



**Vlaamse Dienst voor Arbeidsbemiddeling en  
Beroepsopleiding**

# **C# TEST DRIVEN DEVELOPMENT**

Deze cursus is eigendom van de VDAB©

---

## Inhoudsopgave

<b>1</b>	<b>INLEIDING</b>	<b>1</b>
1.1	Doelstelling	1
1.2	Vereiste voorkennis	1
1.3	Nodige software	1
1.4	De noodzaak van testen	1
1.5	Geautomatiseerd testen	1
1.6	Unit tests en integration tests	1
1.7	Plaats van de unit tests binnen de solution	2
<b>2</b>	<b>KENNISMAKING MET TDD</b>	<b>3</b>
2.1	Algemeen	3
2.2	Het voorbeeldproject	3
2.3	De te testen class	3
2.4	De unit test	4
2.5	De unit test uitvoeren	6
2.6	Één testmethod uitvoeren	7
2.7	De unit test als documentatie	7
2.8	De methods van de class Assert	8
<b>3</b>	<b>EERST TESTS SCHRIJVEN, DAARNA IMPLEMENTATIE</b>	<b>9</b>
3.1	Algemeen	9
3.2	De te testen class	10
3.3	Voorbeeld	10
3.3.1	De te testen class	10
3.3.2	De unit test	11
3.3.3	Een eerste implementatie	12
3.3.4	Refactoring	13
3.3.5	Samenvatting van de stappen bij test driven development	13
<b>4</b>	<b>TEST FIXTURES</b>	<b>14</b>
4.1	Algemeen	14
4.2	[TestInitialize]	14
<b>5</b>	<b>TO TEST OR NOT TO TEST</b>	<b>16</b>

---

<b>5.1</b>	<b>Exceptions testen</b>	<b>16</b>
<b>5.2</b>	<b>Grenswaarden en extreme waarden testen</b>	<b>17</b>
5.2.1	Algemeen	17
5.2.2	Eerste voorbeeld	17
5.2.3	De class	18
5.2.4	De unit test	18
5.2.5	Een eerste implementatie	19
5.2.6	Refactoring	20
5.2.7	Voorbeeld met een verzameling	20
5.2.8	De unit test	21
5.2.9	Een eerste implementatie	21
5.2.10	Refactoring	22
<b>5.3</b>	<b>Testen bijhouden</b>	<b>22</b>
<b>6</b>	<b>MEERDERE UNIT TESTS UITVOEREN</b>	<b>23</b>
6.1	Algemeen	23
<b>7</b>	<b>DEPENDENCIES – STUBS</b>	<b>24</b>
7.1	Dependency	24
7.1.1	Algemeen	24
7.1.2	Voorbeeld	25
7.1.3	Problemen bij het testen van een technische class met dependencies	26
7.1.4	De dependency uitdrukken in een interface	26
7.1.5	Dependency injection	27
7.1.6	Stub	27
<b>8</b>	<b>MOCK</b>	<b>30</b>
8.1	Algemeen	30
8.1.1	Resultaat van method oproep	30
8.1.2	Verificaties	30
8.2	Moq	30
8.2.1	Algemeen	30
8.2.2	De Moq library toevoegen	30
8.2.3	Een mock aanmaken	30
8.2.4	De mock trainen	31
8.2.5	Verificaties	32
<b>9</b>	<b>COLOFON</b>	<b>33</b>

# 1 INLEIDING

## 1.1 Doelstelling

Code testen met de hulp van Visual Studio.

Code ontwikkelen op de manier van Test driven development.

## 1.2 Vereiste voorkennis

- C# PF

## 1.3 Nodige software

- Visual Studio 2012

## 1.4 De noodzaak van testen

Uit een studie van het National Institute of Standards and Technology blijkt dat softwarefouten jaarlijks 59,5 miljard (59.500.000.000) dollar kosten aan de economie van de Verenigde Staten.

Uit de studie blijkt ook dat men een derde van deze kosten kan vermijden door de manier van testen te verbeteren. Ook blijkt dat hoe vroeger men een fout ontdekt in de ontwikkeltijd van een applicatie, hoe minder deze fout kost.

Testen is dus een belangrijk onderdeel van softwareontwikkeling.

Test driven development (TDD) is een manier van software ontwikkelen waarbij

- testen een centraal onderdeel van de ontwikkeling zijn
- testen vroeg in de ontwikkeltijd worden toegepast

## 1.5 Geautomatiseerd testen

Wanneer je nieuwe code schrijft moet je de werking van die code testen.

Als je de code wijzigt of corrigeert, moet je de werking van de code terug testen.

Testen is dus een repetitieve taak.

Een ontwikkelaar automatiseert elke repetitieve taak, door deze taak als een programma te schrijven. Bij TDD doet de ontwikkelaar dit ook voor een test. Hij doet de test niet met manuele handelingen, maar met een testprogramma: code dat de goede werking van een andere code test.

Visual Studio helpt om zo'n testprogramma te schrijven.

## 1.6 Unit tests en integration tests

Een class waarvan je de code wil testen heet een *unit*.

Een class die de tests bevat voor één te testen class heet een *unit test*.

Een synoniem voor een *unit test* is een *test case*.

Terwijl je een class ontwikkelt, kan je al de unit test van die class schrijven en uitvoeren. Je krijgt zo snel feedback over fouten in de class.

Als de class volledig geschreven is, is ze ook volledig getest.

De kans is minimaal dat ze nog fouten bevat. Je zal zo minder moeten debuggen.

Dit is zeer voordelig: debuggen is een tijdrovende, vervelende taak.

Een integration test is een class waarin je de samenwerking van meerdere classes test.

In het onderdeel “Klant toevoegen” van een website werken deze classes samen:

- Een web form `KlantToevoegen.aspx`
- Een entity class `Klant`
- Een class `KlantenManager` die de code bevat voor databasetoegang

Een ontwikkelaar gebruikt zowel unit tests als integration tests

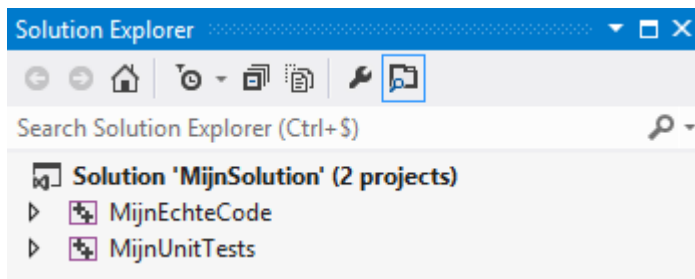
- Hij gebruikt tijdens het ontwikkelen van een class een unit test om fouten te corrigeren in die ene class.
- Als alle classes van een programma onderdeel af zijn, gebruikt hij een integration test om de samenwerking van die classes te testen.

Je leert in deze cursus unit tests kennen.

## 1.7 Plaats van de unit tests binnen de solution

De unit tests komen in dezelfde solution als de te testen code, maar de unit tests bevinden zich in een apart project binnen die solution.

Als je programma af is, hoeft je het project met de unit tests niet mee te leveren aan de klant.



## 2 KENNISMAKING MET TDD

### 2.1 Algemeen

Je leert in dit hoofdstuk een unit test schrijven.

### 2.2 Het voorbeeldproject

Je maakt in Visual Studio een Class Library Project

- Je kiest FILE, New, Project
- Je kiest links Visual C#, Windows
- Je kiest rechts Class Library
- Je tikt TDDCursusLibrary bij Name
- Je tikt TDDCursusSolution bij Solution Name en je kiest OK
- Je verwijdert Class1.cs

### 2.3 De te testen class

De voorbeeld te testen class is de class Jaar.

Je geeft aan de constructor een jaartal mee.

De method isSchrikkeljaar geeft true terug als dit jaar een schrikkeljaar is.

Anders geeft de method false terug.

Jaar
+Jaar(jaar: int) +IsSchrikkeljaar(): bool

Deze method is gebaseerd op de regels van een schrikkeljaar

- Als een jaar deelbaar is door 4 is het een schrikkeljaar.
- Als een jaar deelbaar is door 100 is het echter geen schrikkeljaar.
- Als een jaar deelbaar is door 400 is het echter wel een schrikkeljaar.

Je voegt deze class toe aan het project

- Je kiest PROJECT, Add Class
- Je tikt Jaar bij Name en je kiest Add

Je wijzigt deze class als volgt

```
namespace TDDCursusLibrary
{
    public class Jaar
    {
        private int jaar;
        public Jaar(int jaar)
        {
            this.jaar = jaar;
        }
        public bool IsSchrikkeljaar
        {
            get
            {
                if (jaar % 400 == 0)
```

```

        {
            return true;
        }
        if (jaar % 100 == 0)
        {
            return false;
        }
        return jaar % 4 == 0;
    }
}
}
}

```

Zonder unit test zou je deze class gebruiken in een console applicatie, WPF applicatie of website en zou je de class testen door in deze user interface van de applicatie jaartallen te tikken en te controleren of je een juist bericht ziet.

Jaartal: 2013  Dit is geen schrikkeljaar

Deze manier van testen heeft nadelen

- Er is veel tijd tussen het schrijven van de class en het testen van de class. Het zou beter zijn dat je de class kunt testen terwijl je ze schrijft, want op dat moment zijn je gedachten helemaal geconcentreerd op de class.
- Het tikken van jaartallen en het klikken op de knop Controleer jaar is een manuele activiteit die tijd vraagt en na enkele keren saai is.

Met een unit test kan je de class onmiddellijk testen en hoef je de jaartallen niet in een invoervak in te tikken. Je geeft de jaartallen vanuit de unit test door aan de Jaar constructor.

## 2.4 De unit test

Je maakt een unit test JaarTest. Je test daarmee de werking van de class Jaar.

Je voegt een test project toe aan de solution

- Je kiest FILE, Add, New Project
- Je kiest links Visual C#, Test
- Je kiest rechts Unit Test Project
- Je tikt TDDCursusLibraryTest bij Name en je kiest OK
- Je hernoemt UnitTest1.cs naar JaarTest.cs

Je legt een verwijzing naar het project TDDCursusLibrary

- Je kiest PROJECT, Add Reference
- Je kiest links Solution
- Je plaatst rechts een vinkje bij TDDCursusLibrary en je kiest OK

Je wijzigt de source JaarTest

```

using Microsoft.VisualStudio.TestTools.UnitTesting;
using TDDCursusLibrary;

namespace TDDCursusLibraryTest
{
    [TestClass] (1)
    public class JaarTest
    {
        [TestMethod] (2)
        public void EenJaarDeelbaarDoor400IsEenSchrikkeljaar() (3)
        {
            var jaar = new Jaar(2000); (4)
        }
    }
}

```

```

        Assert.AreEqual(true, jaar.IsSchrikkeljaar);
    }
    [TestMethod]
    public void EenJaarDeelbaarDoor100IsGeenSchrikkeljaar()
    {
        var jaar = new Jaar(1900);
        Assert.AreEqual(false, jaar.IsSchrikkeljaar);
    }
    [TestMethod]
    public void EenJaarDeelbaarDoor4IsEenSchrikkeljaar()
    {
        var jaar = new Jaar(2012);
        Assert.AreEqual(true, jaar.IsSchrikkeljaar);
    }
    [TestMethod]
    public void EenJaarNietDeelbaarDoor4IsGeenSchrikkeljaar()
    {
        var jaar = new Jaar(2015);
        Assert.AreEqual(false, jaar.IsSchrikkeljaar);
    }
}
}

```

- (1) Een unit test is een public class waarvoor je [TestClass] schrijft.  
Je mag de naam van deze class vrij kiezen. De conventie is dat de naam gelijk is aan de te testen class, gevolgd door Test.
- (2) Je schrijft één test als één method. Je schrijft [TestMethod] bij deze method.  
Visual Studio aanziet enkel methods voorzien van [TestMethod] als tests.  
Vanaf nu noemen we een method, waarbij [TestMethod] staat, een testmethod.
- (3) Een method die een test voorstelt is public, heeft als returntype void en heeft geen parameters. Je kan de naam van de method vrij kiezen.  
Het is aan te raden dat de naam van de method aangeeft wat je in die method gaat testen. De naam van de method bij (4) duidt aan dat je in deze method test dat een jaar dat deelbaar is door 400 een schrikkeljaar is.
- (4) Je maakt een object van de te testen class.  
Je zorgt er voor dat dit object in de toestand komt die je wilt testen.  
De toestand is in dit geval een voorbeeldjaartal dat deelbaar is door 400.  
Je gebruikt dus in elke test een voorbeeldwaarde die past bij die test.  
Je vindt zo'n waarde in de analyse, in voorbeelddocumenten, op het internet, ... of je bedenkt zelf een correcte waarde die past bij de test.  
Deze waarde is hard gecodeerd. Het is niet de bedoeling dat je in de test zware berekeningen of algoritmes uitwerkt om deze waarde in te stellen.  
Je maakt dan kans in de test zelf denkfouten te maken.  
Een foutieve test kan de werking van de gewone class nooit correct testen !
- (5) Je voert de method uit waarvan je de werking wil testen: IsSchrikkeljaar.  
Als deze property correct geschreven is, is de returnwaarde true. De class Assert bevat enkele static methods waarmee je dit controleert. De meest gebruikte method is de method AreEqual. Je geeft aan deze method twee waarden mee: de waarde die je verwacht (true) en de echte waarde die je krijgt bij de method oproep.  
Als deze twee waarden aan mekaar gelijk zijn, zal Visual Studio deze test aanzien als een test die correct verlopen is.  
Als deze twee waarden verschillen van mekaar, zal Visual Studio deze test aanzien als een test die verkeerd loopt.



De overige testmethods testen andere aspecten van de method `IsSchrikkeljaar`.

## 2.5 De unit test uitvoeren

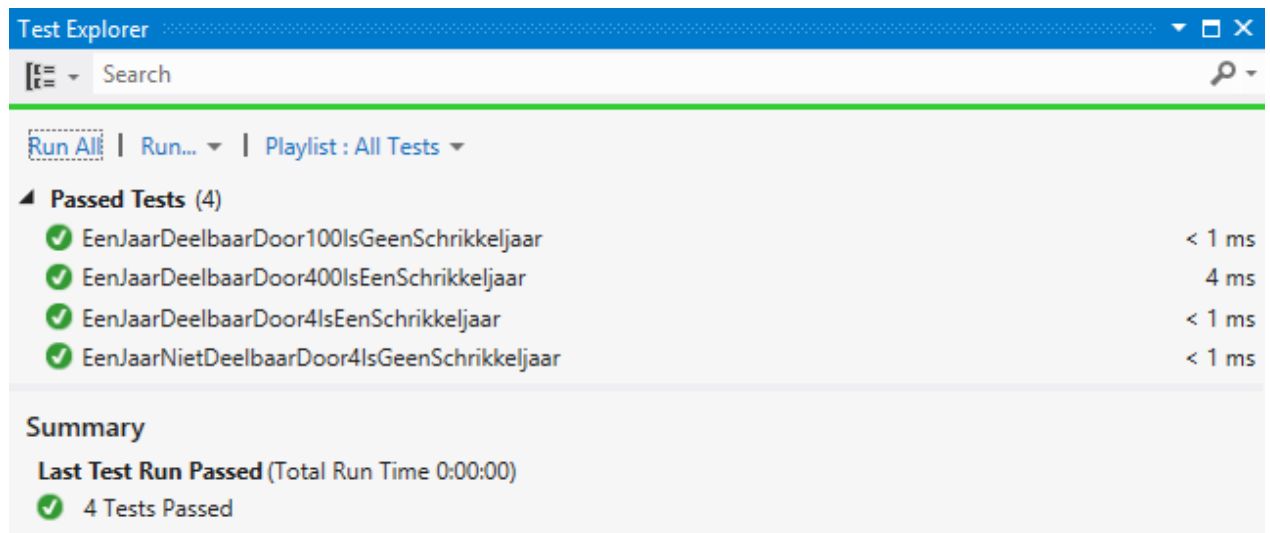
Je laat Visual Studio de unit test uitvoeren.

Visual Studio voert hierbij alle test methods van die unit test uit.

- Je kiest **TEST, Run, All Tests**

Visual Studio voert de unit test uit

en toont een rapport in een venster **Test Explorer**:



De groene horizontale lijn duidt aan dat alle tests geslaagd zijn.

Een groen vinkje duidt aan dat een test geslaagd is.

### Opmerking

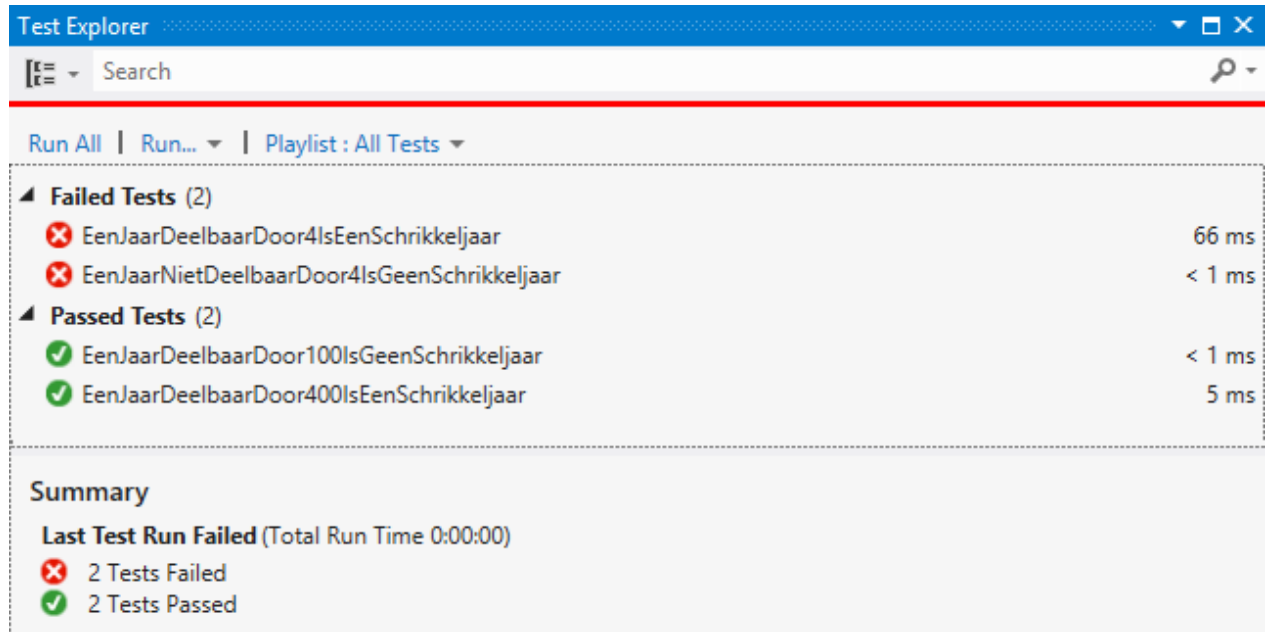
Bemerk dat Visual Studio de test methods niet uitvoert in de volgorde zoals ze geschreven zijn in de unit test `JaarTest`. De volgorde is onbepaald.

Je tikt een fout in de property `IsSchrikkeljaar`, om te zien hoe Visual Studio dan reageert. Je vervangt in de laatste opdracht van deze method 4 door 5.

Je slaat de gewijzigde source op.

Je voert de unit test opnieuw uit met **Run All** in het venster **Test Explorer**.

Je ziet volgend rapport



De rode horizontale lijn duidt aan dat minstens 1 test mislukt is.

Een rood vinkje duidt aan dat een test mislukt is

Je corrigeert de fout in de method `IsSchrikkeljaar`.

Je vervangt in de laatste opdracht van deze method 5 terug door 4.

Je voert de unit test opnieuw uit. Alle tests slagen.

Je ziet met deze voorbeelden al hoe een programmeur bij TDD werkt.

Iedere keer hij een method van een class wijzigt, voert hij de unit test van die class uit. Hij krijgt zo direct feedback of die wijziging correct was.

Code schrijven en onmiddellijk feedback krijgen over die code is een productieve manier om code te schrijven !

## 2.6 Één testmethod uitvoeren

Je voert tot nu alle testmethods van een unit test uit.

Je kan ook één testmethod uitvoeren

- Je klikt met de rechtermuisknop op één van de test methods in het venster Test Explorer en je kiest Run Selected Tests

## 2.7 De unit test als documentatie

Het rapport van een unit test van een class is tegelijk ook prima documentatie over de werking van die class.

```
EenJaarDeelbaarDoor100IsGeenSchrikkeljaar
EenJaarDeelbaarDoor400IsEenSchrikkeljaar
EenJaarDeelbaarDoor4IsEenSchrikkeljaar
EenJaarNietDeelbaarDoor4IsGeenSchrikkeljaar
```

Als ontwikkelaar Caroline de class `Jaar` en de unit test `JaarTest` geschreven heeft, kan haar collega Philippe de werking van de class `Jaar` leren kennen door het rapport van de bijbehorende unit test `JaarTest` in te kijken.

Hij ziet bijvoorbeeld dat een jaar dat deelbaar is door 400 een schrikkeljaar is. Dit lukt enkel als de namen van de testmethods in “mensentaal” geschreven zijn: `eenJaarDeelbaarDoor400IsEenSchrikkeljaar`

## 2.8 De methods van de class Assert

Je gebruikt tot nu in de test methods de method `AreEqual` van de class `Assert`.

De class `Assert` bevat nog methods die je kan gebruiken in testmethods

- `AreNotEqual(verwachteExpressie, teTestenExpressie)`  
De test lukt als `teTestenExpressie` verschilt van `verwachteExpressie`
- `IsTrue(teTestenExpressie)`  
De test lukt als `teTestenExpressie` gelijk is aan `true`.
- `IsFalse(teTestenExpressie)`  
De test lukt als `teTestenExpressie` gelijk is aan `false`
- `IsNull(teTestenExpressie)`  
De test lukt als `teTestenExpressie` gelijk is aan `null`
- `IsNotNull(teTestenExpressie)`  
De test lukt als `teTestenExpressie` verschilt van `null`
- `AreSame(verwachteReferenceVariabele, teTestenReferenceVariabele)`  
De test lukt als `teTestenReferenceVariabele` naar hetzelfde object wijst als `verwachteReferenceVariabele`
- `AreNotSame(verwachteReferenceVariabele, teTestenReferenceVariabele)`  
De test lukt als `teTestenReferenceVariabele` niet naar hetzelfde object wijst als `verwachteReferenceVariabele`

Je kan als voorbeeld `JaarTest` korter schrijven met `IsTrue` en `IsFalse`

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace TDDCursusLibraryTest
{
    [TestClass]
    public class JaarTest
    {
        [TestMethod]
        public void EenJaarDeelbaarDoor400IsEenSchrikkeljaar()
        {
            var jaar = new Jaar(2000);
            Assert.IsTrue(jaar.IsSchrikkeljaar);
        }
        [TestMethod]
        public void EenJaarDeelbaarDoor100IsGeenSchrikkeljaar()
        {
            var jaar = new Jaar(1900);
            Assert.IsFalse(jaar.IsSchrikkeljaar);
        }
        [TestMethod]
        public void EenJaarDeelbaarDoor4IsEenSchrikkeljaar()
        {
            var jaar = new Jaar(2012);
            Assert.IsTrue(jaar.IsSchrikkeljaar);
        }
        [TestMethod]
        public void EenJaarNietDeelbaarDoor4IsGeenSchrikkeljaar()
        {
            var jaar = new Jaar(2015);
            Assert.IsFalse(jaar.IsSchrikkeljaar);
        }
    }
}
```



Taak Palindroom: zie takenbundel

## 3 EERST TESTS SCHRIJVEN, DAARNA IMPLEMENTATIE

### 3.1 Algemeen

Bij de classes `Jaar` en `JaarTest` heb je eerst de class `Jaar` volledig geschreven (geïmplementeerd). Pas daarna schreef je de bijbehorende unit test `JaarTest`.

In dit hoofdstuk wijzigt deze volgorde

- Je schrijft eerst de unit test van een gewone class.
- Pas daarna schrijf je de code van die gewone class.

Deze nieuwe volgorde wordt aangeraden bij test driven development.

#### Opmerking

Je mag wel al de method declaraties schrijven in de gewone class.  
Je laat de binnenkant van deze methods leeg.

Deze nieuwe volgorde heeft volgende voordelen

- Je schrijft de testen op basis van de analyse van de class.  
In de analyse van de class `Jaar` bleken volgende regels, die je uitschrijft als bijbehorende testmethods
  - als een jaar deelbaar is door 4 is het een schrikkeljaar.  
`public void EenJaarDeelbaarDoor4IsEenSchrikkeljaar()`
  - als een jaar niet deelbaar door 4 is geen schrikkeljaar.  
`public void EenJaarNietDeelbaarDoor4IsGeenSchrikkeljaar()`
  - als een jaar deelbaar is door 100 is het echter geen schrikkeljaar.  
`public void EenJaarDeelbaarDoor100IsGeenSchrikkeljaar()`
  - als een jaar deelbaar is door 400 is het echter wel een schrikkeljaar.  
`public void EenJaarDeelbaarDoor400IsEenSchrikkeljaar()`

Je bent zo verplicht de vereisten uit de analyse nog eens grondig te bestuderen vooraleer je die uitwerkt in code in de class `Jaar`.

Je zal ook geen overbodige functionaliteit in de class `Jaar` schrijven: zodra alle testen slagen heb je genoeg code in de class geschreven.

- Je roept binnen de testmethods de methods op van de te testen class.  
Als deze method oproepen vanuit de testmethods vreemd aandoen, wegens een verkeerd gekozen methodnaam, te veel of te weinig parameters, ... kan je snel de signatuur van deze methods wijzigen, omdat ze enkel vanuit de testmethods zijn opgeroepen.  
Het kan ook dat je tijdens het uitvoeren van de testmethods merkt dat er nog methods ontbreken in de te testen class.
- Tijdens het schrijven van de unit test denk je na over de werking van de te testen class. Pas daarna schrijf je deze class uit.  
Je schrijft betere code als je eerst nagedacht hebt over de te schrijven code.
- Als je eerst code schrijft en pas daarna de tests voor die code, volg je in de tests onbewust hetzelfde algoritme dat je in de code geschreven hebt.  
Als dit algoritme foutief is, schrijf je de test onbewust ook foutief en is de test dus niet correct. Als je de test schrijft voor de code, gebeurt deze fout niet.

## 3.2 De te testen class

Als je geen enkele letter code zou mogen schrijven van de te testen class, kan je ook niet naar deze class verwijzen binnen de unit test.

Je schrijft daarom een minimale versie van de te testen class: de class zelf en de methods van de class. De methods bevatten echter geen code.

Sommige methods van de te testen class zijn geen `void` methods, maar geven een waarde (een `int`, een `String`, een `decimal`) terug. Als je zo'n method geen `return` opdracht bevat kan je de class niet compileren.

Je zou dan ook niet naar de class kunnen verwijzen vanuit de unit test.

De oplossing is als volgt: je schrijft initieel in elke method één opdracht: `throw new NotImplementedException();`

Nu kan je de class wel compileren.

De exception `NotImplementedException` geeft expliciet aan dat bij een oproep van de method deze method momenteel nog niet uitgewerkt (*implemented*) is.

## 3.3 Voorbeeld

### 3.3.1 De te testen class

Je past deze techniek toe op de voorbeeldclass `Rekening`

Rekening
-saldo: decimal
+Storten(bedrag: decimal)
+Saldo(): decimal

Je schrijft de class met zijn methods, maar je schrijft in elke method enkel `throw new NotImplementedException();`

`using System;`

```
namespace TDDCursusLibrary
{
    public class Rekening
    {
        public void Storten(decimal bedrag)
        {
            throw new NotImplementedException();
        }
        public decimal Saldo
        {
            get
            {
                throw new NotImplementedException();
            }
        }
    }
}
```

### 3.3.2 De unit test

Je schrijft nu de unit test voor deze class, op basis van de analyse.

Uit deze analyse blijkt het volgende

- Het saldo van een nieuwe rekening is 0 €.
- Als je een eerste bedrag stort op een nieuwe rekening, is het saldo gelijk aan dit eerste bedrag.
- Als je een eerste bedrag en daarna een tweede bedrag stort op een nieuwe rekening, is het saldo gelijk aan de som van die twee bedragen.


Je maakt de unit test `RekeningTest` als volgt

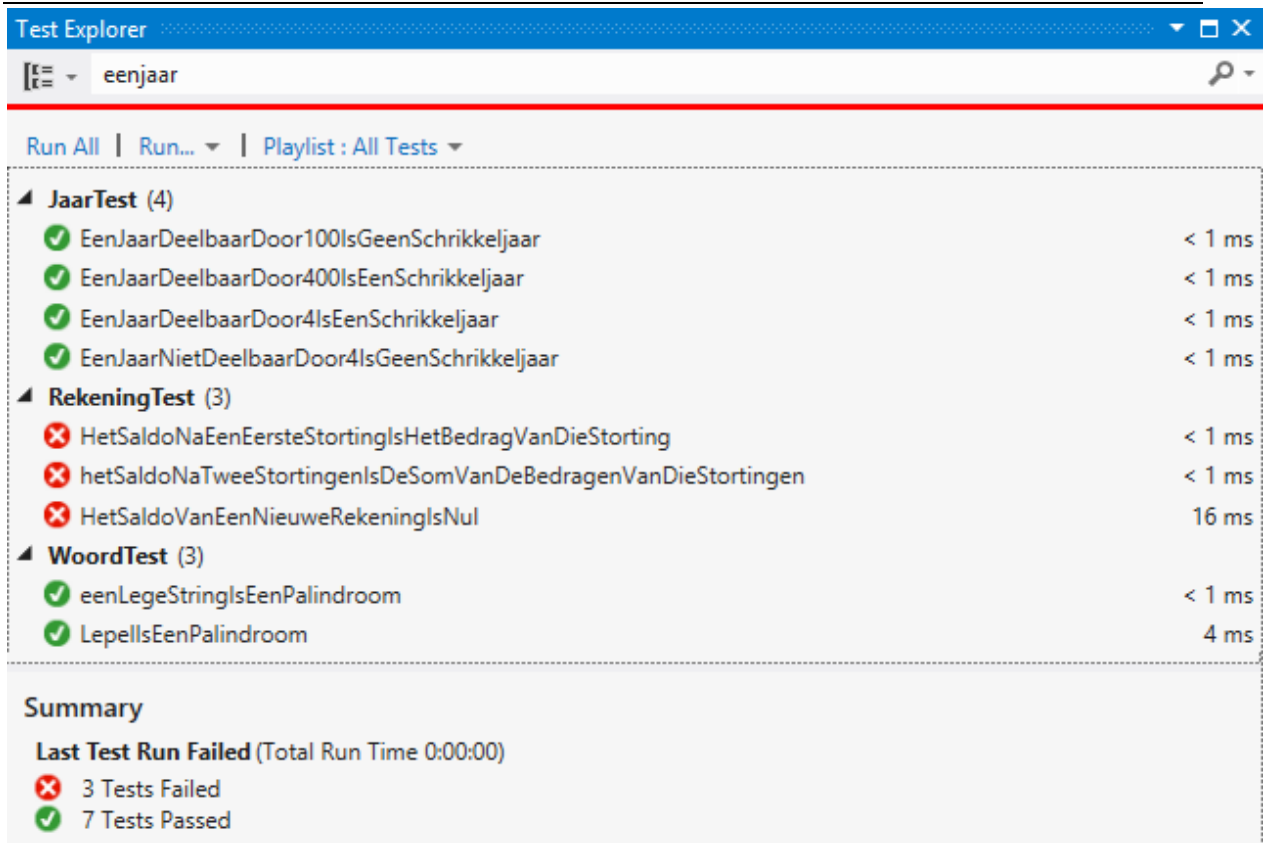
```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System;
using TDDCursusLibrary;

namespace TDDCursusLibraryTest
{
    [TestClass]
    public class RekeningTest
    {
        [TestMethod]
        public void HetSaldoVanEenNieuweRekeningIsNul()
        {
            var rekening = new Rekening();
            Assert.AreEqual(decimal.Zero, rekening.Saldo);
        }
        [TestMethod]
        public void HetSaldoNaEenEersteStortingIsHetBedragVanDieStorting()
        {
            var rekening = new Rekening();
            var bedrag = 2.5m;
            rekening.Storten(bedrag);
            Assert.AreEqual(bedrag, rekening.Saldo);
        }
        [TestMethod]
        public void hetSaldoNaTweeStortingenIsDeSomVanDeBedragenVanDieStortingen()
        {
            var rekening = new Rekening();
            rekening.Storten(2.5m);
            rekening.Storten(1.2m);
            Assert.AreEqual(3.7m, rekening.Saldo);
        }
    }
}
```

Je controleert als volgende stap of je aan de verleiding kon weerstaan om methods in de class `Rekening` te implementeren (er code in te schrijven).

Je doet dit door de unit test uit te voeren. Je krijgt een rapport.

Alle nieuwe tests mislukten. Je kan de tests per test class zien als je in de Test Explorer de knop  aanklikt en kiest Class.



### 3.3.3 Een eerste implementatie

Je schrijft nu code in de methods van de class `Rekening`.

De eerste versie van die code moet niet noodzakelijk de optimaalste code zijn (naar performantie, geheugengebruik, leesbaarheid, onderhoudbaarheid ... toe)

`using System;`

```
namespace TDDCursusLibrary
{
    public class Rekening
    {
        private decimal saldo;
        public void Storten(decimal bedrag)
        {
            saldo += bedrag;
        }
        public decimal Saldo
        {
            get
            {
                return saldo;
            }
        }
    }
}
```

Je voert de bijbehorende unit test (`RekeningTest`) uit en alle tests slagen. Dit betekent dat de class `Rekening` voldoet aan de vereisten van de analyse.

### 3.3.4 Refactoring

Nu je een werkende versie hebt van de class `Rekening`, zie je na of je in deze class refactoring kan toepassen.

Bij refactoring wijzig je code om de leesbaarheid en onderhoudbaarheid te verbeteren. Het kan ook dat je gedurende refactoring de performantie of geheugengebruik verbetert.

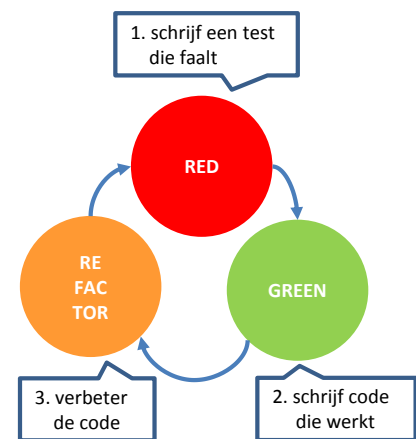
Na het refactoren voer je de unit test opnieuw uit.

- Als de tests slagen, is je refactoring OK.
- Als een test mislukt, zie je de refactoring na en voer je de tests opnieuw uit, tot alle tests slagen.

Je kan in de class `Rekening` geen refactoring doen.

### 3.3.5 Samenvatting van de stappen bij test driven development

- Je doet analyse van de te schrijven class.
- Je schrijft de class en zijn methods.  
De methods bevatten één opdracht:  
`throw new NotImplementedException();`
- Je schrijft de unit test van de class, gebaseerd op de vereisten voor de class die je ontdekte in de analyse.
- Je voert de unit test uit, de tests mislukken.
- Je schrijft echte code in de class en voert de unit tests uit, tot de tests slagen.
- Je past op de class refactoring toe en controleert met unit tests of deze geen nieuwe fouten introduceren, tot je vindt dat de code van de class optimaal is.



Taak Veiling: zie takenbundel



## 4 TEST FIXTURES

### 4.1 Algemeen

Het gebeurt regelmatig dat je in meerdere test methods van een unit test soortgelijke objecten aanmaakt. Je initialiseert in `RekeningTest` bijvoorbeeld in iedere test method eenzelfde `Rekening` object:

```
var rekening = new Rekening();
```

Zo'n object heet een test fixture.

### 4.2 [TestInitialize]

Code herhalen is niet goed, dus herhalend test fixtures initialiseren is niet goed.

Je vermijdt dit herhalen in twee stappen.

Als eerste stap declareer je in de unit test een `private` variabele per test fixture. In `RekeningTest` wordt dit dus `private Rekening rekening`;

Als tweede stap initialiseer je de variabele in een `testinitialize` method.

Dit is een `public void` method, zonder parameters, waarvoor je `[TestInitialize]` tikt. De naam van de method is vrij te kiezen.

In `RekeningTest` wordt dit dus een method

```
[TestInitialize]
public void Initialize()
{
    rekening = new Rekening();
}
```

Visual Studio voert voor iedere test method de `testinitialize` method van dezelfde unit test uit.

Als Visual Studio de tests van `RekeningTest` uitvoert, voert hij volgende methods uit

- `[TestInitialize]`      `Initialize()`
- `[TestMethod]`        `HetSaldoNaTweeStortingenIsDeSomVanDeBedragenVanDieStortingen()`
- `[TestInitialize]`      `Initialize()`
- `[TestMethod]`        `hetSaldoNaEenEersteStortingIsHetBedragVanDieStorting()`
- `[TestInitialize]`      `Initialize()`
- `[TestMethod]`        `hetSaldoVanEenNieuweRekeningIsNul()`

Je moet nu binnen de test method geen `Rekening` object meer initialiseren, maar je kan de `private` variabele `rekening` gebruiken. Visual Studio initialiseert deze variabele bij iedere oproep van de `testinitialize` method met een `Rekening` object.

De volledig aangepast `RekeningTest`

```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System;
using TDDCursusLibrary;

namespace TDDCursusLibraryTest
{
    [TestClass]
    public class RekeningTest
    {
        private Rekening rekening;
```

```
[TestInitialize]
public void Initialize()
{
    rekening = new Rekening();
}
[TestMethod]
public void HetSaldoVanEenNieuweRekeningIsNul()
{
    Assert.AreEqual(decimal.Zero, rekening.Saldo);
}
[TestMethod]
public void HetSaldoNaEenEersteStortingIsHetBedragVanDieStorting()
{
    var bedrag = 2.5m;
    rekening.Storten(bedrag);
    Assert.AreEqual(bedrag, rekening.Saldo);
}
[TestMethod]
public void hetSaldoNaTweeStortingenIsDeSomVanDeBedragenVanDieStortingen()
{
    rekening.Storten(2.5m);
    rekening.Storten(1.2m);
    Assert.AreEqual(3.7m, rekening.Saldo);
}
}
```

Je kan de unit test opnieuw uitvoeren.

Je mag het Rekening object, waar de private variabele rekening naar verwijst, wijzigen in een test method. Bij de volgende test method bevat de variabele rekening een vers geïnitieerd Rekening object: Visual Studio voert tussen de test methods de testinitialize method uit.

#### Opmerking

Je kan in een unit test ook een method opnemen waarvoor je [TestCleanup] tikt. Visual Studio voert deze testcleanup method uit na iedere test method.

Je hebt een testcleanup method enkel nodig als je in de testinitialize method iets aanmaakt buiten het RAM geheugen (bijvoorbeeld een bestand), dat je tijdens de test method gebruikt en dat je na de test method wil verwijderen. Je verwijdert dit bestand in een testcleanup method.



Taak Test fixtures: zie takenbundel

## 5 TO TEST OR NOT TO TEST

### 5.1 Exceptions testen

Als je een verkeerde waarde meegeeft aan de constructor of een method van een class, werkt deze constructor of method enkel juist als hij een exception werpt.

Je kan ook dit testen met Visual Studio.

Uit de analyse van de class Rekening blijkt een extra regel:  
het bedrag dat je stort moet een positief getal zijn.

Zoniet moet de method storten een ArgumentException werpen.

Je schrijft eerst extra tests voor deze nieuwe vereiste in RekeningTest.

```
[TestMethod, ExpectedException(typeof(ArgumentException))] (1)
```

```
public void HetBedragVanEenStortingMagNietNulZijn()
```

```
{  
    rekening.Storten(decimal.Zero); (2)
```

```
}
```

```
[TestMethod, ExpectedException(typeof(ArgumentException))] (1)
```

```
public void HetBedragVanEenStortingMagNietNegatiefZijn()
```

```
{  
    rekening.Storten(-1m);
```

```
}
```

- (1) Je test in deze test method een situatie die tot een ArgumentException moet leiden: je probeert 0 € te storten bij (2).

Je vermeldt deze exceptie als de parameter van ExpectedException.

Enkel als deze exception optreedt tijdens het uitvoeren van de test method is deze test volgens Visual Studio geslaagd.

Je voert de test uit.

De twee extra test methods mislukken.

Dit betekent dat je extra code moet schrijven in de class Rekening, om aan deze extra vereiste uit de analyse te voldoen.

Je wijzigt in de class Rekening de method storten

```
public void Storten(decimal bedrag)
```

```
{  
    if (bedrag <= decimal.Zero)  
    {  
        throw new ArgumentException();  
    }  
    saldo += bedrag;  
}
```

Je voert de unit test RekeningTest opnieuw uit. Alle tests slagen.

De code in de class Rekening voldoet dus aan de extra vereiste uit de analyse.

## 5.2 Grenswaarden en extreme waarden testen

### 5.2.1 Algemeen

Hoe meer tests, hoe zekerder je bent van de kwaliteit van de geteste code.

Hierbij is het ook belangrijk dat je grenswaarden test

- waarden die juist op de grens liggen van wat mag volgens de analyse
- waarden die juist boven de grens liggen van wat mag volgens de analyse
- waarden die juist onder de grens liggen van wat mag volgens de analyse

Het is ook belangrijk dat je extreme waarden test

- de waarde `null` meegeven aan een parameter bij een method oproep
- Een lege string meegeven aan een parameter van het type `string`
- `Integer.MaxValue` meegeven aan een parameter van het type `int`

### 5.2.2 Eerste voorbeeld

Je maakt een class `Rekeningnummer`.

Deze class stelt een Belgisch bankrekeningnummer voor.

Rekeningnummer
-nummer: String +Rekeningnummer(nummer: String) +ToString(): String

Uit de analyse blijken volgende regels

- Het nummer moet 12 cijfers bevatten.  
Je moet dus een nummer met de grenswaarde 12 cijfers testen, een nummer met de grenswaarde 13 cijfers en een nummer met de grenswaarde 11 cijfers.
- Na de eerste 3 cijfers en de volgende 7 cijfers komt een - teken: 063-1547564-61  
Je moet dus een nummer testen met een eerste grenswaarde: een nummer mét streepjes. Je moet ook een nummer testen met een tweede grenswaarde: een nummer zonder streepje.
- Het getal, voorgesteld door de laatste 2 cijfers, moet gelijk zijn aan het getal, voorgesteld door de eerste 10 cijfers modulus 97.  
Je moet dus een nummer testen met een eerste grenswaarde: een nummer waarbij deze berekening klopt. Je moet ook een nummer testen met een tweede grenswaarde: een nummer waarbij deze berekening juist niet klopt.

Als extreme waarden doe je ook een test waarbij je `null` meegeeft aan de constructor en een test waarbij je een lege string meegeeft aan de constructor.

### 5.2.3 De class

Je schrijft de class Rekeningnummer

```
using System;

namespace TDDCursusLibrary
{
    public class Rekeningnummer
    {
        public Rekeningnummer(string nummer)
        {
            throw new NotImplementedException();
        }
        public override string ToString()
        {
            throw new NotImplementedException();
        }
    }
}
```

### 5.2.4 De unit test

Je schrijft RekeningnummerTest als volgt

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using TDDCursusLibrary;

namespace TDDCursusLibraryTest
{
    [TestClass]
    public class RekeningNummerTest
    {
        [TestMethod]
        public void NummerMet12CijfersMetCorrectControleIsOK()
        {
            new Rekeningnummer("063-1547564-61");
            // dit nummer mag geen exception veroorzaken
        }
        [TestMethod, ExpectedException(typeof(ArgumentException))]
        public void NummerMet12CijfersMetVerkeerdeControleIsVerkeerd()
        {
            new Rekeningnummer("063-1547564-62");
        }
        [TestMethod, ExpectedException(typeof(ArgumentException))]
        public void NummerMet12CijfersZonderStreepjesMetCorrectControleIsVerkeerd()
        {
            new Rekeningnummer("063154756461");
        }
        [TestMethod, ExpectedException(typeof(ArgumentException))]
        public void NummerMet13CijfersIsVerkeerd()
        {
            new Rekeningnummer("063-1547564-623");
        }
        [TestMethod, ExpectedException(typeof(ArgumentException))]
        public void NummerMet11CijfersIsVerkeerd()
        {
            new Rekeningnummer("063-1547564-6");
        }
        [TestMethod, ExpectedException(typeof(ArgumentException))]
        public void LeegNummerIsVerkeerd()
        {
            new Rekeningnummer("");
        }
    }
}
```

```
[TestMethod, ExpectedException(typeof(ArgumentNullException))]
public void nummerMetNullIsVerkeerd()
{
    new Rekeningnummer(null);
}
[TestMethod]
public void toStringMoetHetNummerTeruggeven()
{
    var nummer = "063-1547564-61";
    var rekeningnummer = new Rekeningnummer(nummer);
    Assert.AreEqual(nummer, rekeningnummer.ToString());
}
}
```

Je controleert als volgende stap of je aan de verleiding kon weerstaan om methods in de class Rekeningnummer te implementeren

Je doet dit door de unit test uit te voeren.

Je krijgt een rapport waarin al deze tests mislukten. Dit is goed nieuws.

### 5.2.5 Een eerste implementatie

Je schrijft nu code in de methods van de class Rekeningnummer

```
using System;
using System.Text.RegularExpressions;
namespace TDDCursusLibrary
{
    public class Rekeningnummer
    {
        private static Regex regex = new Regex("^\\d{3}-\\d{7}-\\d{2}$");           (1)
        private String nummer;
        public Rekeningnummer(String nummer)
        {
            if (!regex.IsMatch(nummer))                                         (2)
            {
                throw new ArgumentException();
            }
            long eerste10Cijfers =
                long.Parse(nummer.Substring(0, 3) + nummer.Substring(4, 7));
            long laatste2Cijfers = long.Parse(nummer.Substring(12, 2));
            if (eerste10Cijfers % 97L != laatste2Cijfers)
            {
                throw new ArgumentException();
            }
            this.nummer = nummer;
        }
        public override string ToString()
        {
            return nummer;
        }
    }
}
```

- (1) Dit object stelt een regular expression (tekstpatroon) voor.  
In een regular expression hebben bepaalde tekens een speciale betekenis.  
Sommige van deze tekens worden in het voorbeeld gebruikt
- |     |   |
|-----|---|
| ^   | het begin van de tekst                                    |
| \$  | het einde van de tekst                                    |
| \\d | een cijfer  |
| {x} | x aantal keer het vorige teken uit de regular expression. |

We leggen nu de regular expression uit, die een rekeningnummer voorstelt

- `^` het begin van het rekeningnummer
- `\\d{3}` daarna komen 3 cijfers
- `-` daarna komt letterlijk het teken -
- `\\d{7}` daarna komen 7 cijfers
- `-` daarna komt letterlijk het teken -
- `\\d{2}` daarna komen 2 cijfers
- `$` het einde van het rekeningnummer

- (2) De method `IsMatch` geeft `true` terug als de `Regex` waarop je deze method uitvoert past bij de `string` die je als parameter meegeeft. Anders geeft de method `false` terug.

Je voert de bijbehorende unit test (`RekeningnummerTest`) uit en alle tests slagen. Dit betekent dat de class `Rekeningnummer` voldoet aan de vereisten van de analyse.

### 5.2.6 Refactoring

Nu je een werkende versie hebt van de class `Rekeningnummer`, zie je na of je in deze class refactoring kan toepassen.

Dit is niet het geval. De class `Rekeningnummer` is dus af.

### 5.2.7 Voorbeeld met een verzameling

Als je een method test die een parameter met een verzameling (`array`, `List`, ...) bevat, moet je ook volgende grenswaarden en extreme waarden testen:

- ☞ Een verzameling met één element
- ☞ Een lege verzameling
- ☞ In de plaats van een verzameling de waarde `null` meegeven

Je maakt een class `Statistiek`. Deze bevat een static method `Gemiddelde`.

Je geeft aan deze method een array van `decimals` mee.

Je krijgt het gemiddelde van deze `decimals` terug.

Statistiek
<u>+Gemiddelde(getallen: decimal[]): decimal</u>

```
using System;
```

```
namespace TDDCursusLibrary
```

```
{
    public static class Statistiek
    {
        public static decimal Gemiddelde(decimal[] getallen)
        {
            throw new NotImplementedException();
        }
    }
}
```

### 5.2.8 De unit test

Je schrijft de unit test StatistiekTest als volgt

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using TDDCursusLibrary;

namespace TDDCursusLibraryTest
{
    [TestClass]
    public class StatistiekTest
    {
        [TestMethod]
        public void HetGemiddeldeVan10en15is12punt5()
        {
            decimal[] getallen = { 10m, 15m };
            Assert.AreEqual(12.5m, Statistiek.Gemiddelde(getallen));
        }
        [TestMethod]
        public void HetGemiddeldeVanEenGetalIsDatGetal()
        {
            decimal enigGetal = 1.23m;
            decimal[] getallen = { enigGetal };
            Assert.AreEqual(enigGetal, Statistiek.Gemiddelde(getallen));
        }
        [TestMethod, ExpectedException(typeof(ArgumentException))]
        public void hetGemiddeldeVanEenLegeVerzamelingKanJeNietBerekenen()
        {
            decimal[] legeVerzameling = { };
            Statistiek.Gemiddelde(legeVerzameling);
        }
        [TestMethod, ExpectedException(typeof(ArgumentNullException))]
        public void HetGemiddeldeVanNullKanJeNietBerekenen()
        {
            Statistiek.Gemiddelde(null);
        }
    }
}
```

Je controleert als volgende stap of je aan de verleiding kon weerstaan om methods in de class Statistiek te implementeren

Je doet dit door de unit test uit te voeren.

Je krijgt een rapport waarin alle tests mislukten. Dit is goed nieuws.

### 5.2.9 Een eerste implementatie

Je schrijft nu code in de method van de class Statistiek.

```
using System;

namespace TDDCursusLibrary
{
    public static class Statistiek
    {
        public static decimal Gemiddelde(decimal[] getallen)
        {
            if (getallen == null)
            {
                throw new ArgumentNullException();
            }
        }
    }
}
```



```
        if (getallen.Length == 0)
        {
            throw new ArgumentException();
        }
        var totaal = decimal.Zero;
        foreach (var getal in getallen)
        {
            totaal += getal;
        }
        return totaal / getallen.Length;
    }
}
```

Je voert de bijbehorende unit test (StatistiekTest) uit en alle tests slagen.  
Dit betekent dat de class Statistiek voldoet aan de vereisten van de analyse.

### 5.2.10 Refactoring

Nu je een werkende versie hebt van de class Statistiek,  
zie je na of je in deze class refactoring kan toepassen.

Dit kan, je vervangt de regels vanaf `var totaal ...` tot en met `return ...` door  
`return getallen.Average();`

## 5.3 Testen bijhouden

Je verwijdert nooit unit tests, tenzij ze niet meer relevant zijn  
omdat de analyse van de te testen class wijzigt.

Iedere keer je de te testen class achteraf nog bijwerkt,  
kan je nazien of die aanpassing geen nieuwe fouten introduceerde,  
door de bijbehorende unit test uit te voeren.



Taak ISBN: zie takenbundel


## 6 MEERDERE UNIT TESTS UITVOEREN

### 6.1 Algemeen

Een ordered test is een test die een verzameling testmethods uit één of meerdere unit tests bevat.

Je maakt een ordered test die de testmethods bevat van `RekeningTest` en van `RekeningnummerTest`.

- Je selecteert in de Solution Explorer het project `TDDCursusLibraryTest`
- Je kiest PROJECT, Add Ordered Test

Je krijgt een venster met links alle beschikbare testen. Je kan zo'n test aanduiden en met de knop  overbrengen naar rechts, waar je de ordered test ziet.

Je sorteert eerst de tests op hun ID, door de derde hoofding aan te klikken. Zo staan alle test methods per unit test bij mekaar.

Je brengt alle tests van `RekeningNummerTest` en `RekeningTest` over naar rechts.

Je wijzigt in de Solution Explorer de naam van de ordered test naar `Rekening(Nummer).orderedtest`

Je voert alle tests uit en je ziet in het resultaat venster ook de ordered test staan. Je kan deze ordered test aanklikken met de rechtermuisknop en `Run Selected Tests` kiezen. Nu voer je enkel die ordered test uit.

#### Opmerking

Een ordered test kan andere ordered tests bevatten.

## 7 DEPENDENCIES – STUBS

### 7.1 Dependency

#### 7.1.1 Algemeen

Als een class A methods oproept van een class B, heeft de class A een dependency (“afhankelijkheid”) op de class B.

```
public class A {  
    public void a1() {  
        ...  
        B b = new B();  
        b.b1();  
        ...  
    }  
    public int a2() {  
        ...  
        B b = new B();  
        b.b1();  
        b.b2();  
        ...  
    }  
}
```

In een class diagram druk je deze dependency uit met een pijl, met gestippelde lijn, van de class A naar de class B



Heel veel classes hebben een dependency op één of meerdere andere classes.

Je leert in dit hoofdstuk hoe je de problemen oplost bij het testen van zo’n classes. Deze problemen stellen zich niet als de classes A of B entities of value objecten (concepten uit de werkelijkheid) voorstellen.

Deze problemen stellen zich wel als de classes A of B technische classes zijn (WinForm classes, WPF classes, WebForm classes, ASP.NET controller classes, service classes, classes die de database aanspreken).

Technische classes zijn thread-safe (aanroepbaar vanuit meerder threads).

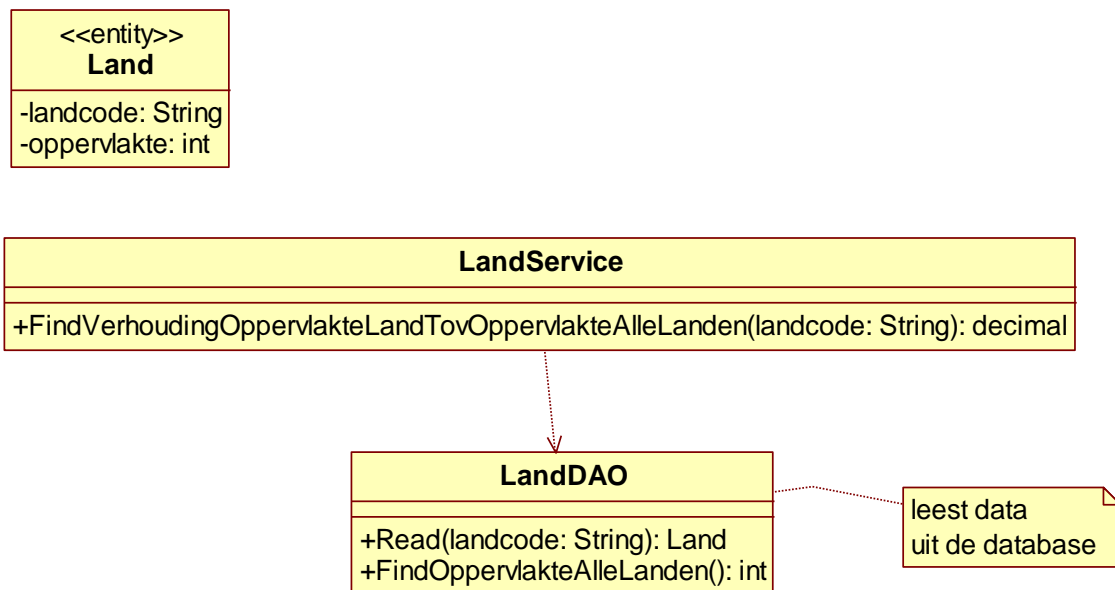
Je moet dan niet in elke method van de class A een instance maken van de class B. Je kan één instance van de class B in een `private` variabele in de class A bijhouden. Je gebruikt deze instance in alle methods van de class A.

Je maakt deze `private` variabele ook best `readonly`. Je verhindert zo dat je (per ongeluk) de variabele na het initialiseren nog wijzigt (bvb. op `null` plaatst).

```
public class A {  
    private readonly B b = new B();  
    public void a1() {  
        ...  
        b.b1();  
        ...  
    }  
    public int a2() {  
        ...  
        b.b1();  
        b.b2();  
        ...  
    }  
}
```

### 7.1.2 Voorbeeld

Je zal de problemen bij het testen van een technische class met dependencies leren kennen aan de hand van volgend voorbeeld



De class `LandService` heeft een dependency op de class `LandDAO`.

De method `FindVerhoudingOppervlakteLandTovOppervlakteAlleLanden` geeft een `decimal` terug met de verhouding van de oppervlakte van één land ten opzichte van de oppervlakte van alle landen. Je bepaalt dit land door de bijbehorende `landcode` als parameter mee te geven.

Je zal in deze method de `LandDAO` method `Read` oproepen.

Je krijgt de detailinformatie van dat land (waaronder de oppervlakte) terug.

Je zal in deze method ook de `LandDAO` method `FindOppervlakteAlleLanden` oproepen. Je krijgt de totale oppervlakte van alle landen terug.

Je zal daarna de oppervlakte van het ene land delen door de oppervlakte van alle landen.

Je maakt de class `Land`

```
namespace TDDCursusLibrary
{
    public class Land
    {
        public string Landcode { get; set; }
        public int Oppervlakte { get; set; }
    }
}
```

Je maakt de class `LandService`.

```
using System;
```

```
namespace TDDCursusLibrary
{
    public class LandService
    {
        public decimal VerhoudingOppervlakteLandTovOppervlakteAlleLanden(string landcode)
        {
            throw new NotImplementedException();
        }
    }
}
```

```
}  
}  
}
```

### 7.1.3 Problemen bij het testen van een technische class met dependencies

Er zijn drie problemen bij het testen van de class `LandService`

- In een groot team van programmeurs kan iedere programmeur zijn “specialiteit” hebben. “front-end” programmeurs zijn gespecialiseerd in user-interface code. Ze programmeren WebForms, WPF, HTML, CSS. “back-end” programmeurs zijn gespecialiseerd in databasetoegangcode en programmeren DAO classes. Het is dus mogelijk dat jij de class `LandService` schrijft en Philippe de class `LandDAO`. Jij kan echter de class `LandService` niet schrijven zolang de Philippe de class `LandDAO` niet heeft afgewerkt. Als Philippe verlof heeft, ziek is of andere werk heeft met hogere prioriteit, kan jij niet verder werken.
- Jij test `LandService` met een unit test voor deze class. Deze test voert indirect ook code uit van de `LandDAO` methods. Jij kan dus geconfronteerd worden met fouten in `LandDAO` methods. Dit is niet de bedoeling: jij wilt fouten opsporen in jouw code, niet in de code van Philippe. Zelfs als jij zowel de class `LandService` als de class `LandDAO` schrijft, wil je bij een unit test van `LandService` niet geconfronteerd worden met fouten in `LandDAO`. Je schrijft een aparte unit test voor de class `LandDAO`.
- Je test `LandService` met een unit test voor deze class. Deze test voert indirect ook code uit van de `LandDAO` methods. Gezien deze methods de database aanspreken, loopt de unit test traag. Dit maakt test driven development vervelend: je wilt de feedback van een test snel zien (binnen de seconde). Dit geldt ook als jij zowel de class `LandService` als de class `LandDAO` schrijft.

### 7.1.4 De dependency uitdrukken in een interface

De eerste stap om deze problemen op te lossen is de functionaliteit van de dependency uit te drukken in een interface.

```
<<interface>>  
ILandDAO  
  
+Read(landcode: String): Land  
+FindOppervlakteAlleLanden(): int
```

Je maakt de interface `ILandDAO`

```
namespace TDDCursusLibrary  
{  
    public interface ILandDAO  
    {  
        Land Read(string landcode);  
        int OppervlakteAlleLanden();  
    }  
}
```

Jij kan deze interface ontwerpen, of Philippe, of jullie analyst, ...

Als tweede stap gebruik je deze interface in elke class die zo’n dependency heeft.

...

```
public class LandService
{
private LandDAO landDAO = new LandDAO();
    private readonly ILandDAO landDAO;
    ...
}
```

In deze nieuwe versie bevat de variabele `landDAO` de waarde `null`.

Als je op deze variabele een method oproept, krijg je een `NullReferenceException`.

### 7.1.5 Dependency injection

Je voegt aan dezelfde class een constructor toe, om dit probleem op te lossen

```
...
public class LandService
{
    private readonly ILandDAO landDAO;
    public LandService(ILandDAO landDAO) {
        this.landDAO = landDAO;
    }
    ...
}
```

Je geeft, bij het aanmaken van een `LandService` object, aan de constructor een object mee waarvan de class de interface `ILandDAO` implementeert.

De private variabele `landDAO` wijst daarna naar dit object.

De oproepen `landDAO.read(landcode);` en `landDAO.findOppervlakteAlleALanden();` verder in `LandService`, gebeuren op dit object.

Zo treedt geen `NullReferenceException` meer op.



De class `LandService` maakt dus zelf geen object meer dat de dependency voorstelt, maar krijgt dit object aangereikt (als constructor parameter). Dit heet “dependency injection”.

#### ☞ Opmerking

Een class kan meerdere dependencies hebben.

De class bevat dan per dependency een extra `private final` variabele en een extra parameter in de constructor.

### 7.1.6 Stub

Wanneer je een unit test schrijft voor de class `LandService`, maak je in die unit test een `LandService` object aan.

Je roept hierbij de `LandService` constructor op.

Je moet als constructor parameter een object meegeven waarvan de class de interface `ILandDAO` implementeert.

De class die Philippe zal schrijven, zal deze interface implementeren.

Philippe zal deze class bijvoorbeeld `LandDAOImpl` noemen.

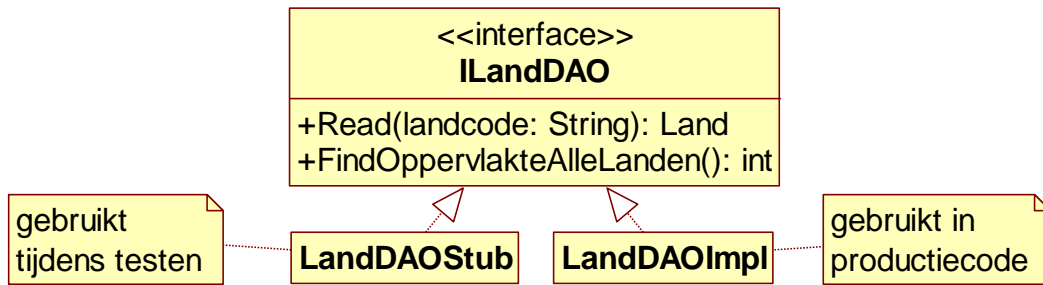
Maar jij gebruikt in de unit test van `LandService` geen object van deze class.

Anders heb je de problemen beschreven in dit hoofdstuk.

Je gebruikt in de plaats een stub. Een stub is een object.

De class van dit object implementeert dezelfde interface (`ILandDAO`) als de echte class (`LandDAOImpl`). Maar de code in de stub is minimaal. De code zorgt er enkel voor dat jij je unit test (voor de class `LandService`) kan schrijven.

De stub heet bijvoorbeeld `LandDAOStub`



- De class `LandDAOImpl` leest data uit de database.
- De class `LandDAOStub` maakt dummy data in het interne geheugen. Deze data moet geen reële data zijn. Deze data dient enkel om de class `LandService` te kunnen testen.

Je kan een stub vergelijken met een crash test dummy die auto-ontwerpers gebruiken bij het uittesten van een auto.

Je maakt stubs in het project `TDDCursusLibraryTest`: je gebruikt stubs enkel in tests.

Je maakt de class `LandDAOStub`

```

using TDDCursusLibrary;

namespace TDDCursusLibraryTest
{
    class LandDAOStub : ILandDAO
    {
        public Land Read(string landcode)
        {
            return new Land { Landcode = landcode, Oppervlakte = 5 };
        }
        public int OppervlakteAlleLanden()
        {
            return 20;
        }
    }
}

```

(1) De stub geeft een `Land` object terug met de gevraagde landcode en een fictieve oppervlakte (5).

(2) De stub geeft een fictieve totale oppervlakte terug (20).

Je maakt de class `LandServiceTest`

```

using Microsoft.VisualStudio.TestTools.UnitTesting;
using TDDCursusLibrary;
using System;

namespace TDDCursusLibraryTest
{
    [TestClass]
    public class LandServiceTest
    {
        private ILandDAO landDAO;
        private LandService landService;
        [TestInitialize]
        public void Initialize()
        {
            landDAO = new LandDAOStub();
            landService = new LandService(landDAO);
        }
    }
}

```

```
[TestMethod]
public void VerhoudingOppervlakteLandTovOppervlakteAlleLanden()
{
    Assert.AreEqual(0.25m,
        landService.VerhoudingOppervlakteLandTovOppervlakteAlleLanden("B"));
}
}
```

- (1) Je maakt een stub.
- (2) Je geeft deze stub mee aan de constructor van de te testen class (dependency injection).
- (3) Op basis van de data in de stub moet de verhouding 0.25 zijn.

Je voert de unit test uit.

Deze mislukt, omdat LandService nog geen echte code bevat.

Je schrijft nu echte code in de method

FindVerhoudingOppervlakteLandTovOppervlakteAlleLanden in de class LandService

```
public decimal VerhoudingOppervlakteLandTovOppervlakteAlleLanden(string landcode)
{
    Land land = landDAO.Read(landcode);
    int oppervlakteAlleLanden = landDAO.OppervlakteAlleLanden();
    return (decimal) land.Oppervlakte / oppervlakteAlleLanden;
}
```

Je voert de unit test opnieuw uit. Deze lukt.

Je kan dus je class LandService schrijven én testen, terwijl de class LandDAOImpl nog niet geschreven is door Philippe.



Taak Stub: zie takenbundel



## 8 MOCK

### 8.1 Algemeen

Een mock is een stub met twee extra eigenschappen

- Resultaat van method oproep
- Verificaties

#### 8.1.1 Resultaat van method oproep

Iedere oproep van een method van een stub geeft eenzelfde resultaat.

Als je op de `LandDAOStub` de method `ReadLand("")` oproept, krijg je een `Land` object terug. Bij bepaalde testen zou het interessant zijn dat je dan `null` terugkrijgt.

Bij een mock kan het resultaat van een method oproep variëren, afhankelijk van de parameterwaarden die je meegeeft bij de method oproep.

#### 8.1.2 Verificaties

Nadat je in een testmethod een method hebt opgeroepen van de te testen class, kan je bij een mock verifiëren of deze class zijn dependency aangesproken heeft

```
public void VerhoudingOppervlakteLandTovOppervlakteAlleLanden() {
    Assert.AreEqual(0.25m,
        landService.VerhoudingOppervlakteLandTovOppervlakteAlleLanden("B"));
    // hier gaan we straks verifiëren of landService de methods
    // read("B") en OppervlakteAlleALanden() heeft opgeroepen op landDAO.
}
```

### 8.2 Moq

#### 8.2.1 Algemeen

Als je de twee extra eigenschappen van een mock zelf programmeert in een class (bijvoorbeeld de class `LandDAOStub`), moet je meer en meer code schrijven. Dit is tijdrovend en er is een kans dat je fouten schrijft in deze code.

Je lost dit op door een mocking library te gebruiken.

Een mocking library maakt een class aan die een mock voorstelt en maakt ook een mock: een object van deze class. Je kan deze mock daarna gebruiken in je test.

Er bestaan meerdere mocking libraries in .net.

Moq is één van de elegantste en veel gebruikte mocking libraries in .net.

#### 8.2.2 De Moq library toevoegen

- Je kiest `TOOLS`, `Library Package Manager`, `Manage NuGet Packages for Solution`
- Je kiest links `Online`.
- Je tikt rechtsboven (in `Search Online`) `moq`
- Je kiest in het middendeel `Install` bij `Moq`
- Je verwijdert het vinkje bij het project `TDDCursusLibrary` en je kiest `OK`
- Je kiest `Close`

#### 8.2.3 Een mock aanmaken

Je vervangt in `LandServiceTest` de stub door een mock, aangemaakt door `Mockito`.

Je voegt boven in de source volgend regel toe

```
using Moq;
```

Je voegt een private variabele toe:

```
private Mock<ILandDAO> mockFactory;
```

Je vervangt in de method `Initialize` de opdracht

```
landDAO = new LandDAOStub();
```

door

```
mockFactory = new Mock<ILandDAO>();
```

(1)

```
landDAO = mock.Object;
```

(2)

(1) Een `Moq` object zal bij (2) een mock object voor je aanmaken.

Je geeft tussen `<>` de interface mee die de mock bij (2) moet implementeren.

(2) Je vraagt een mock object met de property `Object`.

### 8.2.4 De mock trainen

De class waar de variabele `landDAO` naar verwijst, is gebaseerd op een class die `Moq` maakte. Deze aangemaakte class implementeert de interface `LandDAO`.

Deze aangemaakte class bevat dus de methods beschreven in deze interface.

Deze method implementaties zijn zeer eenvoudig.

Je ziet hier onder welke waarde ze teruggeven.

Deze waarde hangt af van het returntype van de method in de interface.

Returntype method in de interface	Waarde teruggeven door method in de "mock" class
Bool	false
byte, short, int, long, float, double, decimal	0 (nul)
Een class of interface	null
Void	niets

De aangemaakte class die de interface `LandDAO` implementeert bevat dus

- de method `Read`, die `null` teruggeeft
- de method `FindOppervlakteAlleLanden`, die `0 (nul)` teruggeeft

Deze waarden zijn zelden bruikbaar in je unit test.

Als je bijvoorbeeld `LandServiceTest` uitvoert, krijg je een `NullReferenceException`.

Gelukkig kan je de class trainen. Je geeft hierbij een waarde aan die een method van de aangemaakte class moet teruggeven.

Je doet dit met extra opdrachten in de method `Initialize`, na de opdracht `landDAO = mock.Object;`

```
mockFactory.Setup(eenLandDAO => eenLandDAO.OppervlakteAlleLanden()).Returns(20);(1)  
mockFactory.Setup(eenLandDAO => eenLandDAO.Read("B")).Returns(new Land {  
    Landcode = "B", Oppervlakte = 5 });(2)
```

(1) Je traint de class met de method `Setup` van de class `Mock`.

Je geeft als parameter een lambda expressie mee.

De parameter van deze expressie verwijst naar een `LandDAO` mock object.

De return waarde van de lambda is de method die je wilt trainen.

De method `Setup` geeft je een object terug.

Je roept daarop de method `Returns` op. Je geeft de waarde mee die de method (die het resultaat is van de lambda expressie) moet teruggeven.

De mock zal dus de waarde 20 teruggeven als je er de method `OppervlakteAlleALanden` op uitvoert.

- (2) Je traint de mock zodat hij een `Land` object met een landcode "B" en een oppervlakte 5 teruggeeft als je op de mock de method `Read` oproept met een parameter "B".

Als je de unit test nu uitvoert, slaagt de test.

☞ **Opmerking 1**

Je kan de aangemaakte class trainen, zodat haar methods verschillende waarden teruggeven, afhankelijk van de parameterwaarden die ze binnenkrijgen. Als je in de method `Initialize` volgende opdracht toevoegt

```
mockFactory.Setup(eenLandDAO => eenLandDAO.Read("NL")).Returns(new Land {
    Landcode = "NL", Oppervlakte = 6 });
```

zal de method `read` een ander `Land` object teruggeven, naargelang je de landcode "B" of "NL" meegeeft als parameter.

☞ **Opmerking 2**

Je kan de aangemaakte class trainen, zodat haar methods een exception werpen als ze een bepaalde parameterwaarde binnenkrijgen:

```
mockFactory.Setup(eenLandDAO => eenLandDAO.Read(string.Empty)).Throws(
    new ArgumentException());
```

## 8.2.5 Verificaties

Je kan verifiëren of de te testen class zijn dependencies heeft opgeroepen tijdens het uitvoeren van zijn methods.

De `LandService` method `findVerhoudingOppervlakteLandTovOppervlakteAlleLanden` moet op zijn `LandDAO` dependency de methods `OppervlakteAlleLanden` en `Read("B")` oproepen. Zoniet is deze method verkeerd geschreven.

Deze verificaties zijn ingebouwd in de mocks die Moq aanmaakt.

Je voegt volgende opdrachten toe als laatste in de test method

```
findVerhoudingOppervlakteLandTovOppervlakteAlleLanden
```

```
mockFactory.Verify(eenLandDAO => eenLandDAO.OppervlakteAlleLanden());           (1)
mockFactory.Verify(eenLandDAO => eenLandDAO.Read("B"));
```

- (1) Je verifieert met de method `verify`. Je geeft als parameter een lambda expressie mee. De parameter van deze lambda expressie staat voor een mock object. De return waarde van de expressie is de method die gedurende de test zou moeten opgeroepen worden. Als dit niet het geval is, mislukt de test.



Taak Mock: zie takenbundel

## 9 COLOFON

**Sectorverantwoordelijke:** Ortaire Uyttersprot

**Cursusverantwoordelijke:** Jean Smits

**Medewerkers:** Hans Desmet

**Versie:** 13/9/2013

**Nummer dotatielijst:**