# Unit testing

### Voorbeeld

### Te gebruiken bij:

-

### Eigenschappen:

-

### Info

een werkend stukje software vergt vaak nogal wat *trial and error*.

Wanneer je denkt dat je de juiste methode hebt geschreven, zul je dat ongetwijfeld controleren door jouw methode meerdere malen aan te roepen met verschillende waardes. Zo check je of de methode doet wat je verwacht en alle afwijkende scenario's hebt gedacht.

Onbewust ben je dus al bezig geweest met het *handmatig* testen van je code.

Handmatig testen is ontzettend langzaam.  
in de praktijk test je hele softwarepakketten, bij iedere update.

Het is in de praktijk gebruikelijk om**automated tests** te gebruiken:

En dit biedt een hoop voordelen:

* Wanneer de tests eenmaal geschreven zijn, is er geen "handarbeid" meer nodig. Ze zijn dus ontzettend goedkoop om (herhaaldelijk) uit te voeren;
* Het is razendsnel, de computer zal er hoogstens een paar milliseconden tot seconden voor nodig hebben;
* Omdat het snel en goedkoop is, kun je heel vaak en veel testen. Dit is bij doorontwikkeling van software erg belangrijk;
* Wanneer je vaak en veel test, kom je er direct achter als je iets per ongeluk stuk hebt gemaakt. Verschillende stukken code hangen in grote projecten nauw met elkaar samen, maar de afhankelijkheden zijn niet altijd even goed te overzien. Het is dus logisch dat je het niet altijd opmerkt dat het aanpassen van de *return value* op plek *x* iets stuk maakt op plek *y*. Automatisch testen zorgt er dus voor dat je minder vaak achteraf ontdekt dat bestaande features niet meer werken. Als dit wel gebeurd, noem je dit **unintended feature change**;

Binnen automatische testen onderscheiden we twee varianten:

* unit testen
  + **Unit testen** zorgen ervoor dat we kleine stukjes code - "units" - kunnen testen op hun werking.
  + Over het algemeen proberen we die units, net als de logica in onze methodes, zo klein mogelijk te houden.
  + Als we alleen een class wil testen
* Integratietesten
  + Bij **integratietesten** wil je vooral testen of de applicatie in hoofdlijnen goed functioneert.
  + Als we echter willen testen of de Btw van de producten correct berekend wordt, zul je snel merken dat een integratietest daar niet geschikt voor is.
  + We moeten namelijk de hele applicatie opstarten
  + Daarnaast moeten we dan het Btw-bedrag uit een webpagina gaan vissen.

#### jUnit

De standaard framework voor het schrijven van unit testen in Java is **JUnit**.

Dit is een open source framework.

Wanneer je gebruik wil maken van JUnit, zul je de volgende code moeten toevoegen aan de pom.xl

<dependencies>

...

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

<version>5.9.1</version>

<scope>test</scope>

</dependency>

</dependencies>

met alleen een testing framework heb je nog geen werkende unit test.  
Iedere test in onze applicatie zullen we zelf moeten schrijven. Hiervoor is het gebruikelijk om de test-code in een speciale test-class te zetten en in een aparte map op te slaan.

We testen de class BtwCalculator dus met de test-class BtwCalculatorTest

We slaan onze classes altijd op in src/main/java, maar onze test-bestanden slaan we op in src/test/java.

src

└── main

└── java

│ └── com

│ └── novi

│ ├── Product.java

│ └── BtwCalculator.java

└── test

└── com

└── novi

├── ProductTest.java

└── BtwCalculatorTest.java

Dit betekent dat een class in src/main/java/nl/novi/BtwCalculator wordt getest met een class in src/main/test/nl/novi/BtwCalculatorTest.

Daarnaast worden de methodes van de test-classes geannoteerd met @Test. Daardoor herkent JUnit (en daarmee ook veel tooling zoals Maven en IntelliJ) dat dit een unit test betreft.

Wanneer je deze annotatie gebruikt, zal IntelliJ je direct vragen om de bijbehorende import toe te voegen. Je zult in IntelliJ ook zien dat je de tests allemaal tegelijk, of individueel kunt draaien met het groene pijltje naast de test!

Afbeelding met tekst, schermopname, software, Besturingssysteem

Automatisch gegenereerde beschrijving

#### Unit tests

Om je uit te kunnen leggen hoe je een unit test schrijft,

de class die verantwoordelijk is voor het berekenen van het Btw-percentage

// BtwCalculator.java

class BtwCalculator {

public double getBTWPercentage(Product product) {

return product.getBTWPercentage();

}

public double getBTWAmount(Product product) {

return product.getPriceExcludingBTW() \* product.getBTWPercentage() / 100.00d;

}

}

De BtwCalculator-class heeft geen properties, maar wel twee methodes

eerste methode haalt het Btw-tarief van het product op en de tweede methode berekent het Btw-bedrag op basis van dat tarief en de productprijs.

1. Wanneer we testen gaan schrijven voor onze code, doen we dit in zo klein mogelijke units.
2. dus logisch zijn om beide methodes individueel te voorzien van een unit test.
3. Bij het schrijven dan de tests ga je vragen stellen over de werking van deze methodes.

"Haalt de methode getBTWPercentage echt het juiste BTW-percentage op?

// BtwCalculatorTest.java

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

class BtwCalculatorTest {

// we maken een instantie van onze BtwCalculator-class,

// zodat we toegang hebben tot de methodes;

private BtwCalculator productCalculator = new BtwCalculator();

@Test

public void testThatBtwCalculatorReturnsBTWPercentage()

{

//arrange:

Product jacket = new Product("Jas", 50.00d, 21.00d);

//act:

double btwPercentage = productCalculator.getBTWPercentage(jacket);

//assert:

assertEquals(21.00d, btwPercentage);

}

}

\*Let op: bovenstaand voorbeeld maakt gebruik van een Product-class die we in dit voorbeeld niet hebben uitgeschreven.

Binnen een testmethode zullen we altijd drie stappen doorlopen om onze test-vraag te beantwoorden:

1. **Arrange**: hier creëren we de context voor onze test. We zetten als het ware alle benodigde informatie klaar. Zo hebben we een werkende Product-instantie nodig om de methodes van onze BtwCalculator mee te kunnen testen.
2. **Act**: hier voeren we de operatie uit. Dat betekent dat we de methode aanroepen of acties uitvoeren. Je kunt het lezen als: "Wanneer we met *deze* Product-instantie de getBTWPercentage-methode aanroepen..."
3. **Assert**: hier maken we een bewering over de verwachte uitkomst. Dit kun je lezen als: "... dan verwacht ik dat de uitkomst 21% is". Dit doen we met behulp van *asserts*. Een assert zegt letterlijk: controleer of een expressie deze verwachtte waarde heeft. Is de uitkomst van de assert *false*? Dan faalt de test. Is de uitkomst van de assert *true*? Dan is de test geslaagd! Je mag zelfs meerdere asserts maken binnen dezelfde test, wanneer je meerdere resultaten wil controleren. Alle asserts moeten dan *true* zijn om de test te laten slagen.

##### Test methode

Laten we nu ook nog een extra testmethode toevoegen om onze andere methode te testen. Hier vragen we ons het volgende af: "Berekent de methode getBTWAmount het juiste Btw-bedrag?":

// BtwCalculatorTest.java

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

class BtwCalculatorTest {

// we maken een instantie van onze BtwCalculator-class,

// zodat we toegang hebben tot de methodes;

private BtwCalculator productCalculator = new BtwCalculator();

@Test

public void testThatBtwCalculatorReturnsBTWPercentage()

{

//arrange:

Product jacket = new Product("Jas", 50.00d, 21.00d);

//act:

double btwPercentage = productCalculator.getBTWPercentage(jacket);

//assert:

assertEquals(21.00d, btwPercentage);

}

@Test

public void testThatBtwCalculatorReturnsBTWAmount()

{

//arrange:

Product jacket = new Product("Jas", 50.00d, 21.00d);

//act:

double btwAmount = productCalculator.getBTWAmount(jacket);

//assert:

assertEquals(10.50d, btwAmount);

}

}

\***Let op:** bovenstaand voorbeeld maakt gebruik van een Product-class die we in dit voorbeeld niet hebben uitgeschreven.

In bovenstaande voorbeeld hebben we telkens de assert assertEquals gebruikt.

assertEquals zijn er nog meer handige asserts, zoals:

* assertTrue: checkt of de conditie *true* is;
* assertFalse: checkt of de conditie *false* is;
* assertNull : checkt of de waarde gelijk is aan *null*;
* assertNotEquals: checkt of de waardes (of *inhoud* van de objecten) niet overeenkomen;
* assertSame: checkt of beide objecten naar hetzelfde origineel wijzen (dus *hetzelfde* zijn). Let op: twee objecten kunnen dezelfde *inhoud* hebben, maar dat maakt ze niet hetzelfde.

##### Mocht je arrays willen vergelijken:

* assertArrayEquals: die checkt of de inhoud van beide arrays overeenkomt.

##### Unit testen bieden veiligheid

Het is belangrijk om altijd te controleren of onze code doet wat we verwachten.

een ander essentieel onderdeel van unit testen is het feit dat wecollega's en onszelf

behoeden voor **unintended feature change**.  
aanpassen van de *return value* op plek *x* iets stuk maakt op plek *y*.

###### first line of defence

De eerste "line of defense" is dat jij bij het maken van deze aanpassing de methode-naam waarschijnlijk veranderd in getPriceIncludingBTW.

Wanneer we in de Product-class deze methodenaam aanpassen, zal de compiler ons waarschuwen dat er in de BtwCalculator-class een methode gebruikt wordt die door jou niet meer bestaat.   
Dit noemen we **compiler safety**.

Als we het hernoemen van de methode niet overal consequent doorvoeren zal de code niet compilen.

###### Second line of defence

"second line of defense". Bij het draaien van de unit tests - vaak iets dat standaard gebeurt wanneer je jouw wijzigingen naar GitHub pusht - faalt onze test:

@Test

public void testThatBtwCalculatorReturnsBTWAmount() {

//arrange:

Product jacket = new Product("Jas", 50.00d, 21.00d);

//act:

double btwAmount = productCalculator.getBTWAmount(jacket);

//assert:

assertEquals(10.50d, btwAmount);

}

Het Btw-bedrag van een jas die €50,- kost komt nu namelijk niet meer overeen met €10,50. Dit zullen we nu snel gaan oplossen voor deze implementatie bij de eindgebruiker komt.

##### Mocking

Unit tests zijn ontzettend krachtig, maar in sommige gevallen niet werkbaar.  
Dit treedt vooral op in situaties waar we gebruik maken van data uit een database, of van gegevens die we ophalen van een externe partij (via een API).

data die we daaruit ophalen kan namelijk op ieder moment veranderen, waardoor onze tests niet betrouwbaar zijn.

In dit soort gevallen maken we gebruik van **mocking.**Mocking wordt gebruikt om specifieke onderdelen van de code te isoleren door er een soort nep-versie van te maken.  
Zo kunnen we de code testen zonder afhankelijk te zijn van de echte databron.

ook de vrijheid om specifieke scenario's of situaties te simuleren, zodat we de functionaliteit kunnen testen in een gecontroleerde omgeving.

Het mocken van classes is echt een vak apart.

Laten we een situatie schetsen waarin we de producten uit onze webshop niet zelf beheren, maar hier een externe service voor gebruiken.

// BtwCalculator.java

class BtwCalculator {

private final ProductService productService;

public BTWCBtwCalculator(ProductService productService) {

this.productService = productService;

}

public double getBTWPercentage(String productCode) {

Product product = productService.getByProductCode(productCode);

return product.getBTWPercentage();

}

public double getBTWAmount(String productCode) {

Product product = productService.getByProductCode(productCode);

return product.getPriceExcludingBTW() \* product.getBTWPercentage() / 100.00d;

}

}

Hoe zou in dit geval onze test eruit moeten zien?

Wanneer het bedrijf stopt met het produceren van producten met de code "xyz", mag het niet zo zijn dat onze tests ineens falen.

Om dit op te lossen zullen we dus gebruik moeten maken van mocking. Dit doen we door een mock-class te maken: die gedraagt zich als een echte class, maar kunnen we vanuit de unit test "configureren". Om dit voor elkaar te krijgen, gebruiken we het framework **Mockito**.

Het is bijna altijd mogelijk om een class te mocken: de enige voorwaarde is dat de mock zich aan dezelfde interface moet houden als de originele class.

Om Mockito te kunnen gebruiken in jouw project, zul je de volgende XML moeten toevoegen aan de pom.xml in de root-map:

...

<dependencies>

...

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>5.0.0</version>

<scope>test</scope>

</dependency>

</dependencies>

Nu kunnen we onze bestaande test-class omschrijven naar testen die gebruik maken van een mock van de Product-service.

We vervangen private ProductService productService; door private ProductService productService = Mockito.mock(ProductService.class);.

// BtwCalculatorTest.java

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

class BtwCalculatorTest {

// we maken een mock van de productService

private ProductService productService = Mockito.mock(ProductService.class);

// we maken een instantie van onze BtwCalculator-class

private BtwCalculator productCalculator = new BtwCalculator(productService);

// testmethodes...

}

Nu hebben we een mock-ProductService die we kunnen controleren. Zo kunnen we bijvoorbeeld zelf een Product-instantie maken die de mock-service moet teruggeven wanneer een product met code "xyz" wordt opgevraagd. Dit doen we volgens de schrijfwijze Mockito.when(<methodcall>).thenReturn(<returnValue>); :

// We maken een jas aan om als product te gebruiken

Product jacket = new Product("Jas", 50.00d, 21.00d);

// Wanneer het product met code "xyz" wordt opgevraagd,

// Geeft onze mock-product-service altijd de jas terug

Mockito.when(productService.getByProductCode("xyz")).thenReturn(jacket)

We hebben onszelf er nu van verzekerd dat wanneer de methodes van de BtwCalculator in onze testclass een product opvragen met code "xyz", dat altijd hetzelfde product wordt teruggegeven. Dit zorgt ervoor dat onze tests de onderliggende functionaliteit kunnen blijven testen, ongeacht of we dit type jas over een paar jaar nog verkopen!

// BtwCalculatorTest.java

import static org.junit.jupiter.api.Assertions.assertEquals;

import org.junit.jupiter.api.Test;

import org.mockito.Mockito;

class BtwCalculatorTest {

// we maken een mock van de productService

private ProductService productService = Mockito.mock(ProductService.class);

// we maken een instantie van onze BtwCalculator-class

private BtwCalculator productCalculator = new BtwCalculator(productService);

@Test

public void testThatBtwCalculatorReturnsBTWPercentage()

{

//arrange:

Product jacket = new Product("Jas", 50.00d, 21.00d);

// Zorgt ervoor dat wanneer het product met code "xyz" wordt opgevraagd,

// Het product dat wij zojuist hebben aangemaakt wordt teruggegeven.

Mockito.when(productService.getByProductCode("xyz")).thenReturn(jacket);

//act:

double btwPercentage = productCalculator.getBTWPercentage("xyz");

//assert:

assertEquals(21.00d, btwPercentage);

}

@Test

public void testThatBtwCalculatorReturnsBTWAmount()

{

//arrange:

Product product = new Product("Jas", 50.00d, 21.00d);

// Zorgt ervoor dat wanneer het product met code "xyz" wordt opgevraagd,

// Het product dat wij zojuist hebben aangemaakt wordt teruggegeven.

Mockito.when(productService.getByProductCode("xyz")).thenReturn(product);

//act:

double btwAmount = subject.getBTWAmount("xyz");

//assert:

assertEquals(10.50d, btwAmount);

}

}