Unit Testing

Unit Testing is het testen van individuele stukken code. Dit is meestal het kleinste testbare stukje. In Procedural Programming kan dat een hele module zijn, maar vaker een functie of procedure. In object-oriented programming, een unit is vaak een hele interface, zoals een class, maar het kan ook een methode zijn.

Unit Testing wordt gebruikt om te zien of een stuk code daadwerkelijk doet en werkt zoals bedoelt is.

Hier wordt het nogmaals duidelijk uitgelegd:

<https://codeutopia.net/blog/2015/03/01/unit-testing-tdd-and-bdd/>

# Test Driven Development

Test Driven Development, ookwel TDD, is een process voor wanneer je je tests schrijft en runt. Met gebruik van TDD kan je een heel hoog test-coverage quota halen.

TDD wordt gedaan in de volgende stappen:

1. Start met het schrijven van een test
2. Run de test en alle andere tests die je hebt. Hier zou de nieuwe test moeten falen. Als dat niet het geval is zit er waarschijnlijk een bug in de test.
3. Schrijf het minimum aantal code dat nodig is om de test succesvol te laten wezen.
4. Run de test en check of de nieuwe test succesvol is. Zo ja: ga door naar stap 5. Zo nee: los het probleem op en voer daarna stap 4 uit.
5. Optimaliseer de code en check of test nog succesvol verloopt.
6. Herhaal vanaf stap 1.

Dit is in principe ook de red-green-refactor-cycle.

Hier een kata om dit te oefenen

<http://osherove.com/tdd-kata-1/>

**Unit testing via IntelliJ**

Als je in je IntelliJ bestand klassen of methodes met @Test markeert en je de nieuwste versie van JUnit in bezit hebt dan worden deze in de testfase van maven uitgevoerd. Ook worden ze uitgevoerd in IntelliJ als je op run whole project drukt of als je het testbestand runt.

Hier is wat code als voorbeeld voor het testen van een erg simpel stukje code (gejat van Déjan, bedankt).

|  |
| --- |
| package com.oose;  public class App {  public static void main(String[] args) {  System.out.println(getHelloWorld());  }   public static String getHelloWorld() {  return "Hello World";  } } |

Als je dit zou packagen en runnen dan zou het hello world printen.

Dit wil je natuurlijk wel testen voordat je het packaged zodat je niet tijd verspild in het geval dat het niet werkt en in plaats van hello world misschien wel goodbye world print.

Om dit te testen maak je een extra bestand aan in IntelliJ (of je pakt gewoon de automatisch gemaakte appTest). ←- volgens mij moet je per klasse een nieuwe testklasse maken. Bijvoorbeeld voor de class Calculator, maak je een testklasse CalculatorTest. Dit dan om meer structuur en duidelijk te krijgen in je code.

In die appTest doe je dan weer een stukje code schrijven:

|  |
| --- |
| package com.oose;  import org.junit.Assert; import org.junit.Test;  import static org.junit.Assert.assertTrue;  public class AppTest {  int value1 = 2;  int value2 = 3;  @Test  public void testPrintHelloWorld() {  Assert.assertEquals(App.getHelloWorld(), "Hello World");  }  @Test  public void testAdd(){  int result = value1 + value2;  assertTrue(result == 5);  } } |

In dit stukje code kijk je eerst of de methode getHelloWorld wel “Hello World” uitprint. Als de tekst exact overeenkomt met wat je bij assertequals aangeeft dan is de test succesvol uitgevoerd.

In het tweede stukje voeren we de test uit of 2 variabelen samen 5 zijn dit doen we hier met asserttrue. Als je meer wilt weten over assert dan lees deze link <https://github.com/junit-team/junit4/wiki/Assertions>

**Code coverage**

Code coverage is een afmeting om te laten zien hoeveel code van jou getest is. Dit word meestal aangeduid met een procentuele waarde, waarbij er gestreefd wordt naar 100% code coverage zodat je weet dat al jouw code getest is.

Om te kijken of je code coverage wel goed is in IntelliJ druk je op run (Alt+U) en dan op “run ‘appTest’ with coverage”. Dan laat hij zien hoeveel % van elke class, method en line gedekt is.

Dit knopje maakt een coverage rapport: 

Als je dan dubbelklikt op de coverage app dan gaat het kijken welke files specifiek gedekt zijn in plaats van het hele project. Dus dan zie je bijvoorbeeld dat App.java helemaal perfect is maar dat je andere klassen vol zitten met fouten.

# In unit tests checken of de juiste exceptions op het juiste moment komt

Je kunt op verschillende manieren testen of er een exceptie gegeven wordt door een functie:

Vanaf JUnit 4.7 (uitgebracht in 2009) kun je gebruik maken van de ExpectedException, waarop je dan weer de expect methode op kan aanroepen.

|  |
| --- |
| public class EenTest {  @Rule  public final ExpectedException exception = ExpectedException.none();   @Test  public void doIetsDatIndexOutOfBoundsExceptionThrowt() {  EenAndereKlasse eenKlasse = new EenAndereKlasse();  exception.expect(IndexOutOfBoundsException.class);  eenKlasse.doStuff();  } } |

Als je liever een eerdere versie van JUnit 4 wilt gebruiken kun je de verwachte waarde invullen in het Test keyword, zoals hieronder gebeurd:

|  |
| --- |
| @Test(expected = IndexOutOfBoundsException.class) public void testIndexOutOfBoundsException () {  ArrayList emptyList = new ArrayList();  Object o = emptyList.get(0); } |

Of als je helemaal niet gebruik wilt maken van JUnit om jouw code te testen, dan kan je natuurlijk een normale try-catch gebruiken.

**JUnit test framework** is wat we gebruiken:

Dit betekent dat we even deze site hier moeten lezen:

<https://www.tutorialspoint.com/junit/junit_test_framework.htm>

Een testcase is een aantal tests in een groepje dat gebruik maakt van dezelfde setup en teardown. Dit betekent dat je tests krijgt die makkelijk te herhalen zijn met dezelfde situaties die achter elkaar uitgevoerd kunnen worden.

Je kan het voorstellen als een testcase voor als admin inloggen en een voor als gebruiker inloggen. Het enige dat verschilt in de setup is dat je andere inloggegevens/methode gebruikt, maar in het programma zelf kan de admin veel meer. Je kan dan heel veel tests schrijven die allemaal de admin gegevens gebruiken, of je kan een keer in de setup de gegevens neerzetten en dan kan je die bij elke test in die testcase gebruiken.