Testen

Unit testing is als je huis opruimen. Eigenlijk is het belang ervan wel duidelijk, maar in de praktijk schiet het er toch te vaak bij in. Het helpt wanneer je met weinig inspanning je tests kunt schrijven en onderhouden. Spock biedt daarvoor de uitkomst.

Expressief testen met Spock framework

Test- en mockingframework voor Java

Michel Vollebregt
Senior softwareontwikkelaar bij VX Company

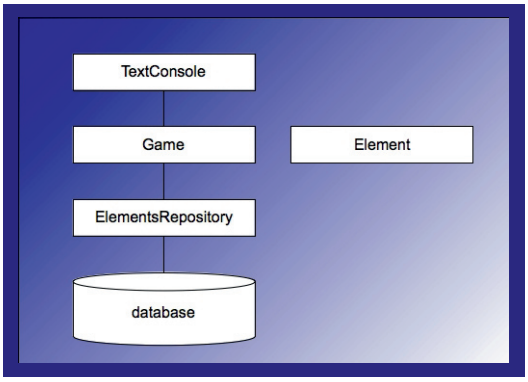
Spock is een test- en mockingframework voor Java. Het heeft een eenvoudige en expressieve syntax waarin je snel en leesbaar tests schrijft. Dit artikel laat het gebruik van Spock zien aan de hand van een voorbeeld. Daarbij wordt Spock uitgebreid vergeleken met JUnit.

Het artikel laat zien hoe je tests schrijft voor het spel Alchemy. In dit spel combineer je als speler de elementen vuur, water, lucht en aarde. Hierbij creëer je steeds nieuwe elementen, die je vervolgens weer verder kunt combineren. Het spel is een eenvoudige variant van het bestaande spel op iPhone en Android. De code van het spel bevat constructies die je ook in enterpriseapplicaties tegenkomt. De database wordt benaderd door een Repositoryklasse en ook spellogica en presentatie zijn van elkaar gescheiden. Een structuurdiagram van de code vind je in afbeelding 1.

Om de voorbeelden in het artikel te compileren heb je een aantal libraries nodig.

- spockframework.org
- junit.org
- mockito.org
- cglib.sf.net
- hsqldb.org

Je kunt gebruik maken van een IDE naar keuze. Dankzij de ingebouwde JUnit Runner integreert Spock goed met alle bekende IDE's.
Het volledige uitgewerkte Javaproject is beschikbaar op github (github.com/VXCompany/SpockSnippets). Het bevat een aantal aanvullende voorbeelden, waaronder een databasetest en een integratietest. Ook bevat het een Maven-objectmodel, dat je kunt gebruiken als uitgangspunt voor je eigen Mavenprojecten met Spock.



Een structuurdiagram van de code van Alchemy.

Eerste test: een getter

Laten we beginnen met een eenvoudige test. Hieruit zullen de meest in het oog springende verschillen van Spock met JUnit meteen duidelijk worden. In het spel draait alles om elementen. De speler vraagt het spel welke elementen beschikbaar zijn, combineert deze en het spel vraagt nieuwe elementen op aan de database. Element is de belangrijkste klasse van het spel. Elementen zijn herkenbaar aan hun naam, bijvoorbeeld lucht, aarde vuur of water. Dat leidt tot de eerste test:
Van ieder element kunnen we de naam opvragen.

De meeste Java-ontwikkelaars zullen deze test in JUnit zonder al te veel moeite schrijven.

```
@Test
public void newElement_getName_nameIsCorrect()
{ Element element = new Element("name of element");
assertEquals("Nameshould equal name in constructor",
"name of element", element.getName());
}
```

De naam van de methode geeft een korte systematische beschrijving van de test. De assert geeft

een leesbaardere beschrijving in een volzin: 'name should equal name in constructor'. Het equivalent in Spock is als volgt:

```
def "on new element, calling getName should return
correct name"() {
  when:
    def element = new Element("name of element")
  then:
    element.getName() == "name of element"
```

Wat valt op? Allereerst is er een duidelijk verschil in syntax. Spock is gebaseerd op Groovy, een taal die alle constructies van Java ondersteunt maar ook een aantal versimpelingen toestaat. Zo mag je expliciete typedeclaraties vervangen door *def* en hoef je niet iedere regel af te sluiten met een puntkomma. Als je dat niet prettig vindt, is standaard Javasyntax ook toegestaan. De te testen code zelf is gewoon in Java geschreven.

Ten tweede bestaat de naam van de test uit een volledige volzin. Mede hierdoor is Spock heel geschikt om ook specificaties te schrijven. Java ondersteunt deze mogelijkheid niet. Ook dit is echter optioneel. Als je de voorkeur geeft aan compactheid, staat Spock ook de standaard Javanaamgeving van testmethoden toe, net als JUnit.

Ten slotte is ook de structuur van de test anders. De Spocktest bestaat uit duidelijk afgebakende blokken met een eigen functie. Het *when*-blok beschrijft het uitgangspunt van de test. Het *then*-block beschrijft de voorwaarde waaraan de test moet voldoen. Verderop in het artikel laat ik je nog enkele andere blokken zien.

Omdat het *then*-blok zo'n uitgesproken functie heeft, kun je het *assert*-keyword weglaten. Het is voldoende om simpelweg de boolean expressie op te schrijven die geverifieerd moet worden. De uiteindelijke test leest bijna als een volzin: 'wanneer element een instantie is van Element, dan is element.getName() gelijk aan de naam van het element'.

Een laatste verschil wordt zichtbaar bij het draaien van de tests. JUnit geeft de bekende melding:

```
java.lang.AssertionError: Name should equal name in
constructor expected:<name of element> but was:<null>
```

Spock drukt alle waarden af die het bij de evaluatie van de conditie tegenkomt:

```
Condition not satisfied:

element.getName() == "name of element"
|         |         |
|         null      false
com.vxcompany.spocksnippets.Element@5815338

at com.vxcompany.spocksnippets.ElementSpec.on new
element, calling getName should return correct
name(ElementSpec.groovy:31)
```

Spock geeft dus wat meer informatie dan JUnit en presenteert de informatie bovendien overzichtelijker.

Een test met een stub

Je hebt nu gezien hoe je tests kunt schrijven voor één enkele klasse. De meeste programmacode bestaat echter uit meerdere klassen die met elkaar interacteren. Om zulke code te testen kun je gebruik maken van mocks en stubs. Ik zal je dit laten zien aan de hand van de volgende test:

Wanneer we aarde en vuur combineren, wordt lava toegevoegd aan de beschikbare elementen.

Informatie over de elementen is beschikbaar via de ElementsRepository. Wanneer je de Gameklasse vraagt om aarde en vuur te combineren, gebruikt hij de ElementsRepository om informatie over die elementen op te vragen. Om de Gameklasse geïsoleerd te testen, kun je de ElementsRepository vervangen door een stub. Een stub heeft voorgeprogrammeerde functionaliteit die specifiek bedoeld is voor de test. Hiermee maak je het slagen van de test onafhankelijk van de toevallige vulling of beschikbaarheid van de database.

Je kunt stubs zelf uitprogrammeren, maar het is makkelijker om gebruik te maken van een framework. Voor JUnit zijn er diverse mockingframeworks. Mockito is één van de bekendste en is bovendien makkelijk in gebruik. Met Mockito declareer je een stub met een annotatie. Vervolgens specificeer je per methode-aanroep de gewenste uitkomst:

```
@RunWith(MockitoJUnitRunner.class)
public class GameTest {

@Mock ElementsRepository repo;
Element earth = new Element("earth");
Element fire = new Element("fire");
Element lava = new Element("lava");

 @Test
 public void gameWithElements_combineTwo_newAdded() {
   // setup
   Game game = new Game(repo);
   Set<Element> elements = new HashSet<Element>();
   Collections.addAll(elements, earth, fire);
   game.setAvailableElements(elements);
   when(repo.getCombinedElement(earth, fire)).
         thenReturn(lava);

   // call function
   game.combine(earth, fire);

   // assert
   Set<Element> resultSet = new HashSet<Element>();
   Collections.addAll(resultSet, earth, fire, lava);
   assertEquals("combine should give new element",
            resultSet, game.availableElements());
}
```

Spock maakt gebruik van hetzelfde principe, maar de syntax is anders:

```
class GameSpec extends Specification {

def repo = Mock(ElementsRepository)
static earth = new Element("earth")
```

Een stub heeft functionaliteit die specifiek bedoeld is voor de test.

Wat vooral opvalt is dat de Spocktest veel compacter is.

```
 static fire = new Element("fire")
 static lava = new Element("lava")

 def "on game with elements, combining two elements
makes another element available"() {
   given:
     def game = new Game(repo)
     game.availableElements = [earth, fire]
     repo.getCombinedElement(earth, fire) >> lava
   when:
     game.combine(earth, fire)
   then:
     game.availableElements() == [earth, fire, lava]
                                    as Set
}
```

Spock maakt gebruik van een elegante en compacte syntax om het gewenste gedrag van de stub vast te leggen. De syntax van Mockito is langer en minder overzichtelijk. Het declareren van de stub is bij Mockito erg elegant. Je hoeft slechts een annotatie toe te voegen.

Wat vooral opvalt is dat de Spocktest veel compacter is. De feitelijke testmethode bestaat uit 10 regels. Dezelfde test in JUnit beslaat 17 regels, ruim anderhalf keer zoveel. De Spocktest maakt gebruik van Groovy's compacte syntax voor Lists en Sets. Het statement [earth, fire] definieert een ArrayList met de elementen earth en fire. Door daar as Set achter te plakken, transformeer je de ArrayList in één keer naar een Set. In Java heb je voor het creëren en toekennen van dezelfde Set drie regels nodig.

Het vergelijken van Lists en Sets kan met Spock eenvoudig met de ==-operator. In Groovy is deze operator een korte schrijfwijze voor equals(). De laatste regel van de Spocktest controleert dus niet of de twee sets exact dezelfde instantie in het geheugen zijn, maar of de sets dezelfde inhoud bevatten.

Een test met een mock

Een stub imiteert gedrag dat is gewenst voor de test. Soms is dat niet voldoende en vereist de test een controle op het daadwerkelijk aanroepen van een methode op de stub. We spreken in zo'n geval niet van een stub, maar van een mock. In de praktijk worden de termen stub en mock vaak door elkaar gebruikt. Ik laat je nu een test zien waarin een mock noodzakelijk is.
De speler geeft commando's aan het spel via een tekstconsole. Hij krijgt daarop feedback in de vorm van tekst in de output. In de volgende test controleren we of de juiste tekst verschijnt:
Wanneer de speler de beschikbare elementen opvraagt, worden deze afgedrukt in de output.

De test lijkt in JUnit erg op de vorige:

```
@Test
public void textConsole_listCommand_printElements() {

  Game game = mock(Game.class);
  PrintStream output = mock(PrintStream.class);
  TextConsole console = new TextConsole(game, output);
  Set elements = createSet(earth, fire, lava); when(game.
  availableElements()).thenReturn(elements); console.
  eval("list");

  verify(output).println("[earth, fire, lava]");
}
```

Het controleren van de aanroep gebeurt met het verifycommando. De test slaagt wanneer output.println wordt aangeroepen met de exacte parameter "[lava, earth, fire]".
In de bovenstaande test worden de game-stub en de output-mock geïnstantieerd met de mock()-methode in plaats van de @Mock-annotatie. Het resultaat is identiek. Merk op dat we voor het creëren van de Set een hulpmethode createSet hebben geschreven. Dit verhoogt de leesbaarheid van de test aanzienlijk. Ook de Spocktest verschilt niet wezenlijk van de vorige:

```
def "on text console, list command prints available
elements"() {
  given:
   def game = Mock(Game)
   def output = Mock(PrintStream)
   def console = new TextConsole(game, output)
  and:
   game.availableElements() >> [earth, fire, lava]
  when:
   console.eval("list")
  then:
    1 * output.println("[earth, fire, lava]")
}
```

De verwachte aanroep op het output-object is kort en krachtig geformuleerd. In Spock formuleer je het aantal keer dat je de aanroep verwacht. In plaats van een exact aantal, kun je ook een minimum of een maximum aantal aanroepen opgeven.
Bij het draaien van de test geeft Mockito nauwkeurige informatie over de feitelijke output:

```
Expected :printStream.println("[earth, fire, lava]");
Actual   :printStream.println("[lava, earth, fire]");
```

Spock is in dit geval minder uitvoerig in zijn output en geeft slechts de melding dat de methode niet het juiste aantal keer is aangeroepen:

```
Too few invocations for:

1 * output.println("[earth, fire, lava]")
(0 invocations)
```

Fragile tests

De bovenstaande test faalt, omdat de volgorde van de afgedrukte elementen niet overeenkomt met de volgorde in de specificatie. We hebben hier te maken met een zogeheten fragile test. De test is overgespecificeerd. De test legt letterlijk vast welke tekst de speler op het scherm te zien krijgt wanneer hij de beschikbare elementen opvraagt. Als je ook maar een komma aan deze tekst toevoegt, faalt de test en moet die worden herschreven. Dat maakt het onderhouden van de test nodeloos tijdrovend

en vervelend. Beter is het om niet de letterlijke tekst te specificeren, maar de voorwaarden waaraan de tekst moet voldoen. Je kunt daarvoor in Mockito gebruik maken van argument matchers. Een argument matcher is een klasse die verifieert of een argument voldoet aan een voorwaarde. Mockito stelt een aantal standaard matchers beschikbaar voor eenvoudige gevallen. In ingewikkeldere gevallen kun je zelf een argument matcher implementeren of gebruik maken van een argument captor. Het voordeel van de argument captor is dat je de voorwaarde als doodgewone boolean expressie kan opschrijven:

```
ArgumentCaptor<String> captor =
   ArgumentCaptor.forClass(String.class);
verify(output).println(captor.capture());
String text = captor.getValue();
assertTrue(
   "list command should print available elements", text.
   contains("earth")
   && text.contains("fire")
   && text.contains("lava"));
```

De eerste regel creëert de argument captor. Deze houdt het argument van de functieaanroep vast, zodat we het later kunnen inspecteren. assertTrue in de laatste regel bepaalt de precieze verwachting op het argument.
Spock maakt het leven een stuk eenvoudiger:

```
1 * output.println( { text ->
      text.contains("earth")
      && text.contains("fire")
      && text.contains("lava")})
```

In Spock voeg je simpel de boolean expressie toe als argument van output.println(). Alle boilerplatecode die in Mockito nodig is om het argument vast te houden in een argument captor is in Spock overbodig.

Geparametriseerde tests

Laten we nog even teruggaan naar de test voor de Gameklasse. De test verifieert de situatie dat bij het combineren van twee beschikbare elementen een derde element wordt gevormd. De test dekt nog niet alle gevallen af. Wat gebeurt er als twee elementen niet met elkaar reageren? Of als de speler een niet-bestaand element probeert te combineren?
Je kunt voor deze tests aparte testmethoden schrijven. De tests lijken echter nogal op elkaar. Het ligt daarom meer voor de hand om de test te parametriseren. Dezelfde testmethode wordt meerdere keren uitgevoerd, maar met verschillende parameters. Iedere set van parameters representeert een afzonderlijk testgeval.
In Spock kun je een test eenvoudig parametriseren met een *where*-blok:

```
def "game must combine elements correctly"() {
  given:
    def game = new Game(repo)
    game.availableElements = initial
    repo.getCombinedElement(first, secnd) >> combn
  when:
    game.combine(first, secnd)
  then:
    game.availableElements() == result as Set
  where:
initial           | first|secnd|combn| result
[earth,fire]      | earth| fire| lava| [earth,fire,lava]
[earth,fire,lava] | lava |fire | null| [earth,fire,lava]
[earth]           |earth| fire| lava| [earth]
[fire]            | earth| fire| lava| [fire]
}
```

Iedere regel in het *where*-blok legt een afzonderlijk testgeval vast. Je specificeert extra testgevallen door eenvoudig regels toe te voegen.
Ook JUnit kent geparametriseerde tests. In JUnit leg je parameters echter niet vast per testmethode, maar per testklasse. Concreet betekent dit dat je de bovenstaande test alleen kunnen parametriseren wanneer je hem isoleert in een eigen testklasse. Wanneer je veel tests parametriseert, explodeert je aantal testklassen. Geparametriseerde tests zijn daardoor in JUnit veel minder breed toepasbaar.
Een ander framework dat je om deze reden nog zou kunnen overwegen is TestNG. Dit framework kent dezelfde constructies als JUnit, maar biedt een paar extra mogelijkheden. Zo kun je met TestNG tests eenvoudig in groepen indelen, of tests schrijven die van elkaar afhankelijk zijn. En je kunt met TestNG parameters toekennen aan testmethoden:

```
@Test(dataProvider = "combining elements")
public void game_combineElements_correctResult( Set
initial, Element first, Element second, Element
combined, Set result) {
  ...
}

@DataProvider(name = "combining elements")
public Object[][] combineElementsParameters() {
  return new Object[][] {
    {createSet(earth, fire)    , earth, fire, lava,
createSet(earth, fire, lava)},
    {createSet(earth, fire, lava) , lava , fire, null,
createSet(earth, fire, lava)},
   ...
};
}
```

Deze test heeft hetzelfde resultaat als de Spocktest, maar de syntax is erg verschillend. Bij Spock staan de sets van parameters in de test zelf. Bij TestNG staan ze in een afzonderlijke methode. Spock maakt gebruik van een tabelformaat om de parameters vast te leggen. Dit is overzichtelijk en maakt Spock bovendien heel geschikt als specificatieframework. «

Conclusie

Spock heeft een expressieve syntax waarmee je snel tests schrijft. JUnit en Spock hebben ongeveer dezelfde mogelijkheden, maar de syntax van Spock is korter en doelmatiger. Je hebt in Spock minder regels code nodig voor dezelfde test. Spocktests zijn daardoor goed leesbaar en goed onderhoudbaar.
Zelf gebruik ik Spock in de praktijk met veel plezier. Ik ben gemotiveerder om tests te schrijven en verlies minder snel het overzicht. Het uitgewerkte voorbeeld op github.com/VXCompany/SpockSnippets is daarvoor een mooi startpunt.