knóldus

Unit testing with ScalaCheck

Jouko Virtanen

Outline

- Why unit testing?
- Unit testing with JUnit example
- Specification vs. tests
- Tests as specification
- Properties
- Generators
- ScalaCheck example
- More on generators
- Exercise 1
- More on generators
- Spark and ScalaCheck
- More exercises

Link to talk and code

https://github.com/JoukoPresentations/UnitTestingWithScalaCheck

Why unit testing?

- Make sure that program is working
- Make sure that changes don't break the program
- Reduce time finding and fixing bugs
- The development of software will not slow to a crawl as complexity increases

Unit testing with JUnit example

MaxTest.scala

MaxTest.scala | MyMax.scala

```scala
object MyMax {
  def myMax(x: Int, y: Int): Int = if (x > y) x else y
}
```

MyMax

MaxTest.scala

```scala
import junit.framework.TestCase
import org.junit.Test
import org.junit.Assert.assertEquals

class MaxTest extends TestCase {
  @Test
  def testMax(): Unit = {
    val z = MyMax.myMax(1, 0)
    assertEquals( expected = 1, z)
  }
}
```

MaxTest > testMax()

Unit testing with JUnit example

MaxTest.scala

```scala
class MaxTest extends TestCase {
  @Test
  def testMax1(): Unit = {
    val z = MyMax.myMax(1, 0)
    assertEquals( expected = 1, z)
  }

  @Test
  def testMax2(): Unit = {
    val z = MyMax.myMax(0, 1)
    assertEquals( expected = 1, z)
  }

  @Test
  def testMax3(): Unit = {
    val z = MyMax.myMax(-100, 0)
    assertEquals( expected = 0, z)
  }

  @Test
  def testMax4(): Unit = {
    val z = MyMax.myMax(0, -100)
    assertEquals( expected = 0, z)
  }

  @Test
  def testMax5(): Unit = {
    val z = MyMax.myMax(30, 30)
    assertEquals( expected = 30, z)
  }

  @Test
  def testMax6(): Unit = {
    val z = MyMax.myMax(1, 35)
    assertEquals( expected = 35, z)
  }

  @Test
  def testMax7(): Unit = {
    val z = MyMax.myMax(35, 1)
    assertEquals( expected = 35, z)
  }

  @Test
  def testMax8(): Unit = {
    val z = MyMax.myMax(-35, -1)
    assertEquals( expected = -1, z)
  }

  @Test
  def testMax9(): Unit = {
    val z = MyMax.myMax(-1, -35)
    assertEquals( expected = -1, z)
  }
}
```

MaxTest

Specifcation vs. tests

- Specification is the definition of the desired behavior of the code
- Traditional unit tests only look at some number of examples
- We would really like to test if the specifications are met
- Specifications might be proven with logic in some cases
- This can be difficult or impossible and mistakes can be made
- ScalaCheck gets us close to testing specifications

Properties

- ScalaCheck tests specifications by testing properties
- A property gives a more general description of the behavior of some code
- It is something that is always true of the code
- Eg if we have a function to square the number the output will always be positive

Generators

- In order to test properties ScalaCheck generates random input
- The unit tests check if the property holds for all random input
- Input is not completely random, but also includes common edge cases

ScalaCheck example

```
class MaxTest extends Properties( name = "Max") {
  property("max") = forAll { (x: Int, y: Int) =>
    val z = MyMax.myMax(x, y)
    (z == x || z == y) && (z >= x && z >= y)
  }
}
```

The maximum of two numbers has to be equal to one of the two numbers

If the maximum of x and y is x then the maximum is equal to x. If the maximum is y than the maximum is greater than x.

If the maximum of x and y is y then the maximum is equal to y. If the maximum is x than the maximum is greater than y.

ScalaCheck example

```
[info] Done compiling.
x= -1627517184 y= -2147483648
x= -1356396115 y= 1543367840
x= 1590997440 y= -2147483648
x= 1696906038 y= -301351631
x= 1707682334 y= -1
x= 0 y= 89959705
x= 1 y= 574291082
x= -1036460420 y= 0
x= 2108447452 y= -989926386
x= -1 y= -2147483648
x= 1 y= -1
x= 1815128107 y= 2147483647
x= 2147483647 y= 530114203
x= -1 y= 1302644545
x= -1421035958 y= -813824629
x= -2147483648 y= 1
x= 0 y= 1757498861
x= -1 y= -1
```

Bad ScalaCheck example

The property must be sufficiently different than the code being tested

```
class MaxTest extends Properties( name = "Max") {
  property("max") = forAll { (x: Int, y: Int) =>
    val z = MyMax.myMax(x, y)
    z == (if (x > y) x else y)
  }
}
```

More on generators (Chars)

```scala
import org.scalacheck.Properties
import org.scalacheck.Prop.forAll
import org.scalacheck.Gen.{numChar, alphaUpperChar, alphaLowerChar, alphaChar, alphaNumChar}

class GenExamples extends Properties( name = "StringUtils"){
  property("GenExample") = forAll(numChar, //A random digit as a char
    alphaUpperChar, //A random upper case letter as char
    alphaLowerChar, //A random lower case letter as char
    alphaChar,  //A random lower or upper case letter as char
    alphaNumChar) { //A random lower or upper case letter or digit as char
    (s1, s2, s3, s4, s5) =>
      println(s1 + "\t" + s2 + "\t" + s3 + "\t" + s4 + "\t" + s5)
      true
  }
}
```

More on generators (Chars)

```
[info] Done compiling.
7       G       u       a       d
4       L       s       m       w
3       K       m       k       E
2       J       b       p       9
0       V       z       u       r
1       V       h       y       f
8       P       l       m       w
4       O       v       n       F
1       G       b       d       z
0       N       x       b       y
4       F       k       e       a
4       D       i       b       i
7       E       q       a       m
3       S       w       o       i
6       S       s       m       p
5       A       y       j       x
4       K       t       h       3
0       U       j       f       i
9       P       o       d       w
3       O       j       x       f
```

More on generators (Strings)

Example.scala × PreconditionExample.scala × GenNumExample.scala × StringExample.scala × CharExample.scala × SparkExample.scala × build.sbt ×

```scala
import org.scalacheck.Properties
import org.scalacheck.Prop.forAll
import org.scalacheck.Gen.{alphaStr, numStr, identifier, alphaLowerStr, alphaUpperStr}

class StringExample extends Properties( name = "GenString") {

  property("GenStringExample") = forAll(numStr, //A random sequence of digits as a string
    alphaUpperStr, //A random sequence of upper case letters
    alphaLowerStr, //A random sequence of lower case letters
    alphaStr, //A random sequence of lower or upper case letters
    identifier) { //A random lower case letter followed by alphanumeric characters
    (s1, s2, s3, s4, s5) =>
      println(s1 + "\t" + s2 + "\t" + s3 + "\t" + s4 + "\t" + s5)
      true
  }
}
```

Exercise 1

Write two ScalaCheck unit tests for this function
If you think there is a bug in the code fix it

```scala
object StringUtils {
  def contains(str: String, subStr: String): Boolean = {
    str.indexOf(subStr) != -1
  }
}
```

More on generators (Integers)

Example.scala | PreconditionExample.scala | GenNumExample.scala | StringExample.scala | CharExample.scala | SparkExample.scala

```scala
import org.scalacheck.Properties
import org.scalacheck.Prop.forAll
import org.scalacheck.Gen.{posNum, negNum}

class GenNumExample extends Properties( name = "GenNum") {

  property("GenNumExample") = forAll(negNum[Int], posNum[Int]) {
    (n, p) =>
      println(n + "\t" + p)
      n < p
  }
}
```

More on generators (Preconditions)

```
nExample.scala  PreconditionExample.scala  GenNumExample.scala  StringExample.scala  CharExample.scala  SparkExample.scala

import org.scalacheck.Properties
import org.scalacheck.Prop.{forAll, BooleanOperators}

class PreconditionExample extends Properties( name = "Precondition") {

  property("GenPreconditionExample") = forAll { n: Int =>
    (n % 3 == 0) ==> {
      (n + 3) % 3 == 0
    }
  }
}
```

More on generators (Lists)

```scala
import org.scalacheck.Properties
import org.scalacheck.Prop.forAll
import org.scalacheck.Gen.listOf

class ListExample extends Properties( name = "GenList") {

  property("GenListExample") = forAll(listOf(Int)) {
    xs => xs.length >= 0
  }
}
```

More on generators (Custom generators)

```scala
import org.scalacheck.Properties
import org.scalacheck.Prop.forAll
import org.scalacheck.Gen.choose

class MyGenExample extends Properties( name = "MyGen") {

  val myGen = for {
    n <- choose( min = 1,  max = 50)
    m <- choose(n,  max = 2 * n)
  } yield (n, m)

  property("MyGenExample") = forAll(myGen) {
    pair => pair._2 >= pair._1
  }
}
```

Spark and ScalaCheck

```scala
class SparkExample extends FunSuite with SharedSparkContext with Checkers {
  override implicit def reuseContextIfPossible: Boolean = true

  test( testName = "schemas should be the same") {
    val sqlContext = new SQLContext(sc)

    val schema = StructType(List(
      StructField("id", IntegerType, nullable = true),
      StructField("name", StringType, nullable = true)
    ))

    val newSchema = StructType(List(StructField("name", StringType, nullable = true)))

    val dataframeGen = DataframeGenerator.arbitraryDataFrame(sqlContext, schema)

    val property = forAll(dataframeGen.arbitrary) {
      df => {
        val newDf = df.select(col( colName = "name"))
        newDf.schema == newSchema
      }
    }
    check(property)
  }
}
```

Advice on writing unit tests

- Properties don't have to cover all behavior
- If a function has an inverse, apply the inverse to the output and compare to input
- Compare brute force methods with efficient methods
- Relation properties

Exercise 2

- Write a ScalaCheck unit test for this function
- If you think there is a bug in the code fix it

```
class StringUtils {
  def truncate(str: String, n: Int): String = {
    if (str.length <= n) str
    else str.substring(0, n) + "..."
  }
}
```

Exercise 3

```scala
import java.util.StringTokenizer
import scala.annotation.tailrec

object StringUtils {
  @tailrec
  private def tokenizeRecursive(st: StringTokenizer, delimStr: String, result: List[String]): List[String] = {
    if (st.hasMoreTokens) tokenizeRecursive(st, delimStr, result :+ st.nextToken)
    else result
  }

  def tokenize(str: String, delim: Char): List[String] = {
    val delimStr = delim.toString
    val st = new StringTokenizer(str, delimStr)
    val result = List()
    tokenizeRecursive(st, delimStr, result)
  }
```

Thank You

Appendix

Controlling how many times a test is run

```scala
import org.scalacheck.Properties
import org.scalacheck.Prop.forAll
import org.scalacheck.Test.Parameters

class Max1000Test extends Properties( name = "Max") {
  val p = forAll { (x: Int, y: Int) =>
    val z = MyMax.myMax(x, y)
    (z == x || z == y) && (z >= x && z >= y)
  }

  val myParams = Parameters.default.withMinSuccessfulTests( minSuccessfulTests = 1000)

  p.check(myParams)

}
```

Shrinking

```scala
import org.scalacheck.Properties
import org.scalacheck.Prop.forAll
import org.scalacheck.Gen.{listOf, choose}

class ShrinkingExample extends Properties( name = "Shrinking") {
  property("IncorrectProperty") = forAll(listOf(choose( min = 0, max = 100)))
  {
    ls =>
    println(ls)
    !ls.exists{x => x%2 == 0}
  }
}
```

Shrinking

```
\ShrinkingExample\target\scala-2.13\test-classes ...
[info] Done compiling.
List()
List(39)
List(69, 34)
List(69)
List(34)
List()
List(17)
List(-17)
List(8)
List()
List(4)
List()
List(2)
List()
List(1)
List(-1)
List(0)
List()
[info] ! Shrinking.IncorrectProperty: Falsified after 2 passed tests.
[info] > ARG_0: List("0")
[info] > ARG_0_ORIGINAL: List("69", "34")
[info] Failed: Total 1, Failed 1, Errors 0, Passed 0
[error] Failed tests:
[error]         ShrinkingExample
[error] (Test / test) sbt.TestsFailedException: Tests unsuccessful
[error] Total time: 7 s, completed Aug 13, 2019 5:18:04 PM
```

Once ScalaCheck finds an example were the unit test fails it tries to simplify it repeatedly until it cannot find a simpler test case that fails.