



C# 2017
TEST DRIVEN DEVELOPMENT

Versie: 15-dec-2017

INHOUDSOPGAVE

1	INLEIDING	6
1.1	DOELSTELLING	6
1.2	VEREISTE VOORKENNIS	6
1.3	NODIGE SOFTWARE	6
1.4	DE NOODZAAK VAN TESTEN	6
1.5	GEAUTOMATISEERD TESTEN	6
1.6	UNIT TESTS EN INTEGRATION TESTS	6
1.7	PLAATS VAN DE UNIT TESTS BINNEN DE SOLUTION	7
2	KENNISMAKING MET TDD	8
2.1	ALGEMEEN	8
2.2	HET VOORBEELDPROJECT	8
2.3	DE TE TESTEN CLASS	9
2.4	DE UNIT TEST	11
2.5	DE UNIT TEST UITVOEREN	15
2.6	ÉÉN TESTMETHOD UITVOEREN	18
2.7	DE UNIT TEST ALS DOCUMENTATIE	18
2.8	DE METHODS VAN DE CLASS ASSERT	18
2.9	TAAK 1: PALINDROOM	20
3	EERST TESTS SCHRIJVEN, DAARNA IMPLEMENTATIE	21
3.1	ALGEMEEN	21
3.2	DE TE TESTEN CLASS	22
3.3	VOORBEELD	22
3.3.1	DE TE TESTEN CLASS	22
3.3.2	DE UNIT TEST	23
3.3.3	EEN EERSTE IMPLEMENTATIE	24
3.3.4	REFACTORING	25
3.3.5	SAMENVATTING VAN DE STAPPEN BIJ TEST DRIVEN DEVELOPMENT	25

3.4	TAAK 2: VEILING	26
4	TEST FIXTURES	27
4.1	ALGEMEEN.....	27
4.2	[TESTINITIALIZE]	27
4.3	TAAK 3: TEST FIXTURES.....	29
5	TO TEST OR NOT TO TEST.....	30
5.1	EXCEPTIONS TESTEN.....	30
5.2	GRENSWAARDEN EN EXTREME WAARDEN TESTEN	31
5.2.1	ALGEMEEN	31
5.2.2	EERSTE VOORBEELD	31
5.2.3	DE CLASS	32
5.2.4	DE UNIT TEST	32
5.2.5	EEN EERSTE IMPLEMENTATIE	34
5.2.6	REFACTORING	35
5.2.7	VOORBEELD MET EEN VERZAMELING	35
5.2.8	DE UNIT TEST	36
5.2.9	EEN EERSTE IMPLEMENTATIE	37
5.2.10	REFACTORING	38
5.3	TESTEN BIJHOUDEN	39
5.4	TAAK 4 : ISBN.....	39
6	MEERDERE UNIT TESTS UITVOEREN	40
6.1	ALGEMEEN.....	40
7	DEPENDENCIES – STUBS	43
7.1	DEPENDENCY.....	43
7.1.1	ALGEMEEN	43
7.1.2	VOORBEELD	44
7.1.3	PROBLEMEN BIJ HET TESTEN VAN EEN TECHNISCHE CLASS MET DEPENDENCIES	46
7.1.4	DE DEPENDENCY UITDRUKKEN IN EEN INTERFACE	46
7.1.5	DEPENDENCY INJECTION	47
7.1.6	STUB	48
7.2	TAAK 5 : STUB	50
8	MOCK	52

8.1	ALGEMEEN.....	52
8.1.1	RESULTAAT VAN METHOD-OPROEP	52
8.1.2	VERIFICATIES.....	52
8.2	MOQ	53
8.2.1	ALGEMEEN	53
8.2.2	DE MOQ LIBRARY TOEVOEGEN.....	53
8.2.3	EEN MOCK AANMAKEN.....	55
8.2.4	DE MOCK TRAINEN	56
8.2.5	VERIFICATIES.....	58
8.3	TAAK 6 : MOCK	59
9	INTELLITEST	60
9.1	ALGEMEEN.....	60
9.1.1	CODEREN	60
9.1.2	RUN INTELLITEST	61
9.1.3	CREËER EN SAVE HET TEST PROJECT.....	62
10	VOORBEELDOPLOSSINGEN TAKEN.....	65
10.1	HOOFDSTUK 2 : PALINDROOM	65
10.1.1	WOORD	65
10.1.2	WOORDTEST	65
10.1.3	ADD REFERENCE	66
10.1.4	TEST	67
10.2	HOOFDSTUK 3 : VEILING.....	67
10.2.1	DE CLASS VEILING ZONDER ECHTE CODE IN ZIJN METHODS	67
10.2.2	DE UNIT TEST VEILINGTEST	68
10.2.3	DE CLASS VEILING MET ECHTE CODE IN ZIJN METHODS	68
10.2.4	ADD REFERENCE	69
10.2.5	TEST	69
10.3	HOOFDSTUK 4 : VEILING - TEST FIXTURES.....	69
10.3.1	VEILINGTEST.CS	69
10.4	HOOFDSTUK 5 : ISBN	70
10.4.1	DE CLASS ISBN ZONDER ECHTE CODE IN ZIJN METHODS	70
10.4.2	ADD REFERENCE	71
10.4.3	DE UNIT TEST ISBNTEST	71
10.4.4	DE CLASS ISBN MET ECHTE CODE IN ZIJN METHODS.....	72
10.4.5	TESTS	73
10.5	HOOFDSTUK 7 : STUB	73
10.5.1	IOPBRENGSTDIAO.....	73
10.5.2	IKOSTDAO.....	73
10.5.3	WINSTSERVICE	73

10.5.4	OPBRENGSTDAOStub.....	74
10.5.5	KOSTDAOStub.....	74
10.5.6	WINSTSERVICETEST.....	74
10.5.7	ECHTE CODE IN DE PROPERTY WINST VAN WINSTSERVICE	75
10.5.8	TEST	75
10.6	HOOFDSTUK 8 : Mock.....	76
10.6.1	INSTALL MOQ LIBRARY	76
10.6.2	WINSTSERVICETEST.cs	76
10.6.3	TEST	76
11	COLOFON.....	77

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 2017

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 die 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 geschreven is, is ze ook volledig getest. De kans is klein dat ze nog fouten bevat.

Je zal zo minder moeten debuggen. Dit is 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

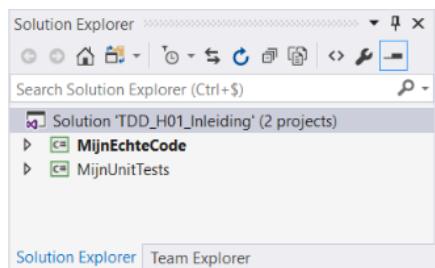
- Hij gebruikt tijdens het ontwikkelen van een class een unit test om fouten te corrigeren in die ene class.
- Als alle classes van een programmaonderdeel af zijn, gebruikt hij een integration test om de samenwerking van die classes te testen.

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 bevinden zich in een apart project binnen die solution.

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



2 KENNISMAKING MET TDD

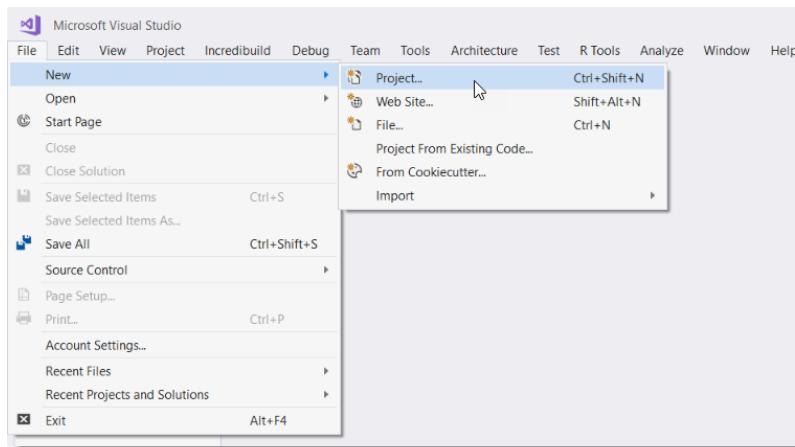
2.1 ALGEMEEN

Je leert in dit hoofdstuk een unit test schrijven.

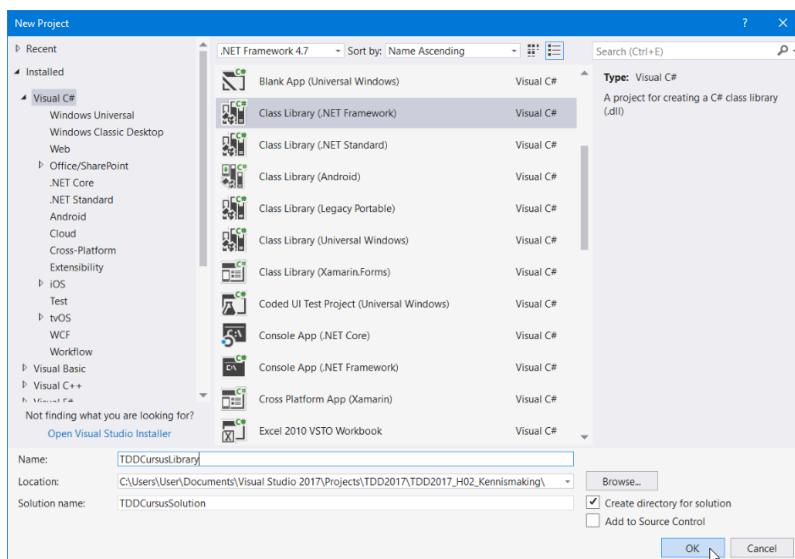
2.2 HET VOORBEELDPROJECT

Je maakt in Visual Studio een Class Library Project:

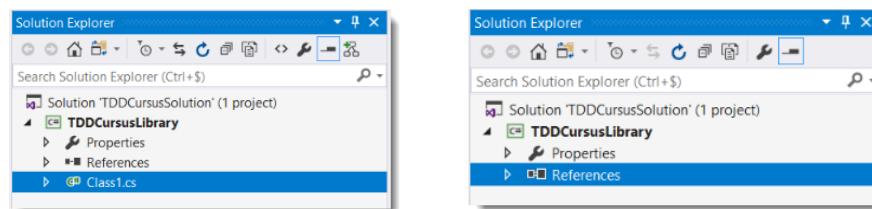
- Je kiest FILE, dan New, dan Project



- Je kiest links Visual C#
- Je kiest rechts Class Library (.NET Framework)
- Je tikt TDDCursusLibrary bij Name
- Je tikt TDDCursusSolution bij Solution Name en je kiest **OK**



- Je verwijderd 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.

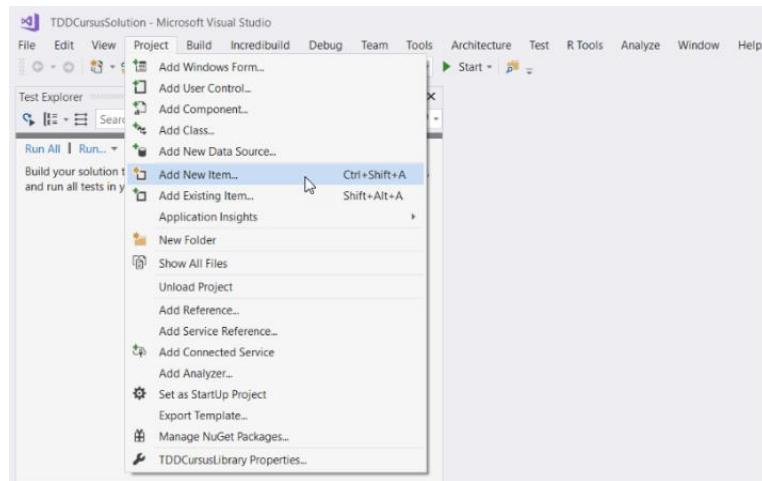
De method **isSchrikkeljaar** geeft **true** terug als dit jaar een schrikkeljaar is. Anders geeft de method **false** terug.

Deze method is gebaseerd op de schrikkeljaarregels:

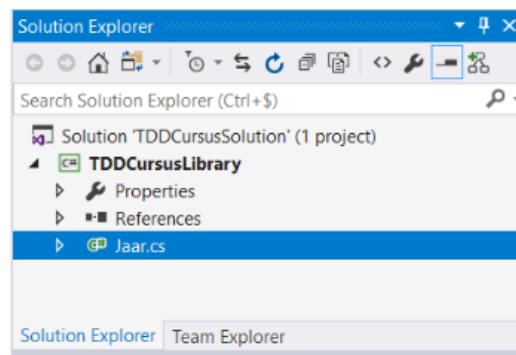
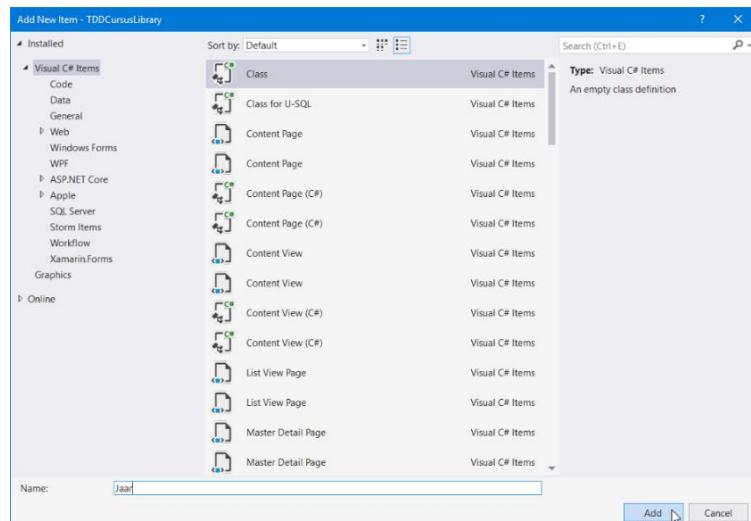
- een jaar deelbaar door 4 is een schrikkeljaar
- een jaar deelbaar door 100 is geen schrikkeljaar, ...
- ... tenzij: een jaar deelbaar door 400 is wel een schrikkeljaar

Je voegt deze class toe aan het project:

- Je kiest **PROJECT**, dan **Add New Item**



- Je kiest **Class**
- Je tikt **Jaar** bij Name en je kiest **Add**



Je wijzigt deze class als volgt:

```
namespace TDDCursusLibrary
{
    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;
            }
        }
    }
}
```

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.

The screenshot shows a simple UI element. On the left is a text input field containing 'Jaartal: 2013'. To its right is a blue button labeled 'Controleer jaar'. To the right of the button is a message box with the text '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.

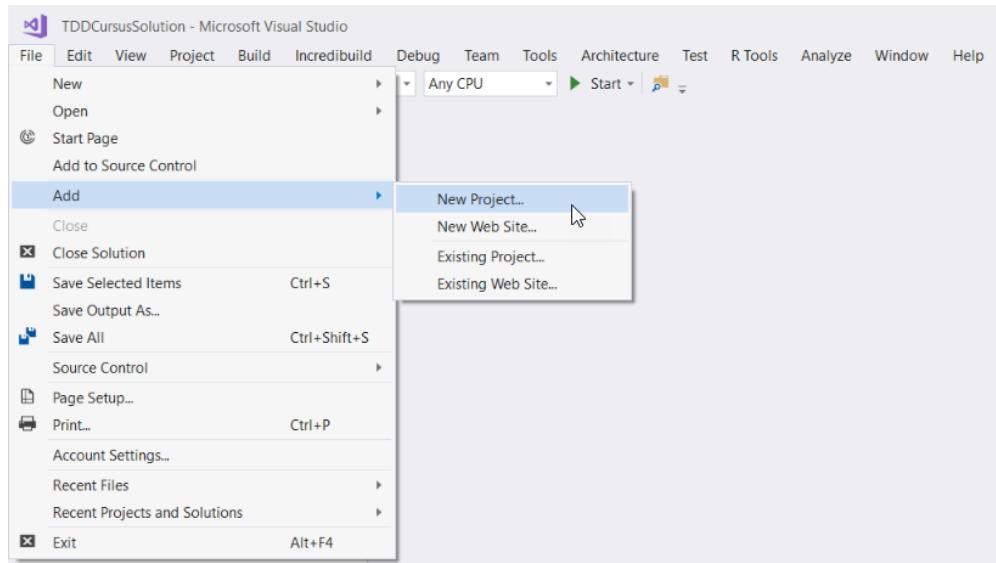
Met een unit test kan je de class onmiddellijk testen en hoeft je de jaartallen niet in een invoervak in te tikken. Je geeft de jaartallen vanuit de unit test door aan de **Jaar**-constructor.

2.4 DE UNIT TEST

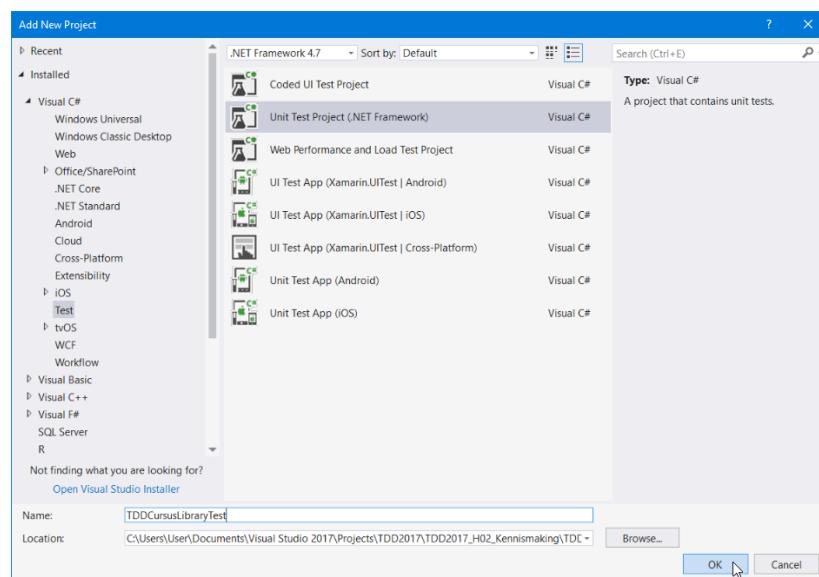
Je maakt een unit test **JaarTest**. Je test daarmee de werking van de class **Jaar**.

Je voegt een test project toe aan de solution

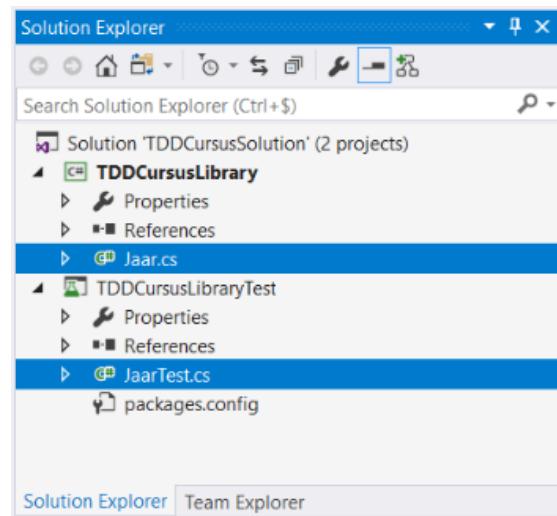
- Je kiest FILE, dan Add, dan New Project



- Je kiest links Visual C#, dan Test en rechts Unit Test Project (.NET Framework)
- Je tikt **TDDCursusLibraryTest** bij Name en je kiest **OK**

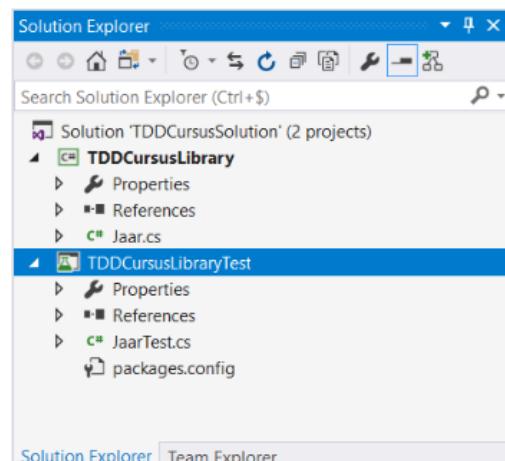


- Je hernoemt **UnitTest1.cs** naar **JaarTest.cs**

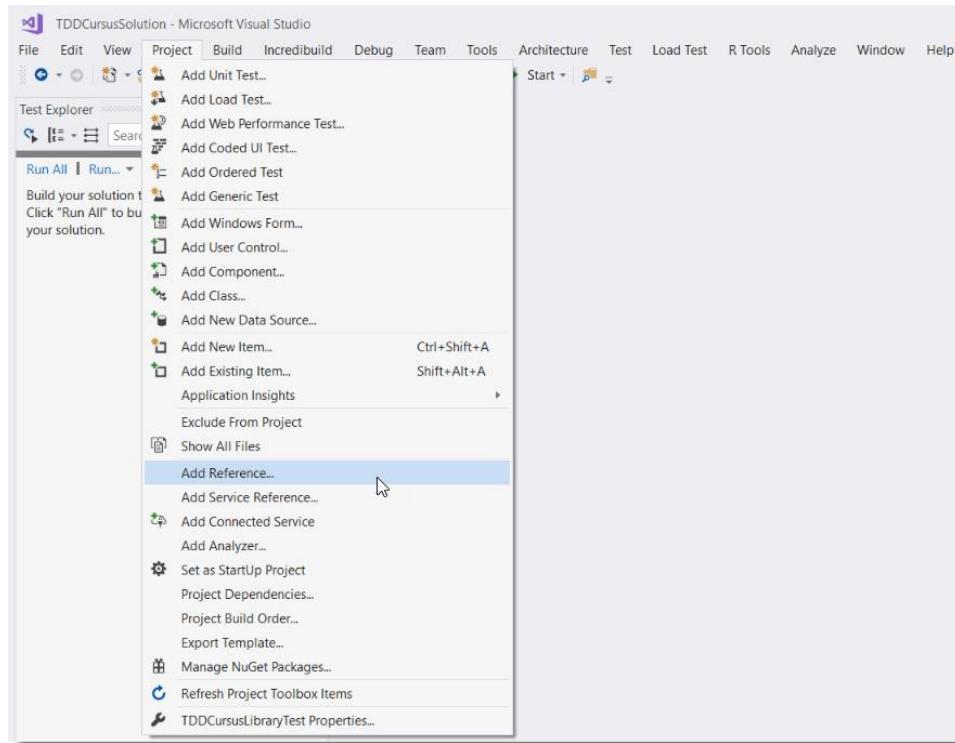


Je legt een verwijzing naar het project **TDDCursusLibrary**

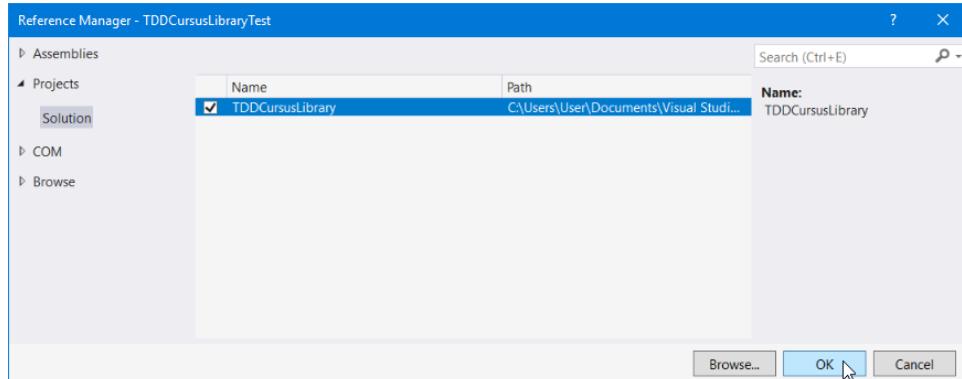
- Selecteer het test-project

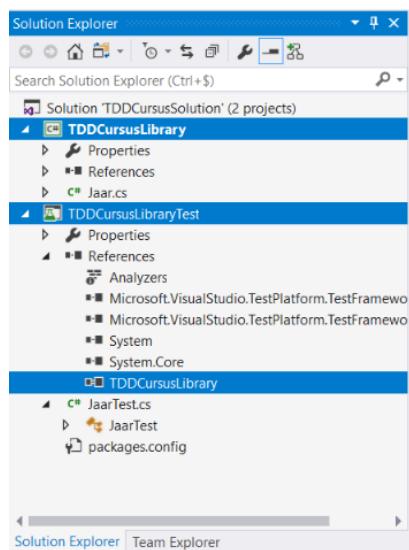


- Je kiest **PROJECT**, dan **Add Reference**



- 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]
    public class JaarTest
    {
        [TestMethod]
        public void EenJaarDeelbaarDoor400IsEenSchrikkeljaar() // (1) // (2)
        {
            Assert.AreEqual(true, new Jaar(2000).IsSchrikkeljaar()); // (3) // (4)
        }

        [TestMethod]
        public void EenJaarDeelbaarDoor100MaarNietDoor400IsGeenSchrikkeljaar()
        {
            Assert.AreEqual(false, new Jaar(1900).IsSchrikkeljaar());
        }

        [TestMethod]
        public void EenJaarDeelbaarDoor4IsEenSchrikkeljaar()
        {
            Assert.AreEqual(true, new Jaar(2012).IsSchrikkeljaar());
        }

        [TestMethod]
        public void EenJaarNietDeelbaarDoor4IsGeenSchrikkeljaar()
        {
            Assert.AreEqual(false, new Jaar(2015).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 aanziert 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 met **new Jaar(2000)** 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!
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.

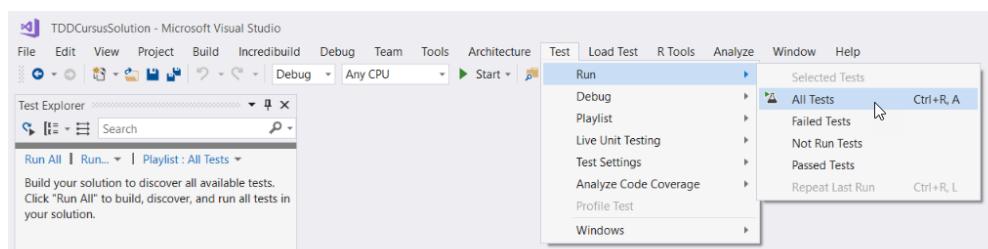
Als deze twee waarden aan elkaar gelijk zijn, zal Visual Studio deze test aanzien als een test die correct verlopen is.
Als deze twee waarden verschillen van elkaar, zal Visual Studio deze test aanzien als een test die verkeerd loopt.

De overige TestMethods testen andere aspecten van de method **IsSchrikkeljaar**.

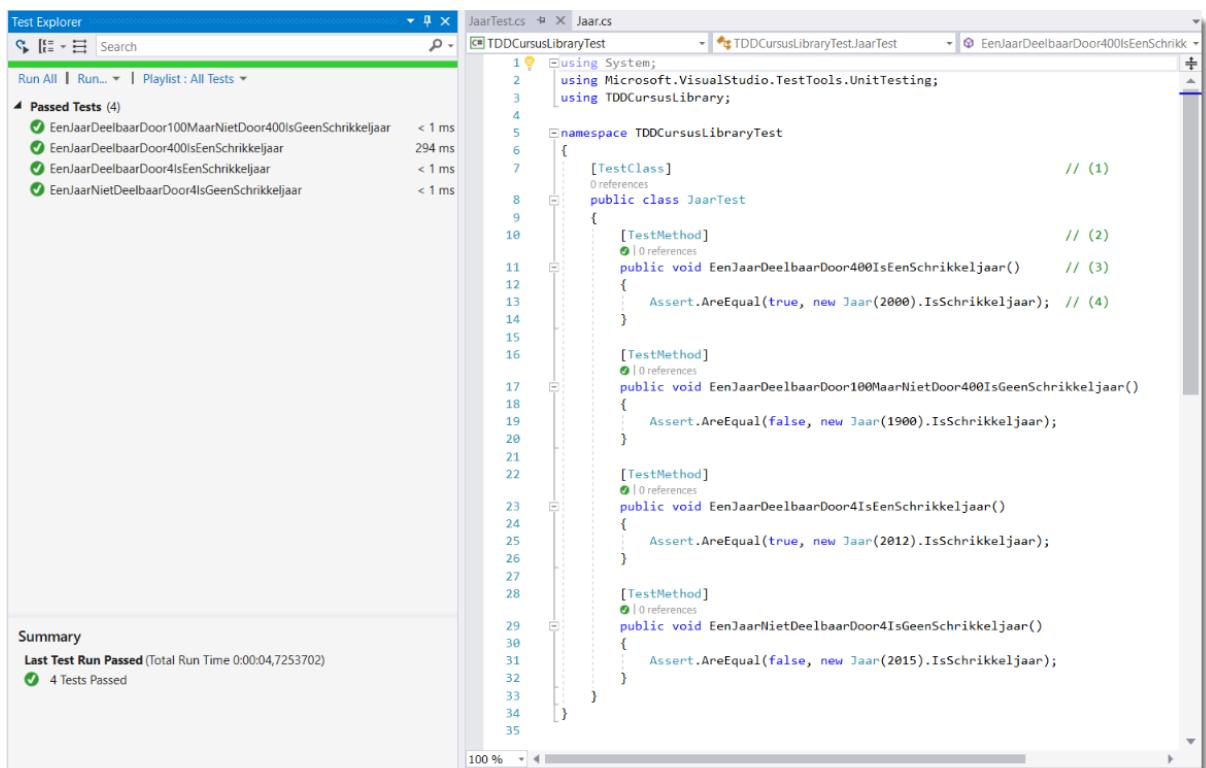
2.5 DE UNIT TEST UITVOEREN

Je laat Visual Studio de unit test uitvoeren. Visual Studio voert hierbij alle TestMethods uit.

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



Visual Studio voert de unit test uit en toont een rapport in een venster **Test Explorer**:



Een groen vinkje in de [Test Explorer](#) en in de [TestClass](#) duidt aan dat een test geslaagd is.

Opmerking:

Bemerk dat Visual Studio de TestMethods 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](#).

```

namespace TDDCursusLibrary
{
    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;
                }

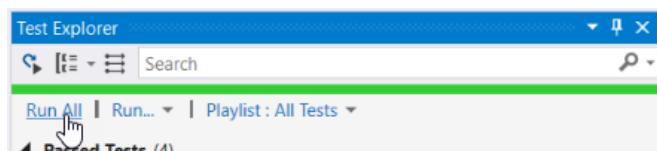
                //return jaar % 4 == 0;
                //return jaar % 5 == 0;
            }
        }
    }
}

```

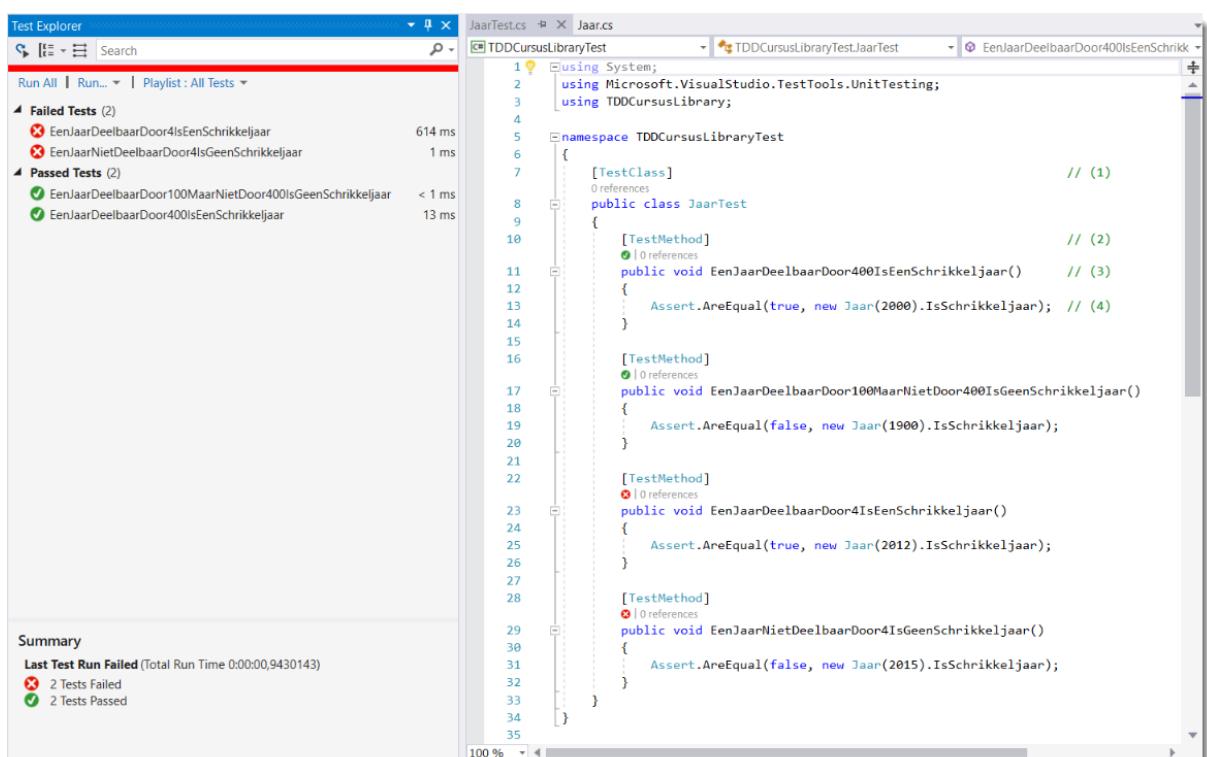
```
        }  
    }  
}
```

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:



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.

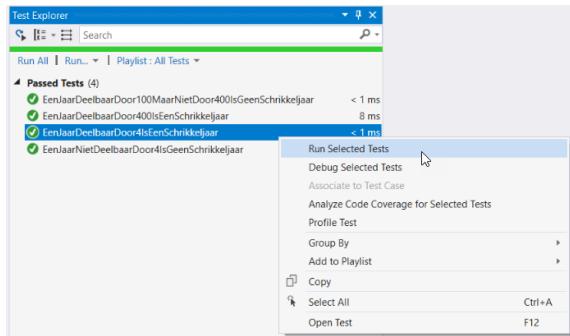
Hij krijgt zo direct feedback of die wijziging correct was.

Code schrijven en hierover onmiddellijk feedback krijgen is productief!

2.6 ÉÉN TESTMETHOD UITVOEREN

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

- Je klikt met de rechtermuisknop op één TestMethod 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.

```
EenJaarDeelbaarDoor100MaarNietDoor400IsGeenSchrikkeljaar
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 gebruikte tot nu in de TestMethods 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;
using TDDCursusLibrary;

namespace TDDCursusLibraryTest
{
    [TestClass]
    public class JaarTest
    {
        [TestMethod]
        public void EenJaarDeelbaarDoor400IsEenSchrikkeljaar() // (1) // (2)
        {
            //Assert.AreEqual(true, new Jaar(2000).IsSchrikkeljaar()); // (4)
            Assert.IsTrue(new Jaar(2000).IsSchrikkeljaar());
        }

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

        [TestMethod]
        public void EenJaarDeelbaarDoor4IsEenSchrikkeljaar()
        {
            //Assert.AreEqual(true, new Jaar(2012).IsSchrikkeljaar());
            Assert.IsTrue(new Jaar(2012).IsSchrikkeljaar());
        }

        [TestMethod]
        public void EenJaarNietDeelbaarDoor4IsGeenSchrikkeljaar()
        {
            //Assert.AreEqual(false, new Jaar(2015).IsSchrikkeljaar());
            Assert.IsFalse(new Jaar(2015).IsSchrikkeljaar());
        }
    }
}
```

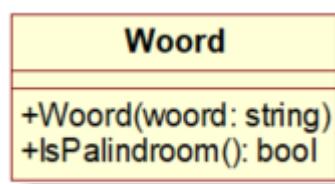
```

1 1? using System;
2 2? using Microsoft.VisualStudio.TestTools.UnitTesting;
3 3? using TDDCursusLibrary;
4
5 4? namespace TDDCursusLibraryTest
6 5? {
7 6?     [TestClass]
8 7?     0 references
9 8?     public class JaarTest
10 9?     {
11 10?         [TestMethod]
12 11?         0 | 0 references
13 12?         public void EenJaarDeelbaarDoor400IsEenSchrikkeljaar() // (3)
14 13?         {
15 14?             //Assert.AreEqual(true, new Jaar(2000).IsSchrikkeljaar()); // (4)
16 15?             Assert.IsTrue(new Jaar(2000).IsSchrikkeljaar());
17 16?         }
18 17?         [TestMethod]
19 18?         0 | 0 references
20 19?         public void EenJaarDeelbaarDoor100MaarNietDoor400IsGeenSchrikkeljaar()
21 20?         {
22 21?             //Assert.AreEqual(false, new Jaar(1900).IsSchrikkeljaar());
23 22?             Assert.IsFalse(new Jaar(1900).IsSchrikkeljaar());
24 23?         }
25 24?         [TestMethod]
26 25?         0 | 0 references
27 26?         public void EenJaarDeelbaarDoor4IsEenSchrikkeljaar()
28 27?         {
29 28?             //Assert.AreEqual(true, new Jaar(2012).IsSchrikkeljaar());
30 29?             Assert.IsTrue(new Jaar(2012).IsSchrikkeljaar());
31 30?         }
32 31?         [TestMethod]
33 32?         0 | 0 references
34 33?         public void EenJaarNietDeelbaarDoor4IsGeenSchrikkeljaar()
35 34?         {
36 35?             //Assert.AreEqual(false, new Jaar(2015).IsSchrikkeljaar());
37 36?             Assert.IsFalse(new Jaar(2015).IsSchrikkeljaar());
38 37?         }
39 38?     }
39 39?

```

2.9 TAAK 1: PALINDROOM

Je maakt een class **Woord**



Je geeft aan de constructor een woord mee.

De method **IsPalindroom** geeft enkel **true** terug als dit woord een palindroom is: een woord dat hetzelfde is als je het van voor naar achter leest en als je het van achter naar voor leest (bvb. lepel).

Je schrijft in een class **WoordTest** de nodige tests voor de class **Woord**

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()
public void EenJaarNietDeelbaarDoor4IsGeenSchrikkeljaar()
```

- als een jaar deelbaar is door 100 maar niet door 400 is het echter geen schrikkeljaar.

```
public void EenJaarDeelbaarDoor100MaarNietDoor400IsGeenSchrikkeljaar()
```

- 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 method-naam, 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 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:

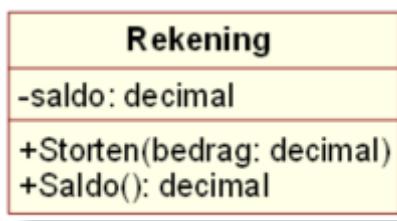
```
throw new NotImplementedException();
```

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 schrijft de class met zijn methods, daarin enkel

```
throw new NotImplementedException();
```

```
using System;

namespace TDDCursusLibrary
{
    public class Rekening
    {
        private decimal saldo;

        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 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 TDDCursusLibrary;

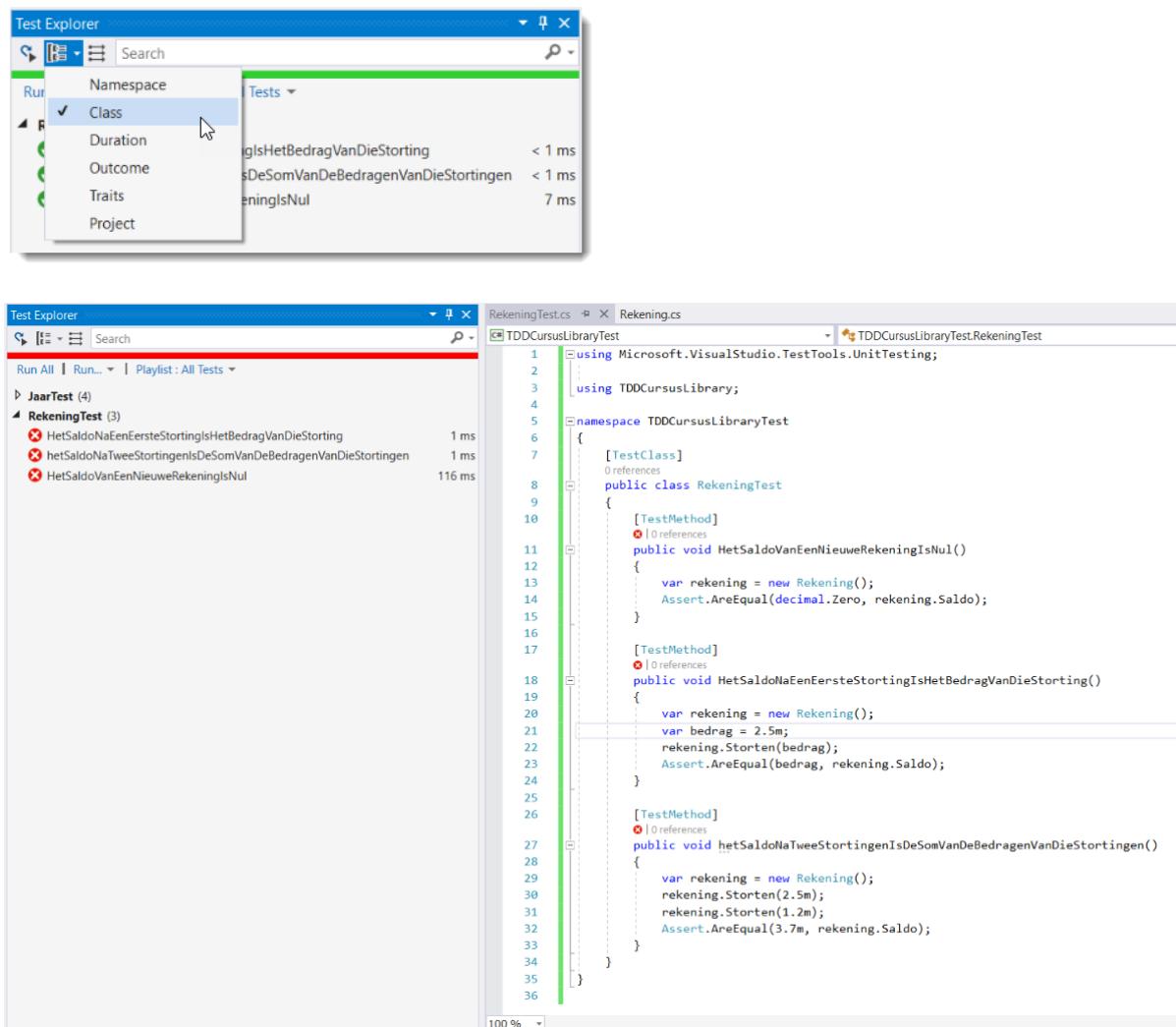
namespace TDDCursusLibraryTest
{
    [TestClass]
    public class RekeningTest
    {
        [TestMethod]
        public void HetSaldoVanEenNieuweRekeningIsNul()
        {
            var rekening = new Rekening();
            Assert.AreEqual(decimal.Zero, rekening.Saldo);
        }

        [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 mislukken.

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



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 TDCCursusLibrary
{
    public class Rekening
    {
        private decimal saldo;

        public void Storten(decimal bedrag)
        {
//            throw new NotImplementedException();
            saldo += bedrag;
        }

        public decimal Saldo
        {
            get
            {
//                throw new NotImplementedException();
                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.

```

Test Explorer
Run All | Run... | Playlist: All Tests
JaarTest (4)
  RekeningTest (3)
    HetSaldoNaEenEersteStortingIsHetBedragVanDieStorting
    hetSaldoNaTweeStortingenIsDeSomVanDeBedragenVanDieStortingen
    HetSaldoVanEenNieuweRekeningIsNul

Summary
Last Test Run Passed (Total Run Time 0:00:00,3273883)
  7 Tests Passed

RekeningTest.cs
1  using Microsoft.VisualStudio.TestTools.UnitTesting;
2
3  using TDDCursusLibrary;
4
5  namespace TDDCursusLibraryTest
6  {
7      [TestClass]
8      public class RekeningTest
9      {
10         [TestMethod]
11         public void HetSaldoVanEenNieuweRekeningIsNul()
12         {
13             var rekening = new Rekening();
14             Assert.AreEqual(decimal.Zero, rekening.Saldo);
15         }
16
17         [TestMethod]
18         public void HetSaldoNaEenEersteStortingIsHetBedragVanDieStorting()
19         {
20             var rekening = new Rekening();
21             var bedrag = 2.5m;
22             rekening.Storten(bedrag);
23             Assert.AreEqual(bedrag, rekening.Saldo);
24         }
25
26         [TestMethod]
27         public void hetSaldoNaTweeStortingenIsDeSomVanDeBedragenVanDieStortingen()
28         {
29             var rekening = new Rekening();
30             rekening.Storten(2.5m);
31             rekening.Storten(1.2m);
32             Assert.AreEqual(3.7m, rekening.Saldo);
33         }
34     }
35 }
36

```

3.3.4 REFACTORING

Nu je een werkende versie hebt van de class **Rekening**, zie je na of je 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.

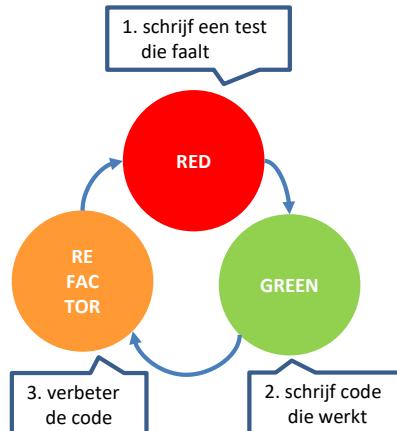
Je kan in de class **Rekening** geen refactoring doen.

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.



3.4 TAAK 2: VEILING

Je maakt met de voorgeschreven stappen van TDD een class die een veiling (verkoop) voorstelt en de bijbehorende unit test.

Veiling
+DoeBod(bedrag: decimal) +HoogsteBod(): decimal

Je kan op een **Veiling**-object meerdere keren de method **DoeBod** oproepen. Je geeft als parameter het bedrag van het bod mee.

Je kan op ieder moment de readonly property **HoogsteBod** oproepen. Deze method geeft je het hoogst geboden bedrag terug.

```

var veiling = new Veiling.Veiling();
veiling.DoeBod(1000);
var hoogsteBod = veiling.HoogsteBod;      // hoogsteBod bevat 1000
veiling.DoeBod(2000);
hoogsteBod = veiling.HoogsteBod;          // hoogsteBod bevat 2000
  
```

Uit de analyse blijkt dat:

- Als nog geen enkel bod werd uitgevoerd, het hoogste bod gelijk is aan 0.
- Als een eerste bod werd uitgevoerd, het hoogste bod gelijk is aan het bedrag van dit bod.
- Al meerdere keren een bod werd uitgevoerd, het hoogste bod gelijk is aan bedrag van het hoogste bod.

4 TEST FIXTURES

4.1 ALGEMEEN

Je maakt regelmatig in meerdere TestMethods van een 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*.

4.2 [TESTINITIALIZE]

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

Je vermindert 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**-methode.

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:

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

Visual Studio voert voor iedere TestMethod de **TestInitialize**-method van dezelfde unit test uit.

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

- **[TestInitialize]** **Initialize()**
HetSaldoNaTweeStortingenIsDeSomVanDeBedragenVanDieStortingen()
- **[TestInitialize]** **Initialize()**
hetSaldoNaEenEersteStortingIsHetBedragVanDieStorting()
- **[TestInitialize]** **Initialize()**
hetSaldoVanEenNieuweRekeningIsNul()
- **[TestMethod]** **rekening**

Je moet nu binnen de TestMethod 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 TDDCursusLibrary;
```

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

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

        [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 kan de unit test opnieuw uitvoeren.

The screenshot shows the Visual Studio interface with the Test Explorer window open. The 'RekeningTest.cs' file is selected in the 'TDDCursusLibraryTest' project. The 'Test Explorer' tab shows a summary: 'Last Test Run Passed (Total Run Time 0:00:00,5715636)' and '7 Tests Passed'. The code editor displays the 'RekeningTest.cs' file with three test methods: 'HetSaldoNaEenEersteStortingIsHetBedragVanDieStorting', 'hetSaldoNaTweeStortingenIsDeSomVanDeBedragenVanDieStortingen', and 'HetSaldoVanEenNieuweRekeningIsNul'. The code uses namespaces 'Microsoft.VisualStudio.TestTools.UnitTesting' and 'TDDCursusLibrary'; classes 'RekeningTest' and 'Rekening'; and methods 'Initialize' and 'TestMethods'.

```
1  using Microsoft.VisualStudio.TestTools.UnitTesting;
2  ...
3  using TDDCursusLibrary;
4  ...
5  namespace TDDCursusLibraryTest
6  {
7      [TestClass]
8      public class RekeningTest
9      {
10         private Rekening rekening;
11
12         [TestInitialize]
13         public void Initialize()
14         {
15             rekening = new Rekening();
16         }
17
18         [TestMethod]
19         public void HetSaldoVanEenNieuweRekeningIsNul()
20         {
21             var rekening = new Rekening();
22             Assert.AreEqual(decimal.Zero, rekening.Saldo);
23         }
24
25         [TestMethod]
26         public void HetSaldoNaEenEersteStortingIsHetBedragVanDieStorting()
27         {
28             var rekening = new Rekening();
29             var bedrag = 2.5m;
30             rekening.Storten(bedrag);
31             Assert.AreEqual(bedrag, rekening.Saldo);
32         }
33
34         [TestMethod]
35         public void hetSaldoNaTweeStortingenIsDeSomVanDeBedragenVanDieStortingen()
36         {
37             var rekening = new Rekening();
38             rekening.Storten(2.5m);
39             rekening.Storten(1.2m);
40             Assert.AreEqual(3.7m, rekening.Saldo);
41         }
42     }
43 }
```

Je mag het **Rekening**-object, waar de **private** variabele **rekening** naar verwijst, wijzigen in een **TestMethod**. Bij de volgende **TestMethod** bevat de variabele **rekening** een vers geïnitialiseerd **Rekening**-object: Visual Studio voert tussen de **TestMethods** de **TestInitialize**-method uit.

Opmerking:

Je kan ook een method opnemen waarvoor je **[TestCleanup]** tikt.

Visual Studio voert deze method uit 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.

Je verwijdert dit bestand in een **TestCleanup**-method.

4.3 TAAK 3: TEST FIXTURES

Je gebruikt een **testinitialize** method in de unit test van de class **Veiling**.

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**.

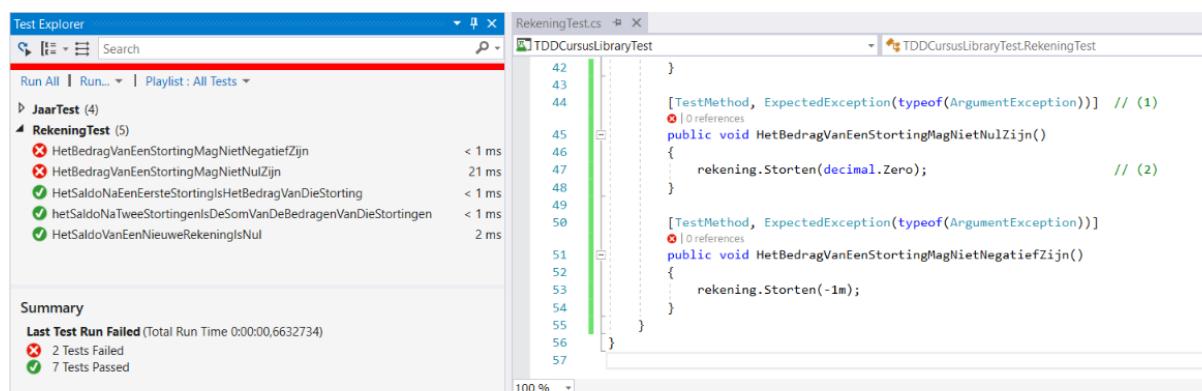
```
using System;
. . . .
[TestMethod, ExpectedException(typeof(ArgumentException))] // (1)
public void HetBedragVanEenStortingMagNietNulZijn()
{
    rekening.Storten(decimal.Zero); // (2)
}
[TestMethod, ExpectedException(typeof(ArgumentException))]
public void HetBedragVanEenStortingMagNietNegatiefZijn()
{
    rekening.Storten(-1m);
}
```

- (1) Je test in deze TestMethod 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 TestMethod is deze test volgens Visual Studio geslaagd.

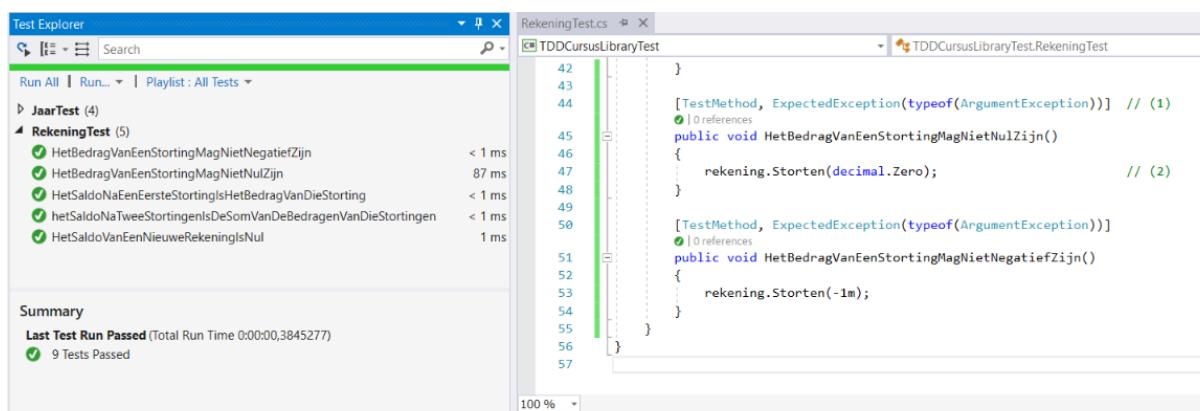
Je voert de test uit. De twee extra TestMethods mislukken. Dit betekent dat je extra code moet schrijven in de class **Rekening**, om aan deze extra vereiste uit de analyse te voldoen.



Je wijzigt in de class **Rekening** de method **storten**

```
using System;
. . .
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

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.

- 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 2^o 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 nr. 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");
}

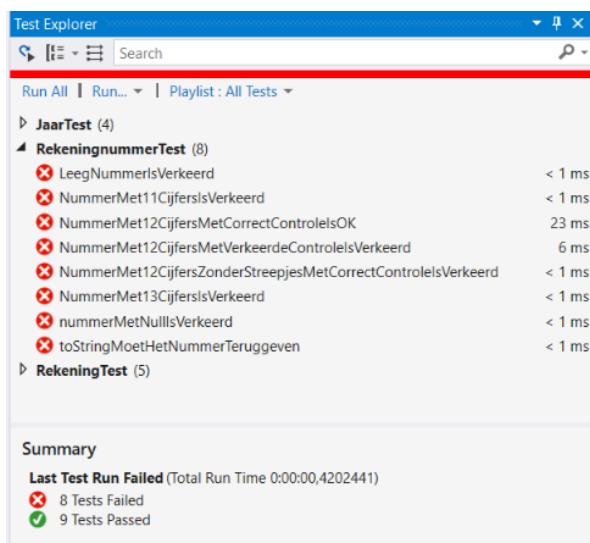
[TestMethod, ExpectedException(typeof(ArgumentException))]
public void LeegNummerIsVerkeerd()
{
    new Rekeningnummer(string.Empty);
}

[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 de 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 TDCCursusLibrary
{
    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)
        {
            // throw new NotImplementedException();

            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));

            if (eerste10Cijfers % 97L != laatste2Cijfers)
            {
                throw new ArgumentException();
            }

            this.nummer = nummer;
        }

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

- (1) Dit object stelt een regular expression (tekstpatroon) voor. In een regular expression hebben bepaalde tekens een speciale betekenis. Sommige worden hier gebruikt

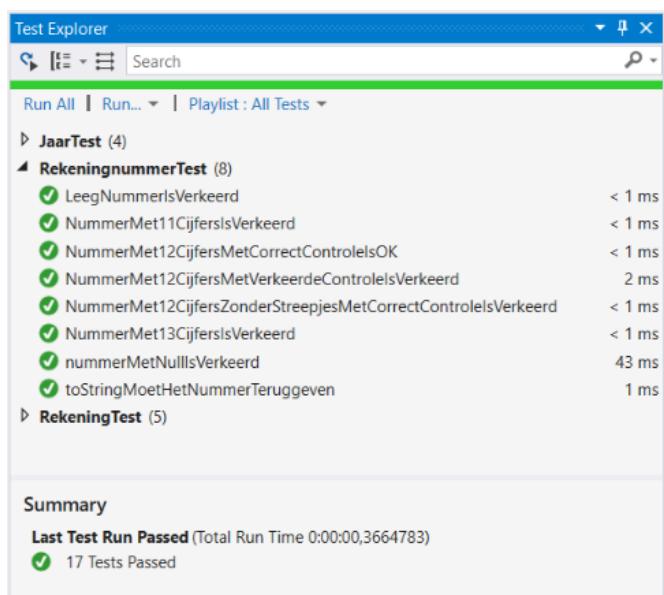
- **^** het begin van de tekst
- **\$** het einde van de tekst
- **\d** een cijfer
- **{x}** x aantal keer het vorige teken uit de regular expression.

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

Statistiek
+Gemiddelde(getallen: decimal[*]): decimal

De class **Statistiek** bevat een static method **Gemiddelde**. Je geeft aan deze method een array van **decimals** mee. Je krijgt het gemiddelde van deze **decimals** terug.

```
using System;

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

5.2.8 DE UNIT TEST

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

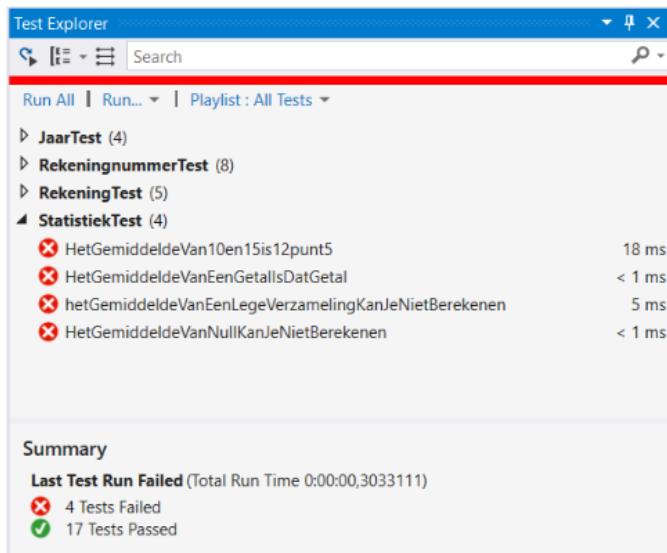
namespace TDDCursusLibraryTest
{
    [TestClass]
    public class StatistiekTest
    {
        [TestMethod]
        public void HetGemiddeldeVan10en15is12punt5()
        {
            Assert.AreEqual(12.5m, Statistiek.Gemiddelde(new decimal[] {10m, 15m}));
        }

        [TestMethod]
        public void HetGemiddeldeVanEenGetalIsDatGetal()
        {
            var enigGetal = 1.23m;
            Assert.AreEqual(enigGetal, Statistiek.Gemiddelde(new decimal[] {enigGetal}));
        }

        [TestMethod, ExpectedException(typeof(ArgumentException))]
        public void hetGemiddeldeVanEenLegeVerzamelingKanJeNietBerekenen()
        {
            Statistiek.Gemiddelde(new decimal[] { });
        }

        [TestMethod, ExpectedException(typeof(ArgumentNullException))]
        public void HetGemiddeldeVanNullKanJeNietBerekenen()
        {
            Statistiek.Gemiddelde(null);
        }
    }
}
```

Je controleert of je aan de verleiding kon weerstaan om methods in class **Statistiek** te implementeren. Je doet dit door de unit test uit te voeren. Alle tests mislukken. Dit is goed nieuws.



5.2.9 EEN EERSTE IMPLEMENTATIE

```
using System;

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

            if (getallen == null)
            {
                throw new ArgumentNullException();
            }

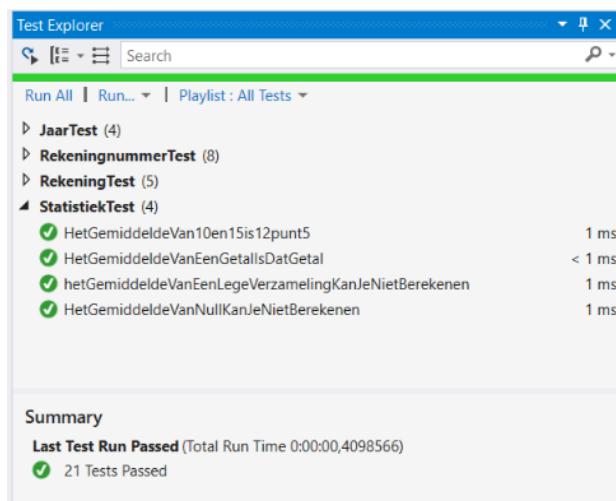
            if (getallen.Length == 0)
            {
                throw new ArgumentException();
            }

            var totaal = decimal.Zero;

            foreach (var aantal in getallen)
            {
                totaal += aantal;
            }

            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();
```

en je voegt boven in de source

```
using System.Linq;
```

toe.

```
using System;
using System.Linq;

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

            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;
        }

        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.

5.4 TAAK 4 : ISBN

Je maakt met de voorgeschreven stappen van TDD een class die een ISBN (Internationaal Standaard Boeknummer) voorstelt en de bijbehorende unit test.

ISBN
+ISBN(numero: long) +ToString(): string

De regels van ISBN zijn als volgt:

- Het bestaat uit 13 cijfers
- Een ISBN bevat een controlemechanisme
 - Je maakt de som van de cijfers op de eerste 6 oneven posities
 - Je maakt de som van de cijfers op de eerste 6 even posities en je vermenigvuldigt deze som maal 3.
 - Je maakt de som van deze twee tussenresultaten.
 - Je maakt het verschil van deze som en het naastgelegen hoger gelegen tiental.
 - Het dertiende cijfer moet gelijk zijn aan dit verschil, tenzij het verschil 10 is. Dan moet het dertiende cijfer gelijk zijn aan 0.

Voorbeeld het ISBN 9789027439642

- de som van de cijfers op de eerste 6 oneven posities
 $9 + 8 + 0 + 7 + 3 + 6 = 33$
- de som van de cijfers op de eerste 6 even posities, maal 3
 $7 + 9 + 2 + 4 + 9 + 4 = 35 \times 3 = 105$
- de som van deze twee tussenresultaten
 $33 + 105 = 138$
- het verschil van deze som en het naastgelegen hoger gelegen tiental
2
- het dertiende cijfer is gelijk aan dit verschil

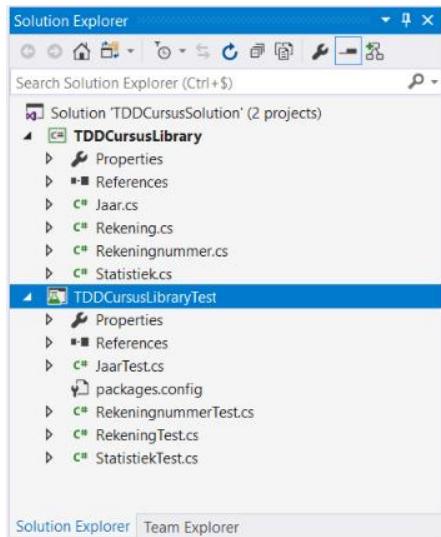
6 MEERDERE UNIT TESTS UITVOEREN

6.1 ALGEMEEN

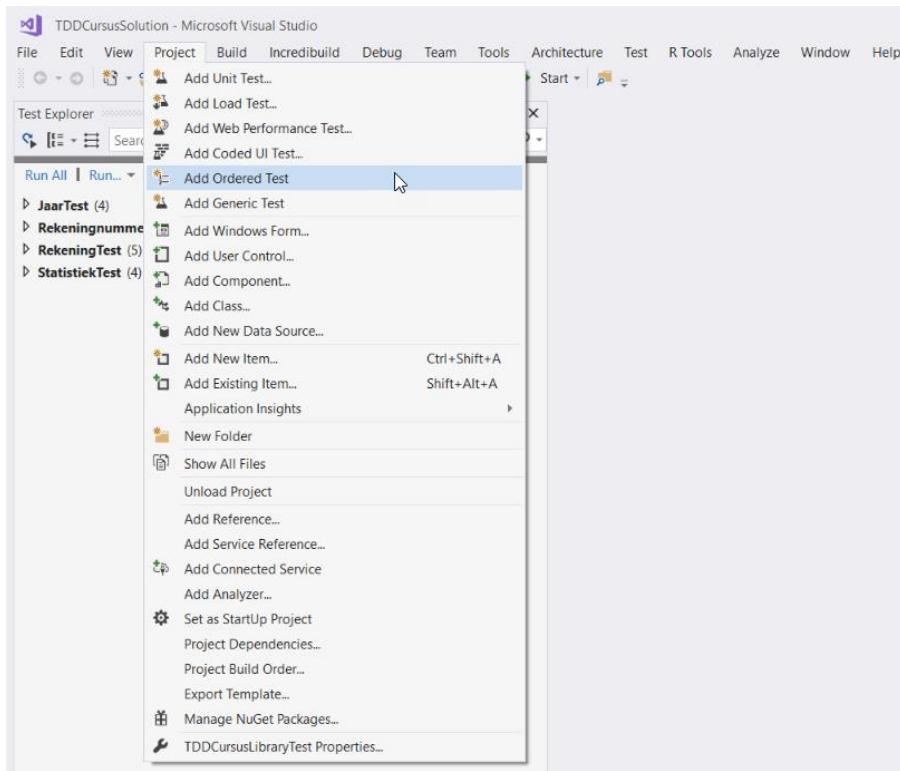
Een ordered test is een test die een verzameling TestMethods uit één of meerdere unit tests bevat.

Je maakt een ordered test die de tests bevat van **RekeningTest** en van **RekeningnummerTest**.

- Je selecteert in de Solution Explorer het project **TDDCursusLibraryTest**



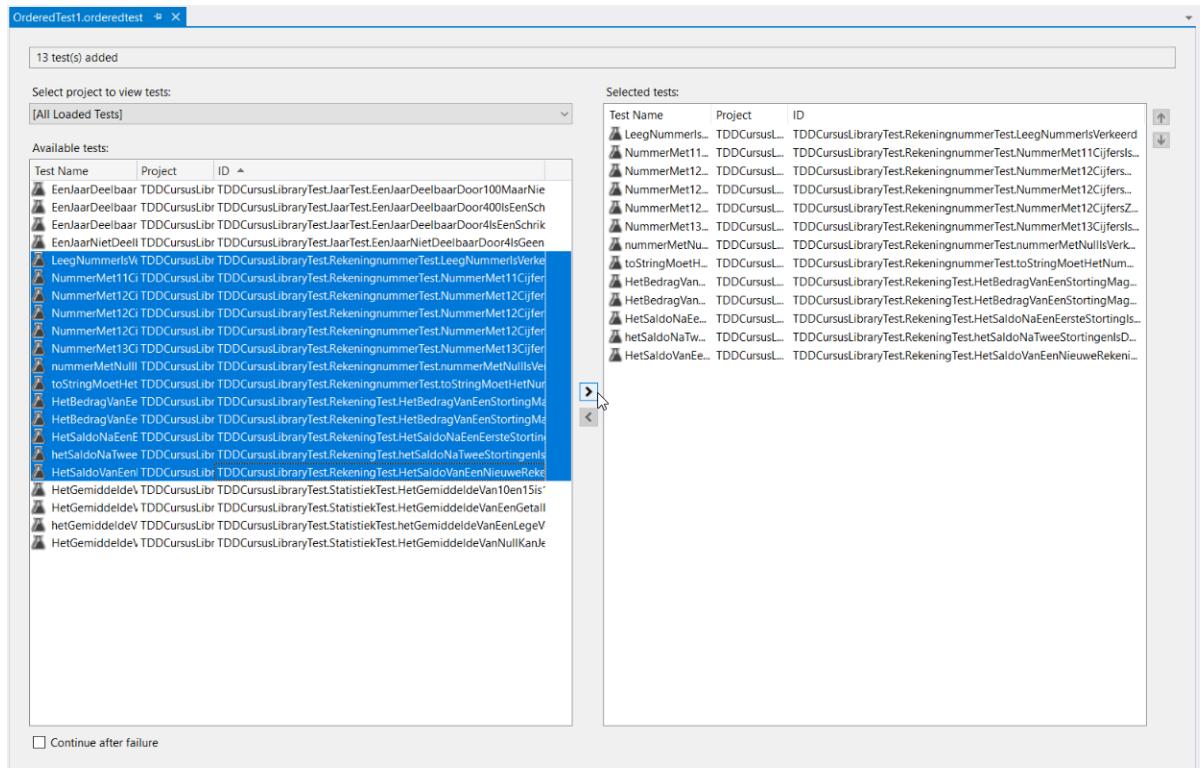
- Je kiest PROJECT, Add Ordered Test



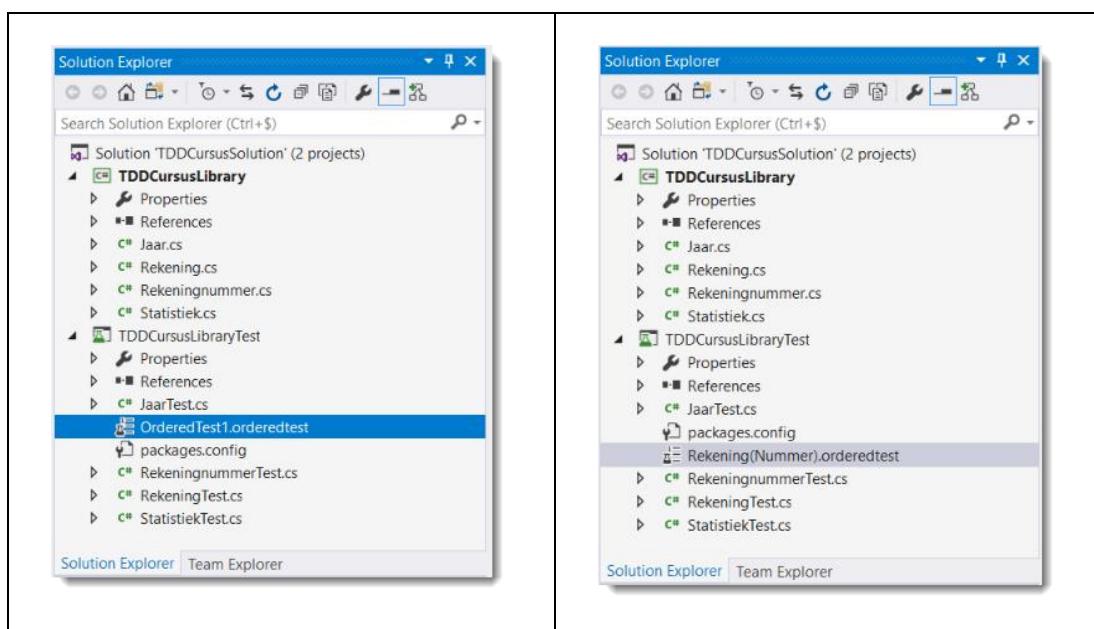
Je krijgt een venster met links alle beschikbare testen. Je kan zo'n test aanduiden en met de knop overbrengen naar rechts, waar je de ordered test ziet.

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

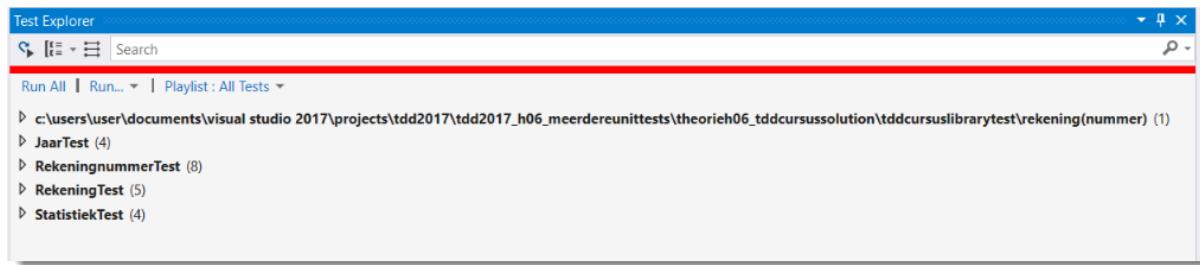
Je brengt alle tests van **RekeningNummerTest** en **RekeningTest** over naar rechts.



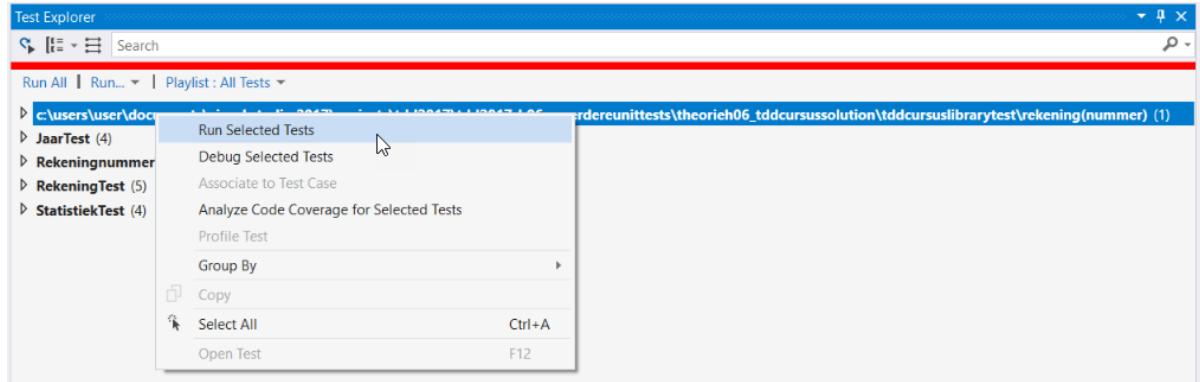
Je wijzigt in de Solution Explorer de naam van de ordered test naar **Rekening(Nummer).orderedtest**



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**.

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



```
namespace TDDCursusLibrary
{
    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;
        }
    }
}
```

```
namespace TDDCursusLibrary
{
    public class B
    {
        public void B1()
        {
            // ...
        }

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

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 or 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 (bv. op null plaatst).

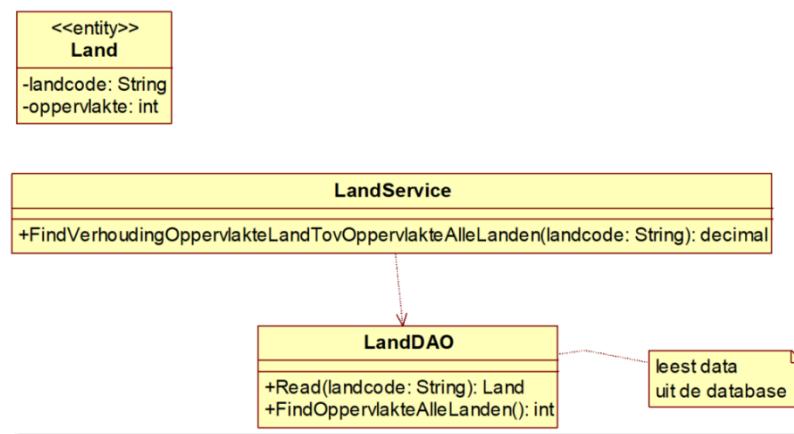
```
namespace TDDCursusLibrary
{
    public class A
    {
        private readonly B b = new B();

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

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

7.1.2 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 **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. Indien 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 **LandService** als **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.



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.

```
using System;

namespace TDDCursusLibrary
{
    public class LandService
    {
//        private LandDAO landDAO = new LandDAO();
        private readonly ILandDAO landDAO;

        public decimal VerhoudingOppervlakteLandTovOppervlakteAlleLanden(string landcode)
        {
            throw new NotImplementedException();
        }
    }
}
```

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

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

7.1.5 DEPENDENCY INJECTION

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

```
using System;

namespace TDDCursusLibrary
{
    public class LandService
    {
//        private LandDAO landDAO = new LandDAO();
        private readonly ILandDAO landDAO;

        // Constructor
        public LandService(ILandDAO landDAO)
        {
            this.landDAO = landDAO;
        }

        public decimal VerhoudingOppervlakteLandTovOppervlakteAlleLanden(string landcode)
        {
            throw new NotImplementedException();
        }
    }
}
```

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 aan dat de dependency voorstelt, maar krijgt dit object aangereikt (als constructor parameter).

Dit heet “**dependency injection**”.

Opmerking:

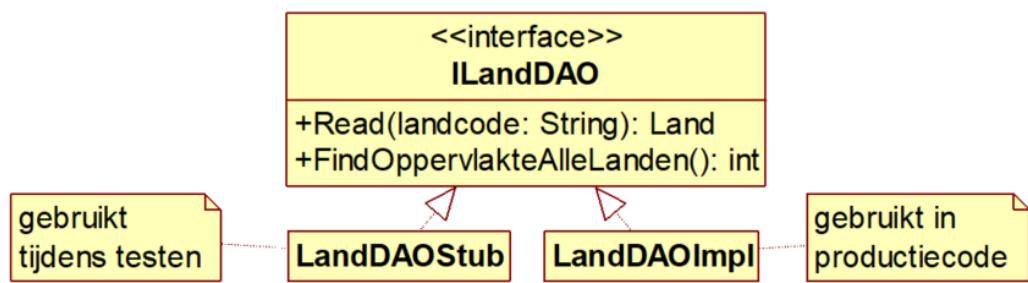
- een class kan meerdere dependencies hebben.
- De class bevat dan per dependency een extra private **readonly** variabele en een extra parameter in de constructor.

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 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.

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 }; // (1)
        }

        public int OppervlakteAlleLanden() // (2)
        {
            return 20;
        }
    }
}
  
```

- (1) De stub geeft een 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;

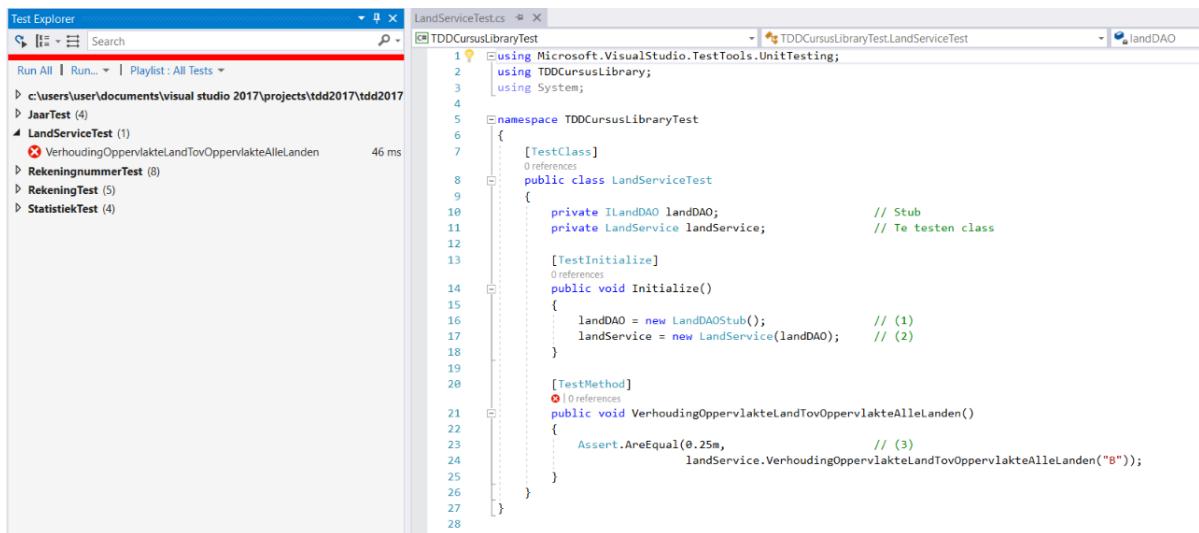
namespace TDDCursusLibraryTest
{
    [TestClass]
    public class LandServiceTest
    {
        private ILandDAO landDAO; // Stub
        private LandService landService; // Te testen class

        [TestInitialize]
        public void Initialize()
        {
            landDAO = new LandDAOStub(); // (1)
            landService = new LandService(landDAO); // (2)
        }

        [TestMethod]
        public void VerhoudingOppervlakteLandTovOppervlakteAlleLanden()
        {
            Assert.AreEqual(0.25m, // (3)
                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

```
using System;

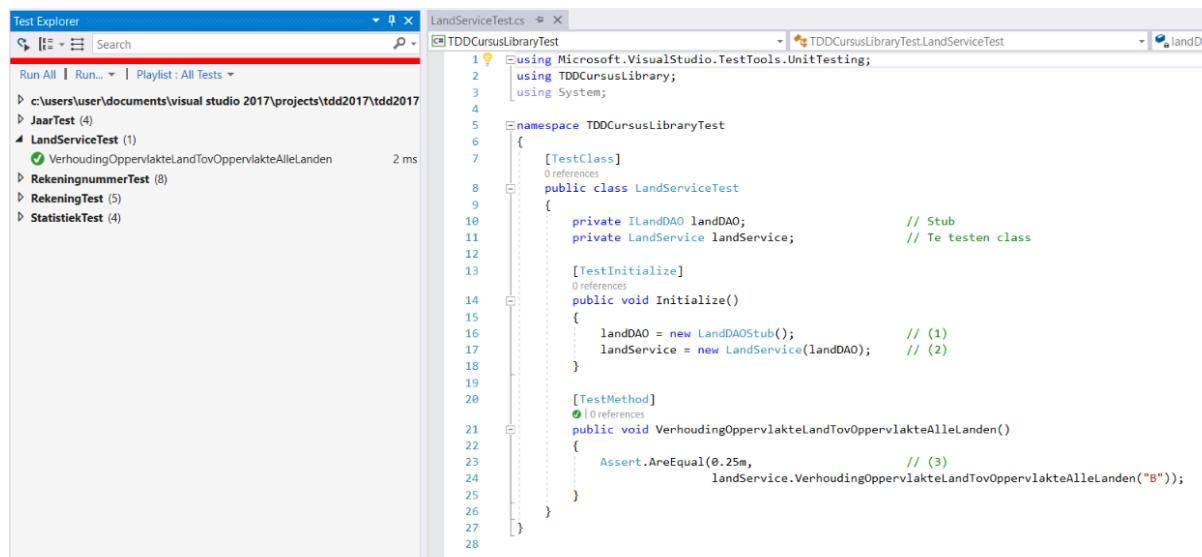
namespace TDDCursusLibrary
{
    public class LandService
    {
        // private LandDAO landDAO = new LandDAO();
        private readonly ILandDAO landDAO;

        // Constructor
        public LandService(ILandDAO landDAO)
        {
            this.landDAO = landDAO;
        }

        public decimal VerhoudingOppervlakteLandTovOppervlakteAlleLanden(string landcode)
        {
            // throw new NotImplementedException();

            var land = landDAO.Read(landcode);
            var 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.

7.2 TAAK 5 : STUB

Je maakt een class **WinstService**.

Deze class heeft een dependency, uitgedrukt in een interface **IOpbrengstDAO**. Deze interface bevat één method-declaratie:

```
decimal TotaleOpbrengst();
```

De class **WinstService** heeft een tweede dependency, uitgedrukt in een interface **IKostDAO**. Deze interface bevat één method-declaratie:

```
decimal TotaleKost();
```

De class **WinstService** bevat **readonly** property:

```
public Decimal Winst
```

De berekening van de winst is: totale opbrengst – totale kost

Je schrijft de class **WinstService** en de bijbehorende unit test.

In deze unit test gebruik je stubs voor **IOpbrengstDAO** en **IKostDAO**.

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("")` oproeft, krijg je een `Land`-object terug. Bij bepaalde testen zou het interessant zijn dat je dan `null` terug krijgt.

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.

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

namespace TDDCursusLibraryTest
{
    [TestClass]
    public class LandServiceTest
    {
        private ILandDAO landDAO; // Stub
        private LandService landService; // Te testen class

        [TestInitialize]
        public void Initialize()
        {
            landDAO = new LandDAOStub(); // (1)
            landService = new LandService(landDAO); // (2)
        }

        [TestMethod]
        public void VerhoudingOppervlakteLandTovOppervlakteAlleLanden()
        {
            Assert.AreEqual(0.25m, // (3)
                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.

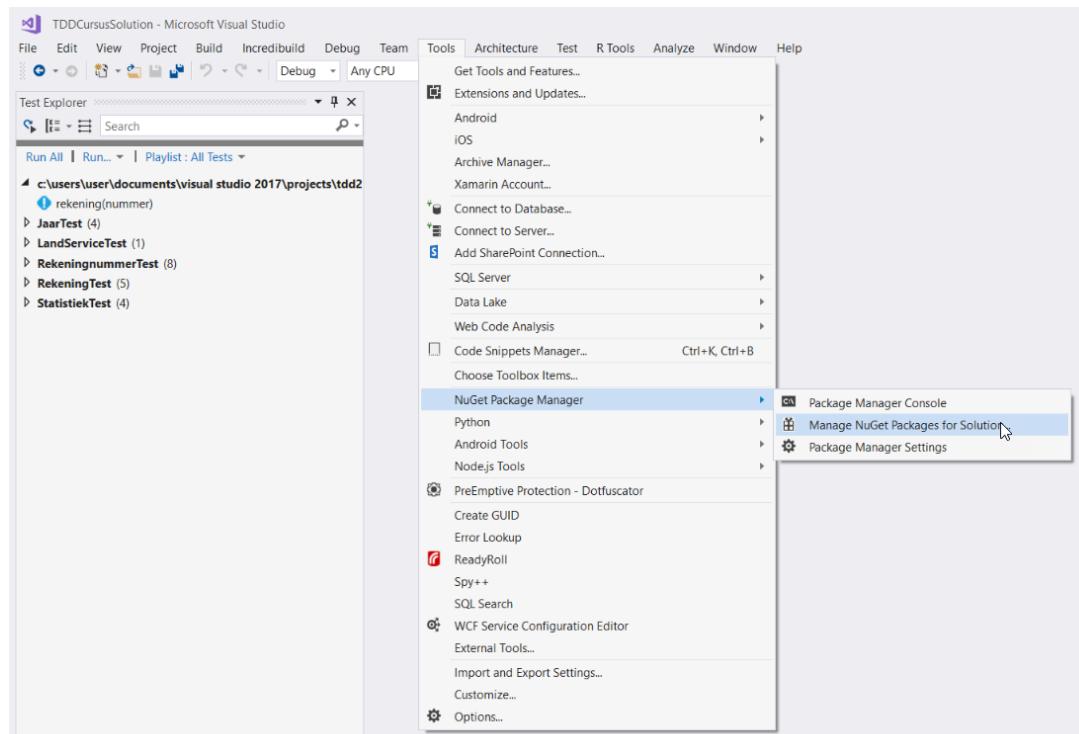
Dit is tijdrovend en er is een kans dat je fouten schrijft in deze code.

Je lost dit op 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.

8.2.2 DE MOQ LIBRARY TOEVOEGEN

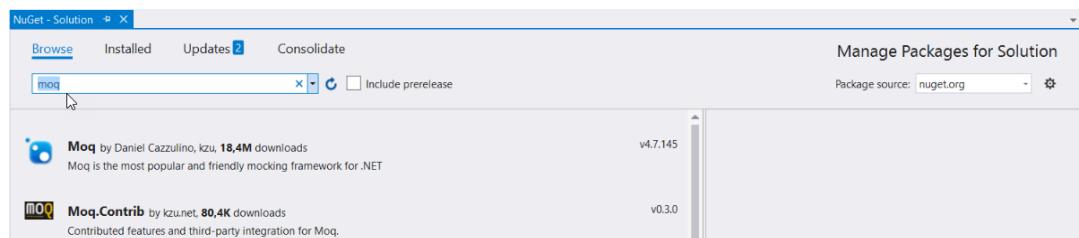
- Je kiest TOOLS, dan NuGet Package Manager, dan Manage NuGet Packages for Solution



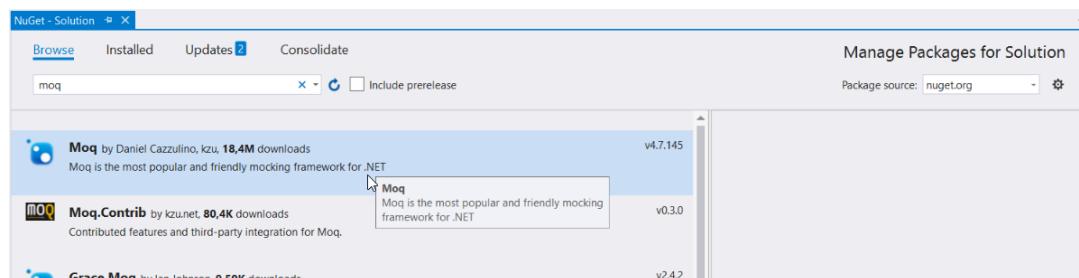
- Je kiest links **Browse**.



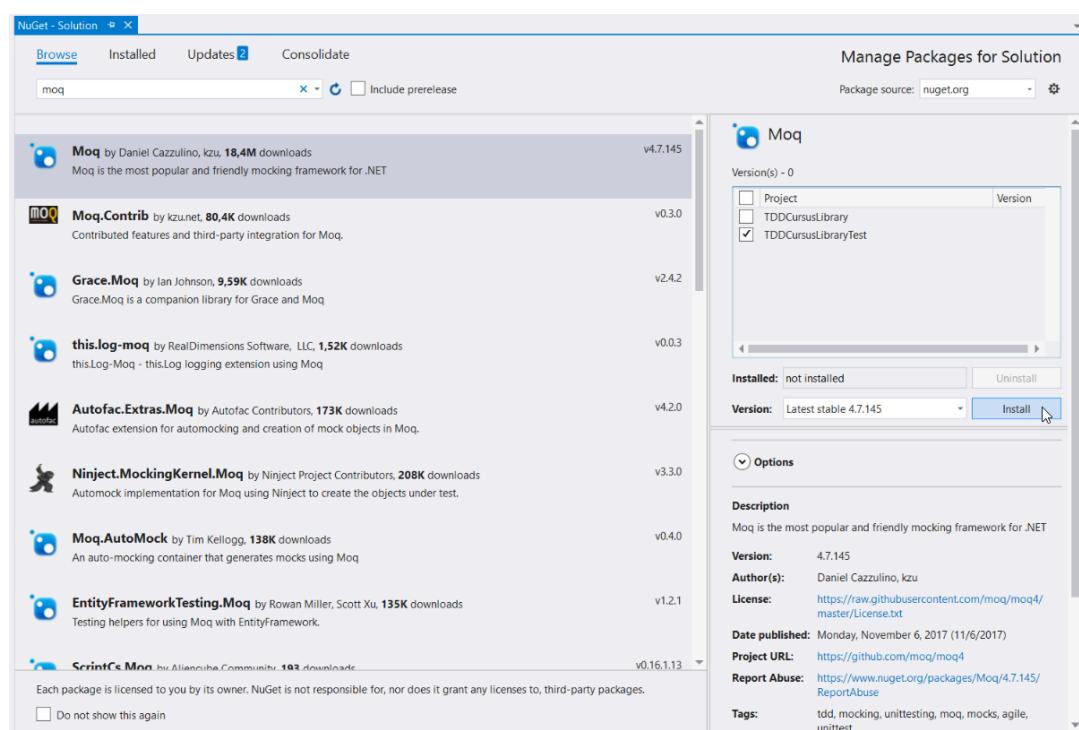
- Je tikt linksboven in het zoekvak **moq**



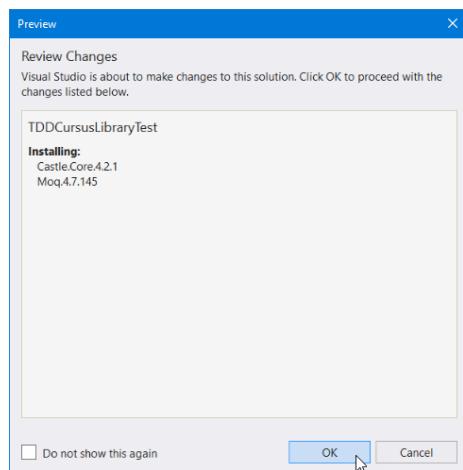
- Je kiest daaronder **Moq by Daniel Cazzulino**



- Je vinkt het project **TDDCursusLibraryTest** aan en klikt op **Install**



- Je klikt op **OK**



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>();           // (4)  
landDAO = mockFactory.Object;                   // (5)
```

```
using Microsoft.VisualStudio.TestTools.UnitTesting;  
using TDDCursusLibrary;  
using Moq;  
  
namespace TDDCursusLibraryTest  
{  
    [TestClass]  
    public class LandServiceTest  
    {  
        private ILandDAO landDAO;                      // Stub  
        private LandService landService;                // Te testen class  
  
        private Mock<ILandDAO> mockFactory;  
  
        [TestInitialize]  
        public void Initialize()  
        {  
            landDAO = new LandDAOStub();                // (1)  
  
            mockFactory = new Mock<ILandDAO>();          // (4)  
            landDAO = mockFactory.Object;                // (5)  
  
            landService = new LandService(landDAO);       // (2)  
        }  
  
        [TestMethod]  
        public void VerhoudingOppervlakteLandTovOppervlakteAlleLanden()
```

```

    {
        Assert.AreEqual(0.25m,                                     // (3)
                        landService.VerhoudingOppervlakteLandTovOppervlakteAlleLanden("B"));

        // hier gaan we straks verifiëren of landService de methods
        // read("B") en OppervlakteAlleALanden() heeft opgeroepen op landDAO.
    }
}
}

```

- (4) Een Moq object zal bij (5) een mock object voor je aanmaken.
 Je geeft tussen <> de interface mee die de mock bij (5) moet implementeren.
 (5) 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 **ILandDAO**.

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

Deze method implementaties zijn zeer eenvoudig.

Je ziet hieronder welke waarde ze teruggeven.

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

Returtype 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 **ILandDAO** 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 = mockFactory.Object;

mockFactory.Setup(eenLandDAO => eenLandDAO.OppervlakteAlleLanden()).Returns(20);           // (6)
mockFactory.Setup(eenLandDAO => eenLandDAO.Read("B")).Returns (new Land{
    Landcode = "B",
    Oppervlakte = 5
});                                                 // (7)

```

- (6) 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 returnwaarde 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.

- (7) 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.

```

Test Explorer
Run All | Run... | Playlist: All Tests
c:\users\user\documents\visual studio 2017\projects\tdd2
  ↗ rekening(nummer) 41 ms
  ↳ JaarTest (4)
  ↳ LandServiceTest (1)
    ↗ VerhoudingOppervlakteLandTotOppervlakteAlle... 479 ms
  ↳ RekeningnummerTest (8)
  ↳ RekeningTest (5)
  ↳ StatistiekTest (4)

LandServiceTest.cs
[TDDCursusLibraryTest]
[TestMethod]
public void VerhoudingOppervlakteLandTotOppervlakteAlleLanden()
{
    Assert.AreEqual(0.25m,                                     // (3)
                    landService.VerhoudingOppervlakteLandTotOppervlakteAlleLanden("B"));

    // hier gaan we straks verifiëren of landService de methods
    // read("B") en OppervlakteAlleALanden() heeft opgeroepen op landDAO.
}

```

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());

```

```

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

namespace TDDCursusLibraryTest
{
    [TestClass]
    public class LandServiceTest
    {
        private ILandDAO landDAO;                                // Stub

```

```

    private LandService landService;           // Te testen class

    private Mock<ILandDAO> mockFactory;

    [TestInitialize]
    public void Initialize()
    {
        landDAO = new LandDAOStub();           // (1)

        mockFactory = new Mock<ILandDAO>();   // (4)
        landDAO = mockFactory.Object;          // (5)

        mockFactory.Setup(eenLandDAO => eenLandDAO.OppervlakteAlleLanden()).
            Returns(20);                      // (6)

        mockFactory.Setup(eenLandDAO => eenLandDAO.Read("B")).
            Returns (new Land
            {
                Landcode = "B",
                Oppervlakte = 5
            });
            // (7)

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

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

    landService = new LandService(landDAO); // (2)
}

[TestMethod]
public void VerhoudingOppervlakteLandTovOppervlakteAlleLanden()
{
    Assert.AreEqual(0.25m,                         // (3)

    landService.VerhoudingOppervlakteLandTovOppervlakteAlleLanden("B"));

    // hier gaan we straks verifiëren of landService de methods
    // read("B") en OppervlakteAlleALanden() heeft opgeroepen op landDAO.
}
}
}

```

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 TestMethod `VerhoudingOppervlakteLandTovOppervlakteAlleLanden`

```

mockFactory.Verify(eenLandDAO => eenLandDAO.OppervlakteAlleLanden()); // (8)
mockFactory.Verify(eenLandDAO => eenLandDAO.Read("B"));

```

- (8) 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 returnwaarde van de expressie is de method die gedurende de test zou moeten opgeroepen worden. Als dit niet het geval is, mislukt de test.

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using TDCCursusLibrary;
using Moq;

namespace TDCCursusLibraryTest
{
    [TestClass]
    public class LandServiceTest
    {
        private ILandDAO landDAO;                                // Stub
        private LandService landService;                          // Te testen class

        private Mock<ILandDAO> mockFactory;

        [TestInitialize]
        public void Initialize()
        {
            landDAO = new LandDAOStub();                         // (1)

            mockFactory = new Mock<ILandDAO>();                // (4)
            landDAO = mockFactory.Object;                        // (5)

            mockFactory.Setup(eenLandDAO => eenLandDAO.OppervlakteAlleLanden()).
                Returns(20);                                    // (6)

            mockFactory.Setup(eenLandDAO => eenLandDAO.Read("B")).
                Returns (new Land
                {
                    Landcode = "B",
                    Oppervlakte = 5
                });
            // (7)

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

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

            landService = new LandService(landDAO); // (2)
        }

        [TestMethod]
        public void VerhoudingOppervlakteLandTovOppervlakteAlleLanden()
        {
            Assert.AreEqual(0.25m,                                // (3)
                            landService.VerhoudingOppervlakteLandTovOppervlakteAlleLanden("B"));

            // hier gaan we straks verifiëren of landService de methods
            // read("B") en OppervlakteAlleALanden() heeft opgeroepen op landDAO.

            mockFactory.Verify(eenLandDAO => eenLandDAO.OppervlakteAlleLanden()); // (8)
            mockFactory.Verify(eenLandDAO => eenLandDAO.Read("B"));
        }
    }
}
```

8.3 TAAK 6 : MOCK

Je vervangt de stubs in **WinstService** door mocks, aangemaakt met Mockito.

Je doet ook verificaties op deze stubs.

9 INTELLITEST

9.1 ALGEMEEN

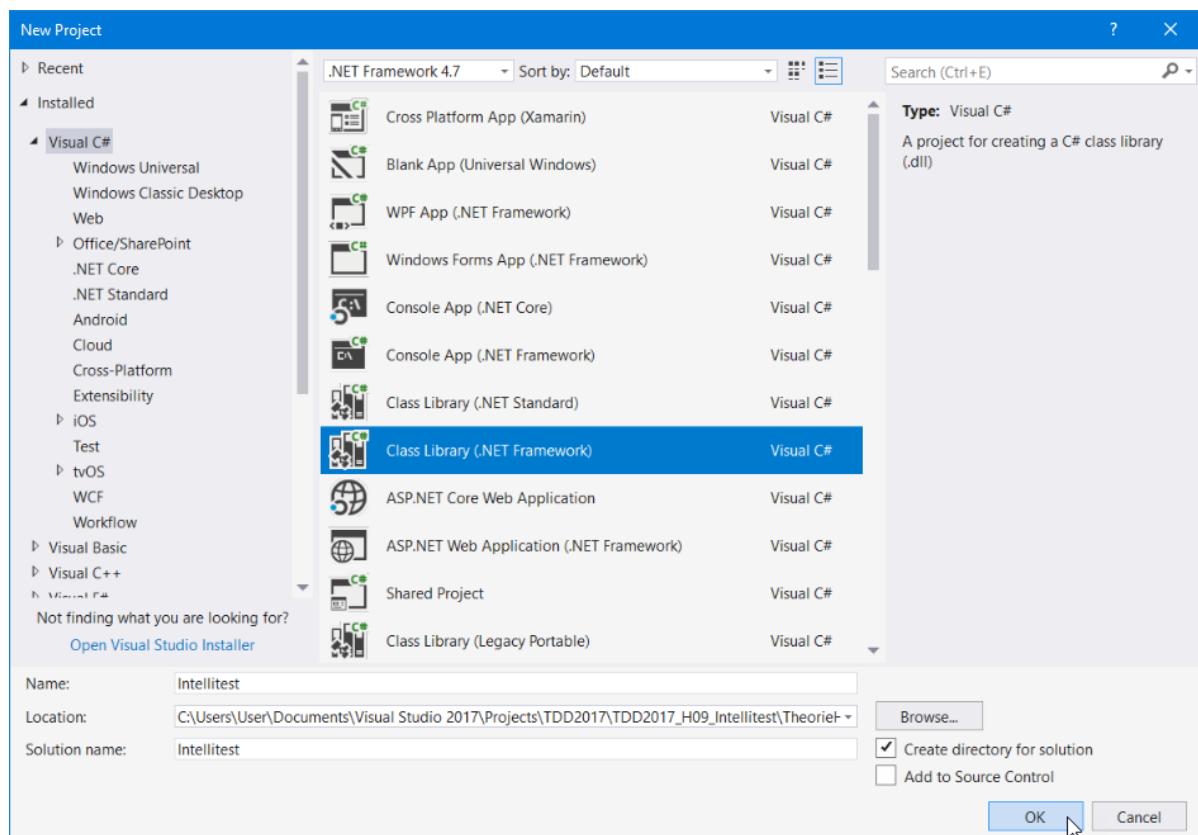
Sinds Visual Studio 2015 kan je automatisch unit tests laten genereren met IntelliTest (eerst werden ze Smart Unit Tests genoemd). Het helpt ons om een maximum aan code te testen (code coverage).

Aangezien je echter eerst je code moet schrijven voor je de tests kan laten genereren, is het niet echt volgens TDD-principes.

Je zal ook altijd nog zelf tests moeten toevoegen om te zien of je code voldoet aan de business rules.

9.1.1 CODEREN

Maak bijvoorbeeld een nieuwe DLL aan: [File](#), dan [New Project](#), dan [Class Library](#).



Geef de solution en het project een plaats en een naam en neem volgende methode op in je class:

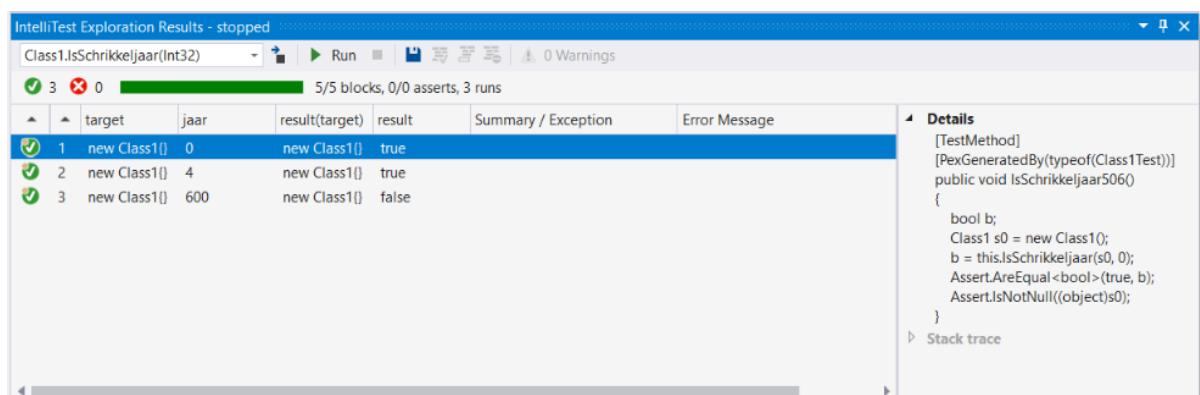
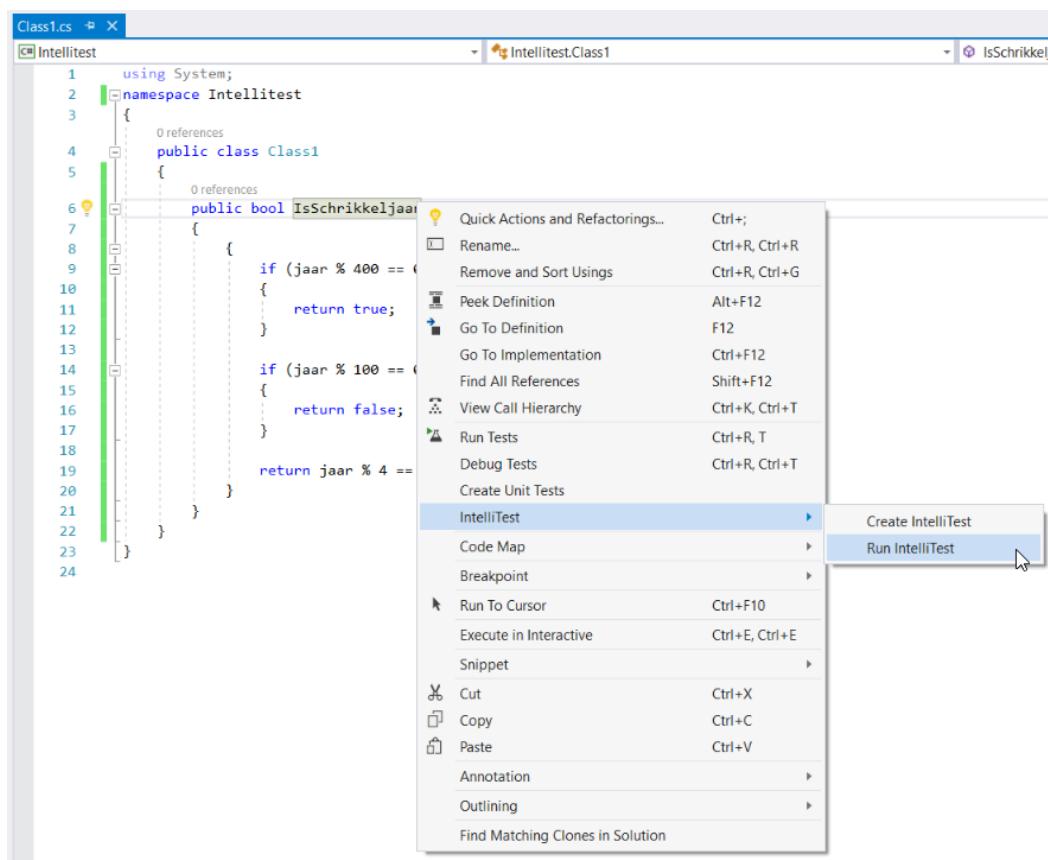
```
using System;
namespace Intellitest
{
    public class Class1
    {
        public bool IsSchrikkeljaar(int jaar)
        {
            if (jaar % 400 == 0)
            {
                return true;
            }
        }
    }
}
```

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

        return jaar % 4 == 0;
    }
}
```

9.1.2 RUN INTELLITEST

Als je nu rechtsklikt op de methode en **Run IntelliTest** klikt in de context-menu, krijg je de IntelliTest Explorator Results te zien:



IntelliTest Exploration Results - stopped

Class1.IsSchrikkeljaar(Int32)

5/5 blocks, 0/0 asserts, 3 runs

target	jaar	result(target)	result	Summary / Exception	Error Message
1 new Class1()	0	new Class1()	true		
2 new Class1()	4	new Class1()	true		
3 new Class1()	600	new Class1()	false		

Details

```
[TestMethod]
[PexGeneratedBy(typeof(Class1Test))]
public void IsSchrikkeljaar9580()
{
    bool b;
    Class1 s0 = new Class1();
    b = this.IsSchrikkeljaar(s0, 4);
    Assert.AreEqual<bool>(true, b);
    Assert.IsNotNull((object)s0);
}
```

Stack trace

IntelliTest Exploration Results - stopped

Class1.IsSchrikkeljaar(Int32)

5/5 blocks, 0/0 asserts, 3 runs

target	jaar	result(target)	result	Summary / Exception	Error Message
1 new Class1()	0	new Class1()	true		
2 new Class1()	4	new Class1()	true		
3 new Class1()	600	new Class1()	false		

Details

```
[TestMethod]
[PexGeneratedBy(typeof(Class1Test))]
public void IsSchrikkeljaar8340()
{
    bool b;
    Class1 s0 = new Class1();
    b = this.IsSchrikkeljaar(s0, 600);
    Assert.AreEqual<bool>(false, b);
    Assert.IsNotNull((object)s0);
}
```

Stack trace

IntelliTest genereert een set van unit tests gebaseerd op de parameters, die zoveel mogelijk code coveren.

Je ziet de parameter-waarden die werden gebruikt voor jaar. Als je op een test klikt, zie je in de **Details** rechts de gecoverde code. Ook grenswaarden worden getest.

9.1.3 CREËER EN SAVE HET TEST PROJECT

Selecteer alle tests en klik op het **Save**-icoontje in het IntelliTest Exploration Results window. Hierdoor wordt er een nieuw Unit Test Project toegevoegd aan de solution.

IntelliTest Exploration Results - stopped

Class1.IsSchrikkeljaar(Int32)

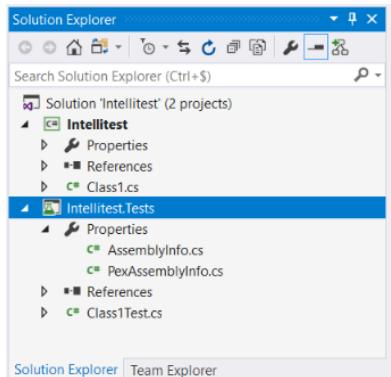
5/5 blocks, 0/0 asserts, 3 runs

target	jaar	result(target)	result	Summary / Exception	Error Message
1 new Class1()	0	new Class1()	true		
2 new Class1()	4	new Class1()	true		
3 new Class1()	600	new Class1()	false		

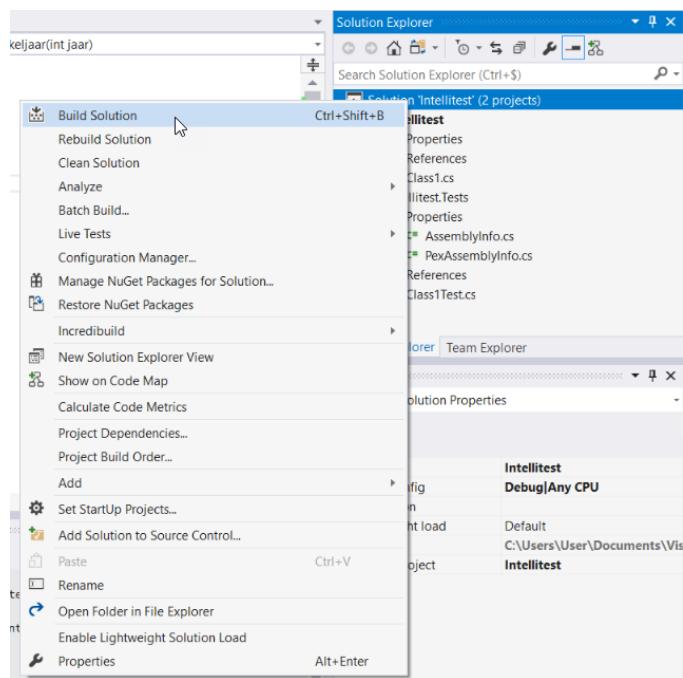
Save 3 runs

Details

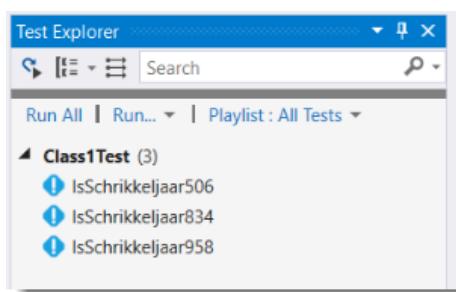
Stack trace



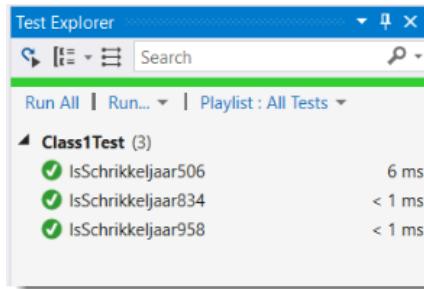
Je kan nu een Build uitvoeren.



Intellitest
Debug|Any CPU



In de Test Explorer zie je de tests. Je kan ze nu runnen.



10 VOORBEELDOPLOSSINGEN TAKEN

10.1 HOOFDSTUK 2 : PALINDROOM

10.1.1 WOORD

```
//using System;
using System.Linq;
namespace Palindroom
{
    public class Woord
    {
        public string Text { get; set; }

        public Woord(string woord)
        {
            Text = woord;
        }

        public bool IsPalindroom()
        {
            var omgekeerd = new string(Text.ToArray().Reverse().ToArray());
            return Text == omgekeerd;
        }

        public bool IsPalindroom01()
        {
            for (var teller = 0; teller < Text.Length / 2; teller++)
            {
                if (Text[teller] != Text[Text.Length - 1 - teller]) return false;
            }

            return true;
        }

        public bool IsPalindroom02()
        {
            return Text == new string(Text.ToCharArray().Reverse().ToArray());
        }
    }
}
```

10.1.2 WOORDTEST

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

using Palindroom;

namespace Palindroom_Test
{
    [TestClass]
    public class WoordTest
    {
        [TestMethod]
        public void LepelIsEenPalindroom()
        {
            Assert.IsTrue(new Woord("lepel").IsPalindroom());
        }

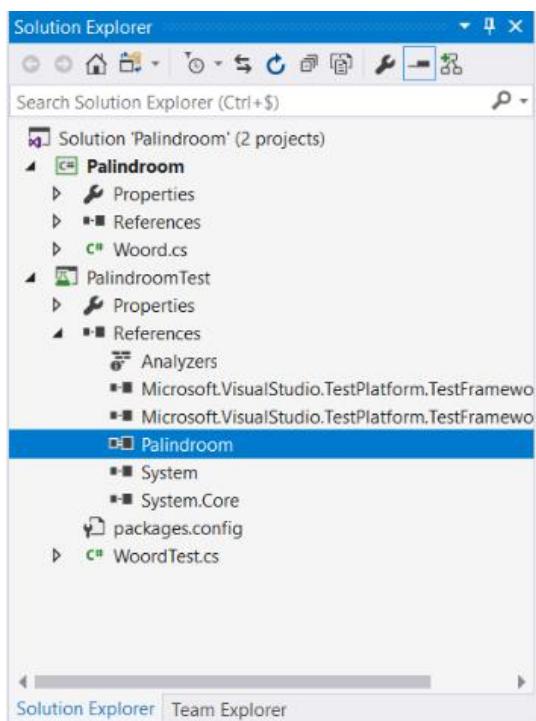
        [TestMethod]
    }
}
```

```
public void VorkIsGeenPalindroom()
{
    Assert.IsFalse(new Woord("vork").IsPalindroom());
}

[TestMethod]
public void eenLegeStringIsEenPalindroom()
{
    Assert.IsTrue(new Woord(String.Empty).IsPalindroom());
}

}
```

10.1.3 ADD REFERENCE



10.1.4 TEST

The screenshot shows the Visual Studio Test Explorer window. On the left, under 'Passed Tests (3)', there are three entries: 'eenLegeStringIsEenPalindroom' (1 ms), 'LepelIsEenPalindroom' (12 ms), and 'VorkIsGeenPalindroom' (< 1 ms). Below this is a 'Summary' section stating 'Last Test Run Passed (Total Run Time 0:00:00,853)' and '3 Tests Passed'. On the right, the code editor displays the 'WoordTest.cs' file, which contains the following C# code:

```
1  using System;
2  using Microsoft.VisualStudio.TestTools.UnitTesting;
3
4  using Palindroom;
5
6  namespace Palindroom_Test
7  {
8      [TestClass]
9      public class WoordTest
10     {
11         [TestMethod]
12         public void LepelIsEenPalindroom()
13         {
14             Assert.IsTrue(new Woord("lepel").IsPalindroom());
15         }
16
17         [TestMethod]
18         public void VorkIsGeenPalindroom()
19         {
20             Assert.IsFalse(new Woord("vork").IsPalindroom());
21         }
22
23         [TestMethod]
24         public void eenLegeStringIsEenPalindroom()
25         {
26             Assert.IsTrue(new Woord(String.Empty).IsPalindroom());
27         }
28     }
29 }
30
```

10.2 HOOFDSTUK 3 : VEILING

10.2.1 DE CLASS VEILING ZONDER ECHTE CODE IN ZIJN METHODS

```
using System;

namespace Veiling
{
    public class Veiling
    {
        private decimal hoogsteBod;

        public void DoeBod(decimal bedrag)
        {
            throw new NotImplementedException();
        }

        public decimal HoogsteBod
        {
            get
            {
                throw new NotImplementedException();
            }
        }
    }
}
```

10.2.2 DE UNIT TEST VEILINGTEST

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

using Veiling;

namespace VeilingTest
{
    [TestClass]
    public class VeilingTest
    {
        [TestMethod]
        public void HetHoogsteBodVanEenNieuweVeilingStaatOpNul()
        {
            Assert.AreEqual(0m, new Veiling.Veiling().HoogsteBod);
        }

        [TestMethod]
        public void NaEenEersteBodIsHetHoogsteBodGelijkAanHetBedragVanDitBod()
        {
            var veiling = new Veiling.Veiling();
            veiling.DoeBod(100m);
            Assert.AreEqual(100m, veiling.HoogsteBod);
        }

        [TestMethod]
        public void NaMeerdereBiedingenIsHetHoogsteBodGelijkAanHetBedragVanDitBod()
        {
            var veiling = new Veiling.Veiling();
            veiling.DoeBod(100m );
            veiling.DoeBod(200m);
            veiling.DoeBod(150m);
            Assert.AreEqual(200, veiling.HoogsteBod);
        }
    }
}
```

10.2.3 DE CLASS VEILING MET ECHTE CODE IN ZIJN METHODS

```
//using System;

namespace Veiling
{
    public class Veiling
    {
        private decimal hoogsteBod;

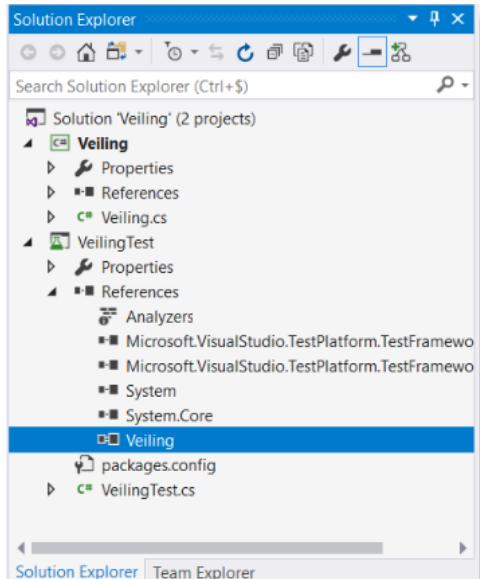
        public void DoeBod(decimal bedrag)
        {
            //throw new NotImplementedException();

            if (bedrag > hoogsteBod)
            {
                hoogsteBod = bedrag;
            }
        }

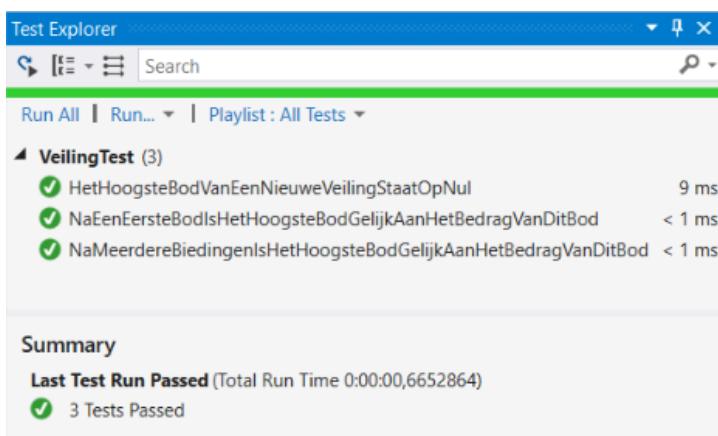
        public decimal HoogsteBod
        {
            get
            {
                //throw new NotImplementedException();

                return hoogsteBod;
            }
        }
    }
}
```

10.2.4 ADD REFERENCE



10.2.5 TEST



10.3 HOOFDSTUK 4 : VEILING - TEST FIXTURES

10.3.1 VEILINGTEST.CS

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

using Veiling;

namespace VeilingTest
{
    [TestClass]
    public class VeilingTest
    {
        private Veiling.Veiling veiling;

        [TestInitialize]
```

```
public void Initialize()
{
    veiling = new Veiling.Veiling();
}

[TestMethod]
public void HetHoogsteBodVanEenNieuweVeilingStaatOpNul()
{
    Assert.AreEqual(0m, new Veiling.Veiling().HoogsteBod);
}

[TestMethod]
public void NaEenEersteBodIsHetHoogsteBodGelijkAanHetBedragVanDitBod()
{
//    var veiling = new Veiling.Veiling();
//    veiling.DoeBod(100m);
//    Assert.AreEqual(100m, veiling.HoogsteBod);
}

[TestMethod]
public void NaMeerdereBiedingenIsHetHoogsteBodGelijkAanHetBedragVanDitBod()
{
//    var veiling = new Veiling.Veiling();
//    veiling.DoeBod(100m);
//    veiling.DoeBod(200m);
//    veiling.DoeBod(150m);
//    Assert.AreEqual(200, veiling.HoogsteBod);
}
}
```

10.4 HOOFDSTUK 5 : ISBN

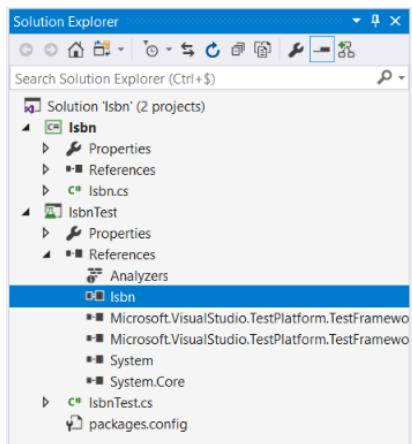
10.4.1 DE CLASS ISBN ZONDER ECHTE CODE IN ZIJN METHODS

```
using System;

namespace TDDCursusLibrary
{
    public class Isbn
    {
        public Isbn(long nummer)
        {
            throw new NotImplementedException();
        }

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

10.4.2 ADD REFERENCE



10.4.3 DE UNIT TEST ISBNTEST

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

namespace TDDCursusLibraryTest
{
    [TestClass]
    public class IsbnTest
    {
        [TestMethod, ExpectedException(typeof(ArgumentException))]
        public void HetNummer0IsVerkeerd()
        {
            new Isbn(0);
        }

        [TestMethod, ExpectedException(typeof(ArgumentException))]
        public void EenNegatiefNummerIsVerkeerd()
        {
            new Isbn(-9789027439642L);
        }

        [TestMethod, ExpectedException(typeof(ArgumentException))]
        public void EenNummerMet12CijfersIsVerkeerd()
        {
            new Isbn(978902743964L);
        }

        [TestMethod, ExpectedException(typeof(ArgumentException))]
        public void EenNummerMet14CijfersIsVerkeerd()
        {
            new Isbn(97890274396421L);
        }

        [TestMethod, ExpectedException(typeof(ArgumentException))]
        public void EenNummerMet13CijfersMetVerkeerdControleGetal2()
        {
            new Isbn(8789027439642L);
        }

        [TestMethod]
        public void EenNummerMet13CijfersMetCorrectControleGetal2()
        {
            new Isbn(9789027439642L);
        }

        [TestMethod, ExpectedException(typeof(ArgumentException))]
        public void EenNummerMet13CijfersMetVerkeerdControleGetal0()
        {
            new Isbn(7789227439640L);
        }
    }
}
```

```
        }

        [TestMethod]
        public void EenNummerMet13CijfersMetCorrectControleGetal0()
        {
            new Isbn(9789227439640L);
        }
    }
```

10.4.4 DE CLASS ISBN MET ECHTE CODE IN ZIJN METHODS

```
using System;

namespace TDDCursusLibrary
{
    public class Isbn
    {
        private const long GrootsteGetalMet13_Cijfers = 9999999999999L;
        private const long KleinsteGetalMet13_Cijfers = 1000000000000L;
        private long nummer;

        public Isbn(long nummer)
        {
            throw new NotImplementedException();

            if (nummer < KleinsteGetalMet13_Cijfers || nummer >
GrootsteGetalMet13_Cijfers)
            {
                throw new ArgumentException();
            }

            var somEvenCijfers = 0L;
            var somOnEvenCijfers = 0L;
            var teVerwerkenCijfers = nummer / 10;

            for (int teller = 0; teller != 6; teller++)
            {
                somEvenCijfers += teVerwerkenCijfers % 10;
                teVerwerkenCijfers /= 10;
                somOnEvenCijfers += teVerwerkenCijfers % 10;
                teVerwerkenCijfers /= 10;
            }

            var controleGetal = somEvenCijfers * 3 + somOnEvenCijfers;
            var naastGelegenHoger10Tal = controleGetal - controleGetal % 10 + 10;
            var verschil = naastGelegenHoger10Tal - controleGetal;
            var laatsteCijfer = nummer % 10;

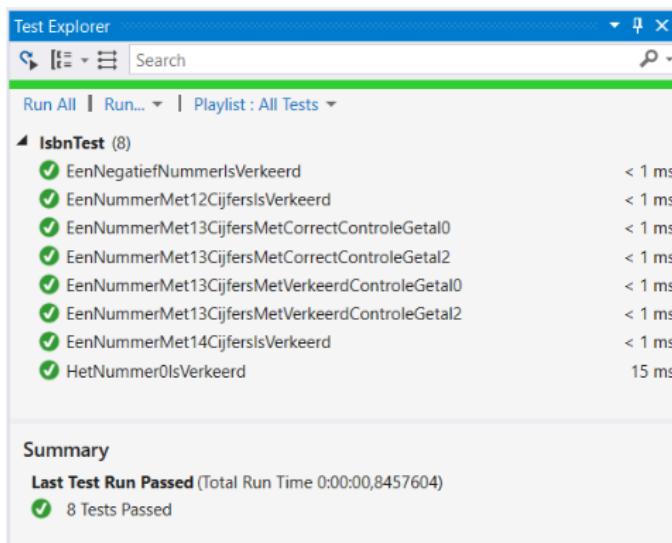
            if (verschil == 10)
            {
                if (laatsteCijfer != 0)
                {
                    throw new ArgumentException();
                }
            }
            else
            {
                if (laatsteCijfer != verschil)
                {
                    throw new ArgumentException();
                }
            }

            this.nummer = nummer;
        }

        public override string ToString()
        {
            throw new NotImplementedException();
            return nummer.ToString();
        }
    }
}
```

```
        }  
    }
```

10.4.5 TESTS



10.5 HOOFDSTUK 7 : STUB

10.5.1 IOPBRENGSTDÃO

```
namespace TDDCursusLibrary  
{  
    public interface IOpbrengstdao  
    {  
        decimal TotaleOpbrengst();  
    }  
}
```

10.5.2 IKOSTDAO

```
namespace TDDCursusLibrary  
{  
    public interface IKostDAO  
    {  
        decimal TotaleKost();  
    }  
}
```

10.5.3 WINSTSERVICE

```
using System;  
  
namespace TDDCursusLibrary  
{  
    public class WinstService  
    {  
        private readonly IOpbrengstdao opbrengstdao;  
        private readonly IKostDAO kostDAO;  
  
        public WinstService(IOpbrengstdao opbrengstdao, IKostDAO kostDAO)  
        {
```

```
        this.opbrengstDAO = opbrengstDAO;
        this.kostDAO = kostDAO;
    }

    public Decimal Winst
    {
        get
        {
            throw new NotImplementedException();
        }
    }
}
```

10.5.4 OPBRENGSTDIAOSTUB

```
using TDDCursusLibrary;

namespace TDDCursusLibraryTest
{
    class OpbrengstDAOSTub : IOpbrengstDAO
    {
        public decimal TotaleOpbrengst()
        {
            return 200m;
        }
    }
}
```

10.5.5 KOSTDAOSTUB

```
using TDDCursusLibrary;

namespace TDDCursusLibraryTest
{
    class KostDAOSTub : IKostDAO
    {
        public decimal TotaleKost()
        {
            return 169m;
        }
    }
}
```

10.5.6 WINSTSERVICETEST

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

namespace TDDCursusLibraryTest
{
    [TestClass]
    public class WinstServiceTest
    {
        private WinstService winstService;
        private IKostDAO kostDAO;
        private IOpbrengstDAO opbrengstDAO;

        [TestInitialize]
        public void Initialize()
        {
            kostDAO = new KostDAOSTub();
            opbrengstDAO = new OpbrengstDAOSTub();
            winstService = new WinstService(opbrengstDAO, kostDAO);
        }
    }
}
```

```
        }

        [TestMethod]
        public void WinstIsOpbrengstMinKost()
        {
            Assert.AreEqual(31m, winstService.Winst);
        }
    }
}
```

10.5.7 ECHTE CODE IN DE PROPERTY WINST VAN WINSTSERVICE

```
using System;

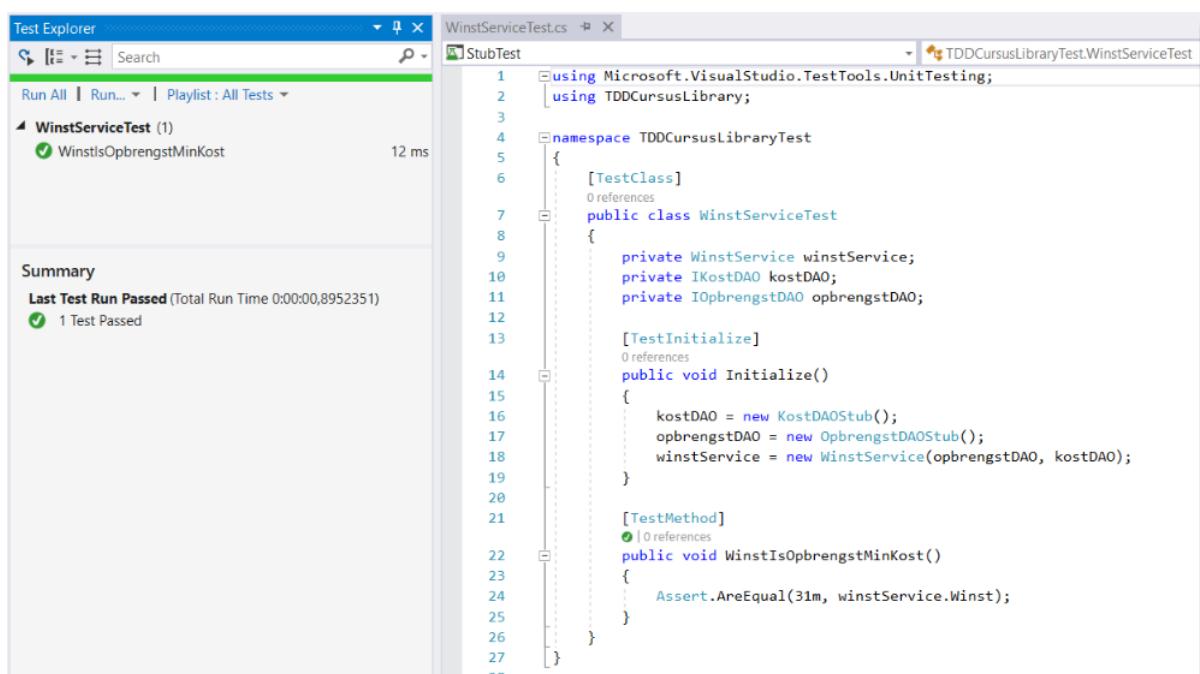
namespace TDDCursusLibrary
{
    public class WinstService
    {
        private readonly IOpbrengstDAO opbrengstDAO;
        private readonly IKostDAO kostDAO;

        public WinstService(IOpbrengstDAO opbrengstDAO, IKostDAO kostDAO)
        {
            this.opbrengstDAO = opbrengstDAO;
            this.kostDAO = kostDAO;
        }

        public Decimal Winst
        {
            get
            {
                // throw new NotImplementedException();

                return opbrengstDAO.TotaleOpbrengst() - kostDAO.TotaleKost();
            }
        }
    }
}
```

10.5.8 TEST



10.6 HOOFDSTUK 8 : MOCK

10.6.1 INSTALL MOQ LIBRARY

10.6.2 WINSTSERVICETEST.CS

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

namespace TDDCursusLibraryTest
{
    [TestClass]
    public class WinstServiceTest
    {
        private WinstService winstService;
        private IKostDAO kostDAO;
        private IOpbrengstDAO opbrengstDAO;

        private Mock<IKostDAO> mockKostDAO;
        private Mock<IOpbrengstDAO> mockOpbrengstDAO;

        [TestInitialize]
        public void Initialize()
        {
            // kostDAO = new KostDAOStub();
            // opbrengstDAO = new OpbrengstDAOStub();

            mockKostDAO = new Mock<IKostDAO>();
            mockOpbrengstDAO = new Mock<IOpbrengstDAO>();

            kostDAO = mockKostDAO.Object;
            opbrengstDAO = mockOpbrengstDAO.Object;

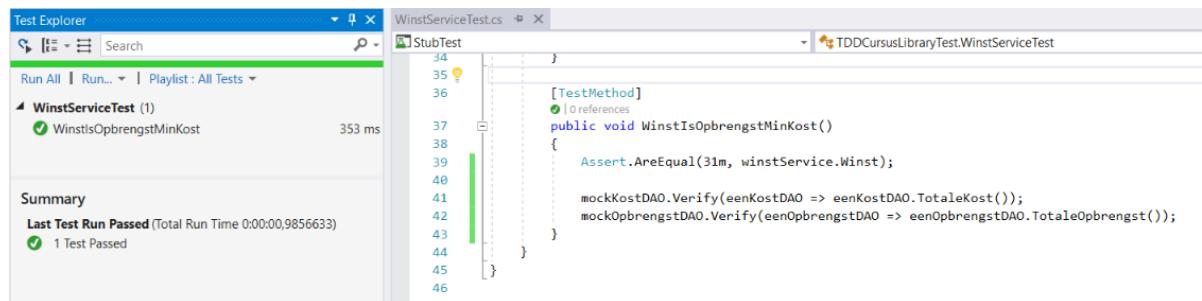
            mockOpbrengstDAO.Setup(
                eenOpbrengstDAO => eenOpbrengstDAO.TotaleOpbrengst()).Returns(200m);
            mockKostDAO.Setup(eenKostDAO => eenKostDAO.TotaleKost()).Returns(169m);

            winstService = new WinstService(opbrengstDAO, kostDAO);
        }

        [TestMethod]
        public void WinstIsOpbrengstMinKost()
        {
            Assert.AreEqual(31m, winstService.Winst);

            mockKostDAO.Verify(eenKostDAO => eenKostDAO.TotaleKost());
            mockOpbrengstDAO.Verify(eenOpbrengstDAO => eenOpbrengstDAO.TotaleOpbrengst());
        }
    }
}
```

10.6.3 TEST



11 COLOFON

Domeinexpertisemanager: Jean Smits
Moduleverantwoordelijke: Hans Desmet
Medewerkers: Hans Desmet
Johan Vandaele
Versie: 15-dec-2017
Nummer dotatielijst: