Software Engineering II (IB)
Testen in Scala und Play

Prof. Dr. Oliver Braun

Letzte Änderung: 15.05.2017 07:39

Moderne Unittest-Frameworks: Spezifikationen

- mit dem Ziel Tests besser lesbar zu machen, geht der Trend immer mehr zu Frameworks, die es ermöglichen **ausführbare Spezifikationen** zu schreiben
- in Scala populär: specs2
- die Idee kommt von **Behavior Driven Development (BDD)** (verhaltensgetriebene Softwareentwicklung)

Specs2

Disclaimer: Die Specs2-Libs sind in Play schon integriert!
(Zunächst) bitte nichts mehr konfigurieren.

Specs2 bietet 2 Arten von Spezifikationen:

1. Acceptance Specifications
   - gesamter Text an einer Stelle, Code an anderer
   - damit besser für Nicht-Entwickler lesbar
2. Unit Specifications
   - Text und Code unmittelbar zusammen
   - Struktur nahe an klassischen Unit-Tests

Beispiel: Acceptance Specifications

```
class MySpecification
    extends org.specs2.Specification
        { def is = s2"""

 this is my specification
   where example 1 must be true          $e1
   where example 2 must be true          $e2
                                         """


  def e1 = 1 must_== 1
  def e2 = 2 must_== 2
}
```

Beispiel: Unit Specifications

```
class MySpecification
    extends org.specs2.mutable.Specification {
  "this is my specification" >> {
    "where example 1 must be true" >> {
      1 must_== 1
    }
    "where example 2 must be true" >> {
      2 must_== 2
    }
  }
}
```

Expectations

- in Acceptance Specifications üblicherweise eine Expectation pro Beispiel
- in Unit Specifications auch mehrere möglich

Functional Expectations

Achtung: in Acceptance Specifications wird nur die letzte Expectation zurück gegeben / ausgeführt.

```
// this will never fail!
s2"""
  my example on strings $e1
"""
  def e1 = {
    // because this expectation will
    // not be returned,...
    "hello" must have size (10000)
    "hello" must startWith("hell")
  }
```

Lösung: mit and verbinden:

```
s2"""
  my example on strings $e1
"""
  def e1 = ("hello" must have size (10000)) and
           ("hello" must startWith("hell"))
```

Thrown expectations

in Unit Specifications wird bei einer Expectation die nicht gilt eine Exception geworfen:

```
class MySpecification
    extends org.specs2.mutable.Specification {
  "This is my example" >> {
    1 must_== 2 // this fails
    1 must_== 1 // this is not executed
  }
}
```

NoThrownExpectations

Dieses Verhalten lässt sich ändern, wenn Sie den Trait `org.specs2.execute.NoThrownExpectations` hineinmixen:

```
class MySpecification
    extends org.specs2.mutable.Specification
    with org.specs2.execute.NoThrownExpectations {
  "This is my example" >> {
    1 must_== 2
    1 must_== 1
  }
}
```

Matchers

- üblicherweise beschreiben wir das Verhalten in specs2 mit *Matchers*
- eine Anweisung wird ausgeführt und überprüft ob das Ergebnis dem erwarteten entspricht
- Beispiel:

```
// describe the functionality
s2"""the directoryPath method should
     return well-formed paths          $e1"""

// give an example with some code
def e1 = Paths.directoryPath("/tmp/path/to/dir"
    ) must beEqualTo("/tmp/path/to/dir/")
```

- der `must`-Operator
  - hat als ersten Operand den berechneten Wert
  - und als zweiten einen Matcher

Gleichheit

- es gibt verschiedene Arten (Geschmackssache) um auf Gleichheit zu testen

```
1 must beEqualTo(1)
1 must be_==(1)
1 must_== 1
1 mustEqual 1
1 should_== 1
1 === 1
1 must be equalTo(1)
```

und noch viele mehr:

- für Strings

```
startWith(b)
beEqualTo(b).ignoreCase
contain(b)
// ...
```

- für Zahlen

```
1 must be_<=(2)
1.0 must beCloseTo(1, 0.5)
5 must beBetween(4, 6).excludingBounds
// ...
```

- für Exceptions:

```
throwA[ExceptionType]
```

- und noch viel mehr unter org.specs2.guide.Matchers
- sie können sich auch eigene definieren!

Runners

- Specifications lassen sich mit `sbt`/`activator`, ScalaIDE, IntelliJ ausführen
- Ausgabe ist möglich z.B. auf der Console, als JUnit XML, HTML, Markdown
- siehe org.specs2.guide.Runners

Context

- die Beispiele in einer Specification sind simple, aber benötigen manchmal einen Kontext
- Beispiele:
  - ein Zustand der vor dem Ausführen eines Beispiels oder aller vorliegen muss
  - ein Zustand der nach dem Ausführen eines Beispiels oder aller aufgeräumt werden muss
  - ein Datenbank-Kontext
- dafür gibt es wieder Traits

BeforeEach

```
import org.specs2.specification.BeforeEach
class BeforeSpecification
    extends mutable.Specification
        with BeforeEach {
  // you need to define the "before" action
  def before = println("before")
  "example 1" >> {
    println("example1"); ok
  }
  "example 2" >> {
    println("example2"); ok
  }
}
```

AfterEach

```
import org.specs2.specification.AfterEach
class BeforeSpecification
    extends mutable.Specification
        with AfterEach {
  // you need to define the "after" action
  def after = println("after")
  "example 1" >> {
    println("example1"); ok
  }
  "example 2" >> {
    println("example2"); ok
  }
}
```

außerdem

- `ArroundEach`
- `ForEach`
- `BeforeAll`
- `AfterAll`
- `BeforeAfterAll`
- und spezielle in Play (siehe später)

Spezielle Kontexte in Play

- `WithApplication`
- `FakeApplication` um globale Konfigurationsparameter auf die Testausführung anzupassen, z.B. Memory Database statt der sonst genutzten
- `WithBrowser`
- `WithServer`

siehe auch ScalaFunctionalTestingWithSpecs2

ScalaCheck (in Specs2)

- ScalaCheck ist ein von QuickCheck inspiriertes Testframework das *property-based testing* für Scala und Java bietet
- Beispiele

```
prop { (a: Int) => a + a == 2 * a }
prop { (a: Int) => a + a must_== 2 * a }
prop { (a: Int) => (a > 0) ==>
            (a + a must_== 2 * a) }
```

- Erzeugt Zufallswerte mit denen getestet wird.
- Es ist möglich eigene Zufallsgeneratoren z.B. für eigene Klassen zu entwickeln.

Mocking mit Mockito

- Mockito: “Tasty mocking framework for unit tests in Java”
- Objekte die noch nicht da sind, oder die nicht mit getestet werden sollen, können gemockt werden, z.B.

```
val m = mock[java.util.List[String]]
```

Damit haben wir ein Objekt `m`, dass den Typ einer Liste hat.

- durch *Stubbing* sagen wir dem Objekt, was es machen soll, wenn eine Methode aufgerufen wird, z.B.

```
m.get(0) returns "one"
m.get(2) throws new
    RuntimeException("forbidden")
```

es können sogar bei den nächsten Aufruf andere Werte zurück gegeben werden

```
m.get(1) returns "one" thenReturns "two"
```

Achtung: Scala `objects` können nicht direkt gemockt werden.
Trick: aus dem `object` einen `trait` machen und ein `object` vom