TOOLS

JUnit is een van de meest bekende en waarschijnlijk ook een van de meest gebruikte open source frameworks. Na de verschijning van versie 3.8 is er lang niets meer gebeurd met dit framework. Het is zelfs zo dat de afgelopen jaren Test-NG bij sommige software-ontwikkelaars de plaats van JUnit heeft ingenomen.

De flexibiliteit van JUnit 4

Unittesten vereenvoudigd

Levente Bokor
is Java Sofware Architekt bij Capgemini.

Met de komst van JUnit versie 4 is hier verandering in gekomen. Deze nieuwe versie van JUnit is gebaseerd op Java 5. Zo maakt deze nieuwe versie 4 gebruik van de verschillende features van Java 5 (zoals annotaties, autoboxing, static imports …). Net zoals andere tools en frameworks die gebaseerd zijn op Java 5 maakt ook JUnit versie 4 extensief gebruik van annotaties. De meeste nieuwe features van deze versie zijn op annotaties gebaseerd. De functionaliteit van de annotaties was in de vorige versies van JUnit gebaseerd op Java reflection en naming conventions. In het vervolg zal blijken dat JUnit 4 een aantal eigenschappen introduceert zoals eenvoud in het gebruik, flexibiliteit en veel nieuwe features.

Voorbeeld

Met het onderstaande voorbeeld zal ik de toegevoegde waarde van de versie 4 proberen te verduidelijken:

```
package org.bool;

public class BooleanExpression {

    public BooleanExpression(boolean val){
      this.val = val;
    }

    private boolean val;           // Static variabele waar de boolean waarde wordt bewaard

    public BooleanExpression and(boolean a){
        val = val && a;
        return this;
    }

    public BooleanExpression or(boolean a){
        val = val && a;                  // Fout in de code: het moet zijn val= val || a
        return this;
    }

    public BooleanExpression xor(boolean a){
        val = val ^ a;
        return this;
    }

  public BooleanExpression neg(){          // Nog niet geimplementeerd
      return null;
    }

    public BooleanExpression and(boolean[] a) throws IllegalArgumentException {
        for (;;);                              // Bug: eindloos loop
    }

    public BooleanExpression or(boolean[] a) throws IllegalArgumentException {
        if ( a.length==0 )
            throw new IllegalArgumentException();
        else {
            for (int i=0; i<a.length; i++){
                val = val || a[i];
            }
        }
        return this;
    }

  public boolean getValue() {
      return val;
  }
}
```

In dit artikel zal steeds een vergelijking gemaakt worden tussen de testcase gebaseerd op JUnit versie 3.8 en JUnit versie 4. De testcase codevoorbeelden hebben echter alleen betrekking op de versie 4, waarbij ik ervan uit ga dat de meeste lezers voldoende bekend zijn met versie 3.8. Het

Op het Run-configuratiescherm zal de Test runner op JUnit 4 ingesteld moeten worden

voorbeeld is vrij eenvoudig en bevat ook een aantal fouten om bepaalde eigenschappen van JUnit versie 4 te verduidelijken.
De or(boolean, boolean) methode geeft het resultaat van een disjunctie gebaseerd op twee beweringen terug. De and(boolean, boolean) methode geeft het resultaat van een conjunctie terug, die ook gebaseerd is op twee beweringen. De neg(boolean a) methode geeft het resultaat terug van een negatie van een bewering. De and(boolean[]a) methode bevat een eindeloze loop en zal gebruikt worden om één van de nieuwe features van de 4.x versie te verduidelijken. De or(boolean[] a) methode zal bij de verkeerde input parameters een exception werpen en heeft tot doel de toegevoegde waarde aan te tonen van de JUnit versie 4 betreffende het testen van de exception handling.
De testcase implementatie gebaseerd op de 3.8 versie van het JUnit framework bevat weinig verassingen voor de meesten van ons. In de testcase implementatie die gebaseerd is op de versie 4 van JUnit zijn een aantal zaken die duidelijk opvallen, waaronder een andere naamgeving en gebruik van annotaties.

Definiëren van een test

Als we bovenaan in de JUnit versie 4 implementatie van de testcase kijken, valt het direct op dat de package waaronder deze JUnit classes te vinden zijn is veranderd van `junit.framework.*` naar `org.junit.*`, maar wegens backward compatibility zijn de oude classes en packages ook in de nieuwe versie aanwezig.

```
import org.junit.*;
import static org.junit.Assert.*;
```

Naast de nieuwe package naam is nog een ander import statement dat opvalt:

```
import static org.junit.Assert.*;.
```

Zodoende kunnen dezelfde assert methodes gebruikt worden als in de versie 3.8, ondanks dat een testcase niet meer afgeleid is van de *TestCase*. Door het gebruik van static imports met Java 5 is het niet meer nodig de klasse mee te geven bij het gebruik van static variabelen en methoden. In versie 4 is het dus mogelijk om iedere willekeurige class als testcase te gebruiken.
Ook is het mogelijk om in tegenstelling tot JUnit versie 3.8 een protected methode te unittesten, door de testcase af te leiden van de class met de betreffende protected methode. De enige restrictie is dat een testcase minimaal een testmethode moet hebben. Het uitvoeren van een testcase met methodes die voorzien zijn van *@Before* en *@After* annotaties, zonder een enkele methode die voorzien is van de *@Test* annotatie, zal resulteren in een *java.lang.Exception: No runnable methods*. Bij het verder vergelijken van de testcases valt het op dat er een verschil is tussen signature van de testmethodes.

```
public class TestBooleanExpression
{
  @Test
  public void and(){
    ...
```

Het verschil heeft niet alleen betrekking op de naamgeving van de verschillende testmethodes, maar de nieuwe versie van de methode is ook voorzien van de al eerder genoemde *@Test* annotatie. De namen van de testmethodes zijn verschillend; vanaf versie 4 is het niet meer verplicht om de testmethode namen met het woord 'test' te laten beginnen. Voor JUnit versie 3.8 en 4.x geldt dat de testmethode een *void* als resultaat moet opleveren en geen parameters mag hebben. Een aantal attributen kan in combinatie met de *@Test* annotatie gebruikt worden (hierover later meer).

setUp en tearDown

Andere belangrijke onderdelen van een testcase zijn de *setUp* en *tearDown* methodes. De *setUp* methode wordt altijd uitgevoerd voor een testmethode en de *tearDown* methode wordt altijd uitgevoerd na een testmethode.
In een testcase die gebaseerd is op een van de voorgaande versies van JUnit zullen de setUp en

de tearDown methodes altijd hetzelfde signature moeten hebben: respectievelijk public void setUp() en public void tearDown().

```
@Before
public void opbouw(){
  truev = new BooleanExpression(true);
  falsev = new BooleanExpression(false);
}

@After
public void afbouwen(){
  truev = null;
  falsev = null;
}
```

Door gebruik te gaan maken van de JUnit versie 4 kan iedere willekeurige tekst als methodenaam gebruikt worden voor de setUp-methode, zolang deze methode is voorzien van de *@Before* annotatie. Dit geldt uiteraard ook voor de tearDown-methode met het verschil dat deze zal moeten worden voorzien van de *@After* annotatie. Een van de grote verschillen tussen de twee versies op dit punt is, dat versie 3.8 slechts één setUp en één tearDown methode kon bevatten. Versie 4 laat meerdere van dit soort methodes toe.
Bij versie 3.8 was ook vaak behoefte om een bepaald stukje(voornamelijk initialisatie-) code alleen tijdens het opstarten van de testcase of voor het beëindigen van de testcase uit te voeren, maar dit was helaas niet mogelijk. Denk daarbij aan een bepaalde code waarvan de uitvoering meer tijd in beslag neemt, zoals bijvoorbeeld het opzetten van een database verbinding. In tegenstelling tot eerdere versies van JUnit is het vanaf versie 4 mogelijk gebruik te maken van de *@BeforeClass* en *@AfterClass* annotaties.
Door de *@BeforeClass* annotatie aan een public static methode te koppelen zal deze methode altijd alleen bij het opstarten van de testcase uitgevoerd worden. Hetzelfde geldt voor de methode voorzien van de *@AfterClass* annotatie, deze wordt eenmalig voor het afsluiten van de testcase uitgevoerd.

```
@BeforeClass
public static void initDB(){
  . . .
}

@AfterClass
public static void cleanupDB(){
  . . .
}
```

De *@BeforeClass* methodes van de superclasses zullen altijd vóór de methodes van de huidige class, en de *@AfterClass* methodes van de superclasses zullen altijd ná de methodes van de huidige class uitgevoerd worden. Wanneer de uitvoe-

Een normale testmethode kan automatisch worden aangeroepen uit een lijst met vooraf gedefinieerde testparameters

ring van de *@BeforeClass* methode een exception werpt, zal het geen consequenties hebben op de uitvoerbaarheid van de *@AfterClass* methode. Een testcase mag maar één *@BeforeClass* en één *@AfterClass* methode bevatten, in tegenstelling tot de setUp en tearDown methodes.

Exception handling

Een andere belangrijke eigenschap van JUnit versie 4 is de manier waarop de exceptions getest kunnen worden. In de oudere versies van JUnit moest de ontwikkelaar zelf voor het opvangen van de geworpen exceptions zorgen en voor de juiste afhandeling daarvan.
De *@Test* annotatie kan uitgebreid worden met verschillende parameters. Eén van de mogelijke parameters is de *expected* parameter. De waarde van de *expected* parameter bevat het type van de Exception, dat als resultaat van deze testmethode uitvoering verwacht wordt. Als het uitvoeren van deze testmethode in een andere Exception resulteert dan de exception gespecificeerd in de *expected* parameter of helemaal geen exception dan zal deze test falen.

```
@Test(expected=IllegalArgumentException.class)
  public void orArrayAndException(){
  . . .
```

Wanneer er meer informatie gewenst is over de Exception of wanneer andere eigenschappen getest moeten worden, dan zal toch de oudere try-catch stijl moeten worden gebruikt.

Tests automatisch overslaan en performance tests

Als we de annotatie *@Test* in *@Ignore* veranderen, is dit de manier om aan te geven dat de test methode in beschouwing moet worden genomen, maar niet moet worden uitgevoerd (bijvoorbeeld wanneer een test te lang duurt of de benodigde resources niet beschikbaar zijn). Deze test zal dan worden opgenomen in het testrapport met de status 'niet uitgevoerd'.

```
@Ignore("Nog niet geimplementeerd")
@Test
public void neg(){}
```

Ook is het mogelijk om aan de *@Ignore*-annotatie een String-parameter toe te voegen, die in het rapport de reden aangeeft waarom de test niet uitgevoerd is. Zo kunnen nog niet geïmplementeerde testmethoden wel ingevuld zijn. Houd er rekening mee dat de *@Ignore* annotatie slechts voor tijdelijk gebruik bedoeld is. Er zal altijd een reden zijn dat deze tests worden gedefinieerd. Bij het blijven negeren van deze tests kan de code, die deze methode veronderstelt te testen, breken. Dit betekent dat de fout niet zal worden ontdekt. Iedereen heeft wel eens meegemaakt dat het uitvoeren van een test veel te lang duurt en uiteindelijk de desbetreffende test faalt; dit zal bijvoorbeeld kunnen gebeuren bij het opzetten van een database of een remote host verbinding.

```
@Test(timeout=5000)
public void andArray(){
  . . .
```

In eerdere versies van JUnit was er ook geen enkele vorm van ondersteuning voor het maken van performance tests. Met de komst van JUnit versie 4 is hier duidelijk verbetering in gekomen. Door een time-out parameter aan de *@Test* annotatie toe te voegen met de juiste waarde (het aantal milliseconden) kan de gebruiker aangeven dat het uitvoeren van de geannoteerde testmethode niet langer mag duren dan het aantal milliseconden gespecificeerd als annotatie-argument. Zodoende heeft de gebruiker de mogelijkheid om zijn testcases vaker aan te roepen. De uitvoering van de wat zwaardere methodes duurt niet langer dan de gespecificeerde time-out.

Geparameteriseerde tests

Een andere handige eigenschap van JUnit versie 4 is de mogelijkheid, voor het maken van geparameteriseerde tests. Dit betekent hoofdzakelijk dat een normale testmethode automatisch kan worden aangeroepen uit een lijst met vooraf gedefinieerde testparameters. De vorige versies van JUnit

hadden beperkte mogelijkheden om dit te bereiken.
Bijvoorbeeld:

- door binnen de testmethode met behulp van een loop dezelfde code aan te roepen met de verschillende parameters (bij het optreden van een fout echter stopte het testen van de resterende parameters).
- door een testmethode te definiëren per test parameter

De stappen die doorlopen moeten worden voor het opzetten van een geparametriseerde test met JUnit versie 4 zijn als volgt:

- Eerst moet er een static methode gedefinieerd worden die voorzien is van de *@Parameters* annotatie. Deze methode geeft de verzameling van de testparameters als resultaat terug.
- Dan moet voor iedere testparameter type een class member aangemaakt worden. Deze class members zullen geïnitialiseerd moeten worden in een constructor van de testcase die al deze parameters als input paramaters heeft.
- Als laatste stap moet de testcase voorzien worden van de *@RunWith(Parameterized.class)* annotatie.

```
@RunWith(Parameterized.class)
public class ParametrizedTestBooleanExpression
{
      static   BooleanExpression   truev   =   new
BooleanExpression(true);
     static   BooleanExpression   falsev   =   new
BooleanExpression(false);
  /** boolean expression test object */
  private BooleanExpression be;
  /** test waarde */
  private boolean val;
  /** test resultaat */
  private boolean resultaat;

  /**
   *
    * Default constructor met test parameters als
methode parameters.
   * Initializeert de test parameters.
   *
   */
  public ParametrizedTestBooleanExpression(BooleanEx
pression be, boolean val,
       boolean resultaat)
  {
    super();
    this.be = be;
    this.val = val;
    this.resultaat = resultaat;
  }

  @Parameters
   public static Collection booleanExpressionTestVa-
lues()
  {
    return Arrays.asList(new Object[][] {
{ truev, true, true },
        { truev, false, true },
{ falsev, true, true },
        { falsev, false, false } });
  }

  @Test
  public void or()
  {
   assertEquals("fout in de or methode:", be.or(val).
getValue(), resultaat);
  }
```

Testsuites

Het definiëren van testsuites met JUnit versie 4 gebeurt door gebruik te maken van de volgende annotaties.

- *@RunWith(Suite.class)* wordt gebruikt om aan te geven dat het om een test suite class gaat.
- *@Suite.SuiteClasses* annotatie wordt gebruikt voor het definiëren van de lijst van testcases die deel uitmaken van de test suite. De testcases moeten in de vorm van een array als attribuut aan de annotatie toegevoegd worden.

```
@RunWith(Suite.class)
@SuiteClasses( {
  EersteTest.class,
  TweedeTest.class,
  DerdeTest.class }
)
```

Asserts

In tegenstelling tot oudere versies van JUnit waarin de *assertEquals* methode alleen op twee objecten aangeroepen kon worden, kan met JUnit versie 4 met behulp van autoboxing(Java 5) de assertEquals op twee objecten aangeroepen worden, maar ook op twee primitieven. De primitieven worden automatisch omgezet door Java 5 naar de corresponderende Java-objecten.

```
@Test
public void or()
{
  assertEquals("fout in de or methode: true | true",
truev.or(true)
         .getValue(), true);
}
```

Een andere toevoeging betreffende de *assertEquals* methode zijn de twee nieuwe *assertEquals* methodes, die bedoeld zijn voor het vergelijken van arrays. Twee arrays van objecten zijn gelijk:

- indien het aantal elementen in de twee verschillende arrays gelijk zijn
- ieder object uit de ene array correspondeert met hetzelfde object uit de andere array.

Integratie met Eclipse

Versie 4 van JUnit is op Java 5 gebaseerd. Om JUnit versie 4 vanuit Eclipse te kunnen gebruiken, zal Eclipse ondersteuning moeten bieden voor het bouwen van applicaties met Java 5. Vanaf Eclipse versie 3.1 is ondersteuning voor Java 5 standaard aanwezig. Het is noodzakelijk om het Compiler compliance level op 5.0 te zetten bij de Java-preferences. Dit is de

manier om Java 5 compliance in Eclipse versie 3.1.x te activeren. Om de JUnit versie 4 tests te kunnen maken en compileren zal de JUnit versie 4 jar binary file aan de build path van het project toegevoegd moeten worden. Om vanuit Eclipse de tests ook uit te kunnen voeren, maken wij gebruik van de welbekende JUnit plugin. Eerst zal echter een testsuite aangemaakt moeten worden. Deze testsuite zal gebruik moeten maken van de JUnit4TestAdapter, aangezien deze testrunner op de JUnit versie 3.8 is gebaseerd. Deze testsuite zal dan een verzameling van de JUnit versie 4 tests bevatten.

```
  @RunWith(Suite.class)
      @SuiteClasses(    {    BooleanExprTest.class,
ArithmeticnExprTest.class })
  public class AllTests {

    public static Test suite() {
      return new JUnit4TestAdapter(AllTests.class);
    }
  }
```

Dit betekent dus, dat vanuit Eclipse versie 3.1.x de JUnit versie 4 tests alleen maar via een testsuite uitgevoerd kunnen worden. Voor de versie 3.2.x/3.3.x van Eclipse geldt dat deze out of the box ondersteuning biedt voor het uitvoeren van JUnit versie 4 tests. Om de JUnit versie 4 tests te kunnen maken en compileren zal de JUnit versie 4 jar binary file aan de build path van het project toegevoegd moeten worden. Als binnen Eclipse versie 3.2.x. voor het eerst een JUnit versie 4 test wordt uitgevoerd, zal de Run-configuratie voor JUnit tests aangepast moeten worden. Op het Run-configuratiescherm zal de Test runner op JUnit4 ingesteld moeten worden.

Conclusie

JUnit versie 4 is dus duidelijk een grote verbetering van dit veelgebruikte framework ten opzichte van de eerdere versies. Bij het opzetten van JUnit versie 4 heeft men rekening gehouden met de gebruikers van de eerdere versies van JUnit. Backward compatibility is één van de belangrijkste kenmerken van JUnit versie 4. Dankzij de backward compatibility kunnen alle bestaande JUnit versie 3.8 testcases ook gebruikt worden in combinatie met JUnit versie 4. Aangezien steeds meer applicaties met Java 5 worden gebouwd is het zeker te overwegen om de overstap te maken. Alle nieuwe features zijn vrij snel aan te leren, dus je zult snel 'up to speed' zijn met het maken van Junit versie 4 testcases, aangezien de meeste features al op Java 5.0 gebaseerd zijn. JUnit versie 4 is in combinatie met een applicatie gebouwd in Java 5.0 de meest voor de handliggende keuze.