HO
GENT

# Unit testing (Intro)

Ludwig Stroobant

# Table of Contents

# 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…

BBC Sign in News Sport Weather Shop Earth Travel M

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


A software update in December appears to be causing problems for the 'smart' thermostat

- 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.

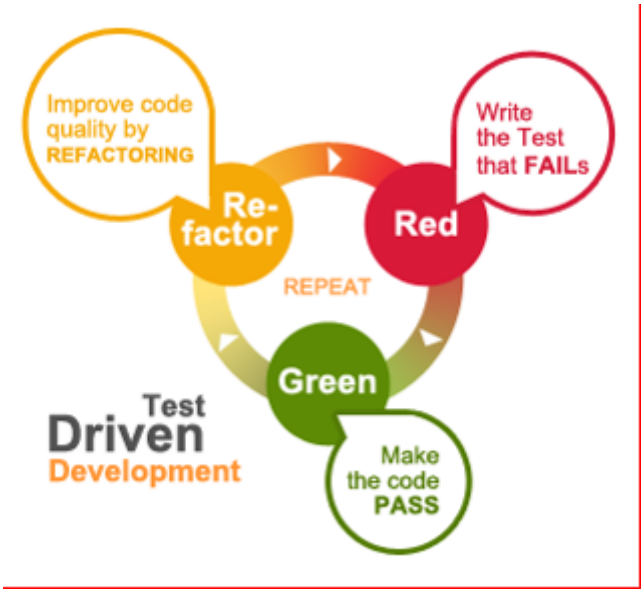
In het ideale geval zijn alle testcases **onafhankelijk** van elkaar.

## 3.1. Test Driven Development (TDD)

Bij Test Drive Development streeft men ernaar om unit testcases te schrijven vóór het schrijven van de eigenlijke code.

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

De TDD workflow ziet er als volgt uit:




- Test First Programming
  - Op basis van analyse en ontwerp schrijf je tests op functionaliteit, vóór je die functionaliteit codeert.
  - Bij het schrijven van de tests denk je na over de te coderen functionaliteit en alle zaken die getest dienen te worden.

Code waarover je vooraf nadenkt, is betere code.

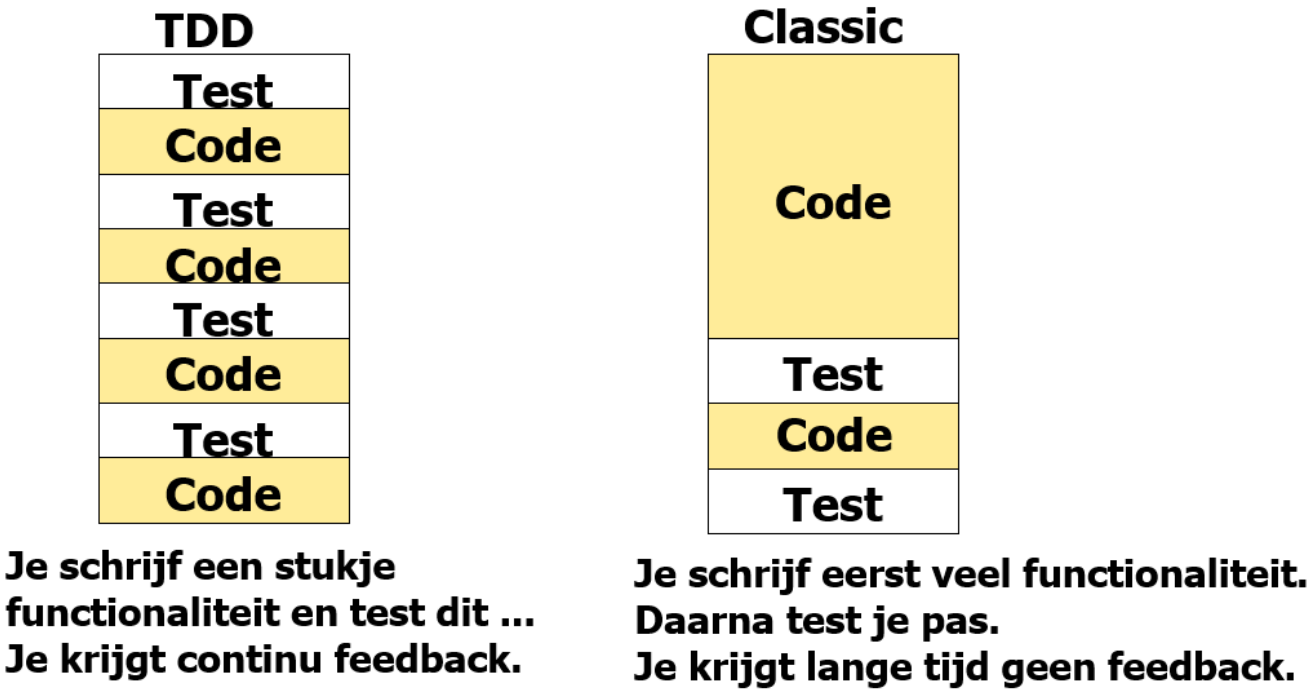
- Je voert deze tests uit
  - Initieel moeten alle tests falen, gezien je de functionaliteit nog niet uitwerkte.

Als je testen schrijft na het implementeren van je functionaliteit heb je de neiging deze tests te

baseren op de code van de functionaliteit, niet op de oorspronkelijke analyse. Risico hierbij is dat je dezelfde denkfouten maakt in je test als in je code: je schrijft dus een test voor mogelijks foutieve code.

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

### 3.1.1. TDD t.o.v. classic development




## 3.2. Unit testen - FIRST eigenschappen

FIRST = **F**ast, **I**solated, **R**epeatable, **S**elf validating, **T**imely

**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.

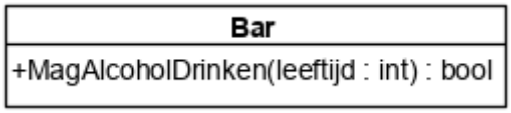
**Timely**

De unit testen worden gecodeerd net voor de programma code wordt geschreven die de test zal doen slagen

# 4. Unit testen - Structuur

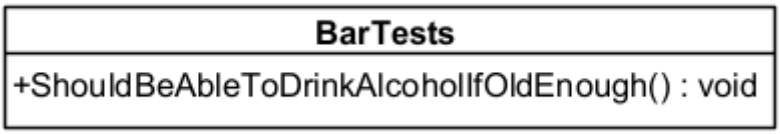
## 4.1. Voorbeeld

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 BarTests zou als volgt kunnen geïmplementeerd worden:

```
1    public class BarTests
2    {
3        [Fact]
```

```
4         public void ShouldReturnTrueIfOudGenoeg()
5         {
6             // Arrange: alles in gereedheid brengen voor de test ①
7             Bar bar = new Bar();
8             int leeftijd = 22;
9
10            // Act: De methode die we willen testen aanroepen ②
11            bool resultaat = bar.MagAlcoholDrinken(leeftijd);
12
13            // Assert: Beslissen of we het verwachte resultaat krijgen ③
14            Assert.True(resultaat);
15        }
16    }
```

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

① **A**rrange: het klaarzetten van de test

② **A**ct: het uitvoeren van de te testen methode

③ **A**ssert: 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.

LET OP: In deze intro schrijven we elke test apart uit, wat leidt tot veel testmethodes. Er zijn technieken om testen samen te nemen waar we in het werkcollege verder op ingaan.

**Afspraken voor testen met xUnit!**

- De naam van het testproject is dezelfde als de naam van het te testen project met het suffix ".Tests"
- De naam van de testklasse is dezelfde als de naam van de te testen klasse, met het suffix "Tests". Voor elke klasse schrijf je een aparte testklasse.
- Laat de naam van de testmethode beginnen "Should" of "Test" en beschrijf wat er getest wordt.
- 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/klasse aangetoond worden

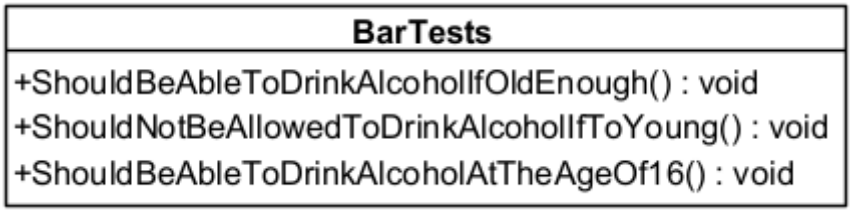
# 5. Unit testen - Ontwerp technieken - Wat te testen?

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)




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 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
    - soms eist men dat de waarden links en rechts van een grenswaarde ook opgenomen worden als testgevallen

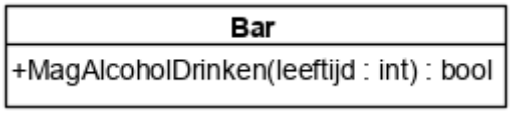
Partities met waarden waar de methode op een normale manier moet kunnen mee omgaan noemen we **geldige partities**

Partities die aanleiding zullen geven tot het werpen van exceptions noemen we **ongeldige partities**

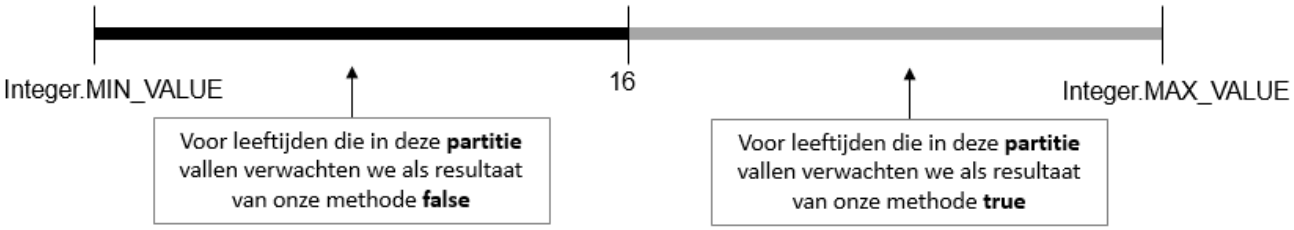
# 5.2. Equivalentiepartitionering

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

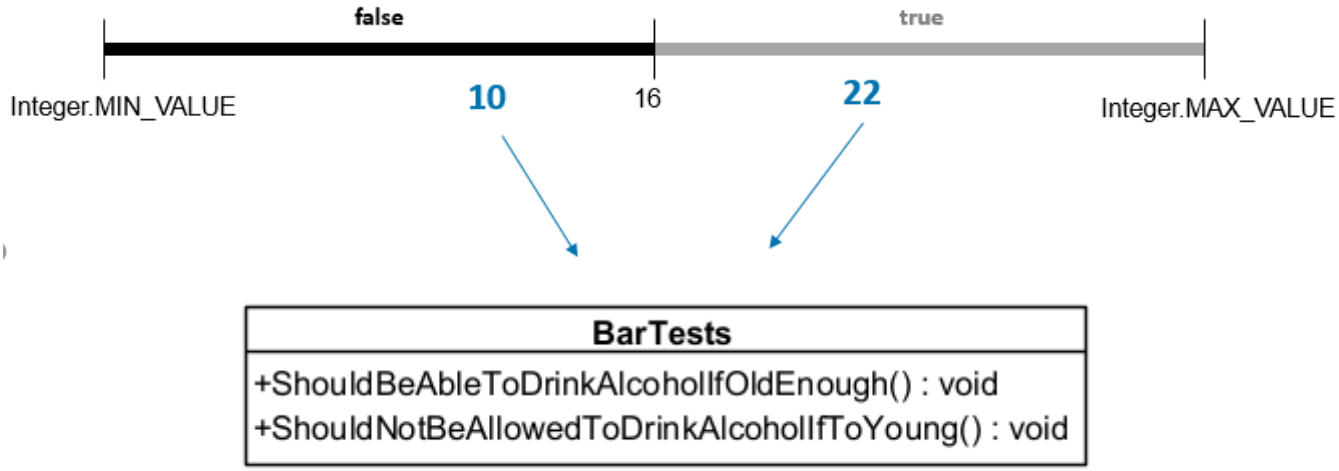
Voorbeeld




Domeinregel: Alcohol drinken mag pas vanaf 16 jaar




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



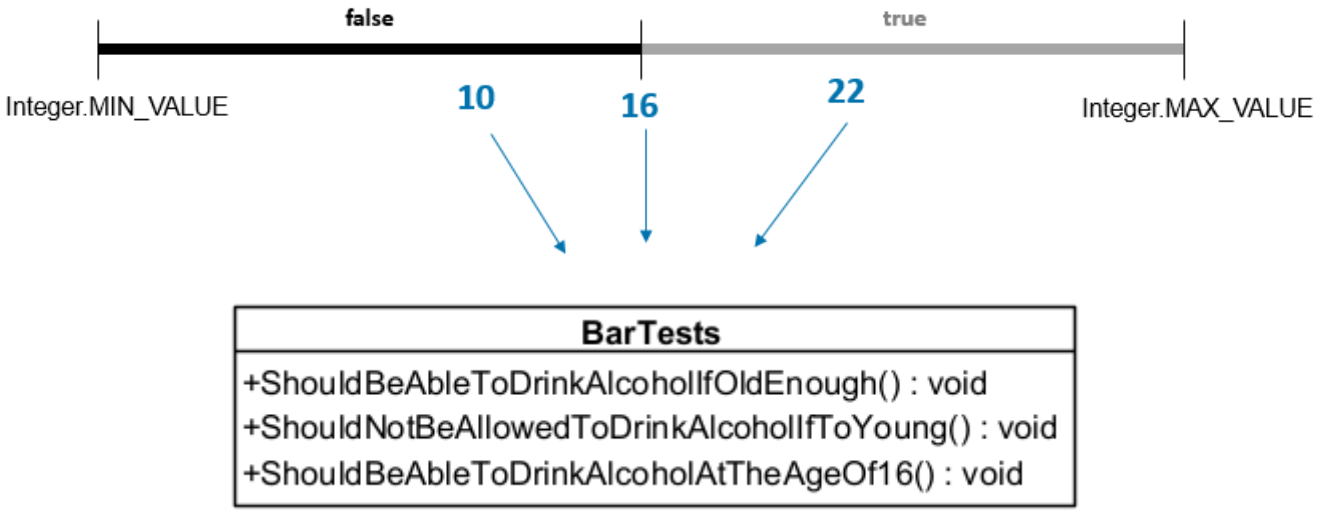

Deze techniek gaat er van uit dat als de methode werkt voor 1 geval uit de partitie, er heel grote

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. Uitwerking methode MagAlcoholDrinken

1. Voer de testen uit, deze zullen rood kleuren.
2. Implementeer nu per test de code, zodat test per test groen kleurt.
3. Op deze manier ga je verder tot alle testen groen kleuren.

# 6. Unit testen - static methoden in klasse Assert

De bibliotheekklasse Assert 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.

> De verschillende Assert methodes worden in het werkcollege toegelicht!