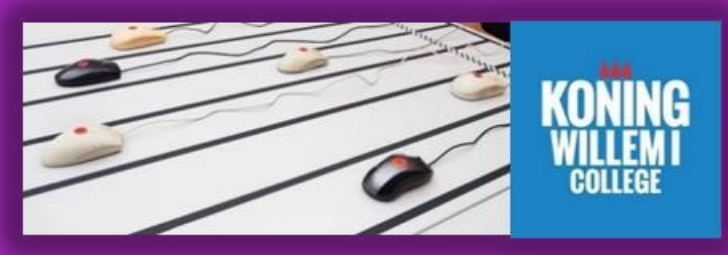
Gevorderd standalone

Testen en verbeteren

hoofdstuk

1

Unit test



Algemene informatie

Onderwerp	Geautomatiseerd testen, de unit test
Leerdoel(en)	1. De student kan voor een C# console applicatie een unit test project opzetten 2. De student oefent met het verzinnen van testmethodes in een testproject 3. De student leert de opbouw (AAA) van een testmethode en begrijpt de toepassing van een assert in een testmethode 4. De student leert hoe je het testproject kan runnen en wat de informatie betekent die in het Test Explorer wordt getoond als de test is uitgevoerd
Vereiste voorkennis	1. De student kan een C# .Net Core console project starten 2. De student heeft de basis kennis van Object georiënteerd programmeren 3. De student is bekend met de Visual Studio IDE
Kwalificatiedossier	☐ B1-K1-W1: Plant werkzaamheden en bewaakt de voortgang ☐ B1-K1-W2: Ontwerpt software ☒ B1-K1-W3: Realiseert (onderdelen van) software ☒ B1-K1-W4: Test software ☐ B1-K1-W5: Doet verbetervoorstellen voor de software ☐ B1-K2-W1: Voert overleg ☐ B1-K2-W2: Presenteert het opgeleverde werk ☐ B1-K2-W3: Reflecteert op het werk

Inhoudsopgave

Introductie

Iedereen weet dat je de code die je maakt goed moet testen. Je slaat een flater bij de klant wanneer er tijdens een demo iets niet naar behoren werkt. Maar, hoe ga je te werk, wat moet je nu precies testen en hoe doe je dat?

Een vorm van tests die je kunt maken zijn *unit-tests*. Een unit is in deze context bijvoorbeeld een component met een bepaalde interface. Een class is hier een voorbeeld van. Alle publiek aanspreekbare zaken in een class zou je kunnen (en moeten) testen. Dat gaat toevallig perfect in een unit test! In deze reader leer je hoe je dat kunt doen en wat daar de grote voordelen van zijn.

Inhoud

Unit-test

→ Code zonder bugs maken is onmogelijk. Door te testen geef je aan wat de kwaliteit is van je software. Niet dat er geen fouten in zitten! Dus, hoe meer je test, des te zekerder van je zaak kunt zijn als je iets zegt over de kwaliteit van je applicatie.

De term geeft het eigenlijk al aan: Unit testen worden gebruikt om een unit van een code te testen. Daarbij wordt het kleinste mogelijke stukje code dat van nut is getest. Dit is in praktijk meestal een methode. Bij een methode verwacht je een vaste output bij een bepaalde input. Een Unit-test wordt geschreven en bedacht door de programmeur zelf. Zij kunnen hiernaar kijken en begrijpen de uitkomst van de test. Het doel van Unit-tests is om te bewijzen dat de unit (een methode) correct is in ieder omstandigheid (verschillende input waardes).

Voorbeeld Unit test met Assert.areEqual()

Stap 0

- Maak een nieuwe console application .NET Core (5/6) project aan genaamd "**UnitTestExample**".
- Maak een nieuwe class aan, genaamd **Calculator.**
- Voeg onderstaande methode toe aan Calculator.cs

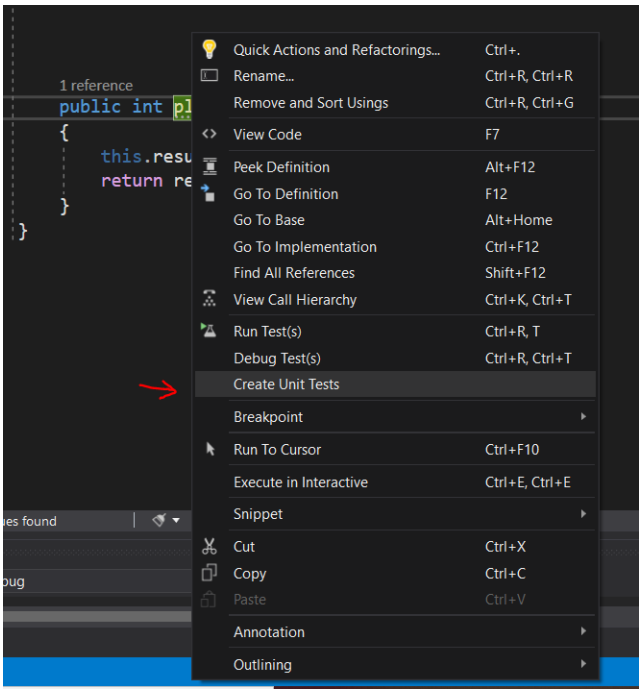
```
2 references | 0 changes | 0 authors, 0 changes
public class Calculator
{
    int result;

    1 reference | 1/1 passing | 0 changes | 0 authors, 0 changes
    public int Plus(int number1, int number2)
    {
        this.result = number1 + number2;
        return result;
    }
}
```

In deze methode worden er twee getallen bij elkaar opgeteld en de uitkomst gereturned.
Nu willen we (zonder een GUI te maken) weten of de methode klopt. Dit kan door een unit test te maken.

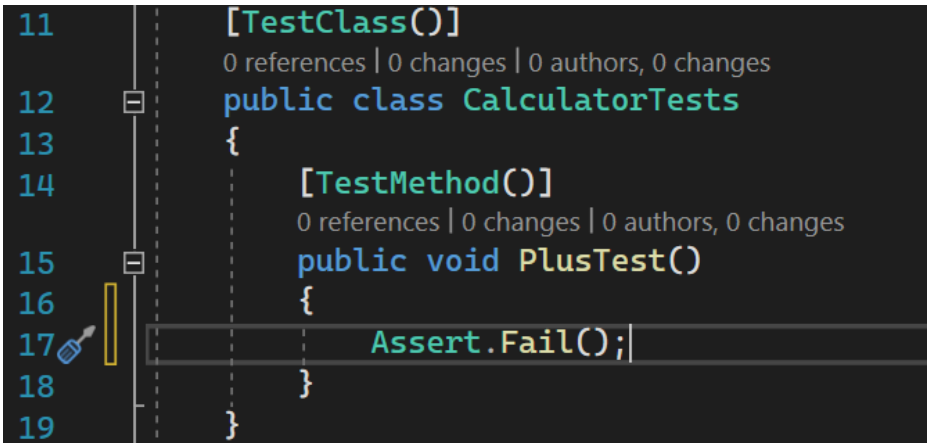
Stap 1 :

Klik met je rechtermuisknop op de methodenaam. Vervolgens klik je op “**Create Unit Tests**”.



Stap 2:

Wanneer je testklas **correct** is aangemaakt, zie je direct al dat de testklasse je methode heeft meegenomen.



Stap 3:

Bij het testen zijn de drie AAA's heel belangrijk. Dit is een pattern voor het correct schrijven van een Unit Test. **In iedere testmethode moeten de 3 onderdelen altijd aanwezig zijn.** (Echter soms zijn Arrange, Act en Assert samengevoegd in 1 commando regel).

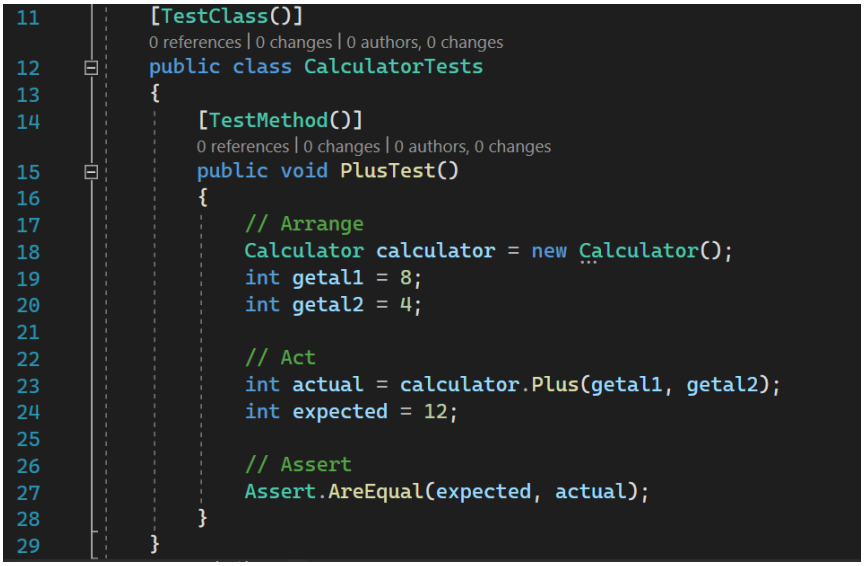
Arrange: In de eerste fase zet je alles op wat nodig is voor het testen met de methode. Denk hierbij aan classes instantiëren, variables aanmaken etc.

Act: Tijdens de tweede fase voer je de methode uit die je wilt testen.

Hierbij maken we variabelen aan, **actual** en **expected.**
De **actual** krijgt de uitkomst van de geteste methode.
De **expected** is de uitkomst die er uit moet komen voor een bepaalde situatie.

Assert: Tenslotte checken we bij de laatste stap met "**Assert**" of de variabelen die we hebben aangemaakt met elkaar overeenkomen.

In dit geval moeten we in de assertion controleren of de actual waarde gelijk is aan de expected waarde is.

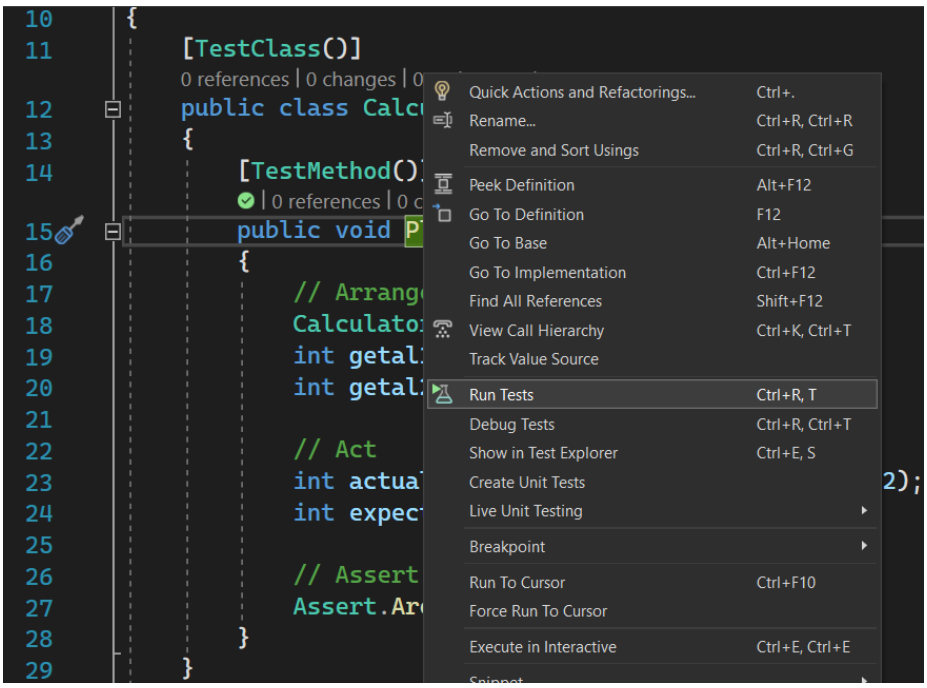


```
11      [TestClass()]
        0 references | 0 changes | 0 authors, 0 changes
12      public class CalculatorTests
13      {
14          [TestMethod()]
            0 references | 0 changes | 0 authors, 0 changes
15          public void PlusTest()
16          {
17              // Arrange
18              Calculator calculator = new Calculator();
19              int getal1 = 8;
20              int getal2 = 4;
21
22              // Act
23              int actual = calculator.Plus(getal1, getal2);
24              int expected = 12;
25
26              // Assert
27              Assert.AreEqual(expected, actual);
28          }
29      }
```

Stap 4:

Nu de test is geschreven moet het alleen nog maar gerunt worden. Dit doe je door met je rechtermuisknop op de methode te klikken en op “**Run Test**” te klikken.

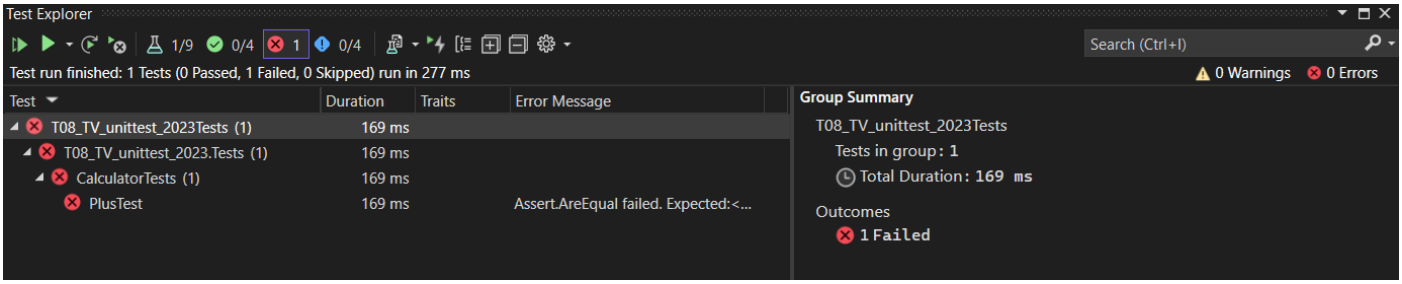
Leuk weetje:
Je kunt op PlusTest( ) de rechter muisknop aanklikken voor Run Tests, maar **ook** op de method Test( ) in class Calculator.



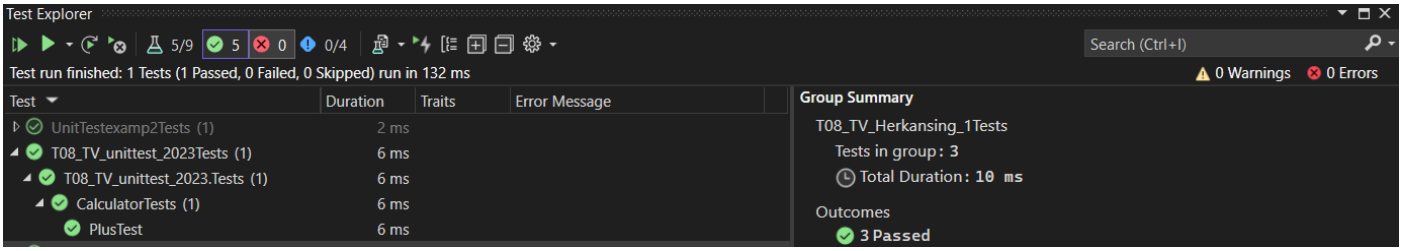
Stap 5:

Tenslotte, controleer of de resultaten correct zijn. Zijn ze niet correct? Lees de error en probeer het opnieuw.

Gefaalde test (Dit betekent dat er een fout is in het geteste onderdeel, analyseer de detail summary) Alle testmethodes krijgen een witte kruis met een rode achtergrond.



Geslaagde test. Dit betekent dat het geteste onderdeel de uitkomst geeft die we verwachten (assertion). Alle testmethodes krijgen een groen vinkje.



Je kunt nu **oefening 1.1** maken.

Lees meer over de standaard opzet van een testmethode (de **3 AAA's**)
https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2019

In een geautomatiseerde test wil je een bevestiging krijgen dat je code volledig werkt. Met een engelse term heet dat **assert. Assert** is ook het commando wat je altijd in een unit test tegenkomt. Een video over de uitleg van een assertion.

Video over het gebruik van assert in code.

Voorbeeld Unit test met Assert.Fail() of Assert.ThrowsException<>()

Unit-test met ongewenste input

Een methode zal soms een foutmelding moeten geven als (één van) de inputparameter(s) een ongewenste waarde is die geen geldige uitvoer kan opleveren. Voorbeeld in het voorbeeld van de rekenmachine is dat we bijvoorbeeld de methode hebben die de WortelUit() een bepaald getal kan berekenen.

Zo is de wortel uit 4 natuurlijk 2, omdat 2 x 2 = 4

En is $\sqrt{16}$ = 4 omdat 4 x 4 = 16

Maar wat is de $\sqrt{-9}$?

Inderdaad dat moet een foutmelding opleveren en er mag geen waarde uitkomen…

Hoe ziet nu de methode WortelUit() en hoe zet je een Unit test op die test op een faal situatie?

Allereerst de methode:

```
/// <summary>
///  Berekent de wortel uit een getal
/// </summary>
/// <param name="number"></param>
/// <returns>Het resultaat van de wortelberekening</returns>
0 references | 0 changes | 0 authors, 0 changes
public double WortelUit(double number)
{
    if (number >= 0)
    {
        return Math.Sqrt(number);
    }
    else
    {
        throw new ArgumentException("Wortel berekenen kan alleen met positieve getallen");
    }
}
```

Hoe ziet de testmethode eruit ?

```
[TestMethod()]
0 references | 0 changes | 0 authors, 0 changes
public void WortelUitTestNegatiefGetal()
{
    //Arrange
    double foutewaarde = -16;
    Calculator calc = new Calculator();

    //Act
    try
    {
        calc.WortelUit(foutewaarde);
    }
    catch (ArgumentException exc)
    {
        //Assert test op de message
        Assert.IsTrue(exc.Message == "Wortel berekenen kan alleen met positieve getallen");
        return;
    }

    //Hier mag ie dan sowieso niet komen
    Assert.Fail();
}
```

1. De testmethode naam wordt de situatie omschrijving meegenomen (testen met een negatief getal)
2. Er is een Arrange, Act en nu **2 ASSERTS**
3. We hebben een try catch constructie gebouwd die de foutmelding (de exceptie) kan opvangen.

4. Met de Assert.IsTrue() kijk ik of de foutmessage overeenkomt met wat ik verwacht.
5. Na de Assert.IsTrue return ik meteen de method zodat het niet verder gaat.
6. **Na de try en catch faalt de test omdat ik met een negatieve input waarde in de catch moet komen.**

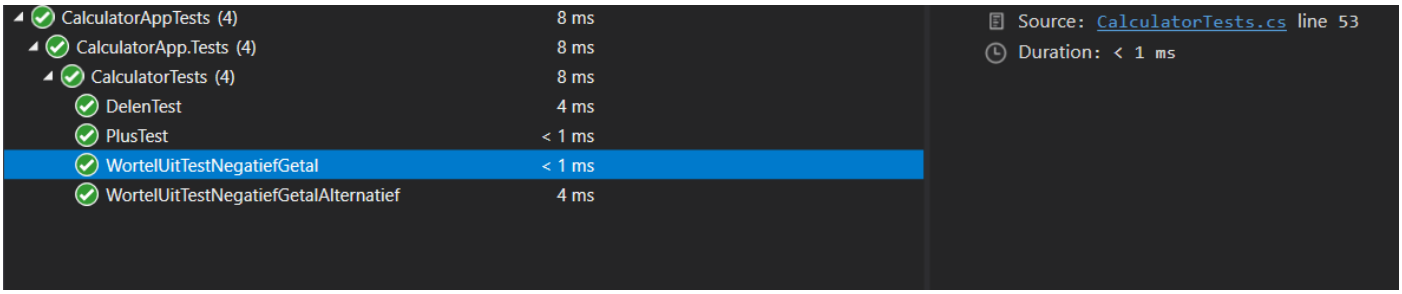
Een alternatief voor de bovenstaande test en een veel kortere notatie is de volgende:

```
[TestMethod()]
0 references | 0 changes | 0 authors, 0 changes
public void WortelUitTestNegatiefGetalAlternatief()
{
    //Arrange
    double foutewaarde = -16;
    Calculator calc = new Calculator();

    //Act en Assert tegelijkertijd
    Assert.ThrowsException<ArgumentException>(() => calc.WortelUit(foutewaarde));

}
```

Het resultaat in de test explorer is hetzelfde.



Je kunt nu **oefening 1.2** maken.

Naslag over de Assert klasse en o.a. de methodes Assert.AreEqual(), Assert.Fail(), Assert.ThrowsException()