

Unit Testing (1)

.NET advanced

Topics

- Waarom software testing ?
- Wat is unit testing ?
- C# unittesting frameworks
- My first test
- Uitvoeren van unit tests
- Schrijven van een unit test
- White box vs. Black box testing
- Wat te testen:
 - Happy path
 - Edge cases
 - Unhappy path
- Code coverage
- Refactoring
- AAA methode
- Voordelen van unit testing
- Wanneer unit testing ?

Waarom software testing ?

- Werkt de software correct ?
- Voldoet de software aan de eisen ?
- Is de software stabiel ?
- Werkt deze ook onder alle omstandigheden ?
- Is deze performant genoeg ?
- Is deze gebruiksvriendelijk ?
-



Meest gekende bugs

A space error: \$370 million for an integer overflow

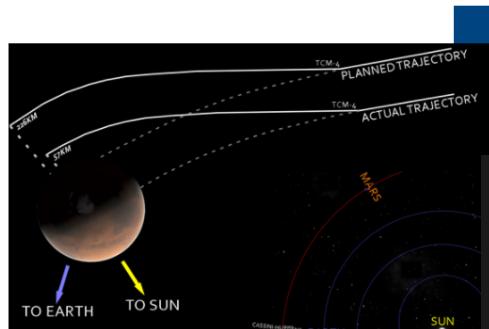
02.09.2016 / HOWNOT2CODE

Start. 37 seconds of flight. KaBOOM! 10 years and 7 billion dollars are turning into dust.



The programmers were to blame for everything.

4 – ¿FEET OR METERS? THE MARS CLIMATE ORBITER NAV BUG



The Mars Climate Orbiter was launched in 1998 with the goal of studying the climate on Mars, although it never managed to fulfill its scientific objectives.

HOW NOT TO CODE

Planned trajectory -vs- Actual trajectory (Source: Wikimedia)

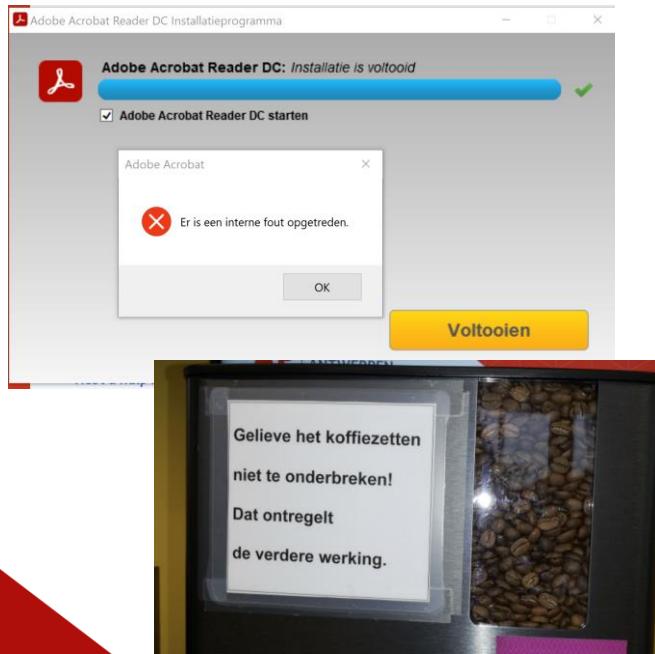
The accidents

The Therac-25 went into service in 1983. For several years and thousands of patients there were no problems. On June 3, 1985, a woman was being treated for breast cancer. She had been prescribed 200 Radiation Absorbed Dose (rad) in the form of a 10 MeV electron beam. The patient felt a tremendous heat when the machine powered up. It wasn't known at the time, but she had been burned by somewhere between 10,000 and 20,000 rad. The patient lived, but lost her left breast and the use of her left arm due to the radiation.

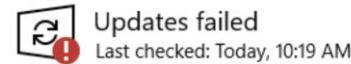
On July 26, a second patient was burned at The Ontario Cancer Foundation in Hamilton, Ontario, Canada. This patient died in November of that year.



Meer bekende problemen...



Windows Update



There were problems installing some updates, but we'll try again later.

Feature update to Windows 10, version 2004 - Error 0xd0000034

Retry



Error



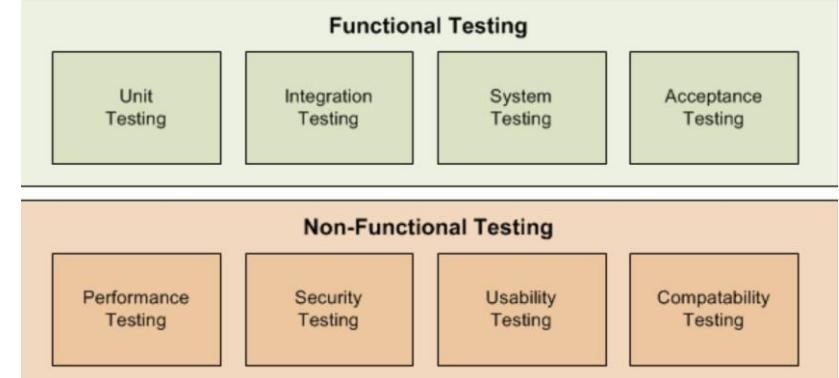
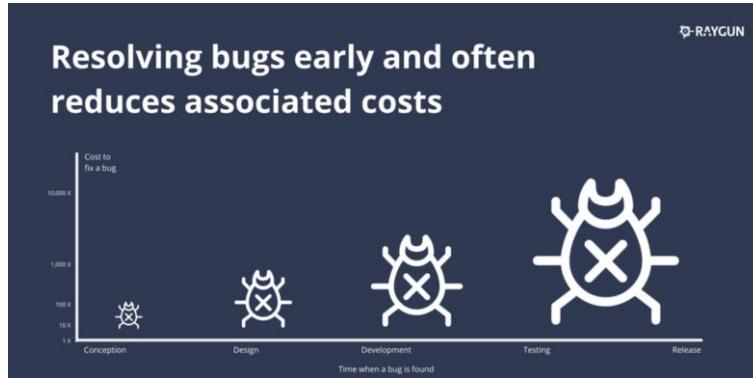
The system cannot find the file specified

OK



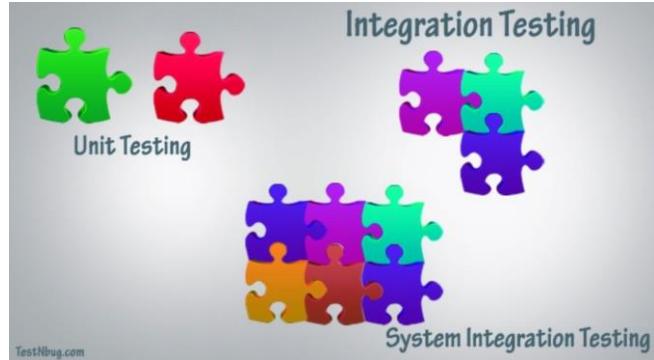
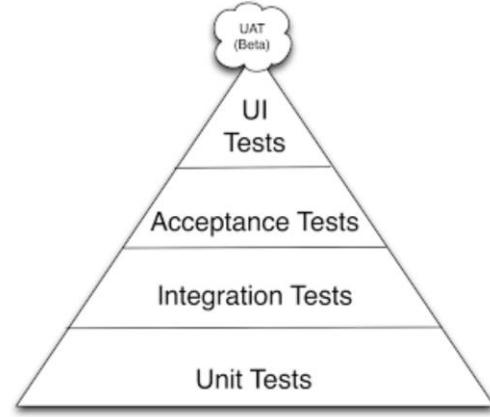
Software testing ?

- Er bestaan heel wat verschillende vormen van testing
- Manual testing vs. automated testing
- Grottere projecten => afzonderlijk QA (test) team
- Maar ook developers kunnen testen uitvoeren !



Wat is Unit Testing ?

- Testen van je code op het laagste niveau
- Testen van de kleinst testbare delen van de applicatie
- Een “unit” kan zijn:
 - Functie
 - Klasse, of een deel van een klasse (bv. Een bepaalde methode)



TestNug.com

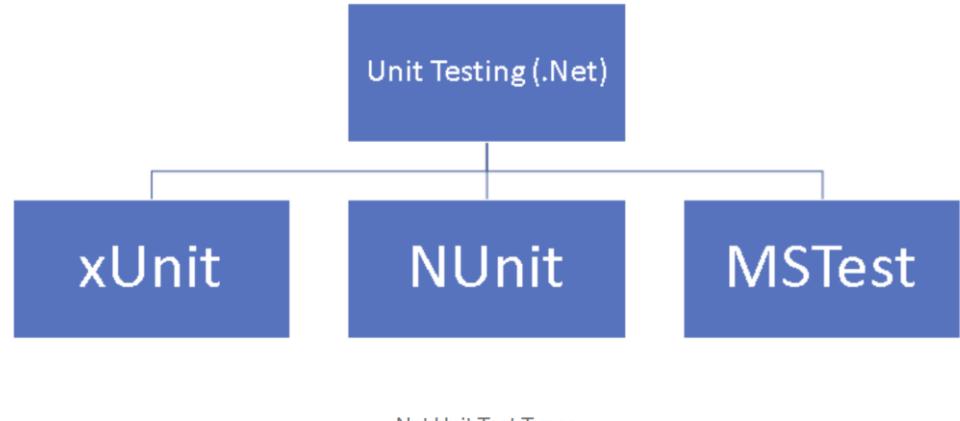
Waarom unit testing ?

Why Test Code

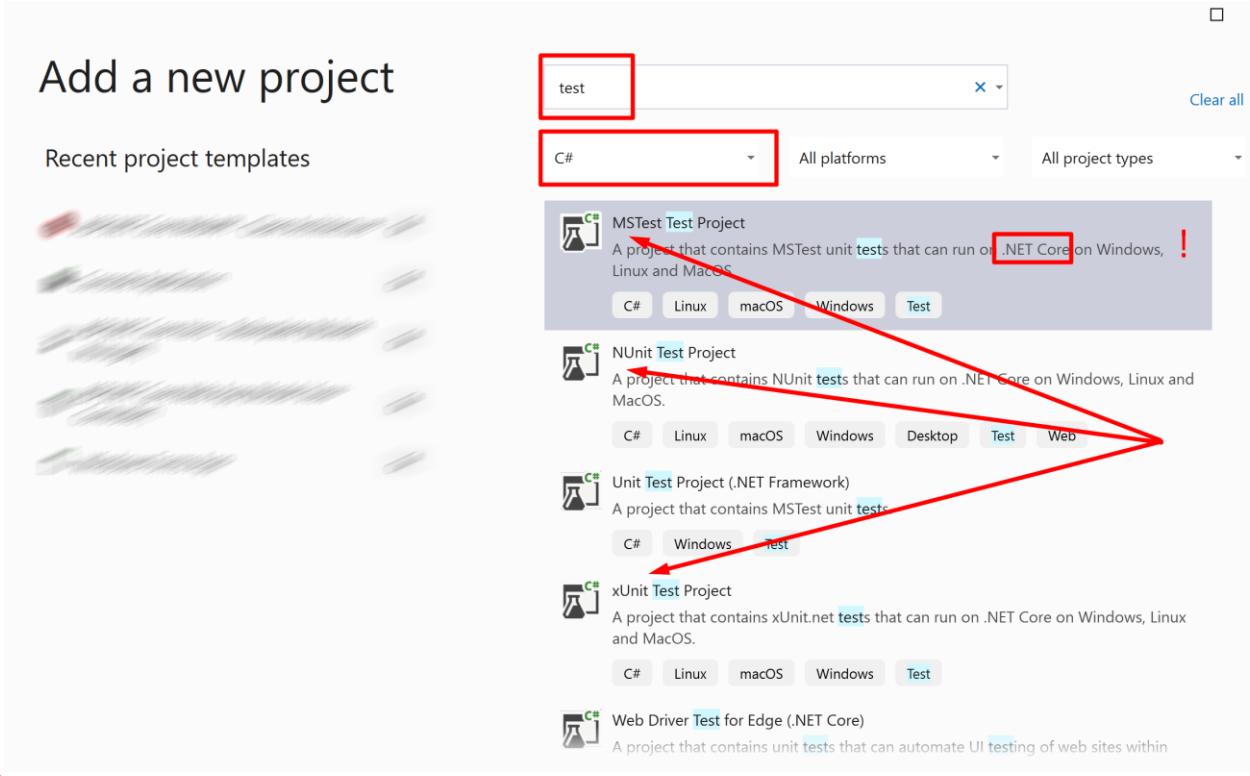


C# unit testing frameworks

- Standaard kan je vanuit Visual Studio opteren om te testen met 1 van deze 3 test frameworks.
 - Wij gaan gebruik maken van MSTest
 - =Microsoft testing framework



Toevoegen van een “Test” Project



My first Unit tests..

- Een unit test is een methode in een klasse...
- De klasse moet echter voorzien worden van het **[TestClass]** attribuut
- De methode moet voorzien worden van het **[TestMethod]** attribuut.



The screenshot shows a portion of a C# code editor. At the top, there are two 'using' statements: 'using Microsoft.VisualStudio.TestTools.UnitTesting;' and 'using MyLibrary;'. Below that is a namespace declaration: 'namespace TestProject'. Inside the namespace, there is a class definition: '[TestClass]' followed by 'public class FactorialTests'. This class contains two methods, both annotated with '[TestMethod]': 'public void CalcFactorialOf5()' and 'public void CalcFactorialOf7()'. A red arrow points from the text 'De klasse moet echter voorzien worden van het [TestClass] attribuut' to the '[TestClass]' attribute above the class definition. Another red box highlights the 'CalcFactorialOf5()' method, and a third red box highlights the 'CalcFactorialOf7()' method.

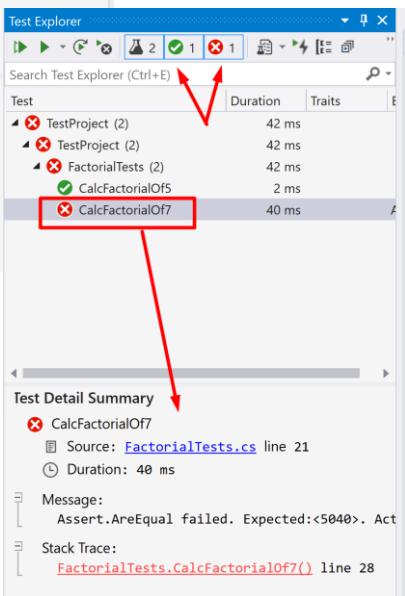
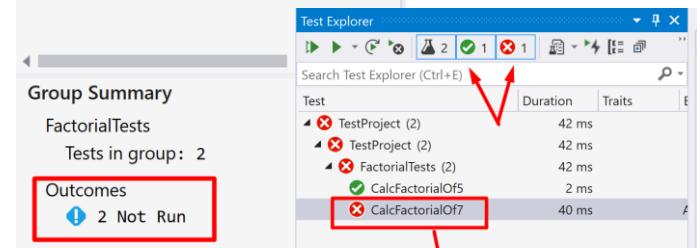
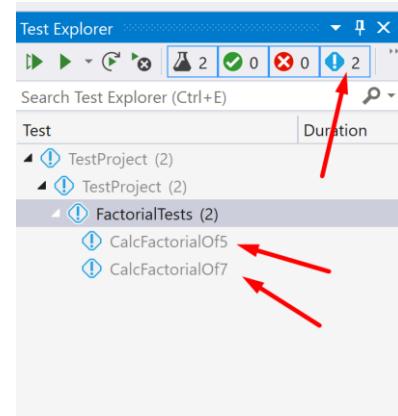
```
using Microsoft.VisualStudio.TestTools.UnitTesting;
using MyLibrary;

namespace TestProject
{
    [TestClass]
    public class FactorialTests
    {
        [TestMethod]
        public void CalcFactorialOf5()
        {
            // Blurred code
        }

        [TestMethod]
        public void CalcFactorialOf7()
        {
            // Blurred code
        }
    }
}
```

Uitvoeren van een unit test

- Via de ‘Test explorer’ krijg je alle unit tests van het test project te zien als boomstructuur.
- Je kan van hieruit (bepaalde / alle) testen laten uitvoeren.
- Je kan zien welke tests reeds werden uitgevoerd en of ze al dan niet gefaald / gelukt waren.



Schrijven van een unit test

- In een unit test gaan we uiteraard de te testen methode of klasse aanroepen net zoals we dat in echte productie code zouden gaan doen.
- We maken een instantie van de unit die we willen testen
- We roepen de methode die we willen testen aan met de nodige parameters

```
namespace TestProject
{
    [TestClass]
    0 references
    public class FactorialTests
    {
        /// <summary>
        /// Test the calculation of the factorial for the number 5
        /// </summary>
        [TestMethod]
        0 | 0 references
        public void CalcFactorialOf5()
        {
            var fac = new Factorial();
            var expectedResult = 120; //= 5 * 4 * 3 * 2 * 1;

            var actualResult = fac.Calculate(5);

            Assert.AreEqual(expectedResult, actualResult);
        }
    }
}
```

Schrijven van een unit test (2)

- Bij een unit test gaan we nu echter het **bekomen resultaat vergelijken met het verwachte referentie resultaat** (waarvan we 100% zeker zijn dat het correct is).
- Via de unit test **Assert helpers** kunnen we de voorwaarden instellen die nodig zijn om de test te doen slagen.

```
namespace TestProject
{
    [TestClass]
    0 references
    public class FactorialTests
    {
        /// <summary>
        /// Test the calculation of the factorial for the number 5
        /// </summary>
        [TestMethod]
        0 references
        public void CalcFactorialOf5()
        {
            var fac = new Factorial();
            var expectedResult = 120; //= 5 * 4 * 3 * 2 * 1;

            var actualResult = fac.Calculate(5);

            Assert.AreEqual(expectedResult, actualResult);
        }
    }
}
```

Gefaalde unit test(s) ?

The screenshot shows the Visual Studio interface with the Test Explorer and the code editor side-by-side.

Test Explorer:

Test	Duration	Traits	Error Message
TestProject (2)	83 ms		
TestProject (2)	83 ms		
FactorialTests (2)	83 ms		
CalcFactorialOf5	43 ms		Assert.AreEqual failed. Expected:<220>. Actual:<120>.
CalcFactorialOf7	40 ms		Assert.AreEqual failed. Expected:<5040>. Actual:<120>.

Test Detail Summary:

- CalcFactorialOf5
 - Source: [FactorialTests.cs](#) line 13
 - Duration: 43 ms
- Message:
Assert.AreEqual failed. Expected:<220>. Actual:<120>.
- Stack Trace:
[FactorialTests.CalcFactorialOf5\(\)](#) line 20

Code Editor:

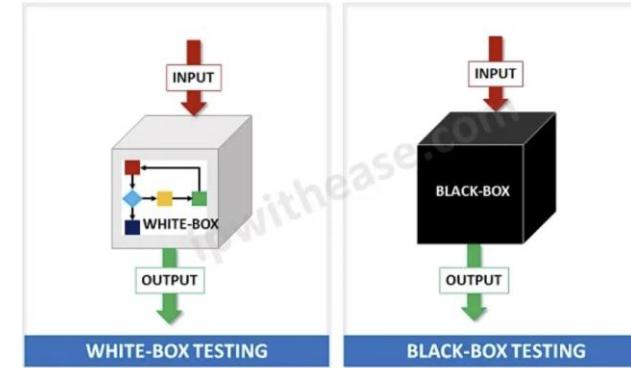
```
[TestClass]
public class FactorialTests
{
    /// <summary>
    /// Test the calculation of the factorial for the number 5
    /// </summary>
    [TestMethod]
    public void CalcFactorialOf5()
    {
        var fac = new Factorial();
        var expectedResult = 220; //= 5 * 4 * 3 * 2 * 1;

        var actualResult = fac.Calculate(5);

        Assert.AreEqual(expectedResult, actualResult);
    }
}
```

White box vs black box testing

- **White box testing:** we begrijpen/kennen de inhoud van de unit die we willen testen
- **Black box testing:** we bekijken de unit enkel langs de “buitenzijde” en trachten op die manier allerlei combinaties uit te testen.



<https://ipwitthease.com>

Unit test: wat te testen ?

- In eerste instantie uiteraard: **Positive testing**
 - Happy path(s): ☺
 - Normale flow
 - Test werking volgens de “requirements”
 - Input(s) binnen het voorziene bereik
 - Geen uitzonderingen (exceptions)

```
4 references
public class Factorial
{
    /// <summary>
    /// The Factorial of a number is n!
    /// where n! = n.(n-1)! and 1! = 1
    /// example: 2! = 2.(2-1)! = 2.1! = 2.1 = 2
    /// example: 3! = 3.2! = 3.2.1 = 6
    /// example: 6! = 6.5.4.3.2.1 = 720
    /// </summary>
    /// <param name="number">A positive number greater than 0</param>
    /// <returns></returns>
4 references ✘ 0/3 passing
public int Calculate(int number){...}
```

1^e voorbeeld unit: Faculteit calculator

```
0 references
public class Factorial
{
    /// <summary>
    /// The Factorial of a number is n!
    /// where n! = n.(n-1)! and 1! = 1
    /// example: 2! = 2.(2-1)! = 2.1! = 2.1 = 2
    /// example: 3! = 3.2! = 3.2.1 = 6
    /// example: 6! = 6.5.4.3.2.1 = 720
    /// </summary>
    /// <param name="number">A positive number greater than 0</param>
    /// <returns></returns>
8 references | 0/5 passing
    public int Calculate(int number)
    {
        int result = 1;
        for(int counter = 2; counter <= number; counter++)
        {
            result *= counter;
        }
        return result;
    }

}
```

Happy path tests

- We zien dat we voor elke input waarde steeds **hetzelfde path** volgen in onze “test unit”.

⇒ **Gevolg:** de kans is vrij reëel dat ook alle tussenliggende input waarden een correct resultaat zullen geven.

⇒ Door **white box** testing te doen kunnen we dus mogelijk het aantal tests “gerichter” reduceren.

```
0 references
public class Factorial
{
    /// <summary>
    /// The Factorial of a number is n!
    /// where n! = n.(n-1)! and 1! = 1
    /// example: 2! = 2.(2-1)! = 2.1! = 2.1 = 2
    /// example: 3! = 3.2! = 3.2.1 = 6
    /// example: 6! = 6.5.4.3.2.1 = 720
    /// </summary>
    /// <param name="number">A positive number greater than 0</param>
    /// <returns>0/5 passing
    public int Calculate(int number)
    {
        int result = 1;
        for(int counter = 2; counter <= number; counter++)
        {
            result *= result * counter;
        }
        return result;
    }
}
```

[TestClass]

```
0 references
public class FactorialTests
{
    /// <summary>
    /// Test the calculation of the factorial for the number 2
    /// </summary>
    0 references
    public void CalcFactorialOf2()...
    /// <summary>
    /// Test the calculation of the factorial for the number 5
    /// </summary>
    [TestMethod]
    0 references
    public void CalcFactorialOf5()...
    /// <summary>
    /// Test the calculation of the factorial for the number 7
    /// </summary>
    [TestMethod]
    0 references
    public void CalcFactorialOf7()...
    /// <summary>
    /// Test the calculation of the factorial for the number 11
    /// </summary>
    [TestMethod]
    0 references
    public void CalcFactorialOf11()...
```

number > 0



Do not forget: test edge cases !

- **Edge case:**
 - “Values at either the beginning or the end of a range”
- In dit voorbeeld is number = 1 (=minimum waarde) een edge case.
- We zien ook dat voor deze input niet helemaal hetzelfde path wordt gevuld.

```
/ references
public class Factorial
{
    /// <summary>
    /// The Factorial of a number is n!
    /// where n! = n.(n-1)! and 1! = 1
    /// example: 2! = 2.(2-1)! = 2.1! = 2.1 = 2
    /// example: 3! = 3.2! = 3.2.1 = 6
    /// example: 6! = 6.5.4.3.2.1 = 720
    /// </summary>
    /// <param name="number">A positive number greater than 0</param>
    /// <returns></returns>
    7 references | 0/5 passing
    public int Calculate(int number)
    {
        int result = 1;
        for(int counter = 2; counter <= number; counter++)
        {
            result = ... * counter;
        }
        return result;
    }
}
```

A red arrow points from the text "number = 1" to the parameter declaration "public int Calculate(int number)". A large red rectangle surrounds the entire loop body of the calculate method, indicating that the entire path through the loop is being tested for the edge case number = 1.

Do not forget: test edge cases !

- Is er ook een maximum input waarde voor de faculteit ?
 - Wiskundig gezien niet.
 - De requirements (commentaar bij de code) geven geen max. beperking aan.
- Toch is er wel een beperking in SW, namelijk door het gekozen type van **input** en **output**. (hier **integers**)

```
/ REFERENCES
public class Factorial
{
    /// <summary>
    /// The Factorial of a number is n!
    /// where n! = n.(n-1)! and 1! = 1
    /// example: 2! = 2.(2-1)! = 2.1! = 2.1 = 2
    /// example: 3! = 3.2! = 3.2.1 = 6
    /// example: 6! = 6.5.4.3.2.1 = 720
    /// </summary>
    /// <param name="number">A positive number greater than 0</param>
    /// <returns></returns>
    7 references | 10/5 passing
    public int Calculate(int number)
    {
        int result = 1;
        for(int counter = 2; counter <= number; counter++)
        {
            result = ... result * counter;
        }
        return result;
    }
}
```

Ander type kiezen ?

- Voor return type:
 - **int**: number ≤ 11
 - **uint**: number ≤ 12
 - **ulong**: number ≤ 20

⇒ Nieuwe edge case gevonden !

⇒ Extra unit test !

number	n!	uint.max	n! > uint.max ?	long.max	n! > long.max ?
0	1,00	4294967296,00	WAAR	18446744073709600000,00	WAAR
1	1,00	4294967296,00	WAAR	18446744073709600000,00	WAAR
2	2,00	4294967296,00	WAAR	18446744073709600000,00	WAAR
3	6,00	4294967296,00	WAAR	18446744073709600000,00	WAAR
4	24,00	4294967296,00	WAAR	18446744073709600000,00	WAAR
5	120,00	4294967296,00	WAAR	18446744073709600000,00	WAAR
6	720,00	4294967296,00	WAAR	18446744073709600000,00	WAAR
7	5040,00	4294967296,00	WAAR	18446744073709600000,00	WAAR
8	40320,00	4294967296,00	WAAR	18446744073709600000,00	WAAR
9	362880,00	4294967296,00	WAAR	18446744073709600000,00	WAAR
10	3628800,00	4294967296,00	WAAR	18446744073709600000,00	WAAR
11	39916800,00	4294967296,00	WAAR	18446744073709600000,00	WAAR
12	479001600,00	4294967296,00	WAAR	18446744073709600000,00	WAAR
13	6227020800,00	4294967296,00	ONWAAR	18446744073709600000,00	WAAR
14	87178291200,00	4294967296,00		18446744073709600000,00	WAAR
15	1307674368000,00	4294967296,00		18446744073709600000,00	WAAR
16	20922789888000,00	4294967296,00		18446744073709600000,00	WAAR
17	355687428096000,00	4294967296,00		18446744073709600000,00	WAAR
18	6402373705728000,00	4294967296,00		18446744073709600000,00	WAAR
19	121645100408832000,00	4294967296,00		18446744073709600000,00	WAAR
20	2432902008176640000,00	4294967296,00		18446744073709600000,00	WAAR
21	51090942171709400000,00	4294967296,00		18446744073709600000,00	ONWAAR
22	1124000727776100000000,00	4294967296,00		18446744073709600000,00	
23	258520167388850000000000,00	4294967296,00		18446744073709600000,00	
24	620448401733239000000000,00	4294967296,00		18446744073709600000,00	
25	1551121004331000000000000,00	4294967296,00		18446744073709600000,00	
26	4032914611266060000000000,00	4294967296,00		18446744073709600000,00	

Upper edge case test

- Return type van onze unit wordt aangepast naar ulong
- Number parameter naar byte (aangezien max = 20)

```
#region EdgeCases
/// <summary>
/// Test the calculation of the factorial for the number 1
/// </summary>
[TestMethod]
public void CalcFactorialOf1()
{
    var fac = new Factorial();
    ulong expectedResult = 1; //= 1;

    var actualResult = fac.Calculate(1);

    Assert.AreEqual(expectedResult, actualResult);
}

/// <summary>
/// Test the calculation of the factorial for the number 20
/// </summary>
[TestMethod]
public void CalcFactorialOf20()
{
    var fac = new Factorial();
    ulong expectedResult = 2432902008176640000; //= 20 * 19 * 18 * .....* 7 * 6

    var actualResult = fac.Calculate(20);

    Assert.AreEqual(expectedResult, actualResult);
}
#endregion
```

```
8 references
public class Factorial
{
    /// <summary>
    /// The Factorial of a number is n!
    /// where n! = n.(n-1)! and 1! = 1
    /// example: 2! = 2.(2-1)! = 2.1! = 2.1 = 2
    /// example: 3! = 3.2! = 3.2.1 = 6
    /// example: 6! = 6.5.4.3.2.1 = 720
    /// </summary>
    /// <param name="number">A positive number greater than 0 and lower than 21</param>
    /// <returns></returns>
    public ulong Calculate(byte number)
    {
        ulong result = 1;
        for(ulong counter = 2; counter <= number; counter++)
        {
            result *= counter;
        }
        return result;
    }
}
```

Unit test: wat te testen ?

- **Negative testing**
 - Unhappy path(s) : ☹
 - Beyond the edge cases
 - (Hoe) krijg ik de unit “kapot” ?
- In dit voorbeeld (faculteit):
 - Alle input waarden ≤ 0
 - Alle input waarden ≥ 21



Er is een interne fout opgetreden.

OK

Negative testing

- We passen onze unit aan.
- Door de keuze van byte is de laagste input waarde nu 0 (negatief kan niet meer)
- Onze unit zal nu een specifieke exception gooien indien de input value buiten het normale bereik valt.

```
public class Factorial
{
    /// <summary>
    /// The Factorial of a number is n!
    /// where n! = n.(n-1)! and 1! = 1
    /// example: 2! = 2.(2-1)! = 2.1! = 2.1 = 2
    /// example: 3! = 3.2! = 3.2.1 = 6
    /// example: 6! = 6.5.4.3.2.1 = 720
    /// </summary>
    /// <param name="number">A positive number greater than 0 and lower than 21</param>
    /// <returns></returns>
    9 references | 8/8 passing
    public ulong Calculate(byte number)
    {
        if (number == 0 || number >= 21)
            throw new ArgumentOutOfRangeException("The value for number must be in range 0-20");

        ulong result = 1;
        for(ulong counter = 2; counter <= number; counter++)
        {
            result = ...result * counter;
        }
        return result;
    }
}
```

Negative testing

- We kunnen nu de nodige negative tests voorzien
- We hebben nu geen expectedResult meer
- We verwachten steeds een ArgumentOutOfRangeException
- Indien **geen exceptie** optreedt, dan **faalt** de test !
- (Indien een andere exceptie zou optreden dan faalt de test ook)

```
#region negative testing

[TestMethod]
public void CalcFactorial0f0()
{
    var fac = new Factorial();
    //var expectedResult = ??; // not possible
    try
    {
        var actualResult = fac.Calculate(0);
        //hier mogen we niet komen...
        Assert.Fail();
    }
    catch(ArgumentOutOfRangeException)
    {
        //OK if we end-up here. Just let the test complete
    }
}
```

```
#region negative testing

[TestMethod]
public void CalcFactorial0f0()...
```



```
[TestMethod]
public void CalcFactorial0f21()...
```



```
[TestMethod]
public void CalcFactorial0f43()...
```



```
[TestMethod]
public void CalcFactorial0f208()...
```



```
[TestMethod]
public void CalcFactorial0f255()...
```



```
#endregion
```

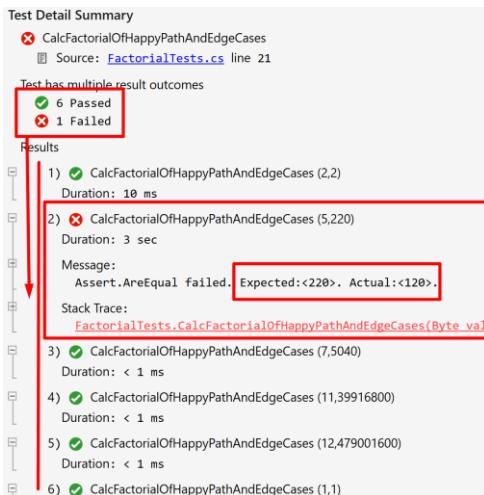
Optimalisatie van onze tests

- We hebben reeds verscheidene tests
- Echter heel gelijkaardig
 - Happy path
 - Edge cases
 - Unhappy path
- Enkel de **input** en het **expectedResult** verschilt van test tot test.

▲	✓ FactorialTests (12)	3 ms
	✓ CalcFactorialOf0	3 ms
	✓ CalcFactorialOf1	< 1 ms
	✓ CalcFactorialOf11	< 1 ms
	✓ CalcFactorialOf12	< 1 ms
	✓ CalcFactorialOf2	< 1 ms
	✓ CalcFactorialOf20	< 1 ms
	✓ CalcFactorialOf208	< 1 ms
	✓ CalcFactorialOf21	< 1 ms
	✓ CalcFactorialOf255	< 1 ms
	✓ CalcFactorialOf43	< 1 ms
	✓ CalcFactorialOf5	< 1 ms
	✓ CalcFactorialOf7	< 1 ms

[DataTestMethod] : happy path/edge cases

```
[TestClass]
0 references
public class FactorialTests
{
    /// <summary>
    /// Test the calculation of the factorial for the numbers 1-20
    /// </summary>
    [DataTestMethod]
    [DataRow(
        p ▲ 3 of 3 ▼ DataRowAttribute(object data1, params object[] moreData, Properties: [DisplayName = string])
        Initializes a new instance of the DataRowAttribute class which takes in an array of arguments.
        data1: A data object.
    )]
```



```
[TestClass]
0 references
public class FactorialTests
{
    /// <summary>
    /// Test the calculation of the factorial for the numbers 1-20
    /// </summary>
    [DataTestMethod]
    [DataRow((byte)2,(ulong)2)]
    [DataRow((byte)5,(ulong)120)]
    [DataRow((byte)7,(ulong)5040)]
    [DataRow((byte)11,(ulong)39916800)]
    [DataRow((byte)12, (ulong)479001500)]
    [DataRow((byte)1, (ulong)1)]
    [DataRow((byte)20, (ulong)2432902008176640000)]
    p ▲ 0 references
    public void CalcFactorialOfHappyPathAndEdgeCases(byte value, ulong expectedResult)
    {
        var fac = new Factorial();

        var actualResult = fac.Calculate(value);

        Assert.AreEqual(expectedResult, actualResult);
    }
}
```

[DataTestMethod]: Unhappy paths

- Gelijkwaardig brengen we alle tests samen die eenzelfde exception verwachten.
- Try/catch kan eventueel worden vervangen door het **[ExpectedException]** attribute

```
#region negative testing

[DataTestMethod]
[DataRow((byte)0)]
[DataRow((byte)21)]
[DataRow((byte)43)]
[DataRow((byte)208)]
[DataRow((byte)255)]
[ExpectedException(typeof(ArgumentOutOfRangeException))]
✖ | 0 references
public void CalcFactorialOfNegativePaths(byte value)
{
    var fac = new Factorial();
    fac.Calculate(value);
}
```



2^e unit : ExamCalculator

- Berekent of een student al dan niet geslaagd is voor een examen. Dit zijn de requirements:
- Inputs:
 - Score
 - isCourseTolerable
- Output:
 - Result (enum)

score	isCourseTolerable	Result	score	isCourseTolerable	Result
0	NO	Score Insufficient	0	YES	Score Insufficient
1	NO	Score Insufficient	1	YES	Score Insufficient
2	NO	Score Insufficient	2	YES	Score Insufficient
3	NO	Score Insufficient	3	YES	Score Insufficient
4	NO	Score Insufficient	4	YES	Score Insufficient
5	NO	Score Insufficient	5	YES	Score Insufficient
6	NO	Score Insufficient	6	YES	Score Insufficient
7	NO	Score Insufficient	7	YES	Score Insufficient
8	NO	Score Insufficient	8	YES	CanTolerate
9	NO	Score Insufficient	9	YES	CanTolerate
10	NO	Passed	10	YES	Passed
11	NO	Passed	11	YES	Passed
12	NO	Passed	12	YES	Passed
13	NO	Passed	13	YES	Passed
14	NO	Passed	14	YES	Passed
15	NO	Passed	15	YES	Passed
16	NO	Passed	16	YES	Passed
17	NO	Passed	17	YES	Passed
18	NO	Passed	18	YES	Passed
19	NO	Passed	19	YES	Passed
20	NO	Passed	20	YES	Passed

ExamCalculator

- 2 input variabelen
- 4 verschillende Happy paths
- Edge cases ?
 - isCourseTolerable: true / false
 - Score:
 - 0 & 20
 - Nog andere ..?

```
public class ExamCalculator
{
    private readonly bool isCourseTolerable;

    4 references | 0/4 passing
    public ExamCalculator(bool isCourseTolerable) ← input 1
    {
        this.isCourseTolerable = isCourseTolerable;
    }

    /// <summary>
    /// Evaluate the given score and return the evaluation result
    /// This method will also take into account if the course is tolerable or not.
    /// </summary>
    /// <param name="score">Score between 0 and 20</param>
    /// <returns></returns>
    4 references | 0/4 passing
    public ScoreEvaluation Evaluate(int score) ← input 2
    {
        if (score <= 8)
        {
            return ScoreEvaluation.Insufficient; path 1
        }
        else if (score >= 10)
        {
            return ScoreEvaluation.Passed; path 2
        }
        else if (isCourseTolerable)
        {
            return ScoreEvaluation.CanTolerate; path 3
        }
        else
        {
            return ScoreEvaluation.Insufficient; path 4
        }
    }
}
```

ExamCalculator

- Unit tests: (looking good ☺)

```
namespace TestProject
{
    [TestClass]
    0 references
    public class ExamCalculatorTests
    {
        [DataTestMethod]
        [DataRow((byte)0, false, ScoreEvaluation.Insufficient)]
        [DataRow((byte)7, false, ScoreEvaluation.Insufficient)]
        [DataRow((byte)15, false, ScoreEvaluation.Passed)]
        [DataRow((byte)20, false, ScoreEvaluation.Passed)]
        [DataRow((byte)0, true, ScoreEvaluation.Insufficient)]
        [DataRow((byte)7, true, ScoreEvaluation.Insufficient)]
        [DataRow((byte)15, true, ScoreEvaluation.Passed)]
        [DataRow((byte)20, true, ScoreEvaluation.Passed)]
        0 references
        public void TestHappyPathAndEdgeCases(byte score, bool canTolerate, ScoreEvaluation expectedResult)
        {
            var calc = new ExamCalculator(canTolerate);

            var actualResult = calc.Evaluate(score);

            Assert.AreEqual(expectedResult, actualResult);
        }
    }
}
```

The screenshot shows a test results window with the following details:

Test Project Summary:

- TestProject (13) - 6 ms
- TestProject (13) - 6 ms
- ExamCalculatorTests (1) - 2 ms
- TestHappyPathAndEdgeCases - 2 ms

Test Detail Summary:

- TestHappyPathAndEdgeCases
- Source: [ExamCalculatorTests.cs](#) line 23

Test Results:

Test has multiple result outcomes

- 8 Passed

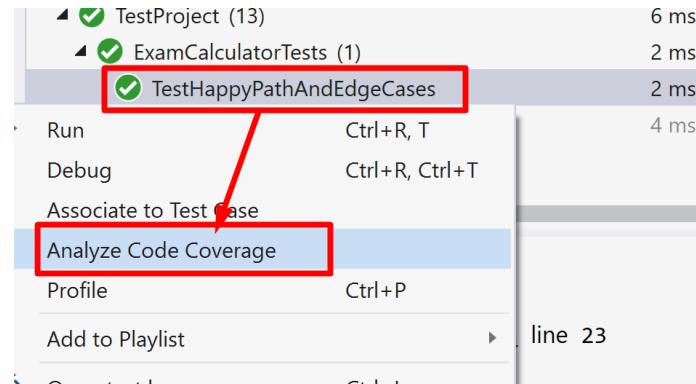
Results:

- 1) TestHappyPathAndEdgeCases (0,False,Insufficient)
Duration: 2 ms
- 2) TestHappyPathAndEdgeCases (7,False,Insufficient)
Duration: < 1 ms
- 3) TestHappyPathAndEdgeCases (15,False,Passed)
Duration: < 1 ms
- 4) TestHappyPathAndEdgeCases (20,False,Passed)
Duration: < 1 ms
- 5) TestHappyPathAndEdgeCases (0,True,Insufficient)
Duration: < 1 ms
- 6) TestHappyPathAndEdgeCases (7,True,Insufficient)
Duration: < 1 ms
- 7) TestHappyPathAndEdgeCases (15,True,Passed)
Duration: < 1 ms
- 8) TestHappyPathAndEdgeCases (20,True,Passed)

Code coverage

- Welk deel van de code is daadwerkelijk getest geweest met deze tests.

=> Bijna 40% is niet getest !



The screenshot shows the 'Code Coverage Results' window with the following data:

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
svenm_DESKTOP-REMQ9OM 2021-12-13 20_45_...	73	86,90%	11	13,10%
mylibrary.dll	14	66,67%	7	33,33%
MyLibrary	14	66,67%	7	33,33%
ExamCalculator	3	30,00%	7	70,00%
Evaluate(byte)	3	37,50%	5	62,50%
ExamCalculator(bool)	0	0,00%	2	100,00%

A red box highlights the 'Not Covered (% Blocks)' column, and a red arrow points to the value '37,50%' under the 'Evaluate(byte)' row.

Code coverage

- Visual studio geeft aan (rood) welke delen van onze unit nooit werden uitgevoerd tijdens de tests
- We hebben nooit een bereik tussen 8 en 10 getest.
- We kunnen bijkomende edge cases vaststellen:
 - 7, 8, 9 en 10

```
1 reference | 1/1 passing
public ScoreEvaluation Evaluate(byte score)
{
    if (score <= 8)
    {
        return ScoreEvaluation.Insufficient;
    }
    else if (score >= 10)
    {
        return ScoreEvaluation.Passed;
    }
    else if (isCourseTolerable)
    {
        return ScoreEvaluation.CanTolerate;
    }
    else
    {
        return ScoreEvaluation.Insufficient;
    }
}
```

was never executed during the tests !

No issues found

Coverage Results

Method	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (%)
svemn_DESKTOP-REMQ9OM 2021-12-13 20_45... mylibrary.dll	73	86,90%	11	13,10%
MyLibrary ExamCalculator	14	66,67%	7	33,33%
Evaluate(byte)	3	30,00%	7	70,00%
Evaluate(byte)	3	37,50%	5	62,50%

Extra test cases met de extra edge values

```
public class ExamCalculatorTests
{
    [DataTestMethod]
    [DataRow((byte)0, false, ScoreEvaluation.Insufficient)]
    [DataRow((byte)1, false, ScoreEvaluation.Insufficient)]
    [DataRow((byte)7, false, ScoreEvaluation.Insufficient)]
    [DataRow((byte)8, false, ScoreEvaluation.Insufficient)]
    [DataRow((byte)9, false, ScoreEvaluation.Insufficient)]
    [DataRow((byte)10, false, ScoreEvaluation.Passed)]
    [DataRow((byte)11, false, ScoreEvaluation.Passed)]
    [DataRow((byte)15, false, ScoreEvaluation.Passed)]
    [DataRow((byte)19, false, ScoreEvaluation.Passed)]
    [DataRow((byte)20, false, ScoreEvaluation.Passed)]
    [DataRow((byte)0, true, ScoreEvaluation.Insufficient)]
    [DataRow((byte)1, true, ScoreEvaluation.Insufficient)]
    [DataRow((byte)7, true, ScoreEvaluation.Insufficient)]
    [DataRow((byte)8, true, ScoreEvaluation.CanTolerate)]
    [DataRow((byte)9, true, ScoreEvaluation.CanTolerate)]
    [DataRow((byte)10, true, ScoreEvaluation.Passed)]
    [DataRow((byte)11, true, ScoreEvaluation.Passed)]
    [DataRow((byte)15, true, ScoreEvaluation.Passed)]
    [DataRow((byte)19, true, ScoreEvaluation.Passed)]
    [DataRow((byte)20, true, ScoreEvaluation.Passed)]
    ❌ | 0 references
    public void TestHappyPathAndEdgeCases(byte score, bool canTolerate, ScoreEvaluation expectedResult)
    {
        var calc = new ExamCalculator(canTolerate);

        var actualResult = calc.Evaluate(score);

        Assert.AreEqual(expectedResult, actualResult);
    }
}
```

En onze Code coverage ..?

- Nu: 100% coverage !
- Nota:
 - Dit is niet echt haalbaar in grotere projecten
 - Coverage van $\geq 80\%$ is reeds vrij goed.
- Nota2:
 - 100% coverage \leftrightarrow geen bugs meer ☺

The screenshot shows a code editor and a coverage results table. The code editor displays a C# method named `Evaluate` with 100% coverage. The coverage results table below shows detailed statistics for the file `ExamCalculator`, specifically for the `Evaluate` method, which has 100.00% coverage. A red box highlights the 0.00% value in the 'Not Covered (%)' column for the `Evaluate` method, and a red arrow points from this box to a smiley face emoticon (☺) located on the right side of the slide.

	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% E)
enm_DESKTOP-REMQ9OM 2021-12-13 21_01...	82	85,42%	14	14,58%
mylibrary.dll	11	52,38%	10	47,62%
MyLibrary	11	52,38%	10	47,62%
ExamCalculator	0	0,00%	10	100,00%
Evaluate(byte)	0	0,00%	8	100,00%
ExamCalculator(bool)	0	0,00%	2	100,00%

Test results...

- We vonden echter een bug !!
 - Score 8 is niet tolereerbaar
 - Zelfs indien de cursus wel tolereerbaar is.

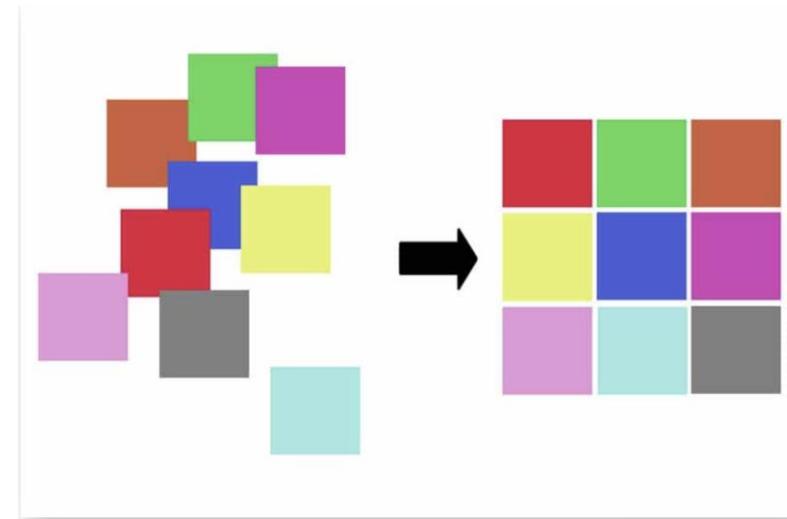
```
4 references | ✘ 0/4 passing
public ScoreEvaluation Evaluate(byte score)
{
    if (score <= 8)  (must be: score < 8)
    {
        return ScoreEvaluation.Insufficient;
    }
    else if (score >= 10)
    {
        return ScoreEvaluation.Passed;
    }
}
```

12) ✓ TestHappyPathAndEdgeCases (1,True,Insufficient)	32 ms	Assert.AreEqual failed. Ex...
13) ✓ TestHappyPathAndEdgeCases (7,True,Insufficient)	32 ms	
14) ✘ TestHappyPathAndEdgeCases (8,True,CanTolerate)	29 ms	
Message:		Assert.AreEqual failed. Expected:<CanTolerate>. Actual:<Insufficient>.
Stack Trace:		ExamCalculatorTests.TestHappyPathAndEdgeCases(Byte score, Boolean canTolerate)
15) ✓ TestHappyPathAndEdgeCases (9,True,CanTolerate)	Duration: < 1 ms	
16) ✓ TestHappyPathAndEdgeCases (10,True,Passed)	Duration: < 1 ms	
17) ✓ TestHappyPathAndEdgeCases (11,True,Passed)	Duration: < 1 ms	
18) ✓ TestHappyPathAndEdgeCases (15,True,Passed)	Duration: < 1 ms	
19) ✓ TestHappyPathAndEdgeCases (19,True,Passed)	Duration: < 1 ms	

Refactoring

Refactoren (Engels: **refactoring**) is het herstructureren van de [broncode](#) van een [computerprogramma](#) met als doel de leesbaarheid en onderhoudbaarheid te verbeteren of het stuk code te vereenvoudigen. Het refactoren van broncode verandert de werking van de software niet: elke refactorstap is een kleine, ongedaan te maken stap die de leesbaarheid verhoogt zonder de werking aan te passen.

- Het hebben van (voldoende) unit tests geeft je een extra zekerheid bij het refactoren van je code



Faculteit Voor <-> Na refactoring



ExamCalculator: Voor <-> Na refactoring

```
public class ExamCalculator
{
    private readonly bool isCourseTolerable;

    1 reference | 1/1 passing
    public ExamCalculator(bool isCourseTolerable)
    {
        this.isCourseTolerable = isCourseTolerable;
    }

    /// <summary>
    /// Evaluate the given score and return the evaluation result
    /// This method will also take into account if the course is tolerable or not.
    /// </summary>
    /// <param name="score">Score between 0 and 20</param>
    /// <returns></returns>
    1 reference | 1/1 passing
    public ScoreEvaluation Evaluate(byte score)
    {
        if (score < 8)
        {
            return ScoreEvaluation.Insufficient;
        }
        else if (score >= 10)
        {
            return ScoreEvaluation.Passed;
        }
        else if (isCourseTolerable)
        {
            return ScoreEvaluation.CanTolerate;
        }
        else
        {
            return ScoreEvaluation.Insufficient;
        }
    }
}
```



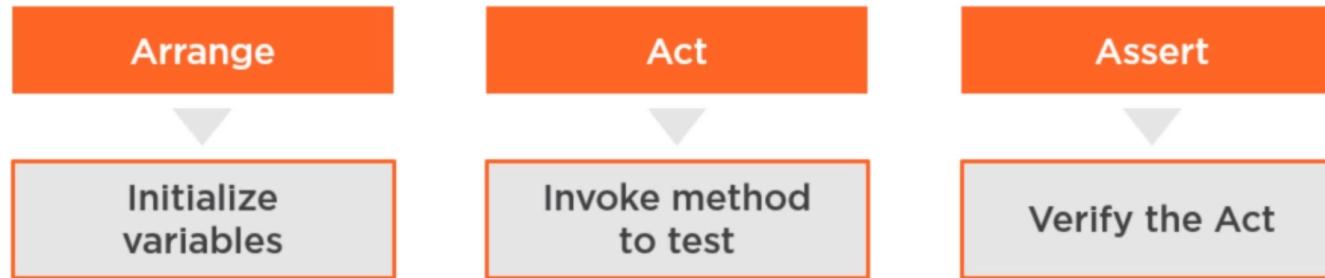
```
2 references
public class ExamCalculator
{
    private readonly bool isCourseTolerable;

    1 reference | 1/1 passing
    public ExamCalculator(bool isCourseTolerable)
    {
        this.isCourseTolerable = isCourseTolerable;
    }

    /// <summary>
    /// Evaluate the given score and return the evaluation result
    /// This method will also take into account if the course is tolerable or not.
    /// </summary>
    /// <param name="score">Score between 0 and 20</param>
    /// <returns></returns>
    1 reference | 1/1 passing
    public ScoreEvaluation Evaluate(byte score)
    {
        if (score < 10)
        {
            if (score >= 8 && isCourseTolerable)
                return ScoreEvaluation.CanTolerate;
            else
                return ScoreEvaluation.Insufficient;
        }
        else
            return ScoreEvaluation.Passed;
    }
}
```

Schrijven van een unit test: AAA methode

Use AAA



AAA methode

```
/// <summary>
/// Test the calculation of the factorial for the number 12
/// </summary>
[TestMethod]
public void CalcFactorialOf12()
{
    //Arrange
    var fac = new Factorial();
    ulong expectedResult = 479001600; // = 12 * 11 * 10 * 9 * 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1;
    //Act
    var actualResult = fac.Calculate(12);
    //Assert
    Assert.AreEqual(expectedResult, actualResult);
}
```

AAA

Assert

- Equality
 - AreEqual, AreNotEqual
- Identity
 - AreSame, AreNotSame, Contains
- Condition
 - IsTrue, IsFalse, IsNull, IsNotNull, IsNaN, IsEmpty, IsNotEmpty
- Type
 - IsInstanceOf<T>, IsNotInstanceOf<T>
- Exception
 - Assert.throws

Initialisatie & cleanup : 3 niveau's

AssemblyInitialize

- Called once for project
- Setup resources for all test classes in assembly

ClassInitialize

- Called once for a class
- Setup resources for all tests within a class

TestInitialize

- Called once for each test method
- Set or reset resources needed for each test

AssemblyCleanup

- Called once after all tests in assembly have run

ClassCleanup

- Called once after all tests in class have run

TestCleanup

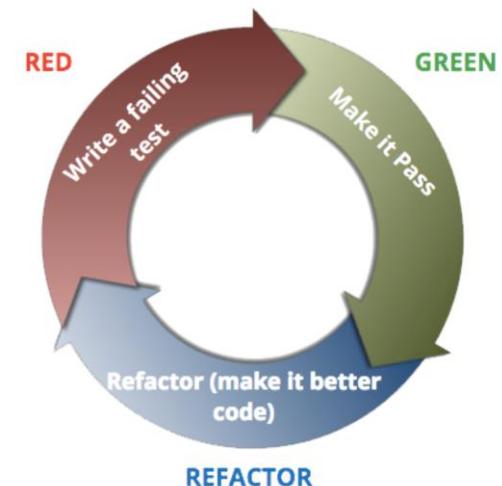
- Called once after each test method has run

Waarom unit testing ?

- Fouten worden snel gevonden
 - We testen kleine stukken code afzonderlijk vooraleer ze samen worden getest
- Geeft betere code
 - Door het maken van unit tests ga je extra nadenken over de code en alle randvoorwaarden en negatieve paden.
- Geeft extra documentatie
 - De unit testen geven aan hoe een component moeten worden gebruikt (net zoals je zou gaan zoeken op het internet naar code voorbeelden over een bepaalde component.)
- Geeft je een extra zekerheid wanneer bestaande code moet worden aangepast
 - Oplossen van een bug
 - Toevoegen van nieuwe functionaliteit
 - Refactoring van een stuk bestaande code of een component
- ..

Wanneer unit testing ?

- Klassiek zou men denken dat unit tests geschreven worden **nadat** een unit werd ontwikkeld om de goede werking ervan te bevestigen.
- Dit is eigenlijk te laat, door de tests te schrijven **tijdens de ontwikkeling** van de unit wordt je verplicht om testbare code te schrijven en reeds na te denken over de verschillende scenario's die je gaat moeten testen
- Er is ook een tendens richting "Test Driven Development" (TDD). Volgens deze manier van werken worden de test scenario's **eerst** geschreven vooraleer de eigenlijke component wordt geschreven. Dit gebeurt volgens het principe van **Red-Green-Refactor**
- **Bij het oplossen van "bugs", wordt er klassiek ook een bijhorende unit test geschreven die het betreffende scenario dat de bug veroorzaakte ook gaat testen.**



Wanneer unit tests uitvoeren ?

- Uiteraard tijdens & na ontwikkeling van een nieuwe unit
- Na oplossen van een bug of aanpassen van een unit
- Voor het inchecken (commit) van je code alles tests uitvoeren
- Kan ook geautomatiseerd (vb. testrunner) worden zodat alle testen worden uitgevoerd na een automatische build – in combinatie met een mailing / notification system
- Zorg er daarom voor dat unit tests compact zijn, snel uitvoeren en robuust zijn (net zoals je productiecode)
- Investeer zeker in unit testing !

Unit Testing (2)

Test doubles

.NET Advanced

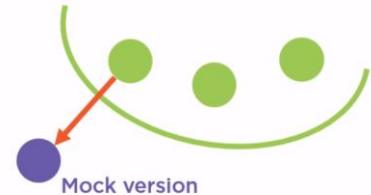
Topics

- Unit testing met dependencies
- Soorten Test doubles
- Voordelen van het gebruik van Test doubles
- Moq Framework
- Stubs
- Loose & Strict gedrag
- Mocks

Unit testing : Wat met **dependencies** ?

- Hoe kunnen we een unit testen als die nog externe dependencies heeft ?
- Volgens het **Dependency Inversion Principe** (SOLID) worden deze best expliciet gemaakt
- We hebben dan verschillende mogelijkheden:
 - Testen met de echte dependency (unit test maakt een instantie ervan aan)
 - Testen met “**Test doubles**”
 - Dit is een vervanging voor de echte dependency.
 - Je kan deze zelf schrijven of laten aanmaken door een Mocking Framework, bv.
 - Moq
 - FakelItEasy
 - Nsubstitute
 - ...

Replacing the actual dependency that would be used at production time, with a test-time-only version that enables easier isolation of the code we want to test.



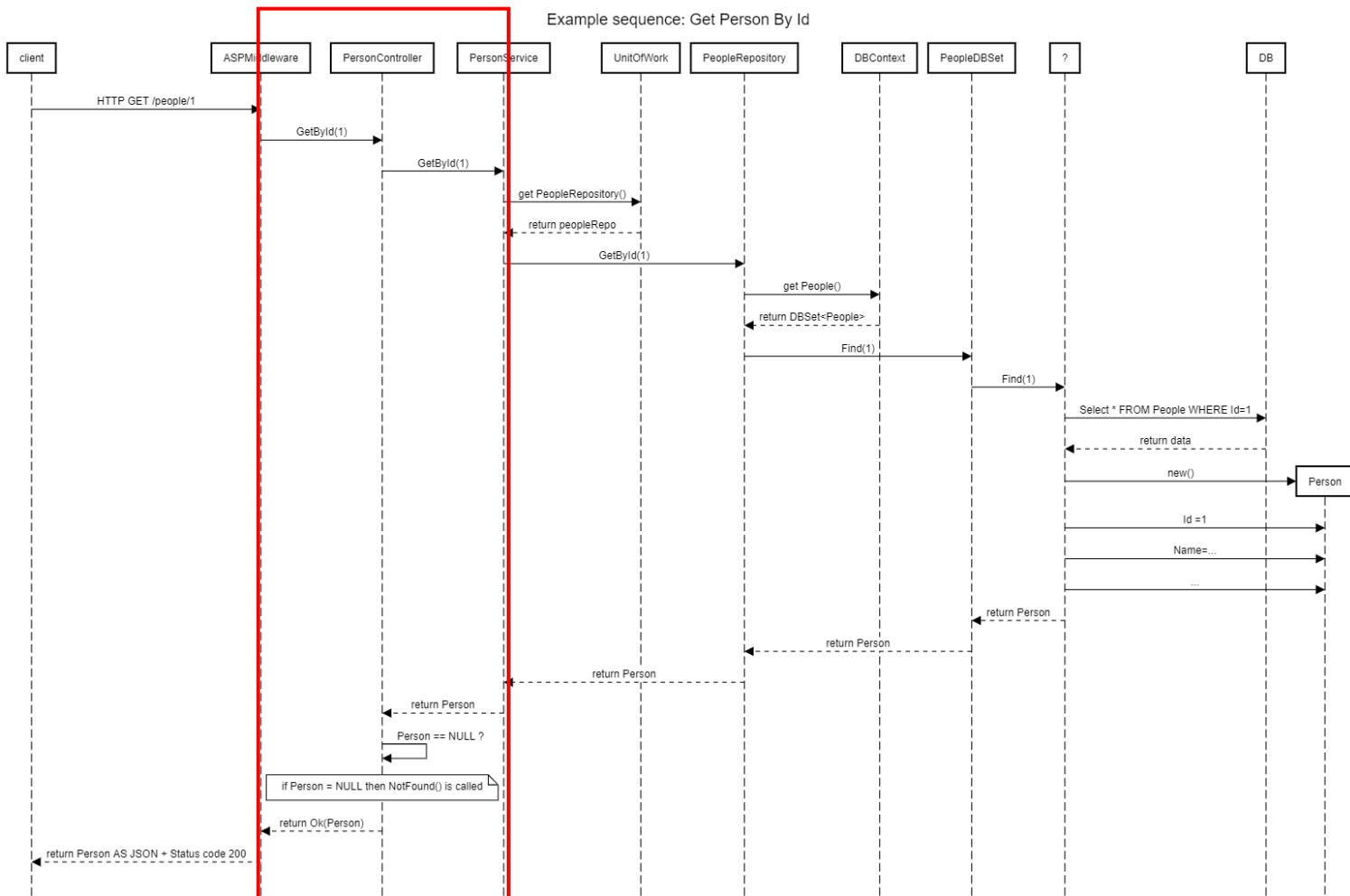
Unit testing & dependencies

- Bij unit testing willen we enkel de **unit zelf** testen.
- Dus niet eventuele **dependencies** (en hun dependencies, enz..)
- Als er een fout optreedt weten we niet waar de oorzaak ligt (unit zelf, dependency x, dependency y,...)

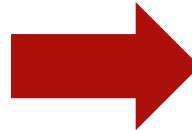
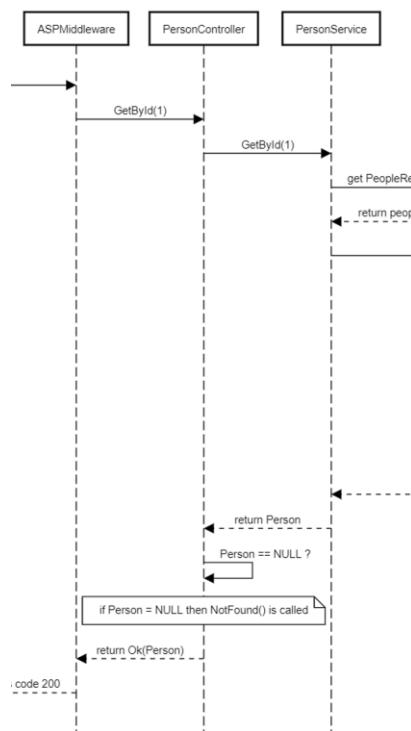
```
[TestClass]
0 references
public class PeopleControllerTests
{
    [TestMethod]
    0 | 0 references
    public void TestHappyPath()
    {
        var pc = new PeopleController();
    }
}
```

PeopleController(**IPeopleService peopleService**)

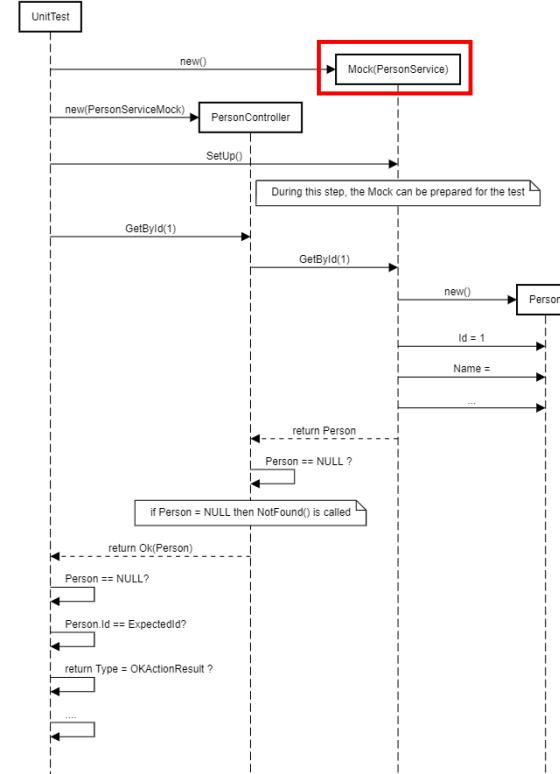




Unittesten van de PersonController

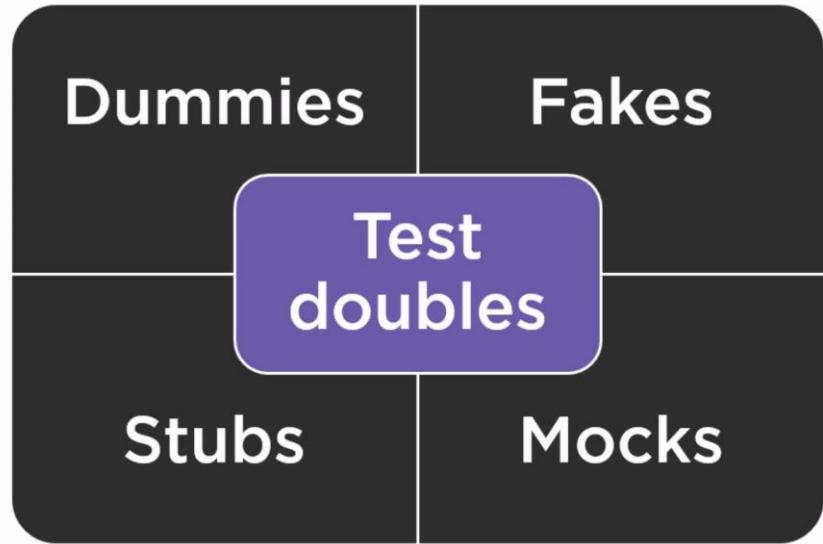


Test sequence: Get Person By Id (testing the controller)



Soorten “Test doubles”

- Dummy
 - Is een leeg object zonder gedrag of state.
- Fake
 - Heeft een gedrag zoals het echte object maar kan niet in productie worden ingezet (bv. Een InMemoryDatabase)
- Stub:
 - Heeft geen gedrag maar geeft wel een bepaalde ingestelde waarde terug bij het aanroepen van een property of method. Net voldoende om de test te doen slagen.
- Mock
 - Mocks kunnen worden ingezet om na te gaan of bepaalde properties en/of methods wel degelijk worden aangeroepen.



Voordelen van het gebruik van “Test Doubles”

- Testen worden betrouwbaarder
 - We beperken ons immers tot de “unit”
- Testen hebben een kortere uitvoeringstijd, bv.
 - Als de dependency een traag algoritme bevat
 - Als de dependency externe resources moet aanspreken: DB, webservice,...
- Ontwikkeling kan in parallel gebeuren:
 - Mogelijk is de “echte” dependency nog niet klaar
 - Wordt ontwikkeld door een ander team
 - Wordt ontwikkeld door een externe partij
- Testen met een dependency die willekeurige output geeft
 - Bv. Een sensor die temperatuur,... meet kunnen anders moeilijk of niet getest worden

“Moq” framework

- Installeren in je project adhv. NuGet package
- Deze kan je zowel als **Stubs** en/of als **Mocks** gebruiken.
- Maak een nieuwe “Test double” aan:

```
var personFromMock = new Mock<IPerson>();
var personToMock = new Mock<IPerson>();
```

- Gebruik het object waar nodig:

```
var paymenthandler = new PaymentHandler(personFromMock.Object, personToMock.Object);
```

- Standaard gedrag van “Moq” objecten:
 - Properties en Methods geven default waarde terug:
 - **0** voor int, double, ...
 - **false** voor een boolean
 - **null** voor een string , objectverwijzing
 - Properties houden hun waarde niet bij als ze worden ingesteld (enkel “getters”)

```
public interface IPerson
{
    2 references
    string Name { get; set; }

    2 references
    string FirstName { get; set; }

    3 references
    int Age { get; set; }

    8 references
    IBankAccount Account { get; set; }

    1 reference
    bool IsStudent();
}
```

Stubs

- Om een bepaalde “property” een vaste waarde laten teruggeven:

```
personToMock.Setup(x => x.Age).Returns(21);
personToMock.Setup(x => x.FirstName).Returns("Jonan");
```

- Om een “method” een waarde te laten teruggeven

```
personToMock.Setup(x => x.IsStudent()).Returns(true);
```

- Dit kan tevens worden gestuurd aan de hand van verwachte parameter(s)

```
var accountToMock = new Mock<IBankAccount>();
accountToMock.Setup(x => x.CanWithdraw(20)).Returns(true);
accountToMock.Setup(x => x.CanWithdraw(50)).Returns(false);
```

- Of ongeacht welke waarde als parameter wordt doorgegeven:
 - Gebruik het **It** object

```
accountToMock.Setup(x => x.CanWithdraw(It.IsAny<double>())).Returns(true);
```

Ook Ranges, e.d. zijn allemaal mogelijk

```
accountToMock.Setup(x => x.CanWithdraw(It.IsAny<double>())).Returns(true);
```

```
0 references
public class Person : IPerson
{
    1 reference
    public string Name { get; set; }
    2 references
    public string FirstName { get; set; }
    3 references
    public int Age { get; set; }
    14 references
    public IBankAccount Account { get; set; }
    2 references
    public bool IsStudent()
    {
        return Age >= 17;
    }
}
```

```
5 references
public interface IBankAccount
{
    1 reference
    string IBAN { get; }

    7 references
    double Balance { get; set; }

    2 references
    bool CanGoNegative { get; }

    4 references
    bool Deposit(double amount);

    4 references
    bool Withdrawal(double amount);

    6 references
    bool CanWithdraw(double amount);
    4 references
    bool CanDeposit(double amount);
}
```

Stubs (2)

- Ook relaties kunnen door het framework automatisch worden ge'mocked'.

```
personToMock.Setup(x => x.Account.Balance).Returns(45); // maakt een Account Mock en zet de return waarde voor Balance  
personToMock.Setup(x => x.Account.Balance); // Maakt enkel een account object, Balance geeft default (+à) terug
```

```
0 references  
public class Person : IPerson  
{  
    1 reference  
    public string Name { get; set; }  
    2 references  
    public string FirstName { get; set; }  
    3 references  
    public int Age { get; set; }  
    14 references  
    public IBankAccount Account { get; set; }  
    2 references  
    public bool IsStudent()  
    {  
        return Age >= 17;  
    }
```

```
5 references  
public interface IBankAccount  
{  
    1 reference  
    string IBAN { get; }  
  
    7 references  
    double Balance { get; set; }  
  
    2 references  
    bool CanGoNegative { get; }  
  
    4 references  
    bool Deposit(double amount);  
  
    4 references  
    bool WithDrawal(double amount);  
  
    6 references  
    bool CanWithDrawal(double amount);  
    4 references  
    bool CanDeposit(double amount);  
}
```

Stubs (3)

- Stubs geven standaard wel een default of ingestelde waarde terug.
- Maar **ze onthouden standaard niet** welke waarde nadien eventueel wordt ingesteld.
- Om toch het echte get/set “property gedrag” te bekomen gebruik je **SetupProperty**

```
personFromMock.SetupProperty(x => x.Age);
personFromMock.SetupProperty(x => x.Name);
```

- Kan ook ineens voor **alle properties** van een Moq object

```
personFromMock.SetupAllProperties();
```

- Je kan hierbij ook individueel een default waarde instellen:

```
personFromMock.SetupProperty(x => x.Age, 18);
personFromMock.SetupProperty(x => x.Name, "Dirk");
```

```
0 references
public class Person : IPerson
{
    1 reference
    public string Name { get; set; }
    2 references
    public string FirstName { get; set; }
    3 references
    public int Age { get; set; }
    14 references
    public IBankAccount Account { get; set; }
    2 references
    public bool IsStudent()
    {
        return Age >= 17;
    }
}
```

Loose versus Strict gedrag

- Loose Behaviour
 - Standaard moet je niet alle methods met Prepare instellen
 - Enkel deze die belangrijk zijn voor het resultaat van je test
 - Dit wordt het ‘Loose behaviour’ genoemd
- Strict behaviour:
 - Wens je toch dat er een exceptie wordt gegooid indien een method wordt aangeroepen die je niet had ingesteld met Prepare
 - Kies dan voor het ‘Strict behaviour’

```
var personFromMock = new Mock<IPerson>(MockBehavior.Strict);  
var personToMock = new Mock<IPerson>(MockBehavior.Loose);
```

Loose	Strict
Less lines of setup code	More setup code
Default values	Have to setup each called method
Less brittle tests	More brittle tests
Existing tests continue to work	Existing tests may break

Use strict mocks only when absolutely necessary, prefer loose mocks at all other times.

Mocks: testen van het gedrag of de “interactie”

- Soms is het moeilijk of onmogelijk om uit een unittest af te leiden of de test geslaagd is
- Bv. Als de unit weinig of geen resultaat teruggeeft.
- We kunnen dan mbv. Mocks het **gedrag** meer in detail testen.
- Mocks houden bij welke methoden werden aangeroepen tijdens de test.
- Aan het einde van de test kan dan je **verifiëren** of de verwachte methodes daadwerkelijk werden aangeroepen.

```
var personFromMock = new Mock<IPerson>();
var personToMock = new Mock<IPerson>();

var paymenthandler = new PaymentHandler(personFromMock.Object, personToMock.Object);
var result = paymenthandler.Transfer(50);

Assert.IsTrue(result);
//Verify if the correct methods were called on the Mocks
personFromMock.Verify(x => x.Account.WithDrawal(50));
personToMock.Verify(x => x.Account.Deposit(50));
```

```
5 references
public interface IBankAccount
{
    1 reference
    string IBAN { get; }

    7 references
    double Balance { get; set; }

    2 references
    bool CanGoNegative { get; }

    4 references
    bool Deposit(double amount);

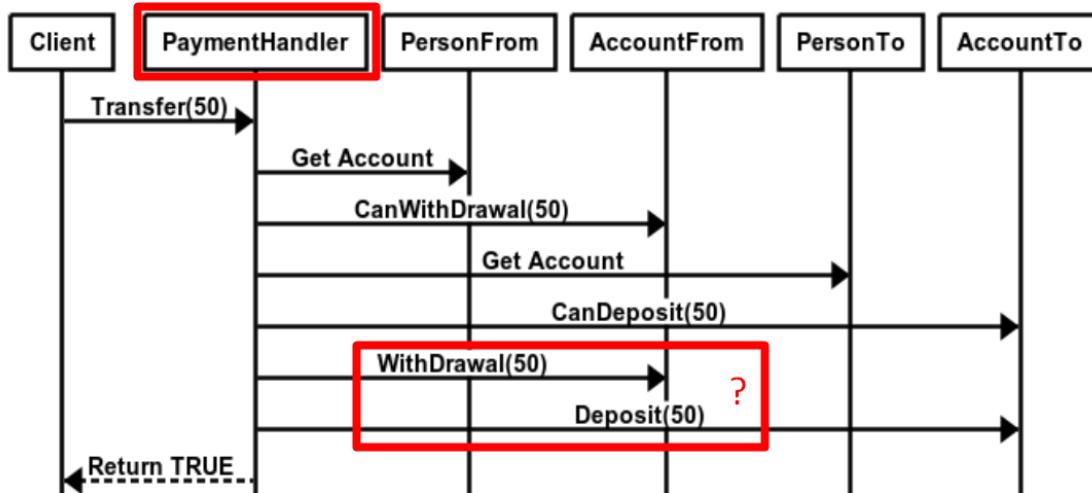
    4 references
    bool WithDrawal(double amount);

    6 references
    bool CanWithDrawal(double amount);
    4 references
    bool CanDeposit(double amount);
}
```

Mocks (2)

- Voorbeeld van een interactie tussen objecten
- De Paymenthandler is de Unit in test. Als het resultaat van de Transfer = TRUE weten we nog niet of het geld daadwerkelijk werd overgeschreven. We kunnen dan daarnaast verifiëren of ook de juiste methods werden aangeroepen op de Account Mocks.

Transfer van een bedrag tussen 2 personen



```
var result = paymenthandler.Transfer(50);  
  
Assert.IsTrue(result);  
//Verify if the correct methods were called on the Mocks  
personFromMock.Verify(x => x.Account.WithDrawal(50));  
personToMock.Verify(x => x.Account.Deposit(50));
```

