Vrije Universiteit Brussel

FACULTEIT INGENIEURSWETENSCHAPPEN & WETENSCHAPPEN

JUnit 4 handleiding

Software Engineering

Nicolas Carraggi, Youri Coppens, Christophe Gaethofs, Pieter Meiresone, Sam Van den Vonder, Fernando Suarez, Tim Witters

Academiejaar 2013-2014

Inhoudsopgave

Hoofdstuk 1

Inleiding

Deze handleiding is opgesteld om het team in te leiden in unittests, zodat zij na het lezen van deze handleiding unittests kunnen schrijven voor hun code. Een unittest is een stukje code die bepaalde functionaliteit in code gaat testen. Unittests worden geschreven voor een kleine hoeveelheid code (m.a.w. een methode of klasse). Door unittests kan verzekerd worden dat code werkt, en blijft werken na het toepassen van veranderingen of bugfixes.

1.1 Wanneer schrijft men tests

In een ideale situatie zijn er tests voor elk stuk code, maar tests schrijven voor alle code is tijdverslindend en overbodig. Voor triviale stukjes code schrijft men dus geen unittests, zoals bijvoorbeeld voor getters en setters die aan simpele assignment doen. Tests worden slechts geschreven voor kritieke en complexe delen van de applicatie. Een goede test suite beschermt ook tegen regressies in bestaande code wanneer je nieuwe features introduceert.

1.2 Verschillende soorten tests

1.2.1 Unittests

Unittests zijn klein en atomisch. Het is de bedoeling dat ze het gedrag van een methode testen, niet van het hele systeem. Wanneer een programmeur klaar is met iets te coderen moet hij op dit deel van de code unittests draaien. Unittests mogen geen rekening houden met het gedrag van geïmporteerde code. Voor deze gevallen gebruikt men integratietests.

1.2.2 Integratietests

Integratietests zijn de tests die de samenwerking testen tussen verschillende componenten. Men zou kunnen zeggen dat integratietests toetsen of de applicatie werkt als een geheel. Deze tests zouden veel meer regressies moeten vangen dan unittests zelf.

1.2.3 Verificatietests

Verificatietests zijn het bewijs dat de applicatie voldoet aan bepaalde functionaliteit. Alle functionaliteit die aangeboden wordt moet kunnen getest worden d.m.v. geautomatiseerde tests.

Hoofdstuk 2

Praktische Informatie

2.1 Installatie

In de ontwikkelingsomgeving Eclipse[1] is JUnit standaard beschikbaar. Het is daarom ook aangeraden om deze ontwikkelomgeving te gebruiken. In alle andere gevallen voorziet de website van JUnit[2] zelf alle informatie om JUnit te gebruiken.

2.2 Inleiding gebruik

Het gebruikte testplatform is JUnit. In dit framework worden methodes geannoteerd om methoden te identificeren die een test specifiëren. Deze methoden bevinden zich gewoonlijk in een klasse die enkel gebruikt wordt om te testen: een test klasse. JUnit neemt aan dat alle tests in arbitraire volgorde uitgevoerd kunnen worden, m.a.w. de programmeur zorgt ervoor dat tests niet afhankelijk zijn van andere tests.

Tests plaatst men gewoonlijk in een aparte source folder zodat de tests zich niet vermengen met de normale code. In de Eclipse ontwikkelomgeving kunnen tests aangemaakt worden via File → New → JUnit Test Case

Neem de volgende Java klasse:

```
1 package junit_test;
2
3 public class testKlasse {
4   public int add(int a, int b) {
5     return a + b;
6   }
7 }
```

We maken nu voor deze klasse een JUnit Test Case die vervolgens in de package junit_test_tests wordt gestopt. Uiteraard wordt de testklasse geïmporteerd in het bestand van JUnit.

```
1 package junit_test_tests;
2
3 import static org.junit.Assert.*;
4 import org.junit.Test;
5
6 import junit_test.testKlasse;
7
8 public class testKlasseTest {
9   @Test
10  public void test() {
```

[1] Eclipse: http://www.eclipse.org/

[2] JUnit download and install: https://github.com/junit-team/junit/wiki/Download-and-Install)

```
11      testKlasse klasse = new testKlasse();
        assertEquals("1 + 1 is 2", 2, klasse.add(1, 1));
13    }
}
```

In deze JUnit file kan in Eclipse op "run"gedrukt worden om alle tests te runnen.

2.3 Annotaties/Labels

Om een test te schrijven met JUnit moet - zoals al opviel in bovenstaande code - de test geannoteerd worden met @Test. De volgende tabel geeft een overzicht van de mogelijke annotaties:

Annotatie	Beschrijving
@Test	De @Test annotatie specifieert dat een methode een test-methode is
@Test (expected = Exception.class)	Faalt als de methode niet de genoemde Exception throwed
@Test(timeout=100)	Faalt als de methode langer dan 100ms duurt
@Before	Geeft aan dat de geannoteerde methode uitgevoerd moet worden voor alle tests, kan bijvoorbeeld nuttig zijn voor het initialiseren van data
@After	Methode met deze annotatie wordt uitgevoerd na elke test. Kan gebruikt worden om de testomgeving op te kuisen (bv. tijdelijke data te verwijderen)
@BeforeClass	Deze methode wordt eenmalig uitgevoerd voor de start van alle tests. Wordt gewoonlijk gebruikt voor activiteiten die veel tijd vragen, zoals een verbinding initialiseren naar een database. Methodes die met @BeforeClass geannoteerd zijn moeten static zijn om te werken met JUnit.
@AfterClass	Wordt eenmalig uitgevoerd na alle tests, bv. om een database-verbinding af te sluiten. De methode hierdoor geannoteerd moet static zijn.
@Ignore	Negeer de methode die hierdoor geannoteerd is. Nuttig wanneer de code van de te testen methode is aangepast, maar de bijhorende test nog niet werkt met deze veranderingen.

2.4 Soorten tests

JUnit voorziet static methodes in de Assert klasse om voor bepaalde condities te testen. Deze assertion methods beginnen gewoonlijk met assert en staan je toe het errorbericht, de verwachte waarde en de eigenlijke resultaat te specifiëren. Een assertion method vergelijkt de verkregen waarde met de verwachte waarde en throwed een AssertionException wanneer de test faalt.

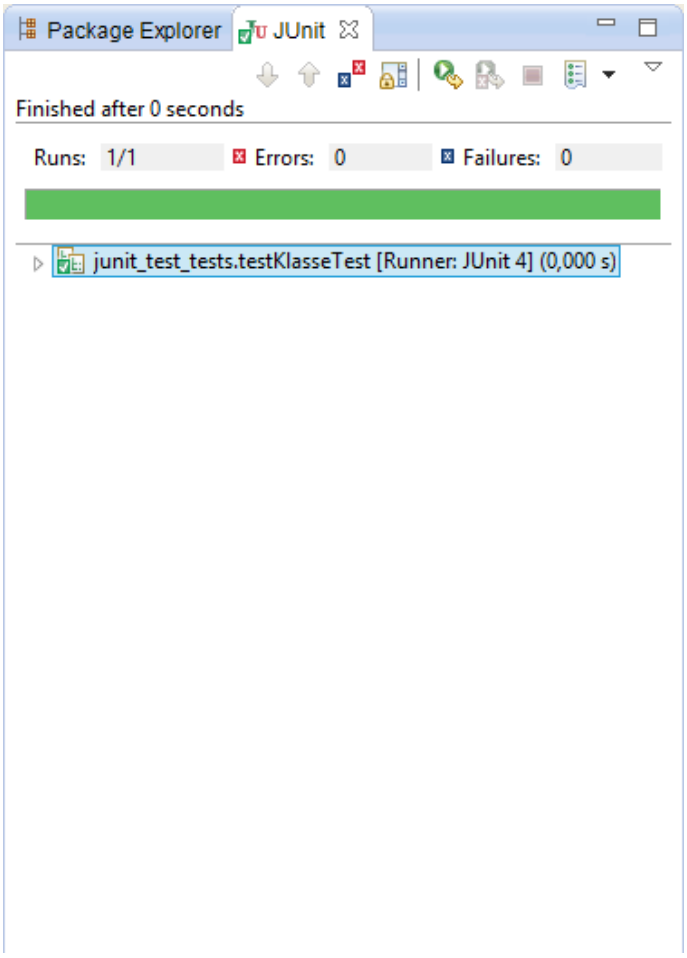
De volgende tabel geeft een overzicht van deze methodes. De parameters tussen rechte haakjes zijn optioneel. Zoals gewoonlijk is de programmeur uiteraard verantwoordelijk voor betekenisvolle berichten in de assertions om andere programmeurs te helpen problemen op te lossen.

Methode	Beschrijving
fail(string)	Wordt gebruikt om ervoor te zorgen dat een bepaald deel van de code niet wordt bereikt, of wanneer code voor een methode nog niet is geïmplementeerd. De String parameter is optioneel.
assertTrue([message], boolean condition)	Check of de booleaanse conditie true is.
assertFalse([message], boolean condition)	Check of the booleaanse conditie false is.
assertEquals([String message], expected, actual)	Test of twee waarden gelijk zijn. Voor arrays wordt de referentie vergeleken, niet de inhoud van de arrays.
assertEquals([String message], expected, actual, tolerance)	Test of double of float waarden hetzelfde zijn. De tolerantie parameter is hoeveel getallen minstnes gelijk moeten zijn.
assertNull([message], object)	Check of het object null is.
assertNotNull([message], object)	Check of het object niet null is.
assertSame([String], expected, actual)	Check of beide variabelen naar hetzelfde object verwijzen.
assertNotSame([String], expected, actual)	Check of beide variabelen naar een ander object wijzen.

Hoofdstuk 3

Tests runnen

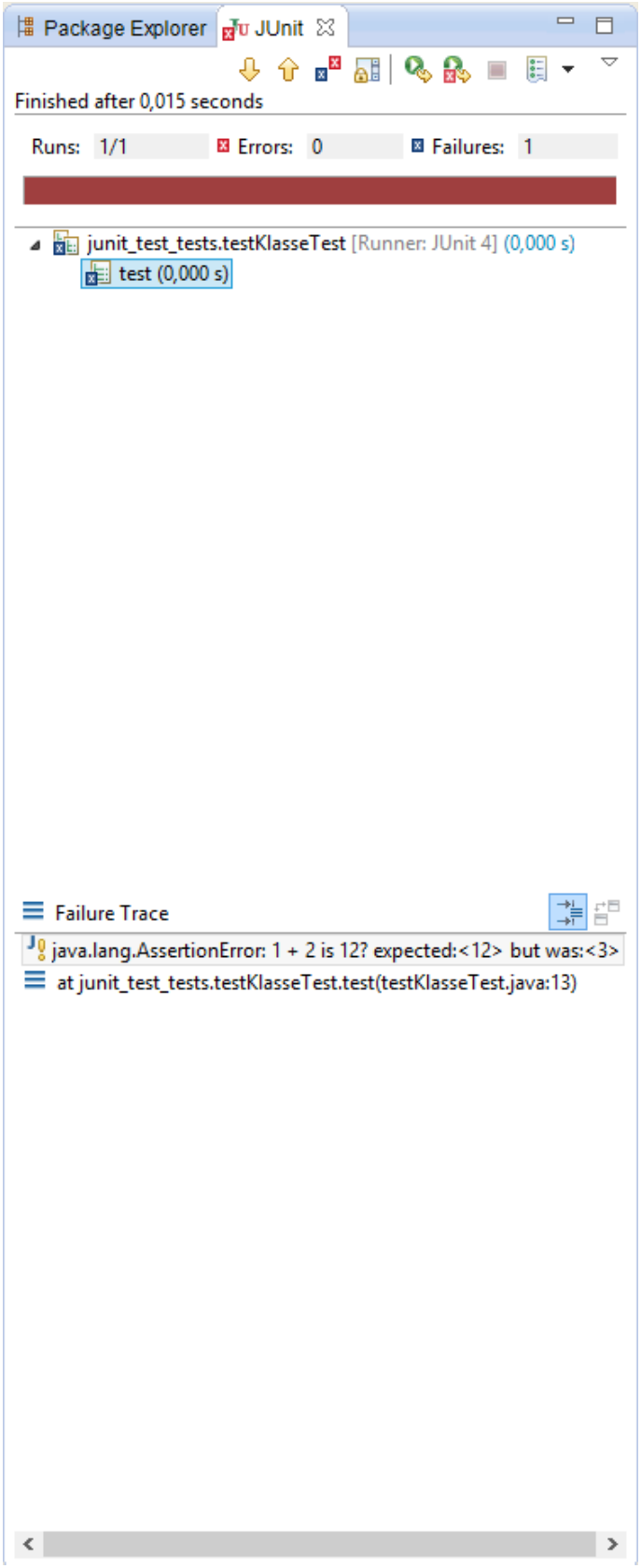
Om een klasse uit te voeren met JUnit in Eclipse kan je uit de menubalk kiezen voor run → run as → JUnit Test. Ook kan je gebruik maken van de keyboard shortcut Alt+Shift+X, T. Wanneer een test slaagt ziet de JUnit-GUI in Eclipse er zo uit:



Figuur 3.1: Alle tests geslaagd

Voor volgende afbeelding werd een falende test toegevoegd die checkt of 1+2 gelijk is aan 12. Merk op dat

onderaan meer info wordt gegeven over de gefaalde test. Er wordt duidelijk aangegeven dat de verwachte waarde 12 is, maar dat 3 de waarde is die de te testen methode returnde.



Figuur 3.2: Gefaalde test