

# HO GENT

H8 Testen

# Table of Contents

1. Doelstellingen .....	1
2. Inleiding .....	1
3. Unit testen .....	4
3.1. UP & TDD .....	4
3.2. Unit testen - FIRST eigenschappen .....	6
4. Unit testen - Structuur .....	7
4.1. Voorbeeld .....	7
4.2. Triple-A patroon .....	8
4.3. Afspraken naamgeving testmethode .....	8
4.4. Timely .....	8
5. Unit testen - Ontwerp technieken .....	9
5.1. Basis test ontwerp technieken .....	9
5.2. Equivalentiepartitionering .....	10
5.3. Grenswaardenanalyse .....	11
5.4. Voorbeeld .....	11
6. Unit testen - static methoden in klasse Assertions .....	14
7. Geparametriseerde testen .....	15
8. @Test methoden - tips .....	18
8.1. Test niet te veel in één @Test methode .....	18
8.2. Test fixture .....	18
8.3. Testen op exceptions .....	19
9. Unit testen - De voordelen op een rijtje .....	19
9.1. Voordelen .....	19
9.2. Nadelen (!) .....	20

# 1. Doelstellingen

- Belang van testen inzien
- Doel, eigenschappen, voor- en nadelen van unit testen kennen
- De 3A regels verstaan
- Testklasse kunnen ontwerpen
- Gebruik kunnen maken van equivalentiepartitionering en grenswaardenanalyse

# 2. Inleiding

- Software doet niet steeds wat het zou moeten doen, het bevat bugs...

## NEWS

[Home](#) | [Video](#) | [World](#) | [UK](#) | [Business](#) | [Tech](#) | [Science](#) | [Magazine](#) | [Entertainment & Arts](#)Technology

# Nest thermostat bug leaves users cold

By Jane Wakefield  
Technology reporter

🕒 14 January 2016 | [Technology](#)

[Share](#)



- De effecten van bugs zijn heel uiteenlopend, van gewoon vervelend, tot bugs die bedrijven en/of klanten heel veel geld kosten, tot bugs die leiden tot verlies van mensenlevens...



- Software ontwikkelaars proberen software te maken die bug-vrij is, een gedegen aanpak van **testen** speelt hierbij een cruciale rol
- Testen is een heel uitgebreide discipline, we beperken ons in dit hoofdstuk tot **unit testen**



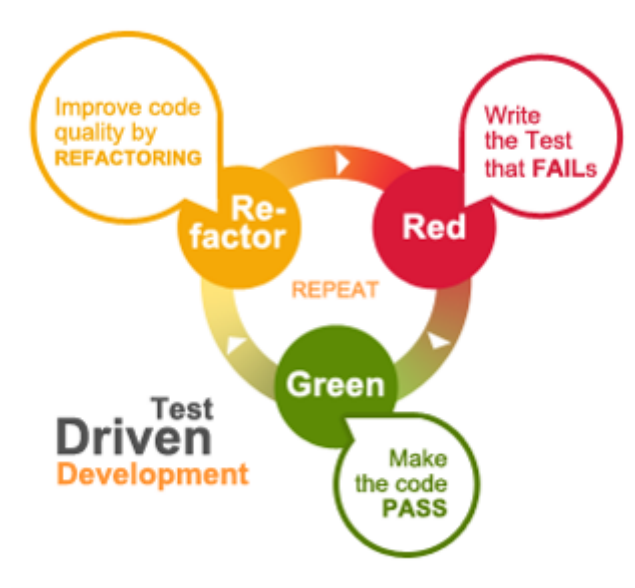
## 3. Unit testen

- Unit testen is een methode om **softwaremodules** of **stukjes broncode** (units) afzonderlijk te testen.
- Bij unittesten zal voor iedere unit **één of meerdere testen** ontwikkeld worden. Hierbij worden dan verschillende **testcases** doorlopen.
- In het ideale geval zijn alle testcases **onafhankelijk** van andere testen.

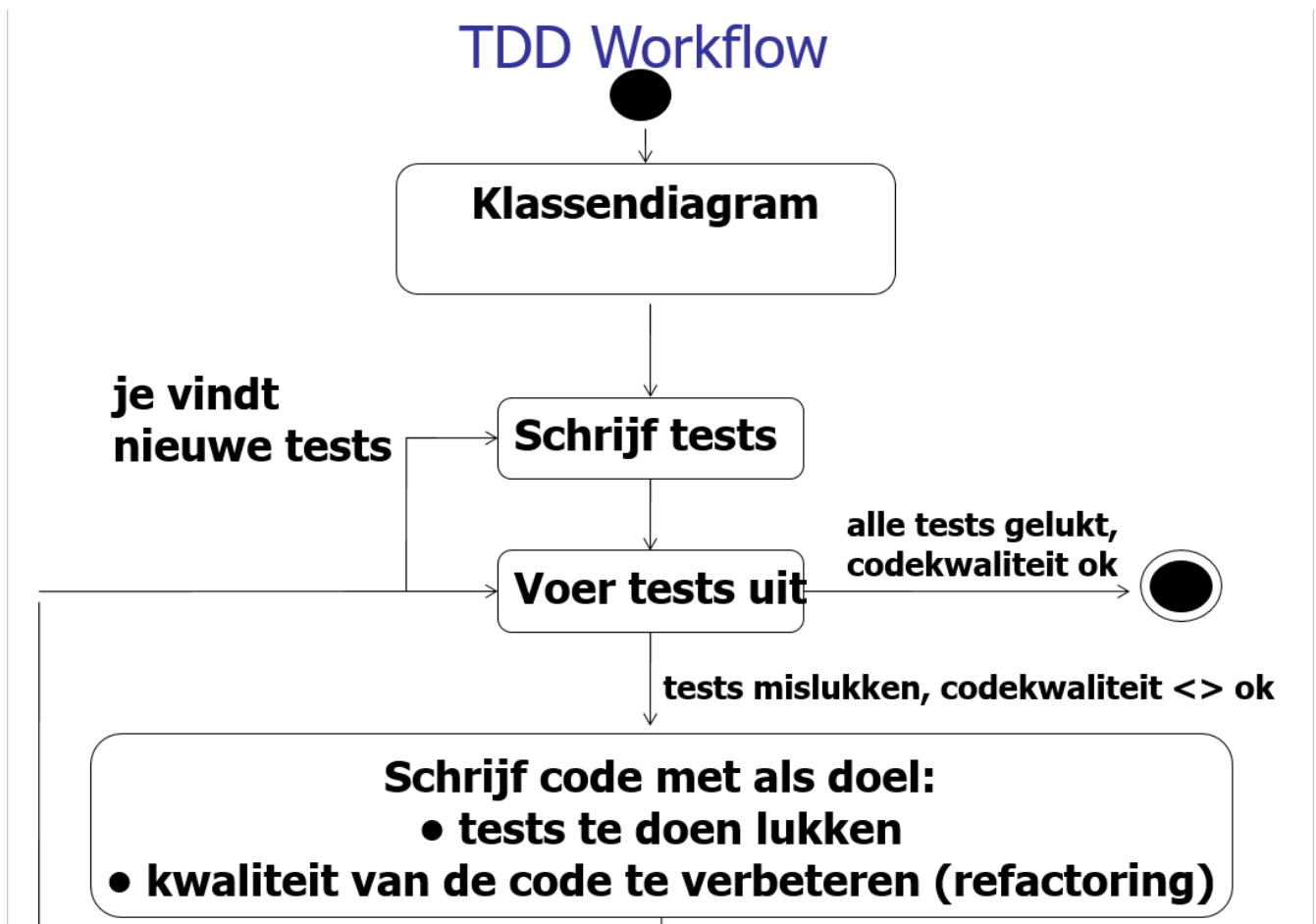
### 3.1. UP & TDD

- Test Driven Development
  - Doorgedreven gebruik van unit testen
  - Het schrijven van unit testen gebeurt vóór het schrijven van de code

- Het uitvoeren van unit testen die falen geven aanleiding tot het schrijven van code



- TDD workflow

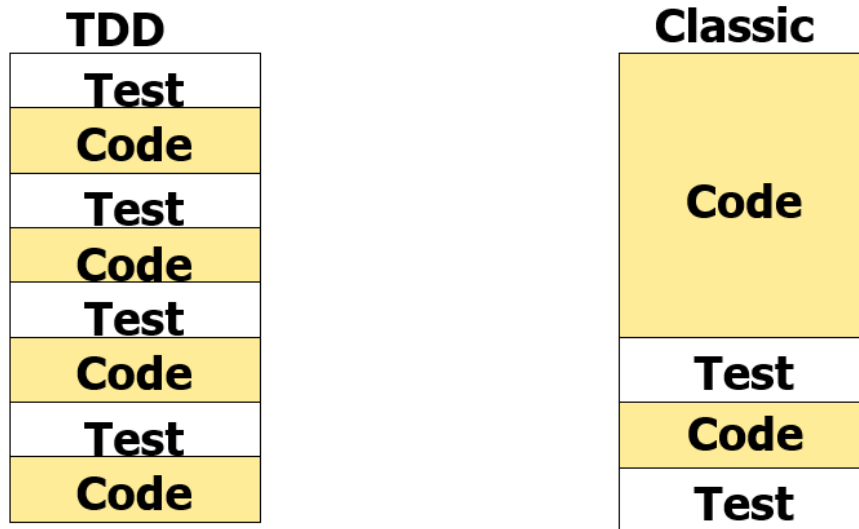


- Test First Programming
  - Je schrijf tests op functionaliteit, vóór je die functionaliteit codeert.
  - Bij het schrijven van de tests denk je na over de te coderen functionaliteit, op basis van de analyse.
  - Code waarover je vooraf nadenkt, is betere code.

- Je voert deze tests uit
  - Alle tests moeten falen, gezien je de functionaliteit nog niet uitwerkte.
- Als je tests schrijft nadat je functionaliteit codeert, heb je de neiging deze tests te baseren op de code van de functionaliteit, niet op de oorspronkelijke analyse.  
Je maakt dan regelmatig dezelfde denkfouten in je test als in je code.

Door eerst de test te schrijven, **kijk je** naar een klasse als een **gebruiker** van een klasse.

- TDD t.o.v. classic development



**Je schrijf een stukje functionaliteit en test dit ...  
Je krijgt continu feedback.**

**Je schrijf eerst veel functionaliteit.  
Daarna test je pas.  
Je krijgt lange tijd geen feedback.**

- JUnit
  - Unit testing framework (open source)
  - Ontworpen door Kent Beck en Erich Gamma in 1997
  - “Standard for unit testing Java applications”
  - JUnit 5

## 3.2. Unit testen - FIRST eigenschappen

- Fast, Isolated, Repeatable, Self validating, Timely
- Fast
  - Unit testen moeten **snel** runnen
    - Een serieus project bevat duizenden testen, en deze worden heel dikwijls uitgevoerd (bv. na aanpassen van klein stukje code, of bij elke build, ...), snelheid is essentieel
- Isolated
  - Unit testen **isoleren bugs** (tight focus)

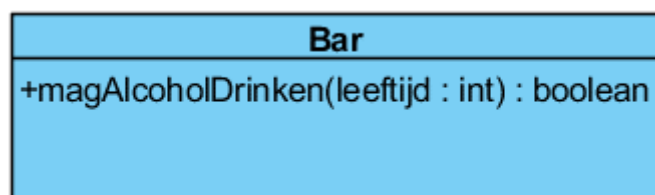


- Met één test, test je een single feature
- Als een test faalt weet je wat er misloopt en waar je moet gaan debuggen
- Unit testen hebben **geen order-of-run** afhankelijkheid
  - Afzonderlijke testen hebben geen invloed op elkaar
  - Een test afzonderlijk runnen, of deze samen met alle andere testen runnen heeft geen invloed op het resultaat van de test
- Repeatable
  - Unit testen kan je **om het even wanneer** runnen, **met zelfde resultaat**
    - Ze hangen niet af van externe services zoals netwerk of database die soms wel/soms niet beschikbaar zijn
    - Opletten met bv.gebruik van 'LocalDate', kan gemakkelijk leiden tot testen die vandaag slagen maar morgen niet...
- Self validating
  - Geen manuele check nodig om te weten of test slaagt of faalt
- Timely
  - De unit testen worden gecodeerd net voor de programma code wordt geschreven die de test zal doen slagen

## 4. Unit testen - Structuur

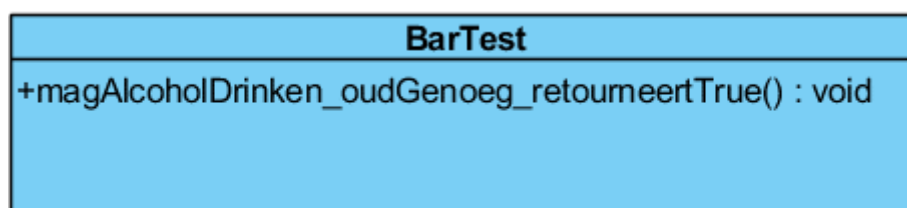
### 4.1. Voorbeeld

In een klasse Bar bestaat een methode magAlcoholDrinken:



Domeinregel: Alcohol drinken mag pas vanaf 16 jaar

Een mogelijke unit test gaat na of de methode *true* retourneert voor iemand die oud genoeg is (bijvoorbeeld 22 jaar)





Voor elke domeinklasse zullen we een testklasse maken die alle unit testen voor methodes uit die domeinklasse bevat

## 4.2. Triple-A patroon

De testmethode uit de klasse BarTest zou als volgt kunnen geïmplementeerd worden:

	<code>@Test</code>	<i>Via deze annotatie wordt deze methode herkend als een unit test</i>
	<code>public void magAlcoholDrinken oudGenoeg retourneertTrue() {</code>	
A	<code>Bar bar = new Bar(); int leeftijd = 22;</code>	<i>Alles in gereedheid brengen voor de test</i>
A	<code>boolean resultaat = bar.magAlcoholDrinken(leeftheid);</code>	<i>De methode die we willen testen aanroepen</i>
A	<code>assertTrue(resultaat);</code>	<i>Beslissen of we het verwachte resultaat krijgen</i>
	<code>}</code>	

We noemen dit het triple-A pattern, **elke unit test volgt dit patroon!**



1. Arrange: het klaarzetten van de test
2. Act: het uitvoeren van de te testen methode
3. Assert: nagaan of de test correct is verlopen

## 4.3. Afspraken naamgeving testmethode

De naam van een testmethode is uiterst belangrijk

- Het geeft een directe indicatie aan de ontwikkelaar indien iets mis loopt
- Namen van testmethodes zijn dan ook dikwijls heel lang

### Afspraken



- Laat de naam van de testmethode beginnen met de naam van de methode die we testen
  - Uitzondering: test je een constructor, dan laat je de naam van de testmethode starten met het prefix `maak` + de naam van de constructor
- Specifieer de waarden waarmee je de methode gaat testen
- Geef een indicatie voor het verwachte resultaat

## 4.4. Timely

Wanneer gaan we nu precies unit testen aanmaken?



Testen worden aangemaakt **na ontwerp** en **vóór het schrijven van de code** van een klasse !

- De ontwikkelaar moet dus vooraf bedenken wat dit stuk code moet doen en wat moet er gebeuren als de code niet (volledig) uitgevoerd kan worden.
- Door duidelijke meldingen en ingebouwde testen kan de goede en foute werking van de unit / class aangetoond worden

## 5. Unit testen - Ontwerp technieken

Tijdens het ontwerpen van unit testen is het belangrijk dat we nadenken over de verschillende waarden waarmee we een methode willen testen



Elke test-case geeft aanleiding tot een testmethode

Voorbeeld:

- Test-case 1: oud genoeg (leeftijd = 22)
- Test-case 2: te jong (leeftijd = 10)
- Test-case 3: net op de grens (leeftijd = 16)

BarTest
+magAlcoholDrinken_oudGenoeg_retourneertTrue() : void
+magAlcoholDrinken_teJong_retourneertFalse() : void
+magAlcoholDrinken_grensGeval16_retourneertTrue() : void



Belangrijk principe: exhaustief testen is onmogelijk!

We kunnen nooit alle mogelijke test cases gebruiken, zelfs niet voor de meest simpele methodes

- Indien we onze simpele methode magAlcoholDrinken zouden willen testen voor elke integer waarde die java kent hebben we nood aan  $2^{32}$  testmethodes...



Via een test techniek gaan we de grote verzameling aan mogelijk test cases vernauwen tot een minimum aan welgekozen test cases.

Indien de code bugs bevat is de kans dat deze naar boven komen via één van deze test cases relatief groot.

### 5.1. Basis test ontwerp technieken

Equivalentiepartitionering en grenswaardenanalyse zijn twee basis test ontwerp technieken die hand in hand gaan:

- in sommige bedrijven eist men dat code op zijn minst volgens deze technieken getest wordt

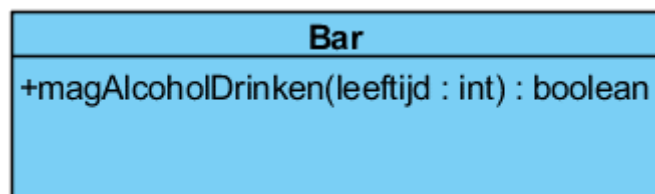
- dit kan in bepaalde gevallen ook een wettelijke eis zijn
- soms eist men dat de waarden links en rechts van een grenswaarde ook opgenomen worden als testgevallen
- partities met waarden waar de methode op een normale manier moet kunnen mee omgaan noemen we **geldige partities**
- partities die aanleiding zullen geven tot het werpen van exceptions noemen we **ongeldige partities**

## 5.2. Equivalentiepartitionering

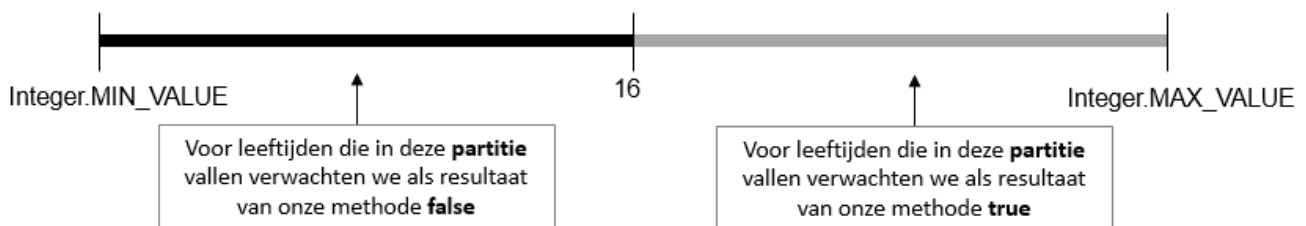
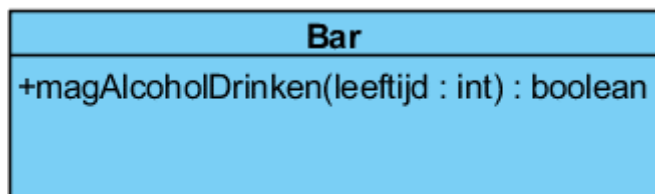


Equivalentiepartitionering is een techniek waarbij je het bereik van waarden gaat opdelen in partities (~delen) waarvoor je eenzelfde resultaat van je methode verwacht

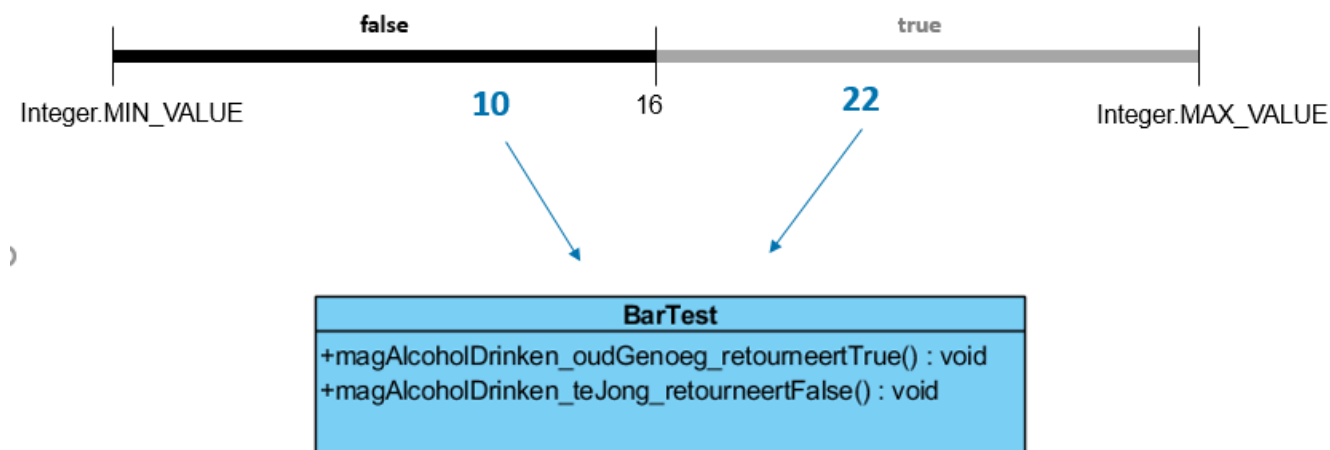
Voorbeeld



Domeinregel: Alcohol drinken mag pas vanaf 16 jaar



We kiezen in elke partitie een willekeurige representant, dit worden onze test cases.



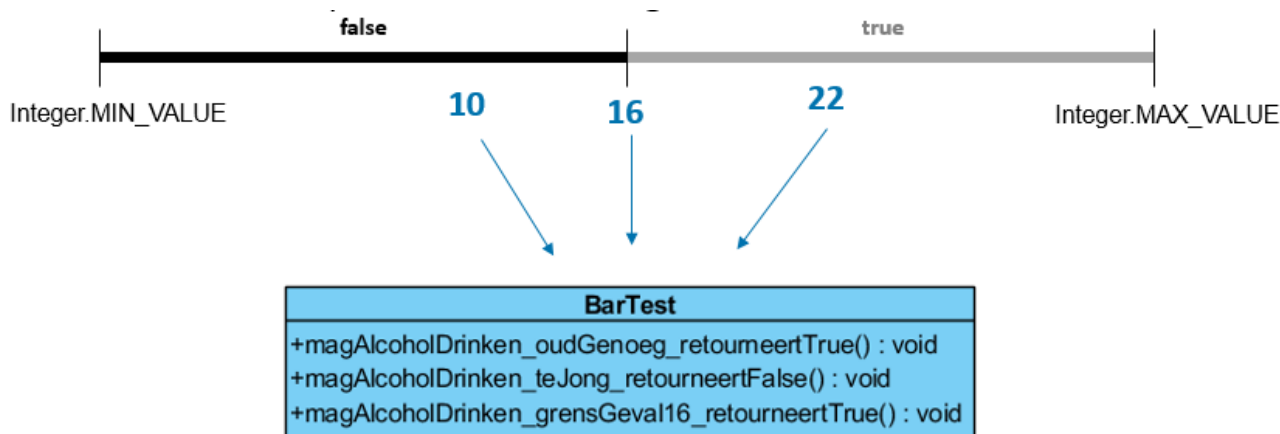
Deze techniek gaat er van uit dat als de methode werkt voor 1 geval uit de partitie, er heel grote kans is dat ze ook zal werken voor andere waarden uit dezelfde partitie.

Indien `magAlcoholDrinken` een correct resultaat oplevert voor 10, dan hoogstwaarschijnlijk ook voor 12, 11, 9, 8, 7, 6, enzoverder, enzoverder, ...

## 5.3. Grenswaardenanalyse



Waarden die op de grens van een partitie liggen zijn ook steeds de moeite om op te nemen als testgeval!



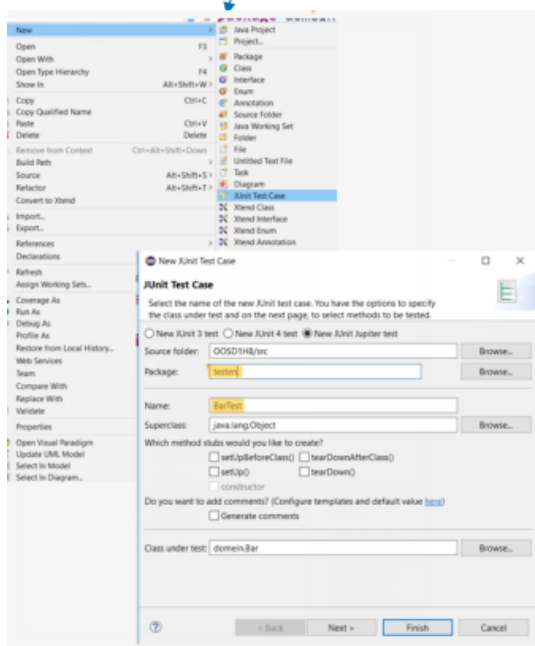
Deze techniek gaat er van uit dat heel veel programmeerfouten gebeuren door het slecht interpreteren van grenswaarden...



Een grens heeft vaak twee zijden, je kan van beide zijden een test case maken.

We zouden ook nog de andere grenzen kunnen opnemen (MIN\_VALUE en MAX\_VALUE) maar dit is in dit voorbeeld weinig relevant, wees in de eerste plaats aandachtig voor grenswaarden die partities scheiden, zoals onze waarde 16...

## 5.4. Voorbeeld



```

BarTest

+magAlcoholDrinken_oudGenoeg_retourneertTrue() : void
+magAlcoholDrinken_teJong_retourneertFalse() : void
+magAlcoholDrinken_grensGeval16_retourneertTrue() : void

```

```

public class BarTest {

    @Test
    public void magAlcoholDrinken_oudGenoeg_retourneertTrue() {
        Bar bar = new Bar();
        int leeftijd = 22;
        boolean resultaat = bar.magAlcoholDrinken(leeftijd);
        assertTrue(resultaat);
    }

    @Test
    public void magAlcoholDrinken_teJong_retourneertFalse() {
        Bar bar = new Bar();
        int leeftijd = 10;
        boolean resultaat = bar.magAlcoholDrinken(leeftijd);
        assertFalse(resultaat);
    }

    @Test
    public void magAlcoholDrinken_grensGeval16_retourneertTrue() {
        Bar bar = new Bar();
        int leeftijd = 16;
        boolean resultaat = bar.magAlcoholDrinken(leeftijd);
        assertTrue(resultaat);
    }
}

```

- Stap1

```

BarTest

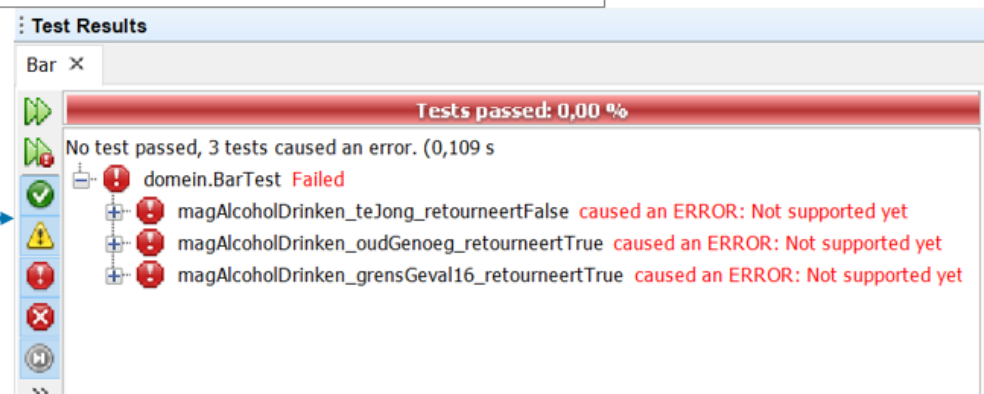
+magAlcoholDrinken_Leeftijd22_RetourneertTrue() : void
+magAlcoholDrinken_Leeftijd10_RetourneertFalse() : void
+magAlcoholDrinken_Leeftijd16_RetourneertTrue() : void

```

```

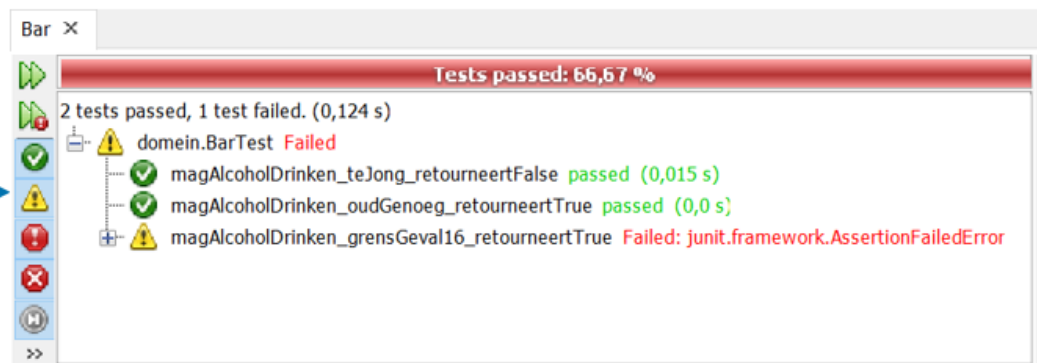
public class Bar {
    public boolean magAlcoholDrinken(int leeftijd){
        throw new UnsupportedOperationException();
    }
}

```



- Stap 2

```
public class Bar {
    public boolean magAlcoholDrinken(int leeftijd){
        return leeftijd > 16;
    }
}
```



Bar X

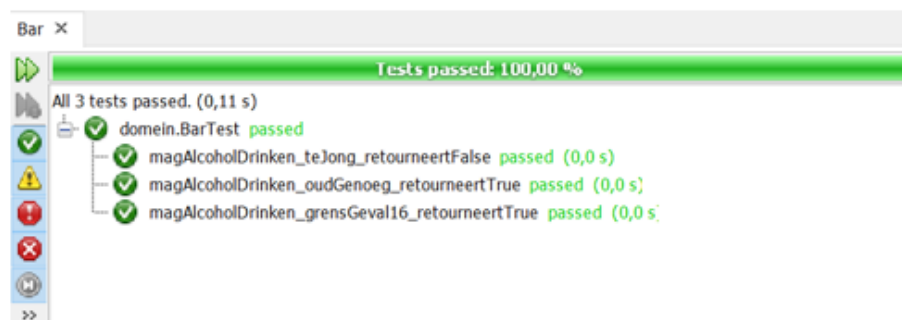
Tests passed: 66,67 %

2 tests passed, 1 test failed. (0,124 s)

- domain.BarTest Failed
  - magAlcoholDrinken\_teJong\_retourneertFalse passed (0,015 s)
  - magAlcoholDrinken\_oudGenoeg\_retourneertTrue passed (0,0 s)
  - magAlcoholDrinken\_grensGeval16\_retourneertTrue Failed: junit.framework.AssertionFailedError

- Stap 3

```
public class Bar {
    public boolean magAlcoholDrinken(int leeftijd){
        return leeftijd >= 16;
    }
}
```



Bar X

Tests passed: 100,00 %

All 3 tests passed. (0,11 s)

- domain.BarTest passed
  - magAlcoholDrinken\_teJong\_retourneertFalse passed (0,0 s)
  - magAlcoholDrinken\_oudGenoeg\_retourneertTrue passed (0,0 s)
  - magAlcoholDrinken\_grensGeval16\_retourneertTrue passed (0,0 s)



Keep the bar green to keep  
your code clean



Code in @BeforeEach methode wordt uitgevoerd vóór elke unit test

```

public class BarTest {
    @Test
    public void magAlcoholDrinken_oudGenoeg_retourneertTrue() {
        Bar bar = new Bar();
        int leeftijd = 22;
        boolean resultaat = bar.magAlcoholDrinken(leeftijd);
        assertTrue(resultaat);
    }

    @Test
    public void magAlcoholDrinken_teJong_retourneertFalse() {
        Bar bar = new Bar();
        int leeftijd = 10;
        boolean resultaat = bar.magAlcoholDrinken(leeftijd);
        assertFalse(resultaat);
    }

    @Test
    public void magAlcoholDrinken_grensGevall6_retourneertTrue() {
        Bar bar = new Bar();
        int leeftijd = 16;
        boolean resultaat = bar.magAlcoholDrinken(leeftijd);
        assertTrue(resultaat);
    }
}

```

```

public class BarTest {
    private Bar bar;

    @BeforeEach
    public void setUp() {
        bar = new Bar();
    }

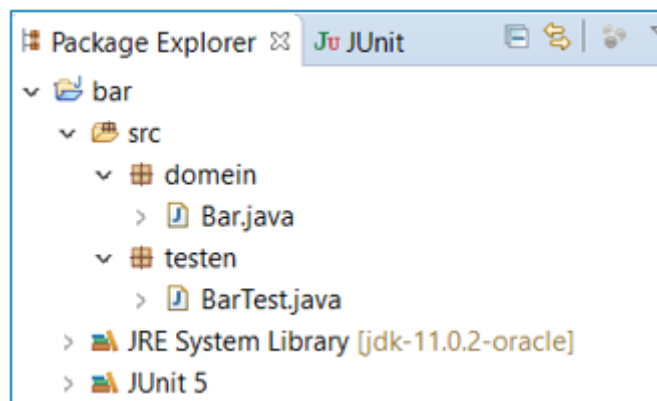
    @Test
    public void magAlcoholDrinken_oudGenoeg_retourneertTrue() {
        int leeftijd = 22;
        boolean resultaat = bar.magAlcoholDrinken(leeftijd);
        assertTrue(resultaat);
    }

    @Test
    public void magAlcoholDrinken_teJong_retourneertFalse() {
        int leeftijd = 10;
        boolean resultaat = bar.magAlcoholDrinken(leeftijd);
        assertFalse(resultaat);
    }

    @Test
    public void magAlcoholDrinken_grensGevall6_retourneertTrue() {
        int leeftijd = 16;
        boolean resultaat = bar.magAlcoholDrinken(leeftijd);
        assertTrue(resultaat);
    }
}

```

- In een Eclipse project vinden we alle testklassen terug in de package 'testen'



## 6. Unit testen - static methoden in klasse Assertions



De bibliotheekklasse Assertions biedt een collectie van methoden aan waarmee we een bepaalde conditie kunnen testen.

Indien niet voldaan is aan de conditie zal de huidige test falen.

### assertEquals(expected, actual)

- Test of expected gelijk is aan actual.
- expected en actual kunnen als type hebben:
  - Alle primitieve datatypes
  - Object (en derived): Dan voert assertEquals expected.equals(actual) uit.

### assertEquals(expected, actual, String message)

- Test of expected gelijk is aan actual



- Toont message als dit niet zo is.

#### **assertEquals(expected, actual, delta)**

- Test of expected actual benadert (delta = maximaal verschil)
- expected, actual en delta zijn van het type double of float.

#### **assertFalse(boolean conditie)**

- Test of conditie gelijk is aan false.

#### **assertFalse(boolean conditie, String message)**

- Test of conditie gelijk is aan false. Toont message als dit niet zo is.

#### **assertTrue(boolean conditie)**

- Test of conditie gelijk is aan true.

#### **assertTrue(boolean conditie, String message)**

- Test of conditie gelijk is aan true. Toont message als dit niet zo is.

#### **assertNull(Object object)**

- Test of object gelijk is aan null.

#### **assertNull(Object object, String message)**

- Test of object gelijk is aan null. Toont message als dit niet zo is.

#### **assertNotNull(Object object)**

- Test of object verschillend is van null.

#### **assertNotNull(Object object, String message)**

- Test of object verschillend is van null. Toont message als dit niet zo is.

## 7. Geparametriseerde testen

Om het aantal uit te schrijven testmethoden te reduceren en hergebruik van code te stimuleren kunnen we gebruik maken van geparametriseerde testen.



Geparametriseerde testen laten toe een test meerdere keren uit te voeren met verschillende argumenten.



Een geparametriseerde testmethode duiden we aan met de annotatie **@ParameterizedTest**. Ook dienen test argumenten (argumenten waarmee de testmethode zal aangeroepen worden) voorzien te worden via een annotatie.

De test argumenten worden voorzien door een **source**.

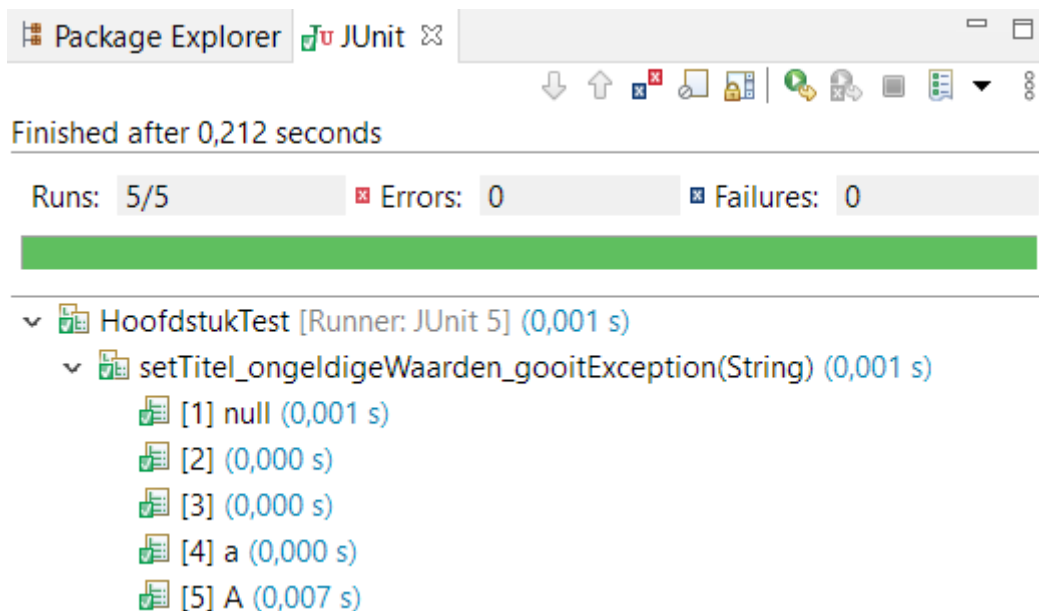
Voorbeeld:

```
1 public class Hoofdstuk
2 {
3     private String titel;
4
5     public void setTitel(String titel)
6     {
7         if (titel == null || titel.trim().isEmpty())
8             throw new IllegalArgumentException("woord moet ingevuld zijn");
9         if (titel.length() < 2)
10            throw new IllegalArgumentException("woord is te klein");
11         this.titel = titel;
12     }
13 }
```

We voorzien enkele testgevallen: null, "", " ", "a", "A"

```
1 public class HoofdstukTest
2 {
3     private Hoofdstuk hoofdstuk;
4
5     @BeforeEach
6     public void before() {
7         hoofdstuk = new Hoofdstuk();
8     }
9
10    @ParameterizedTest // ①
11    @NullAndEmptySource // ②
12    @ValueSource(strings = { " ", "a", "A" }) // ③
13    public void setTitel_ongeldigeWaarden_gooitException(String titel)
14    {
15        Assertions.assertThrows(IllegalArgumentException.class,
16                                () -> { hoofdstuk.setTitel(titel); } ); ④
17    }
18 }
```

Uitvoer:



Welke testen worden uitgevoerd?

- ① **@ParameterizedTest**: aanduiding dat de test meerdere keren zal worden uitgevoerd met verschillende argumenten.
- ② **@NullAndEmptySource**: zal de test uitvoeren met de argumenten **null** en een **lege string**

```
1 setTitel_ongeldigeWaarden_gooitException(null);
```

```
1 setTitel_ongeldigeWaarden_gooitException("");
```

- ③ **@ValueSource**: laat je toe een array van waarden te specificeren die één per één als argument worden doorgegeven. In dit geval al de testmethode drie maal aangeroepen worden met volgende argumenten:

```
1 setTitel_ongeldigeWaarden_gooitException(" ");
```

```
1 setTitel_ongeldigeWaarden_gooitException("a");
```

```
1 setTitel_ongeldigeWaarden_gooitException("A");
```

- ④ De te testen methode, `hoofdstuk.setTitel(titel)`, die herhaaldelijk wordt getest met de verschillende argumenten (= doorgegeven waarden via annotaties 2 en 3)



Deze geparametriseerde test zal uiteindelijk 5 verschillende testcases uitvoeren.

## 8. @Test methoden - tips

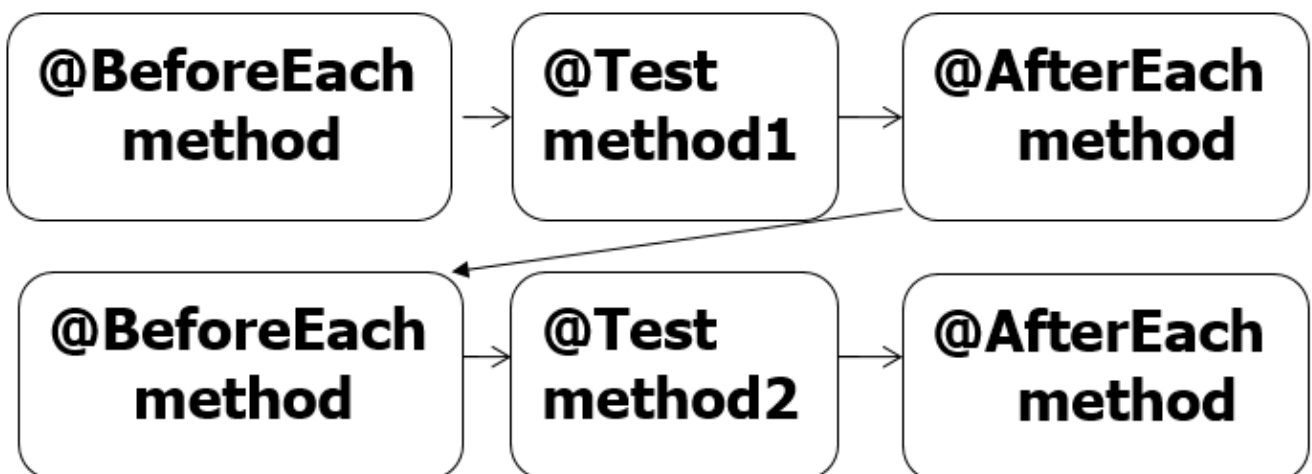
### 8.1. Test niet te veel in één @Test methode

Schrijf niet te veel assertions in één @Test methode

- Nadelen van veel testen in één @Test methode:
  - Als een assertions methode binnen een @Test methode faalt, voert JUnit de rest van die @Test methode niet uit
  - De methode wordt moeilijk te lezen
  - De kans dat je in de methode een bug schrijft is groot
  - De kans dat je de methode moet debuggen is groot
  - De werking van één @Test methode mag **niet afhangen** van de werking van een andere @Test methode
  - Je moet @Test methoden **in willekeurige volgorde** kunnen uitvoeren (JUnit garandeert geen volgorde van het uitvoeren van de @Test methods)

### 8.2. Test fixture

- Test fixture:  
object of resource die je nodig hebt in meerdere @Test methods
- Declareer de variabele om naar de test fixture te verwijzen als private in de Test Case
- Initialiseer de test fixture in een methode voorzien van **@BeforeEach**
- JUnit voert de **@BeforeEach** methode uit vóór iedere **@Test** methode
- Kuis de resource op in een methode voorzien van **@AfterEach**
- JUnit voert deze methode uit na iedere **@Test** methode



```

import org.junit.jupiter.api.BeforeEach;
...
public class ZichtrekeningTest {
    private Zichtrekening rekening;

    @BeforeEach
    public void before() { // de naam van de methode mag je zelf kiezen
        rekening = new Zichtrekening();
    }

    @Test
    public void stortenMoetSaldoAanpassen() {
        BigDecimal bedrag = new BigDecimal(200);
        rekening.storten(bedrag);
        Assertions.assertEquals(bedrag, rekening.getSaldo());
    }
}

```

34

## 8.3. Testen op exceptions

- Als je verkeerde parameterwaarden meegeeft aan de methoden van de te testen klasse, moeten deze methoden Exceptions werpen.
- Assertions.assertThrows
  - Assertions.assertThrows(IllegalArgumentException.class, () → {...})
- Als de @Test methode geen exception werpt van deze exception klasse, krijg je een JUnit failure.

# 9. Unit testen - De voordelen op een rijtje

## 9.1. Voordelen

- Unit testen schrijven dwingt een developer om de **specificaties duidelijk en ondubbelzinnig** te begrijpen
  - door testen te ontwerpen kom je dikwijls tot onduidelijkheden in de specificaties die zo tijdig kunnen opgeklaard worden
- Unit testen geven **feedback** over het design
  - als je, op basis van het UML design van een klasse, moeilijk een test kan schrijven, zit waarschijnlijk het design verkeerd
- Unit testen beschrijven de functionaliteit van een stukje code, ze zijn een vorm van **documentatie**
- Unit testen zijn **geautomatiseerd**
  - niet alle vormen van testen kunnen gemakkelijk geautomatiseerd worden

- Unit testen leiden tot **kwaliteitsvollere code**
  - herbruikbaar, aanpasbaar, leesbaar,
  - developers krijgen zo meer vertrouwen
  - developers spenderen minder tijd aan debuggen
- Unit testen laten toe dat bugs **vroeg** worden gedetecteerd en gecorrigeerd

## 9.2. Nadelen (?!)

- Weerstand bij de ontwikkelaars, mindset 'kost tijd'
  - Het schrijven van unit testen kost inderdaad tijd
  - De tijd die je investeert in unit testen win je dubbel en dik terug omdat je op een veel efficiëntere manier tot code komt die van veel hogere kwaliteit is
- Druk van hogerhand
  - Onder tijdsdruk worden testen dikwijls als wisselgeld gezien
    - time is money
    - “dit project moet tegen volgende week af en we zitten echt te krap in tijd, laat ons nu even code schrijven zonder te unit testen...”