# Software Design Essentials

# Opgave 5 - Junit

## Programmeren zonder Junit

Stel dat we zouden willen testen zonder dat we gebruik zouden maken van een testframework. Zo zouden we bijvoorbeeld testen willen schrijven voor het eenvoudige codevoorbeeld dat hieronder wordt weergegeven:

```
public class Calculator {
	public double add(double n1, double n2) {
		return n1 + n2;
	}
}
```

We zouden hiervoor een testklasse kunnen schrijven met onderstaande code:

```
public class CalculatorTest {

	public void testAdd(){
		Calculator calc = new Calculator();
		double result = calc.add(10,20);
		if(result != 30)
		   throw new IllegalStateException("Wrong result: " + result);
		}

	public static void main(String[] args) {
		CalculatorTest test = new CalculatorTest();
		test.testAdd();
	}
}
```

Deze manier van werken zou eventueel kunnen gebruikt worden voor zeer kleine programma's, maar wanneer een programma verder uitbreid zal deze manier van werken niet meer efficiënt zijn. Daarom werden testframeworks zoals Junit uitgevonden. De 3 belangrijkste doelen van Junit als testframework zijn:

- Het framework helpt ons bij het schrijven van bruikbare testen
- Het framework helpt ons bij het maken van testen die hun waarde behouden op langere termijn
- Het framework helpt ons om sneller testen te schrijven

## Aanmaken van eerste test

In combinatie met JUnit gebruikt men dikwijls het principe van "**T**est-**D**riven **D**evelopment". Dit wil zeggen dat het schrijven van een applicatie eigenlijk wordt gestuwd door het schrijven van tests. Men noemt dit ook wel "Development through testing". TDD is echter niet verplicht in JUnit, maar het wordt wel vaak toegepast. Het idee achter TDD is:

1. Schrijf een test voor het volgende stukje functionaliteit dat je wil toevoegen. Deze test zal moeten slagen wanneer de functionaliteit is toegevoegd.

2. Laat de test compileren door stubs te voorzien voor de klassen en methodes die de test nodig heeft.
3. Run de test. Falen is nu normaal...
4. Implementeer *juist* genoeg functionaliteit om de test te laten slagen.
5. Test opnieuw, de test moet nu slagen!

In onze eerste JUnit test zullen we het principe van "test-driven development" hanteren. We schrijven eerst een test, daarna pas de code. Het resultaat dat we willen bereiken is het berekenen van het maximum van drie meegegeven getallen aan een methode. Deze methode zullen we berekenMax(int a,int b,int c) noemen, ze wordt als statische methode aangemaakt in een klasse MathTool. Maar eerst...de test!

1. Creëer een nieuw Java-Project. Noem dit bv. TestDrivenDevelopment.
2. Maak een test-klasse.
3. In het "New JUnit Test Case" properties scherm, stel volgende zaken in:
    - Kies als naam voor de klasse MathToolTest
    - Kies als package-naam be.ehb.gdt.tools.math. Hier zien we meteen de naming conventions voor package-namen: **begin met de DNS-naam van je bedrijf, in omgekeerde volgorde. Voeg daarna de logische naam voor het package toe.** Het gaat hier over een math package, wat intern zouden kunnen onderbrengen in een package tools. Gebruik **alleen lowercase** letters!
4. Op het einde bij het toevoegen van de test krijg je de melding dat Junit 4 niet in het build path voorkomt. Selecteer de optie om Junit 4 als library toe te voegen tot je build path.

Nu is onze test-klasse aangemaakt. Nu moeten we nog een of meerdere methodes toevoegen die ook effectief zullen testen. In een test-methode roepen we één van de varianten van assertEquals(...) op. Belangrijk is dat we vóór elke testmethode de notatie "@Test" moeten bijschrijven. Hiermee geven we aan dat het om een test gaat die moet uitgevoerd worden door het Junit platform. In ons geval is volgend voorbeeld een mogelijke test:

```
package be.ehb.dt.tools.math;

import org.junit.Assert;
import org.junit.Test;

public class MathToolTest {
	@Test
	public void testMax() {
		int expected = 10;
		int actual = MathTool.berekenMax(10, 5, 2);
		Assert.assertEquals(expected, actual);
	}
}
```

Er is geen verplichting om de testmethode een bepaalde naam te geven maar meestal wordt er wel gekozen voor testXXX() waarbij XXX de naam is van een methode of systeem dat je wil testen. In JUnit 3 was het nog wel verplicht elke testmethode met het woord *test* te laten beginnen.

Om de test te kunnen laten uitvoeren, moet er natuurlijk een berekenMax(...) methode bestaan in de klasse MathTool. We maken de klasse MathTool aan en voegen een methode berekenMax() toe! Voorzie nog geen implementatie maar geef gewoon eender welk getal terug. De methode stuurt immers het resultaat (max) terug.

Nu kan je de test laten lopen en de resultaten bekijken. Je zou 1 failure moeten krijgen.

Implementeer nu je methode berekenMax(...) zodat deze effectief het grootste getal terugstuurt. Test opnieuw. Krijg je nog steeds fouten? Verbeter indien nodig...

## Toevoegen van extra tests

Voeg nu nog een aantal testen toe in de methode testmax(). Dit kan door meerdere assertEquals() statements op te nemen. Wanneer 1 van de assertEquals methoden een fout geeft, wordt de test-methode als niet-geslaagd beschouwd. Voeg zo'n tests toe voor volgende cases:

- berekenMax(1,2,3) geeft 3 als resultaat
- berekenMax(3,2,1) geeft 3 als resultaat
- berekenMax(2,1,3) geeft 3 als resultaat
- berekenMax(3,1,2) geeft 3 als resultaat
- berekenMax(1,3,2) geeft 3 als resultaat
- berekenMax(2,3,1) geeft 3 als resultaat

Je kan ook nieuwe tests toevoegen door het aanmaken van nieuwe testmethoden. Een test-methode moet sowieso de annotatie "@Test" krijgen . Dan wordt de test automatisch door JUnit herkend en uitgevoerd. Voeg bijvoorbeeld een test toe voor een methode berekenGem(...) dat het gemiddelde van 2 floats zal berekenen. Schrijf na het uitwerken van de test de methode zelf.

Merk op dat de volgorde van uitvoering van de verschillende testmethoden niet voorzien kan worden op voorhand. Dit wil dan ook zeggen dat je je testmethode zo moet schrijven dat ze niet afhankelijk is van andere methoden. Er zijn wel enkele mogelijkheden om bepaalde zaken te initialiseren of af te breken voor en na elke test. Deze mogelijkheden worden hieronder besproken.

## De annotations @Before en @After

Bij het schrijven van Unit-testen voor object-georiënteerde systemen zal je dikwijls instanties moeten aanmaken van bepaalde klassen. Wanneer je over meerdere testmethoden wil werken met eenzelfde object, is dat niet echt mogelijk in JUnit 4. Het is in feite ook niet de bedoeling aangezien elke test op zich moet staan. Bijvoorbeeld: (typ de code over)

```
public class Bankrekening {
	private int rekeningnummer;
	private double saldo;

	public Bankrekening(int nr, double startBedrag) {
		rekeningnummer = nr;
		saldo = startBedrag;
	}

	public double getSaldo() {
		return saldo;
	}

	public void stort(double bedrag) {
		saldo += bedrag;
	}

	public void haalAf(double bedrag) {
		saldo -= bedrag;
	}
}
```

Om dit te testen zou je de volgende unit test kunnen gebruiken: (typ in en test!)

```
public class BankrekeningTest {
```

```
	@Test
	public void testStort() {
		Bankrekening rekening = new Bankrekening(1, 100);
		rekening.stort(200);
		assertEquals(300, rekening.getSaldo(), 0);
	}

	@Test
	public void testHaalAf() {
		Bankrekening rekening = new Bankrekening(1, 100);
		rekening.haalAf(50);
		assertEquals(50, rekening.getSaldo(), 0);
	}
}
```

In plaats van telkens (het zijn nu nog maar 2 testen maar dat kunnen er 1000 worden...) een nieuw object rekening aan te maken gebruiken we hetzelfde object:

```
public class BankrekeningTest {
	Bankrekening rekening;

	public BankrekeningTest() {
		rekening = new Bankrekening(1, 100);
	}

	@Test
	public void testStort() {
		rekening.stort(200);
		assertEquals(300, rekening.getSaldo(), 0);
	}

	@Test
	public void testHaalAf() {
		rekening.haalAf(50);
		assertEquals(50, rekening.getSaldo(), 0);
	}

}
```

Je krijgt nog steeds hetzelfde resultaat! Dit komt doordat de constructor van BankRekeningTest voor elke test opnieuw wordt opgeroepen. Test dit zelf uit door bv. een System.out.println("Dit is de constructor") in deze constructor te plaatsen. In JUnit 4 (zie verder) kan een speciale functie worden voorzien die vóór het uitvoeren van alle tests 1x wordt uitgevoerd. Het hergebruiken van objecten voor meerdere testen wordt echter afgeraden, probeer dit dus niet te veel te gebruiken... **Opgelet: de hierboven beschreven methode waarin in de constructor van de test-klasse bepaalde initialisaties gebeuren wordt sterk afgeraden!** Daarvoor gaan we nu enkele constructs bekijken die ons kunnen helpen bij het opzetten en afbreken van testen.

In vele gevallen wil je vóór elke test en na elke test een bepaalde actie altijd uitvoeren, zoals het initialiseren van variabelen, het opzetten van een database connectie, ... We kunnen echter niet uitgaan van een specifieke volgorde die JUnit zal handhaven voor het oproepen van de verschillende testfuncties: dit gebeurt random.

We kunnen onze testklasse echter voorzien van een methode met annotatie @Before en @After. De eerste methode wordt steeds opgeroepen alvorens er een testmethode wordt uitgevoerd. De @After wordt steeds opgeroepen wanneer de test-methode klaar is met uitvoeren. Deze methoden kan je best protected declareren. Je hoeft ze niet zelf op te roepen, dat doet het framework automatisch. We krijgen dus:

```
beforeMethode();

testmethode1();

afterMethode();


beforeMethode();

testmethode2();

afterMethode();


beforeMethode();

testmethode3();

afterMethode();
```

In de @Before methode kan een object geïnstantieerd worden, een file geopend, een database connectie gelegd, ...

In de @After kan een file gesloten worden, een database connectie mooi afgesloten, een tcp-verbinding afgesloten, …

## De mogelijkheden voor Assert

Er zijn een heel aantal methoden waarmee je een bewering (assertion) kan nagaan. De belangrijkste methoden die er in de Assert klasse voorkomen zijn: ([ ] is niet verplicht)

- **assertEquals([String message],expected, actual)**: vergelijkt twee waarden, de test slaagt als de waarden gelijk zijn
- **assertEquals([String message],expected, actual, delta)**: vergelijkt twee waarden met een tolerantie delta (zo veel mogen beide waarden uit elkaar liggen), de test slaagt als de waarden gelijk zijn. Wordt gebruikt voor double en float.
- **assertFalse([message],bool)**: controleert of de waarde van een test false is, indien zo dan slaagt de test
- **assertTrue([message],bool)**: controleert of de waarde van een test true is, indien zo dan slaagt de test
- **assertNull([message],object)**: controleert of de referentie een null-pointer is, indien zo dan slaagt de test
- **assertNotNull([message],object)**: controleert of de referentie een null-pointer is, indien *niet* zo dan slaagt de test
- **assertSame([message],object1,object2)**: vergelijkt 2 referenties, indien gelijk dan slaagt de test (bv. handig voor een singleton)
- **assertNotSame([message],object1,object2)**: vergelijkt 2 referenties, indien ongelijk dan slaagt de test
- **fail()**: doet een test sowieso mislukken, bijvoorbeeld om te testen of je tot deze lijn code zal geraken...

## Uitbreiding van de Bankrekening

We gaan de Bankrekening klasse uitbreiden met enkele nieuwe functionaliteiten:

- Voorzie een methode **haalAfVeilig(bedrag)** die een bepaald bedrag afhaalt van de rekening maar waarbij niets wordt afgehaald en een exception wordt gegooid indien het totaalbedrag onder 0 komt te staan.
- Voorzie een methode **voegSamen(Bankrekening ander)** die de gelden van 2 bankrekeningen samenvoegt.

Schrijf enkele Unit testen voor deze bijgevoegde functionaliteiten.