


scala>

```
propSumInt.check
```



```
scala>
```

```scala
val propUpperThenLower = forAll {
  s: String =>
    s.toUpperCase.toLowerCase == s.toLowerCase
}
```


scala>

```
propUpperThenLower.check
```



```
scala>
```

```scala
val propAbs = forAll {
  i: Int =>
    Math.abs(i) >= 0
}
```


scala>

```
propAbs.check
```



```
scala>
```

```
val propMultCommutative = forAll {
  (i: Int, j: Int) =>
    i * j == j * i
}

val propMultAssociative = forAll {
  (i: Int, j: Int, k: Int) =>
    i * (j * k) == (i * j) * k
}
```


scala>

```
(propMultCommutative && propMultAssociative).check
```



```
scala>
```

Bij property-based testing definieer je eigenschappen die gelden voor de output van je code, gebaseerd op de input, deze worden geverifieerd voor heel veel verschillende gegeneerde input.

Generators

scala>

```scala
val propAbs = forAll {
  i: Int =>
    collect(i) {
      Math.abs(i) >= 0
    }
}

propAbs.check
```


scala>

```scala
import org.scalacheck.Arbitrary.arbitrary
import org.scalacheck.Gen

arbitrary[String].sample
```


scala>

```
arbitrary[Int].sample
```



```
scala>
```

```
arbitrary[(Int, String)].sample
```



```
scala>
```

```
Gen.listOfN(3, arbitrary[Int]).sample
```



```
scala>
```

```
Gen.posNum[Int].sample
```



```
scala>
```

```
Gen.alphaStr.sample
```



```
scala>
```

```
val propUpperThenLowerAlpha = forAll(Gen.alphaStr) {
  s: String =>
    s.toUpperCase.toLowerCase == s.toLowerCase
}
```


scala>

```
propUpperThenLowerAlpha.check
```



```
scala>
```

Bij property-based testing definieer je eigenschappen die gelden voor de output van je code, gebaseerd op de input, deze worden geverifieerd voor heel veel verschillende gegeneerde input, als er een fout gevonden wordt dan wordt er geprobeerd een minimaal tegenvoorbeeld te vinden.

Shrinking

scala>

```scala
val propList = forAll {
  l: List[Int] =>
    l.reverse == l
}
```


scala>

```
propList.check
```



```
scala>
```

```scala
val propUpperThenLower = forAll {
  s: String =>
    println(s)
    s.toUpperCase.toLowerCase == s.toLowerCase
}
```


scala>

```
propUpperThenLower.check
```



```
scala>
```

```scala
import org.scalacheck.Shrink
import org.scalacheck.Shrink.shrink

val shrinkInt = Shrink.shrink(123)
```


scala>

```
shrinkInt.take(10).print
```



```
scala>
```

Enkele patterns


```
scala>
```

"Different paths, same destination"

```
|-----|  y   |-----|
| ABC | ---> |     |
|-----|      |-----|
 x |            | x
   v            v
|-----|      |-----|
|     | ---> | 123 |
|-----|  y   |-----|
```


scala>

```
forAll(Gen.listOf(Gen.posNum[Int])) {
  l: List[Int] => l.map(_ + 1).reverse == l.reverse.map(_ + 1)
}.check
```


scala>

"There and back again"

```
          x
 |-----| ---> |-----|
 | ABC |      |01010|
 |-----| <--- |-----|
       inverse
```


scala>

```
forAll {
  l: List[Int] => l == l.reverse.reverse
}.check
```


scala>

"Some things never change"

```
|-----|      |-----|
| ABC | ---> | CBA |
|-----|      |-----|
```


scala>

```
forAll {
  l: List[Int] => l.length == l.reverse.length
}.check
```


scala>

"The more things change, the more they stay the same"

```
|-----| .... |-----| .... |- - -|
| CAB | ---> | A B |  - > | A B |
|-----|      |-----|      |- - -|
```


scala>

```
forAll {
  l: List[Int] => l.distinct == l.distinct.distinct
}.check
```


scala>

"Solve a smaller problem first"

```
------      -------      --------
|ABCD| ---> |A|BCD| ---> |A|B|CD|
------      -------      --------
```


scala>

```
def checkList(l: List[Int]): Boolean = l match {
    case a :: b :: rest => (a <= b) && checkList(b :: rest)
    case _ => true
}

forAll {
  l:List[Int] => checkList(l.sorted)
}.check
```


scala>

"Hard to prove, easy to verify"

scala>

```scala
def sieve(s: Stream[Int]): Stream[Int] = {
    s.head #:: sieve(s.tail.filter(_ % s.head != 0))
}
val primes = sieve(Stream.from(2))

def isPrime(i: Int): Boolean = !((2 until i-1) exists (i % _ == 0))
```


scala>

```
forAll(Gen.choose(1, 1000)) {
  n: Int => isPrime(primes.drop(n).head)
}.check
```


scala>

```
"The test oracle"

|-----| SUT  |-----|
| ABC | ---> | 123 |
|-----|      |-----|
               ||
               ||
|-----| test |-----|
| ABC | ---> | 123 |
|-----|oracle|-----|
```


scala>

```scala
def bubbleSort(ls: List[Int], size: Int): List[Int] = size match {
  case 0 =>
    ls
  case n =>
    bubbleSort(ls.foldLeft(List.empty[Int]) { (xs, next) =>
      xs match {
        case Nil => next :: xs
        case _ if xs.length >= size => xs :+ next
        case _ if xs.last > next => (xs.init :+ next) :+ xs.last
        case _ => xs :+ next
      }
    }, size - 1)
}

def qsort(list: List[Int]): List[Int] = list match {
  case Nil =>
    Nil
  case head :: tail =>
    qsort(tail.filter(_ < head)) ::: head :: qsort(tail.filter(_ >= head))
}
```


scala>

```
forAll {
  l: List[Int] => qsort(l) == bubbleSort(l, l.length)
}.check
```



```
scala>
```

An order revisited


```
scala>
```

```scala
case class Order(
    id: String,
    orderTime: org.joda.time.DateTime,
    orderState: String,
    customerType: Option[CustomerType],
    orderLines: List[OrderLine]) {
  def addOrderLine(orderLine: OrderLine): Order = {
    this.copy(orderLines = orderLine :: orderLines)
  }
  def total: Int = orderLines.map(x => x.total).sum
}

sealed abstract class CustomerType(override val toString: String)
case object NewCustomer extends CustomerType("new")
case object ExistingCustomer extends CustomerType("existing")

case class OrderLine(
    id: Option[String],
    productId: String,
    productCategory: String,
    quantity: Int,
    netValue: Int) {
  def total: Int = quantity * netValue
}
```


scala>

```scala
case class OrderLine(
    id: Option[String],
    productId: String,
    productCategory: String,
    quantity: Int,
    netValue: Int)

import org.scalacheck.Gen._

def genOrderLineId: Gen[String] = for {
  left ← Gen.listOfN(1, Gen.numChar).map(_.mkString)
  middle ← Gen.const("x")
  right ← Gen.listOfN(5, Gen.numChar).map(_.mkString)
} yield List(left, middle, right).mkString
```


scala>

genOrderLineId.sample

scala>

```scala
case class OrderLine(
    id: Option[String],
    productId: String,
    productCategory: String,
    quantity: Int,
    netValue: Int)

def genOrderLine: Gen[OrderLine] = for {
  id ← Gen.option(genOrderLineId)
  productId ← Gen.resize(3, Gen.alphaStr)
  productCategory ← Gen.resize(5, Gen.alphaStr)
  quantity ← Gen.choose(1, 10)
  netValue ← Gen.choose(0, 1000)
} yield OrderLine(id, productId, productCategory, quantity, netValue)
```


scala>

genOrderLine.sample

scala>

```scala
case class Order(
    id: String,
    orderTime: DateTime,
    orderState: String,
    customerType: Option[CustomerType],
    orderLines: List[OrderLine])

import org.joda.time.{DateTime, DateTimeZone}

def genOrderTime: Gen[DateTime] = {
  val start = new DateTime(2017, 1, 1, 0, 0, 0, 0, DateTimeZone.UTC).getMillis
  val end = new DateTime(2018, 1, 1, 0, 0, 0, 0, DateTimeZone.UTC).getMillis
  Gen.choose(start, end).map(new DateTime(_))
}
```


scala>

genOrderTime.sample

scala>

```
case class Order(
    id: String,
    orderTime: DateTime,
    orderState: String,
    customerType: Option[CustomerType],
    orderLines: List[OrderLine])

def genOrderState = oneOf("ordered", "packaged", "shipped", "delivered")
```


scala>

genOrderState.sample

scala>

Tekst
Tekst
Tekst

```
case class Order(
    id: String,
    orderTime: DateTime,
    orderState: String,
    customerType: Option[CustomerType],
    orderLines: List[OrderLine])

sealed abstract class CustomerType(overTeletext ide val toString: String)
case object NewCustomer extends CustomerType("new")
case object ExistingCustomer extends CustomerType("existing")

def genCustomerType: Gen[CustomerType] = oneOf(NewCustomer, ExistingCustomer)
```


scala>

genCustomerType.sample

scala>

```scala
case class Order(
    id: String,
    orderTime: DateTime,
    orderState: String,
    customerType: Option[CustomerType],
    orderLines: List[OrderLine])

def genOrder: Gen[Order] = for {
  id ← Gen.listOfN(8, Gen.numChar).map(_.mkString)
  orderTime ← genOrderTime
  orderState ← genOrderState
  customerType ← Gen.option(genCustomerType)
  orderLines ← Gen.resize(5, listOf(genOrderLine))
} yield Order(id, orderTime, orderState, customerType, orderLines)
```


scala>

genOrder.sample

scala>

```
forAll(genOrder, genOrderLine) {
  (o: Order, ol: OrderLine) =>
    o.addOrderLine(ol).orderLines.length == o.orderLines.length + 1
}.check
```


scala>

```
forAll(genOrder, genOrderLine) {
  (o: Order, ol: OrderLine) =>
    o.total + ol.total == o.addOrderLine(ol).total
}.check
```


scala>

```scala
val propSerializationRoundtrip = forAll {
  order: Order =>
    order == order.toJson.convertTo[Order]
}
```


scala>

Let's do one more!


```
scala>
```

```scala
def mid(low: Int, high: Int) = (low + high) / 2
```


scala>

```scala
val genIntPairs = for {
    low <- Gen.choose(1, Int.MaxValue)
    high <- Gen.choose(1, Int.MaxValue) if high > low
} yield (low, high)

val propMid = forAllNoShrink(genIntPairs) {
  case (low, high) =>
    mid(low, high) > low
}
```


scala>

```
propMid.check
```



```
scala>
```

```
def midFixed(low: Int, high: Int) = low + ((high - low) / 2)
```


scala>

```scala
val propMidFixed = forAllNoShrink(genIntPairs) {
  case (low, high) =>
    midFixed(low, high) > low
}
```


scala>

```
propMidFixed.check
```



```
scala>
```

"Program testing can be used to show the presence of bugs,
but never to show their absence!" -- Edsger W. Dijkstra


```
scala>
```

```
*****************************************************************************************
*                                                                                       *
*                                                                                       *
*                                                                                       *
*                                                                                       *
*   ScalaCheck                                                                          *
*   https://scalacheck.org                                                              *
*                                                                                       *
*   Choosing properties for property-based testing                                      *
*   https://fsharpforfunandprofit.com/posts/property-based-testing-2                    *
*                                                                                       *
*   Quicksort                                                                           *
*   https://gist.github.com/ferhatelmas/4080888                                         *
*                                                                                       *
*   Bubble sort                                                                         *
*   https://sudheeraedama.blogspot.nl/2013/11/sorting-algorithms-in-scala.html          *
*                                                                                       *
*   Primes sieve                                                                        *
*   https://stackoverflow.com/questions/9711785/find-prime-numbers-using-scala-help-me-to-improve   *
*                                                                                       *
*   Binary search bug                                                                   *
*   https://research.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html   *
*                                                                                       *
*                                                                                       *
*                                                                                       *
*                                                                                 89/90 *
*****************************************************************************************
scala>
```

Philippus Baalman

philippus@gmail.com | https://github.com/philippus

Agile Meetup Twente

https://www.meetup.com/Agile-Meetup-Twente


```
scala>
```