

# Unit Testing en TDD met NUnit in C#

## Inhoud

<b>1</b>	<b>Inleiding</b>	<b>3</b>
1.1	Doelstelling	3
1.2	De noodzaak van testen	3
1.3	Geautomatiseerde testen	3
1.4	Unit tests en integration tests	4
1.5	Maken van een test project in een solution	4
<b>2</b>	<b>TDD – Test Driven Development</b>	<b>4</b>
2.1	Een voorbeeldproject	4
2.2	De te testen class	4
2.3	Een unit test schrijven	5
2.4	Een unit test uitvoeren	7
2.5	Eén TestMethod uitvoeren	9
2.6	De unit test als documentatie	9
2.7	De methods van de class Assert	9
<b>3</b>	<b>TDD: Eerst tests schrijven, daarna implementatie</b>	<b>10</b>
3.1	Voordelen van TDD	11
3.2	De te testen class	11
3.3	Voorbeeld	12
3.4	De unit test schrijven	12
3.5	De code van de te testen class schrijven	13
3.6	Samenvatting van de stappen bij Test Driven Development	13
<b>4</b>	<b>Test Fixtures</b>	<b>14</b>
4.1	SetUp	14
<b>5</b>	<b>Wat wel of niet testen?</b>	<b>16</b>
5.1	Exceptions testen	16
5.2	Grenswaarden en extreme waarden testen	17
5.2.1	Een eerste voorbeeld	17
5.2.2	De te testen klasse	17
5.2.3	De unittest	18
5.2.4	Een eerste implementatie	18

5.2.5	Een tweede voorbeeld.....	19
5.3	De unittest .....	20
5.4	Een eerste implementatie.....	20
5.5	Testen bijhouden .....	21
6	Dependencies.....	21
6.1	Dependency .....	21
6.1.1	Een voorbeeld .....	22
6.2	Problemen bij het testen van een technische class met dependencies .....	23
6.3	De dependency uitdrukken in een interface .....	24
6.4	Dependency injection .....	24
7	Stub.....	25
8	Mock.....	27
8.1	Resultaat van methode-aanroep.....	27
8.2	Verificaties .....	27
8.3	Moq.....	27
8.4	De Moq library toevoegen .....	28
8.5	Een mock aanmaken .....	28
8.6	De mock trainen.....	29
8.7	Verificaties .....	30
9	Referenties .....	31

# 1 Inleiding

## 1.1 Doelstelling

In deze cursus leer je code testen in Visual Studio en software ontwikkelen volgens het Test Driven Development principe.

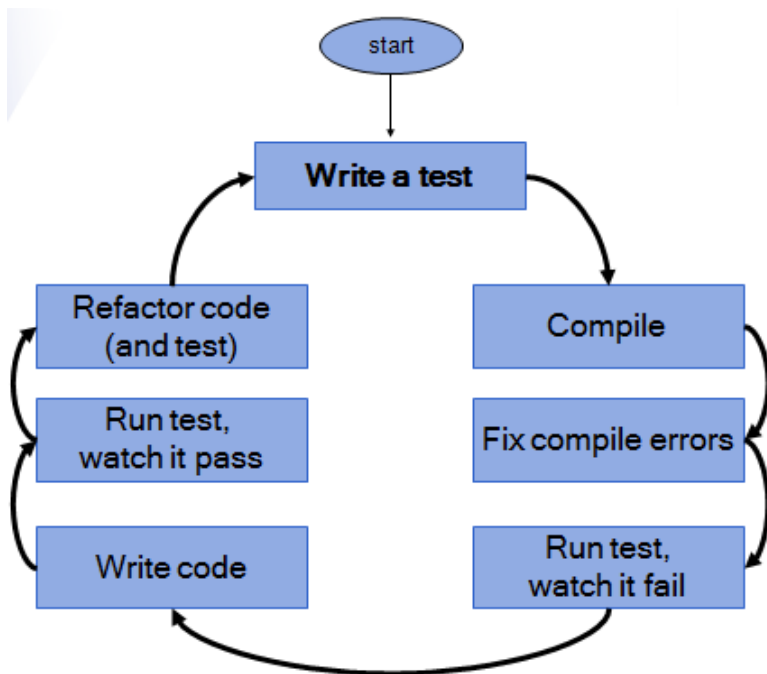
## 1.2 De noodzaak van testen

Uit wetenschappelijke studies 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



**Bron:** <https://www.codeproject.com/Articles/162041/Introduction-to-NUnit-and-TDD>

## 1.3 Geautomatiseerde 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 die de goede werking van een andere code test.

## 1.4 Unit tests en integration tests

Een class methode waarvan je de code wil testen heet een unit. Standaard wordt per class een TestClass geschreven die de methoden van de class worden getest.

Wanneer je een class ontwikkelt, kan je al de unit tests voor die class schrijven en uitvoeren. Op deze manier krijg je snel feedback over fouten in de class. Als de class volledig geschreven is, is ze ook volledig getest. De kans is klein dat ze daarna nog fouten bevat. Je zal zo minder moeten debuggen. Dit is voordelig: debuggen is een tijdrovende, vervelende taak.

Een integration test is een test waarin je de samenwerking van meerdere units test. In het onderdeel "Klant toevoegen" van een website werken deze units samen:

- Een action methode KlantToevoegen in een controller class
- Een class KlantenRepository 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 mbv **NUnit**

## 1.5 Maken van een test project in een solution

De unit tests komen in dezelfde solution als de te testen code, maar bevinden zich in een **apart test project** binnen die solution. Zo'n test project kan een verschillend test-framework gebruiken. Veel gebruikte frameworks voor unit testing in VS zijn **NUnit**, **xUnit** en **MSUnit**.

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

# 2 TDD – Test Driven Development

## 2.1 Een voorbeeldproject

We starten met het aanmaken van een nieuwe **.NET Core Class Library**.

- Start Visual Studio en maak een nieuw project.
- Kies een .NET Core Class Library en klik Next.
- Noem het project **TDDCursusLibrary**, de solution **TDDCursusSolution** en klik op Create.

## 2.2 De te testen class

In het project willen we een class Jaar testen:

Een private variabele jaar kan via een constructor ingevuld worden met een bepaald jaartal. Een property IsSchrikkeljaar is true als het jaar een schrikkeljaar is. Zoniet is het false.

De property IsSchrikkeljaar is gebaseerd op volgende schrikkeljaarregels:

- Een jaar deelbaar door 4 is een schrikkeljaar.
- Een jaar deelbaar door 100 is geen schrikkeljaar,...
- ...tenzij het deelbaar is door 400. Dat is wél een schrikkeljaar.

Je voegt de class Jaar nu toe aan het project.

- Klik met de rechtermuistoets op het project TDDCursusLibrary.
- Kies Add, Class...
- Noem de class Jaar en klik op Add.
- Vul de class als volgt aan:

```
public class Jaar
{
    private readonly 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;
            //of via de methode: return DateTime.IsLeapYear(jaar);
        }
    }
}
```

Om de code te testen **zonder** unit test heb je de volgende **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.
- De user interface moet klaar zijn vooraleer je kan testen.

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.

## 2.3 Een unit test schrijven

Je voegt een test project toe aan de solution. In dat project maak je een unit test JaarTest. Je test daarmee de werking van de class Jaar.

- Klik in de Solution Explorer met de rechtermuistoets op de solution en kies Add, New Project...
- Tik bovenaan in het zoekveld test.

- Kies dan voor de template C# NUnit Test Project (.NET Core) en klik op Next.
- Geef het project de naam **TDDCursusLibraryTest** en je klikt op Create.
- Je hernoemt UnitTest1.cs naar JaarTest.cs.

Je legt nu een project reference vanuit het testproject naar de library.

- Klik in de Solution Explorer met de rechtermuistoets op het testproject en kies AddReference...

- Vink het project TDDCursusLibrary aan en kies OK.

Nu kan je de eerste unittest schrijven in TDDCursusLibraryTest.

- Wijzig de source van **JaarTest.cs** als volgt:

```
[TestFixture]
public class JaarTest // (1)
{
    [Test] // (2)
    public void EenJaarDeelbaarDoor400IsEenSchrikkeljaar() // (3)
    {
        Assert.AreEqual(true, new Jaar(2000).IsSchrikkeljaar);
        // Assert.IsTrue(new Jaar(2000).IsSchrikkeljaar); // (4)
    }
    [Test]
    public void EenJaarDeelbaarDoor100MaarNietDoor400IsGeenSchrikkeljaar()
    {
        Assert.AreEqual(false, new Jaar(1900).IsSchrikkeljaar);
        // Assert.IsFalse(new Jaar(1900).IsSchrikkeljaar);
    }
    [Test]
    public void EenJaarDeelbaarDoor4IsEenSchrikkeljaar()
    {
        Assert.AreEqual(true, new Jaar(2012).IsSchrikkeljaar);
        // Assert.IsTrue(new Jaar(2012).IsSchrikkeljaar);
    }
    [Test]
    public void EenJaarNietDeelbaarDoor4IsGeenSchrikkeljaar()
    {
        // Assert.AreEqual(false, new Jaar(2015).IsSchrikkeljaar);
        Assert.IsFalse(new Jaar(2015).IsSchrikkeljaar);
    }
}
```

- (1) Voor de class methoden die moeten worden getest, maak je een Test class. De naamconventie voor de test class is dat de naam gelijk is aan de te testen class, gevolgd door Test.
- (2) Je schrijft één test als één method. Je schrijft `[Test]` bij deze method. Visual Studio aanziet enkel methoden voorzien van het attribuut `[Test]` als tests.
- (3) Een methode die een test voorstelt is public, heeft als return type void en heeft geen parameters. Je kan de naam van de methode 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 (3) duidt aan dat je in deze method test dat een jaar dat deelbaar is door 400 een schrikkeljaar is.

- (4) Je maakt met `new Jaar(2000)` een object van de te testen class. Je zorgt ervoor 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! Je haalt de waarde op van de property die je wil testen: `IsSchrikkeljaar`.
- (5) Als de property `IsSchrikkeljaar` correct geschreven is, is de waarde `true`. De class **Assert** bevat enkele static methods waarmee je dit controleert. De meest gebruikte methoden zijn de methoden **AreEqual**, **IsTrue** en **IsFalse**. Je geeft aan deze method twee waarden mee: de waarde die je verwacht (**true of false**) en de echte waarde die je krijgt bij de propertyoproep. Als deze twee waarden aan mekaar gelijk zijn, zal Visual Studio deze test aanzien als een test die correct verlopen is. Zoniet dan zal Visual Studio deze test aanzien als een test die verkeerd loopt.

De drie overige testmethods voeren nog meer tests uit.

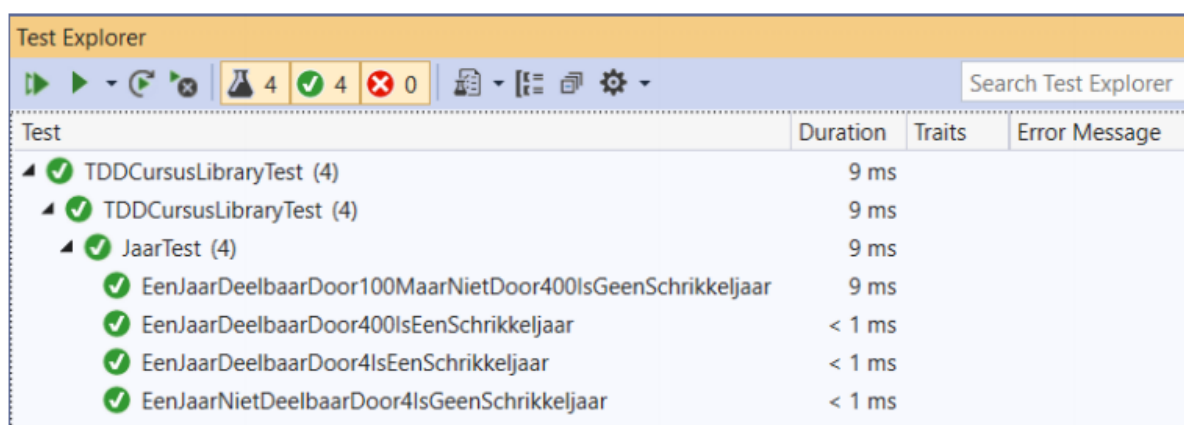
## 2.4 Een unit test uitvoeren

We laten Visual Studio nu de unit test uitvoeren. Visual Studio voert hierbij alle testmethoden uit.

- Kies in het menu Test, Run All Tests.

Visual Studio voert de unit test uit en toont een rapport in een venster Test Explorer. Als dit venster niet zichtbaar is dan kies je in het menu voor View – Test Explorer. Je pint het venster eventueel vast aan de linkerkant van Visual Studio.

- Klap in de Test Explorer de test `TDDCursusLibrary` én de onderverdelingen eronder verder uit zodat je onderstaand beeld krijgt:



Test	Duration	Traits	Error Message
✓ TDDCursusLibraryTest (4)	9 ms		
✓ TDDCursusLibraryTest (4)	9 ms		
✓ JaarTest (4)	9 ms		
✓ EenJaarDeelbaarDoor100MaarNietDoor400IsGeenSchrikkeljaar	9 ms		
✓ EenJaarDeelbaarDoor400IsEenSchrikkeljaar	< 1 ms		
✓ EenJaarDeelbaarDoor4IsEenSchrikkeljaar	< 1 ms		
✓ EenJaarNietDeelbaarDoor4IsGeenSchrikkeljaar	< 1 ms		

Een groen vinkje in de Test Explorer duidt aan dat een test geslaagd is. Ook in de sourcecode van de testmethods zie je bij een geslaagde test een groen vinkje:

```
[Test]
0 references
public void EenJaarDeelbaarDoor100MaarNietDoor400IsGeenSchrikkeljaar()
{
    Assert.AreEqual(false, new Jaar(1900).IsSchrikkeljaar);
    //Assert.IsFalse(new Jaar(1900).IsSchrikkeljaar);
}
```

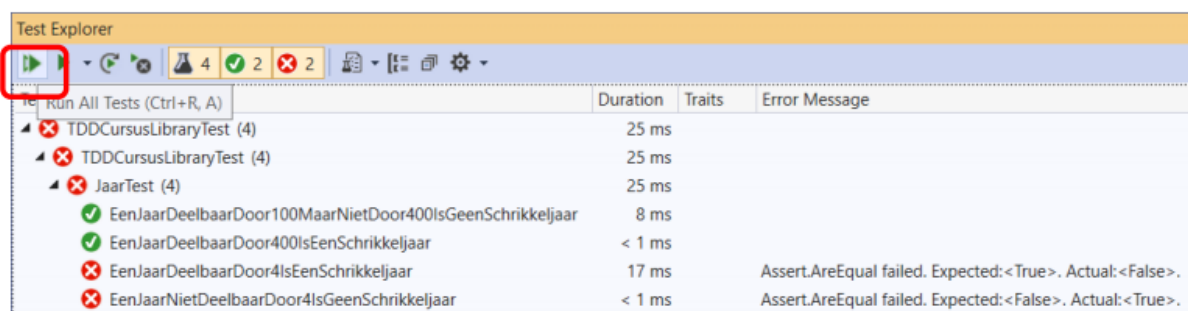
Bemerk dat de testmethods niet noodzakelijk worden uitgevoerd in de volgorde zoals ze geschreven zijn in de unit test JaarTest. De volgorde is onbepaald.

We tikken nu een fout in de property IsSchrikkeljaar om te zien hoe Visual Studio reageert bij het testen.

- Vervang in de class Jaar in het laatste returnstatement de 4 door een 5:

```
public class Jaar {
    ...
    public bool IsSchrikkeljaar {
        get {
            if (jaar % 400 == 0) return true;
            if (jaar % 100 == 0) return false;
            return jaar % 5 == 0;
        }
    }
}
```

- Bewaar de wijziging en voer de tests opnieuw uit door bijvoorbeeld in de Test Explorer te klikken op de knop Run All Tests.



Een rood vinkje duidt aan dat een test mislukt is.

- Je corrigeert de fout in de property IsSchrikkeljaar door de 5 terug te vervangen door een 4.
- Je voert de unit test opnieuw uit. Alle tests slagen.

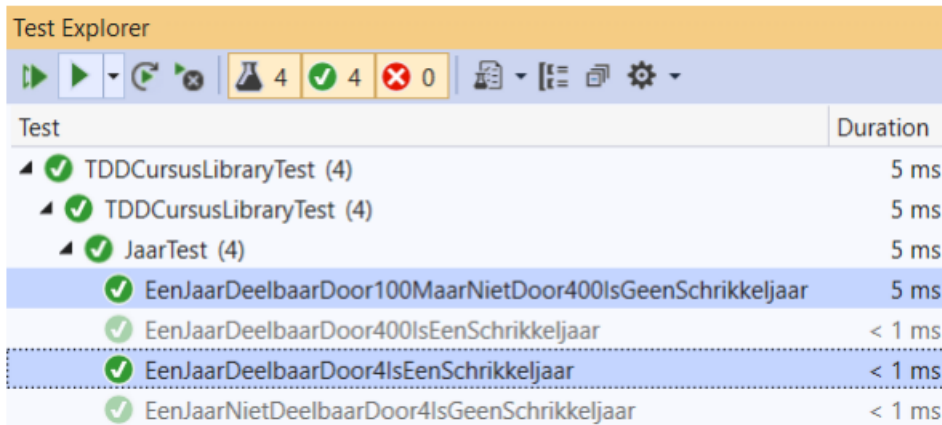
Je ziet met deze voorbeelden al hoe een programmeur bij TDD werkt. Iedere keer hij een method van een class wijzigt, voert hij de unit test van die class uit. Op deze manier krijgt hij direct feedback of die wijziging correct was. Code schrijven en hierover onmiddellijk feedback krijgen is productief.



## 2.5 Eén TestMethod uitvoeren

Je voerde tot nu toe alle TestMethods van een unit test uit. Je kan ook één TestMethod uitvoeren.

- Klik in de Test Explorer met de rechtermuistoets op één of meerdere TestMethods en kies Run om één enkele test te runnen:



Test	Duration
✓ TDDCursusLibraryTest (4)	5 ms
✓ TDDCursusLibraryTest (4)	5 ms
✓ JaarTest (4)	5 ms
✓ EenJaarDeelbaarDoor100MaarNietDoor400IsGeenSchrikkeljaar	5 ms
✓ EenJaarDeelbaarDoor400IsEenSchrikkeljaar	< 1 ms
✓ EenJaarDeelbaarDoor4IsEenSchrikkeljaar	< 1 ms
✓ EenJaarNietDeelbaarDoor4IsGeenSchrikkeljaar	< 1 ms

## 2.6 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.



Als ontwikkelaar Joke de class Jaar en de unittest JaarTest geschreven heeft, kan haar collega Jos de werking van de class Jaar leren kennen door het rapport van de bijbehorende unittest JaarTest in te kijken. Hij ziet dat een jaar dat deelbaar is door 400 een schrikkeljaar is. Dit lukt enkel als de namen van de TestMethoden in "mensentaal" geschreven zijn.

## 2.7 De methods van de class Assert

Je gebruikte tot nu in de test methoden de methoden AreEqual, IsTrue en IsFalse 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, bv:

```
Assert.AreNotEqual(false, new Jaar(2000).IsSchrikkeljaar);
```

- IsNull(teTestenExpressie)

De test lukt als teTestenExpressie gelijk is aan null, bv

```
Assert.IsNull(variable);
```

- `IsNotNull(teTestenExpressie)`

De test lukt als `teTestenExpressie` verschilt van null, bv

```
Assert.IsNotNull(new Jaar(2000));
```

- `AreSame(verwachteReferenceVariabele, teTestenReferenceVariabele)`

De test lukt als `teTestenReferenceVariabele` naar hetzelfde object wijst als `verwachteReferenceVariabele`, bv

```
Jaar jaar1 = new Jaar(2000);
Jaar jaar2 = jaar1;
Assert.AreSame(jaar1, jaar2);
```

- `AreNotSame(verwachteReferenceVariabele, teTestenReferenceVariabele)`

De test lukt als `teTestenReferenceVariabele` niet naar hetzelfde object wijst als `verwachteReferenceVariabele`, bv

```
Jaar jaar1 = new Jaar(2000);
Jaar jaar2 = new Jaar(2000);
Assert.AreNotSame(jaar1, jaar2);
```

Een handig overzicht van alle Assert methoden voor NUnit, xUnit en MSUnit vind je hier:

<https://www.automatetheplanet.com/wp-content/uploads/2018/06/NUnit-Unit-Testing-Framework-Cheat-Sheet.pdf>

### 3 TDD: Eerst tests schrijven, daarna implementatie

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

In dit hoofdstuk wijzigt deze volgorde

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

Deze nieuwe volgorde wordt aangeraden bij **TDD of Test Driven Development**.

#### Opmerkingen:

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

### 3.1 Voordelen van TDD

Eerst de tests schrijven en dan pas de implementatie van de class biedt een aantal voordelen.

Je schrijft de testen op basis van de analyse van de class. In de analyse van de class Jaar bleken onderstaande regels, die je uitschrijft als bijbehorende test methoden:

- Als een jaar deelbaar is door 4 is het een schrikkeljaar

```
public void EenJaarDeelbaarDoor4IsEenSchrikkeljaar()
```

```
public void EenJaarNietDeelbaarDoor4IsGeenSchrikkeljaar()
```

- Als een jaar deelbaar is door 100 maar niet door 400 is het geen schrikkeljaar

```
public void EenJaarDeelbaarDoor100MaarNietDoor400IsGeenSchrikkeljaar()
```

- Als een jaar deelbaar is door 400 is het wel 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 Test Methoden de methoden op van de te testen class. Als deze method-aanroepen vanuit de Test Methoden vreemd zijn, wegens een verkeerd gekozen method-naam, te veel of te weinig parameters, ... kan je snel de signatuur van deze methods wijzigen, omdat ze enkel vanuit de Test Methoden 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 unittest. Je schrijft daarom een minimale versie van de te testen class: de class zelf en de methods van de class zonder verdere inhoud.

Sommige methods van de te testen class zijn geen void methods maar geven een waarde (een int, een string, een decimal,...) terug. Als zo een 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 hiervoor 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 (geïmplemented) is.

### 3.3 Voorbeeld

We passen deze nieuwe werkwijze toe van TDD in een voorbeeld. We maken onderstaande class.

- Maak de class Rekening zoals hieronder weergegeven:

```
public class Rekening
{
    private decimal saldo;
    public void Storten(decimal bedrag)
    {
        //throw new NotImplementedException();
    }
    public decimal Saldo
    {
        get
        {
            //throw new NotImplementedException();
        }
    }
}
```

### 3.4 De unit test schrijven

Je schrijft nu de unittest 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 test class RekeningTest.

- Voeg onderstaande test class toe in het testproject:

```
[TestFixture]
public class RekeningTest
{
    private Rekening rekening;

    [SetUp]
    public void Initialize()
    {
        rekening = new Rekening();
    }
    [Test]
    public void HetSaldoVanEenNieuweRekeningIsNul()
    {
        //var rekening = new Rekening();
        Assert.AreEqual(decimal.Zero, rekening.Saldo);
    }
}
```

```
[Test]
public void HetSaldoNaEenEersteStortingIsHetBedragVanDieStorting()
{
    //var rekening = new Rekening();
    var bedrag = 2.5m;
    rekening.Storten(bedrag);
    Assert.AreEqual(bedrag, rekening.Saldo);
}
}
```

Als volgende stap in het TDD-proces controleer je 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.

✗ TDDCursusLibraryTest (7)	25 ms
✗ TDDCursusLibraryTest (7)	25 ms
✓ JaarTest (4)	11 ms
✗ RekeningTest (3)	14 ms
✗ HetSaldoNaEenEersteStortingIsHetBedragVanDieStorting	14 ms
✗ HetSaldoNaTweeStortingenisDeSomVanDeBedragenVanDieSt...	< 1 ms
✗ HetSaldoVanEenNieuweRekeningIsNul	< 1 ms

### 3.5 De code van de te testen class schrijven

Je schrijft nu code in de methoden van de class Rekening. De eerste versie van die code moet niet noodzakelijk de optimaalste code zijn naar performantie, geheugengebruik, leesbaarheid, onderhoudbaarheid,... toe.

- Wijzig de code in de class Rekening als volgt:

```
public class Rekening
{
    private decimal saldo;
    public void Storten(decimal bedrag)
    {
        //throw new NotImplementedException();
        if (bedrag <= decimal.Zero)
        {
            throw new ArgumentException();
        }
        saldo += bedrag;
    }
    public decimal Saldo
    {
        get
        {
            //throw new NotImplementedException();
            return saldo;
        }
    }
}
```

- Voer nu de tests uit de bijhorende unittest uit. Alle tests slagen.

### 3.6 Samenvatting van de stappen bij Test Driven Development

- Je analyseert de te schrijven class.
- Je schrijft de class en zijn methods. Deze 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.

## 4 Test Fixtures

Je maakt regelmatig in meerdere TestMethods van eenzelfde unit test soortgelijke objecten aan. Je initialiseert in RekeningTest bijvoorbeeld in iedere TestMethods eenzelfde Rekening-object:

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

Zo'n object heet een test fixture.

### 4.1 SetUp

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 de annotation [TestInitialize] tikt. De naam van de method is vrij te kiezen

In RekeningTest wordt dit dus een methode:

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

Visual Studio voert voor iedere Test Methode de TestInitialize-method van dezelfde unit test uit.

In de tests van RekeningTest betekent dit het volgende :

1. Initialize()
2. HetSaldoNaTweeStortingenIsDeSomVanDeBedragenVanDieStortingen()
3. Initialize()
4. HetSaldoNaEenEersteStortingIsHetBedragVanDieStorting()
5. Initialize()

## 6. HetSaldoVanEenNieuweRekeningIsNul()

Door het gebruik van de method `Initialize()` hoef je binnen de testmethod geen Rekening-object meer te initialiseren maar kan je de private variabele rekening gebruiken. Visual Studio initialiseert deze variabele bij iedere oproep van de method voorzien van een `TestInitialize`-annotation. Dit betekent ook dat je de private variabele rekening mag wijzigen in een testmethod. Bij de volgende testmethod bevat de variabele rekening terug een vers geïntialiseerd Rekening-object.

De volledige test-class `RekeningTest` ziet er nu zo uit:

```
public class RekeningTest
{
    private Rekening rekening;

    [SetUp]
    public void Initialize()
    {
        rekening = new Rekening();
    }
    [Test]
    public void HetSaldoVanEenNieuweRekeningIsNul()
    {
        //var rekening = new Rekening();
        Assert.AreEqual(decimal.Zero, rekening.Saldo);
    }
    [Test]
    public void HetSaldoNaEenEersteStortingIsHetBedragVanDieStorting()
    {
        //var rekening = new Rekening();
        var bedrag = 2.5m;
        rekening.Storten(bedrag);
        Assert.AreEqual(bedrag, rekening.Saldo);
    }
    [Test]
    public void HetSaldoNaTweeStortingenIsDeSomVanDeBedragenVanDieStortingen()
    {
        //var rekening = new Rekening();
        rekening.Storten(2.5m);
        rekening.Storten(1.2m);
        Assert.AreEqual(3.7m, rekening.Saldo);
    }
    [Test]
    public void HetBedragVanEenStortingMagNietNulZijn()
    {
        Assert.Throws<ArgumentException>(() => rekening.Storten(decimal.Zero));
    }
    [Test]
    public void HetBedragVanEenStortingMagNietNegatiefZijn()
    {
        Assert.Throws<ArgumentException>(() => rekening.Storten(-1m));
    }
}
```

- Je kan eventueel de unit test opnieuw uitvoeren.

Naast een `TestInitialize` bestaat er ook een `TestCleanup`. Een method die voorzien is van een annotation `[TestCleanup]` wordt automatisch uitgevoerd na iedere testmethod. Je hebt zo'n method enkel nodig als je in de `TestInitialize`-method iets aanmaakt buiten het RAM-

geheugen (bijvoorbeeld een bestand), dat je tijdens de testmethod gebruikt en dat je na de testmethod wil verwijderen.

- Commit jouw sources en publiceer deze op jouw remote repository.

## 5 Wat wel of niet testen?

Wanneer iets fout loopt in een constructor of een method – er wordt bijvoorbeeld een verkeerde waarde meegegeven – dan kan je een exception werpen. Je kan testen of dit ook effectief gebeurt.

### 5.1 Exceptions testen

Uit de analyse van de class Rekening blijkt dat het bedrag dat je stort een positief getal moet zijn. Zoniet moet de method Storten een ArgumentException werpen.

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

- Voeg onderstaande code toe in RekeningTest:

```
[Test]
public void HetBedragVanEenStortingMagNietNulZijn()
{
    Assert.Throws<ArgumentException>(() => rekening.Storten(decimal.Zero));
}
[Test]
public void HetBedragVanEenStortingMagNietNegatiefZijn()
{
    Assert.Throws<ArgumentException>(() => rekening.Storten(-1m));
}
```

(1) Een testmethod waarvan je verwacht dat deze een exception zal throwen geef je de annotation ExpectedException met als parameter het type van de exception. Enkel als deze exception optreedt tijdens het uitvoeren van de testmethod is deze test geslaagd.

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

- Voer de test uit. De twee extra testmethods mislukken. Dit betekent dat je extra code moet schrijven in de class Rekening om aan deze extra vereiste uit de analyse te voldoen.
- Wijzig in de class Rekening de method Storten:

```
public void Storten(decimal bedrag)
{
    //throw new NotImplementedException();
    if (bedrag <= decimal.Zero)
    {
        throw new ArgumentException();
    }
    saldo += bedrag;
}
```

- Voer de test RekeningTest opnieuw uit. De test slaagt. De code in de class Rekening voldoet dus aan de extra vereiste uit de analyse.



## 5.2 Grenswaarden en extreme waarden testen

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

### 5.2.1 Een eerste voorbeeld

De class Rekeningnummer stelt een Belgisch bankrekeningnummer voor.

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.
- Tussen het 3e en 4e cijfer en tussen het 10e en 11e cijfer komt een – teken. Je moet dus een rekeningnummer testen met een eerste grenswaarde : nummer mét streepjes en een rekeningnummer 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. Als de modulus 0 is dan zijn de laatste 2 cijfers 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 niet klopt.

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

### 5.2.2 De te testen klasse

Je schrijft de class Rekeningnummer.

- Voeg onderstaande class Rekeningnummer toe:

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

### 5.2.3 De unittest

Je maakt een nieuwe testclass : RekeningnummerTest.

- Maak een nieuwe class RekeningnummerTest en voorzie deze van volgende code:

```
public class RekeningnummerTest
{
    [Test]
    public void NummerMet12CijfersMetCorrectControleIsOK()
    {
        // dit nummer mag geen exception veroorzaken
        new Rekeningnummer("063-1547564-61");
    }
    [Test]
    public void NummerMet12CijfersMetVerkeerdeControleIsVerkeerd()
    {
        Assert.Throws<ArgumentException>(() => new Rekeningnummer("063-1547564-62"));
    }
    [Test]
    public void NummerMet12CijfersZonderStreepjesMetCorrectControleIsVerkeerd()
    {
        Assert.Throws<ArgumentException>(() => new Rekeningnummer("063154756461"));
    }
    [Test]
    public void NummerMet13CijfersIsVerkeerd()
    {
        Assert.Throws<ArgumentException>(() => new Rekeningnummer("063-1547564-623"));
    }
    [Test]
    public void NummerMet11CijfersIsVerkeerd()
    {
        Assert.Throws<ArgumentException>(() => new Rekeningnummer("063-1547564-6"));
    }
    [Test]
    public void LeegNummerIsVerkeerd()
    {
        Assert.Throws<ArgumentException>(() => new Rekeningnummer(string.Empty));
    }
    [Test]
    public void NummerMetNullIsVerkeerd()
    {
        Assert.Throws<ArgumentNullException>(() => new Rekeningnummer(null));
    }
    [Test]
    public void ToStringMoetHetNummerTeruggeven()
    {
        var nummer = "063-1547564-61";
        var rekeningnummer = new Rekeningnummer(nummer);
        Assert.AreEqual(nummer, rekeningnummer.ToString());
    }
}
```

- Voer de tests uit de test class uit. Deze mislukken aangezien de class Rekeningnummer nog niet geïmplementeerd is.

### 5.2.4 Een eerste implementatie

We voegen een eerste implementatie van de class Rekeningnummer uit.

- Wijzig de class Rekeningnummer als volgt:

```
public class Rekeningnummer
{
    private static readonly Regex regex =
        new Regex("^\\d{3}-\\d{7}-\\d{2}$"); // (1)
    private readonly string nummer;
    public Rekeningnummer(string nummer)
```

```

    {
        if (!regex.IsMatch(nummer))           //(2)
        {
            throw new ArgumentException();
        }
        var eerste10Cijfers = long.Parse(nummer.Substring(0, 3) +
            nummer.Substring(4, 7));
        var laatste2Cijfers = long.Parse(nummer.Substring(12, 2));
        var rest = eerste10Cijfers % 97L;
        if (rest == 0) rest = 97;
        if (rest != laatste2Cijfers)
        {
            throw new ArgumentException();
        }
        this.nummer = nummer;
    }
    public override string ToString()
    {
        return nummer;
    }
}

```

(1) Dit object stelt een regular expression voor. Meer informatie en voorbeelden over gebruik van RegEx en Regular expressions vind je hier: <https://www.c-sharpcorner.com/UploadFile/955025/regular-expression-in-C-Sharp/>

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

- Voer de tests uit de testclass `RekeningnummerTest` uit. Alle tests zouden moeten slagen. Dit betekent dat de class `Rekeningnummer` voldoet aan de vereisten van de analyse.

### 5.2.5 Een tweede voorbeeld

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
- De waarde `null` i.p.v. een verzameling

Als voorbeeld beschouwen we een class `Statistiek` met een static method `Gemiddelde`. Je geeft aan deze method een array van decimals mee. Je krijgt het gemiddelde van deze decimals terug.

- Maak een class `Statistiek` en voeg er volgende code aan toe:

```

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

```

## 5.3 De unittest

We maken een bijhorende unittest.

- Voeg een class StatistiekTest toe.

```
public class StatistiekTest
{
    [Test]
    public void HetGemiddeldeVan10en15is12punt5()
    {
        Assert.AreEqual(12.5m, Statistiek.Gemiddelde(
            new decimal[] { 10m, 15m }));
    }
    [Test]
    public void HetGemiddeldeVanEenGetalIsDatGetal()
    {
        var enigGetal = 1.23m;
        Assert.AreEqual(enigGetal, Statistiek.Gemiddelde(
            new decimal[] { enigGetal }));
    }
    [Test]
    public void HetGemiddeldeVanEenLegeVerzamelingKanJeNietBerekenen()
    {
        Assert.Throws<ArgumentException>(() => Statistiek.Gemiddelde(new decimal[] {
    }));
    }
    [Test]
    public void HetGemiddeldeVanNullKanJeNietBerekenen()
    {
        Assert.Throws<ArgumentNullException>(() => Statistiek.Gemiddelde(null));
    }
}
```

- Voer de tests uit. Voorlopig falen de tests nog.

## 5.4 Een eerste implementatie

We maken een eerste implementatie van de class Statistiek.

- Voeg onderstaande code toe aan de class Statistiek:

```
public 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;
        //of met de linq methode: return getallen.Average();
    }
}
```

- Voer de tests opnieuw uit. Deze keer slagen ze. Dit betekent dat de class Statistiek voldoet aan de vereisten van de analyse.
- Commit de sources en publiceer op jouw remote repository.

## 5.5 Testen bijhouden

Je verwijdt 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.

## 6 Dependencies

### 6.1 Dependency

Als een class A methods oproept van een class B, dan heeft de class A een dependency ("afhankelijkheid") op de class B. Class A is afhankelijk van class B.

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



In code :

```
public class A
{
    public void A1()
    {
        // ...
        B b = new B();
        b.B1();
        // ...
    }

    public int A2()
    {
        // ...
        B b = new B();
        b.B1();
        b.B2();
        // ...
        return 1;
    }
}

public class B
{
    public void B1()
    {
        // ...
    }

    public int B2()
    {
        // ...
        return 1;
    }
}
```

In de class A gebruiken de methods A1() en A2() methods uit 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 zoals 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 methode 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).

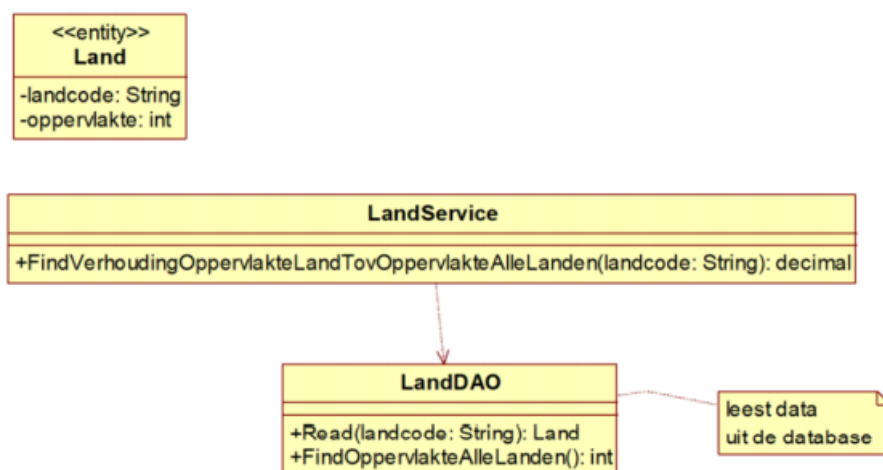
```
public class A
{
    private readonly B b = new B();

    public void A1()
    {
        // ...
        // B b = new B();
        b.B1();
        // ...
    }

    public int A2()
    {
        // ...
        // B b = new B();
        b.B1();
        b.B2();
        // ...
        return 1;
    }
}
```

### 6.1.1 Een voorbeeld

Je leert de problemen bij het testen van een class met dependencies kennen met dit voorbeeld:



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

De method `VindVerhoudingOppervlakteLandTovOppervlakteAlleLanden` 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. Deze method geeft je de detail-informatie van dat land (waaronder de oppervlakte) terug.

Je zal in deze method ook de LandDAO-method VindOppervlakteAlleLanden oproepen. Via deze method krijg je de totale oppervlakte van alle landen terug.

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

- Maak een class Land:

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

- Maak nu ook een nieuwe class LandService:

```
public class LandService
{
    public decimal VindVerhoudingOppervlakteLandTovOppervlakteAlleLanden(
        string landcode)
    {
        throw new NotImplementedException();
    }
}
```

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

Voorlopig is er nog geen probleem : de service gebruikt nog geen LandDAO-class. Wanneer we wél een LandDAO-class gebruiken stellen zich drie problemen bij het testen van de class LandService :

1. In een groot team van programmeurs kan iedere programmeur zijn "specialiteit" hebben.
  - "front-end"-programmeurs zijn gespecialiseerd in user interface-code. Ze programmeren Webpagina's, ontwerpen WPF-schermen, gebruiken HTML en CSS.
  - "back-end"-programmeurs zijn gespecialiseerd in databasetoegangcode en programmeren DAO-classes.

Het is dus mogelijk dat jij de class LandService schrijft en Jos de class LandDAO.

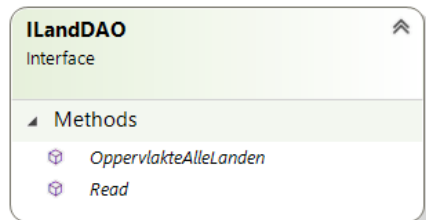
Jij kan echter de class LandService niet schrijven zolang de Jos de class LandDAO niet heeft afgewerkt. Indien Jos verlof heeft, ziek is of andere werk heeft met hogere prioriteit, kan jij niet verder werken.

2. 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 Jos. 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.
3. 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 LandService als LandDAO schrijft.

### 6.3 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.



- Maak een interface ILandDAO en voorzie deze van onderstaande code:

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

Wie deze interface ontwikkelt maakt niet uit. Dit kan de analist zijn, de front-end ontwikkelaar, de back-end ontwikkelaar of een team van deze mensen.

Als **tweede** stap gebruik je deze interface in elke class die zo'n dependency heeft. Je voegt een private variabele toe van het type ILandDAO. Voorlopig geven we deze private variabele nog geen waarde. Dit doen we in de volgende paragraaf via Dependency Injection (DI).

- Vul de class LandService als volgt aan:

In deze nieuwe versie wordt de variabele landDAO niet geïnitieerd en is die dus null.

### 6.4 Dependency injection

Bij dependency injection is het mogelijk om een dependency te injecteren via een parameter van de constructor.

- Wijzig de class LandService als volgt:

```
public class LandService
{
    private readonly ILandDAO landDAO;
    public LandService(ILandDAO landDAO)
    {
        this.landDAO = landDAO;
    }
    public decimal VindVerhoudingOppervlakteLandTovOppervlakteAlleLanden(string landcode)
    {
        throw new NotImplementedException();
    }
}
```

Je geeft bij het aanmaken van een LandService-object aan de constructor een object mee van een type dat de interface ILandDAO implementeert.



(1) Je slaat het object op in de private variabele.

Je kan nu de private variabele `landDAO` gebruiken in de methods van deze class. De class `LandService` zelf maakt geen object meer aan dat de dependency voorstelt. Het wordt aangereikt door de constructor. Dit heet dependency injection via de constructor of kortweg constructor injection.

### Opmerking :

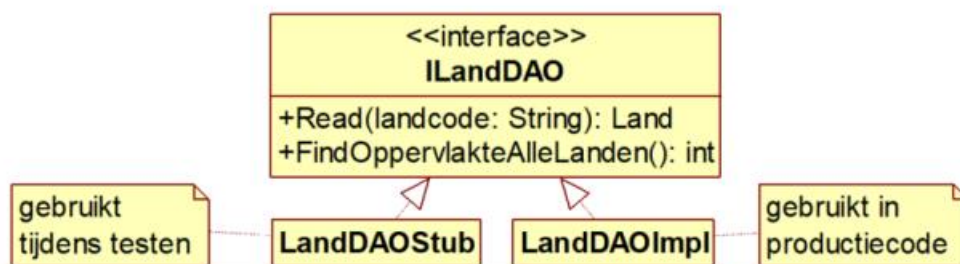
- Een class kan meerdere dependencies hebben.
- De class bevat dan per dependency een extra private readonly variabele en een extra parameter in de constructor.

## 7 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 Jos zal schrijven, zal deze interface implementeren. Jos 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 eerder beschreven in dit hoofdstuk.

Je gebruikt in de plaats een stub. Een stub is een object. De class van dit object implementeert de interface `ILandDAO`, net 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 een echte database.
- De class `LandDAOSTub` maakt dummy data in het interne geheugen. Deze data moeten geen reële data zijn. Deze data dienen 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.

We gebruiken nu een stub in het project `TDDCursusLibraryTest`.

- Voeg in het testproject een class `LandDAOSTub` toe:

```
public class LandDAOSTub : ILandDAO
```

```

{
    public Land Read(string landcode)
    {
        return new Land { Landcode = landcode, Oppervlakte = 5 };
    }
    public int OppervlakteAlleLanden()
    {
        return 20;
    }
}

```

(1) De stubmethod Read() geeft een Land-object terug met de gevraagde landcode en een fictieve oppervlakte.

(2) De stubmethod OppervlakteAlleLanden() geeft een fictieve totale oppervlakte terug.

Je maakt nu de testclass LandServiceTest.

- Voeg in het testproject onderstaande class LandServiceTest toe:

```

public class LandServiceTest
{
    private ILandDAO landDAO;
    private LandService landService;
    [SetUp]
    public void Initialize()
    {
        landDAO = new LandDAOStub(); //(1)
        landService = new LandService(landDAO); //(2)
    }
    [Test]
    public void FindVerhoudingOppervlakteLandTovOppervlakteAlleLanden()
    {
        Assert.AreEqual(0.25m, landService.
            VindVerhoudingOppervlakteLandTovOppervlakteAlleLanden("B")); (3)
    }
}

```

(1) Je maakt een stub.

(2) Je geeft deze stub mee aan de constructor van de te testen class (DI).

(3) Op basis van de data in de stub moet de verhouding 0.25 zijn.

- Voer de unit test uit. Deze mislukt want de class LandService bevat nog geen echte code.

Je schrijft nu echte code in de method

FindVerhoudingOppervlakteLandTovOppervlakteAlleLanden in de class LandService.

Wijzig de class LandService als volgt :

```

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

```

```

        public decimal VindVerhoudingOppervlakteLandTovOppervlakteAlleLanden(string
landcode)
        {
            //throw new NotImplementedException();
            var land = landDAO.Read(landcode);
            var oppervlakteAlleLanden = landDAO.OppervlakteAlleLanden();
            return (decimal)land.Oppervlakte / oppervlakteAlleLanden;
        }
    }
}

```

- Voer de unittest opnieuw uit. Deze keer lukt het.

Je kan dus de class LandService schrijven en testen terwijl de class LandDAOImpl nog niet geschreven is door Jos.

## 8 Mock

Een mock is een stub met twee extra eigenschappen

- Resultaat van method-oproep
- Verificaties

### 8.1 Resultaat van methode-aanroep

Iedere aanroep van een methode van een stub geeft hetzelfde resultaat.

Als je op de LandDAOSTub de methode Read("") aanroept, 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-aanroep variëren, afhankelijk van de parameterwaarden die je meegeeft bij de methode-aanroep.

### 8.2 Verificaties

Nadat je in een Test Methode een method hebt aangeroepen van de te testen class, kan je bij een mock verifiëren of deze class zijn dependency aangesproken heeft.

**Voorbeeld:**

```

[Test]
public void VindVerhoudingOppervlakteLandTovOppervlakteAlleLanden()
{
    Assert.AreEqual(0.25m, landService.
        VindVerhoudingOppervlakteLandTovOppervlakteAlleLanden("B"));
}

```

In bovenstaande code zullen we naar het Assert-statement controleren of landservice de methods Read("B") en OppervlakteAlleLanden() heeft aangeroepen op landDAO.

### 8.3 Moq

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.

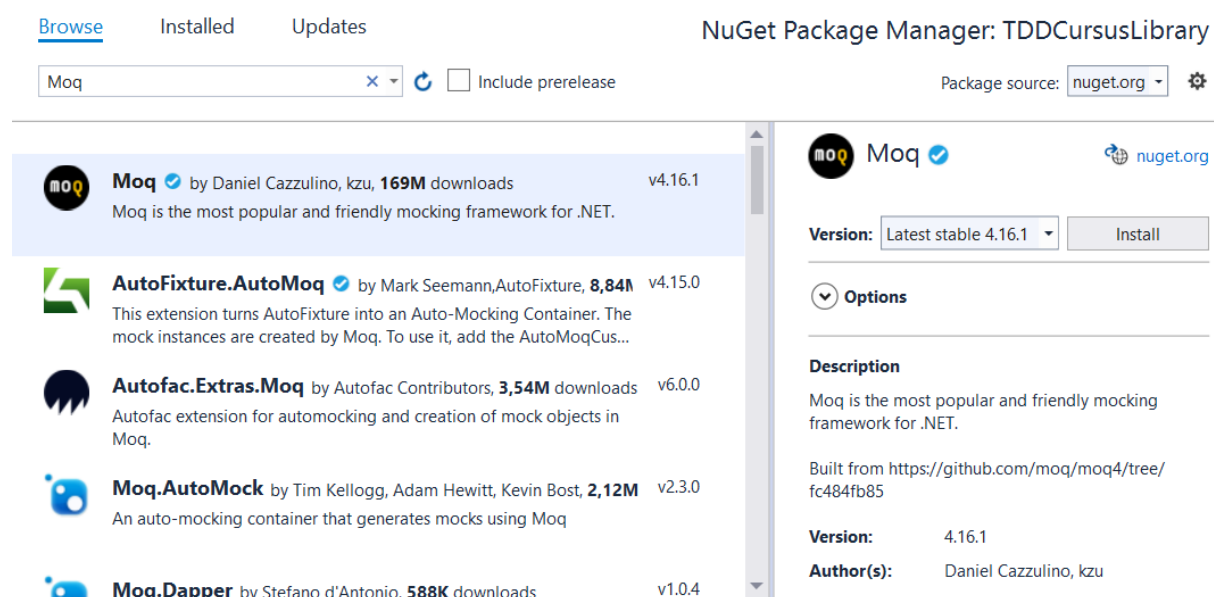
Je lost dit op met een mocking library. Zo'n library maakt een class aan die een mock voorstelt en maakt ook een mock: een object van deze class. Je kan deze mock daarna gebruiken in je test.

Er bestaan meerdere mocking libraries in .NET. Moq is één van de elegantste en veel gebruikte.

## 8.4 De Moq library toevoegen

We voegen de Moq library toe via de NuGet Package Manager

- Open Nuget Package Manager en zoek via het Browse tabblad naar Moq :



- Je kiest de package Moq en installeert deze voor het project TDDCursusLibraryTest.

## 8.5 Een mock aanmaken

In de testclass LandServiceTest zullen we nu de stub vervangen door een mock. We gebruiken hiervoor een Mock-factory.

- Wijzig de class LandServiceTest als volgt :

```
public class LandServiceTest
{
    private ILandDAO landDAO;
    private LandService landService;
    private Mock<ILandDAO> mockFactory;
    [SetUp]
    public void Initialize()
    {
        //landDAO = new LandDAOStub();
        mockFactory = new Mock<ILandDAO>();
        landDAO = mockFactory.Object;
        mockFactory.Setup(eenLandDAO => eenLandDAO.OppervlakteAlleLanden())
            .Returns(20);
        mockFactory.Setup(eenLandDAO => eenLandDAO.Read("B")).
            Returns(new Land
            {
                Landcode = "B",
```

```

        Oppervlakte = 5
    }
    );
    landService = new LandService(landDAO);
}
[Test]
public void VindVerhoudingOppervlakteLandTovOppervlakteAlleLanden()
{
    Assert.AreEqual(0.25m, landService.
        VindVerhoudingOppervlakteLandTovOppervlakteAlleLanden("B"));
    // hier verifiëren we of landService de methods
    // Read("B") en OppervlakteAlleALanden() heeft opgeroepen op landDAO:
    mockFactory.Verify(eenLandDAO => eenLandDAO.OppervlakteAlleLanden());
    mockFactory.Verify(eenLandDAO => eenLandDAO.Read("B"));
}
}

```

(1) In plaats van een LandDAOStub maken we deze keer een mock aan. Dit gebeurt via een nieuwe mockfactory die we bewaren in een private variabele.

(2) Hier maken we de mockfactory aan.

(3) De mockfactory levert ons een mock via de readonly property Object.

## 8.6 De mock trainen

We hebben nu code toegevoegd zodat de private variabele landDAO van de LandServiceTest-class een mock bevat i.p.v. een stub. Deze mock bevat dus de methods die beschreven zijn in de interface ILandDAO. Afhankelijk van het returntype van deze methods retourneren deze methods een welbepaalde waarde. In onderstaande tabel zie je welke waarden dit zijn.

Returntype method in de interface	Returnwaarde van de method in de "mock" class
bool	false
byte, short, int, long, float, double, decimal	0
Een class of interface	null
void	niets

Voor onze mock betekent dit dat de method Read() de waarde null teruggeeft en de method VindOppervlakteAlleLanden() de waarde 0.

Deze waarden zijn echter zelden bruikbaar in een test. Als je bijvoorbeeld LandServiceTest uitvoert krijg je een NullReferenceException.

Je moet eerst de mock trainen door één of meerdere keren de Setup-method van de mock op te roepen. Je doet dit in de [SetUp] methode van de Test class.

Wijzig de testclass LandServiceTest als volgt :

```

[SetUp]
public void Initialize()
{
    //landDAO = new LandDAOStub();
    mockFactory = new Mock<ILandDAO>();
    landDAO = mockFactory.Object;
    mockFactory.Setup(eenLandDAO => eenLandDAO.OppervlakteAlleLanden())
        .Returns(20); //(1)
    mockFactory.Setup(eenLandDAO => eenLandDAO.Read("B")). //(2)
        Returns(new Land
        {
            Landcode = "B",
            Oppervlakte = 5
        });
    landService = new LandService(landDAO);
}

```

(1) Je traint de mock met een oproep van de Setup-method. Als parameter geef je een lambdaexpressie mee. De parameter van deze expressie verwijst naar een LandDAO-mockobject. De returnwaarde van de lambda is de method die je wil trainen. Van die method zeg je dat het een bepaalde waarde teruggeeft. Hier is dit de waarde 20.

(2) Je traint de mock door aan te geven dat de methodoproep Read("B") een Land-object teruggeeft met landcode "B" en oppervlakte 5.

- Voer de tests opnieuw uit. Deze zouden moeten slagen

### Opmerkingen :

– Je kan meerdere keren de Setup-method oproepen op de mock. Door onderstaande code toe te voegen geef je aan dat de method Read() een ander land teruggeeft naargelang je de landcode "B" of "NL" meegeeft als parameter.

```

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

```

– Je kan de mock ook trainen zodat een method een exception werpt als het een bepaalde parameterwaarde meekrijgt.

```

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

```

- Voeg de twee bovenstaande mockfactory-setup statements toe in de Initialize-method

## 8.7 Verificaties

Je kan verifiëren of de te testen class zijn dependencies heeft opgeroepen tijdens het uitvoeren van zijn methods. De LandService method VindVerhoudingOppervlakteLandTovOppervlakteAlleLanden moet op zijn LandDAO

dependency de methods OppervlakteAlleLanden en Read("B") oproepen. Zoniet is deze method verkeerd geschreven. We voegen enkele verificaties toe in de testclass.

- Wijzig de method VindVerhoudingOppervlakteLandTovOppervlakteAlleLanden als volgt :

```
[TestMethod]
public void VindVerhoudingOppervlakteLandTovOppervlakteAlleLanden()
{
    Assert.AreEqual(0.25m, landService.
    VindVerhoudingOppervlakteLandTovOppervlakteAlleLanden("B"));
    // hier verifiëren we of landService de methods
    // Read("B") en OppervlakteAlleALanden() heeft opgeroepen op landDAO:
    mockFactory.Verify(eenLandDAO => eenLandDAO.OppervlakteAlleLanden());
    mockFactory.Verify(eenLandDAO => eenLandDAO.Read("B"));
}
```

## 9 Referenties

<https://docs.nunit.org/>

<https://en.wikipedia.org/wiki/NUnit>

<https://www.testdome.com/questions/c-sharp/account-test/18413>

<https://www.codeproject.com/Articles/178635/Unit-Testing-Using-NUnit>

<https://medium.com/@pjbfgf/title-testing-code-ocd-and-the-aaa-pattern-df453975ab80>

<https://www.codeproject.com/Articles/162041/Introduction-to-NUnit-and-TDD>

[cursus VDAB TDD 2019](#)

<https://dannorth.net/introducing-bdd/>