Inhoud

[Logboek 2](#_Toc88048337)

[Planning 2](#_Toc88048338)

[Onderzoek 3](#_Toc88048339)

[Hoofdstuk 1 4](#_Toc88048340)

[Hoofdstuk 2 5](#_Toc88048341)

[Optellen 5](#_Toc88048342)

[Reset 7](#_Toc88048343)

[Delen 8](#_Toc88048344)

[Vermenigvuldigen 10](#_Toc88048345)

[ZeroException 12](#_Toc88048346)

[Exceptions in 1 test 14](#_Toc88048347)

[Bronnen 16](#_Toc88048348)

# Logboek

|  |  |  |
| --- | --- | --- |
| **Wanneer** | **Wat** | **Duur** |
| 20/09 | Onderzoek | 2u |
| 27/09 | Hoofdstuk 1 | 4u |
| 04/10 | Hoofdstuk 2 | 3u |
| 11/10 | Hoofstuk 2 | 4u |
| 19/10 | Aanpassing document aan de hand van feedback | 2u |
| 24/10 | Aanpassing document aan de hand van feedback | 1u |
| 17/11 | Hoofdstuk 2 | 2u |

# Planning

|  |  |
| --- | --- |
| **Wanneer** | **Wat** |
| 17/11 | Hoofdstuk 2 |
| 21/11 | Hoofdstuk 2 |
| 24/11 | Cursus |

# Onderzoek

Ik heb vandaag nagedacht over wat mij erg zou interesseren om testen voor te schrijven. Ik ben dan tot de conclusie gekomen dat ik c# .net zou kiezen sinds we dit niet leren in onze opleiding en ik geïnteresseerd ben in c# .net. Ik heb een cursus gevonden bij openclassrooms waar de bron van terug te vinden is onderaan dit document bij Bronnen.

# Hoofdstuk 1

Ik heb vandaag al het eerste en tweede hoofdstuk van de cursus doorlopen.

Hierbij bij zou ik bij aanvang vooral begrijpen waarom testing nuttig was in de .net omgeving en dit is ook duidelijk geworden.

Ik heb vandaag onderstaande spullen geleerd.

Er bestaan 3 soorten automatische testen namelijk:

* Unit
* Mock
* Integration tests.

Hierna werd het verschil tussen Test en behavior driven development uitgelegd. Bij TDD wordt eerst een test geschreven en daarna de code zodat deze zeker juist is bij BDD wordt er meer gekeken naar hoe je de werking kan tonen in normale tekst zodat iedereen dit kan begrijpen.

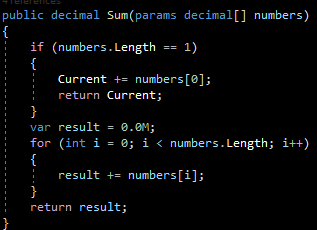
Vervolgens ging het over hoe je een goeie unit test kan schrijven Deze gaan volgens het FIRST principe dit betekent **F**ast, **I**solated, **R**epeatable, **S**elf-validating, **T**imely. Dit betekent dus vooral dat de unit testen snel moeten zijn maar vooral geïsoleerd zodat ze niet afhangen van andere code. En natuurlijk moet de testen blijven kunnen runnen.

# Hoofdstuk 2

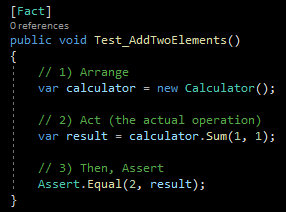
## Optellen

Hierna was er het 2e hoofdstuk hier werden aan de hand van code uitgelegd hoe je testen moet schrijven. Ik heb deze dan gevolg en deze kunt u vinden met de naam CalculatorProgram.

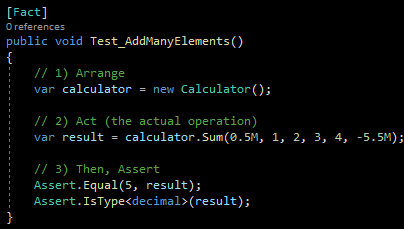
Eerst werd er een methode in de klasse calculator geschreven die we nodig gaan hebben om onze testen te schrijven namelijk Sum. Hieronder ziet u dat we decimale nummers hier gaan meegeven. Dan wordt eerst een variabel resultaat op 0 gezet en vervolgens gaan we met een for loop door al de nummers heen gaan en dat nummer altijd optellen bij resultaat. Als dit gedaan is geven we het resultaat terug.



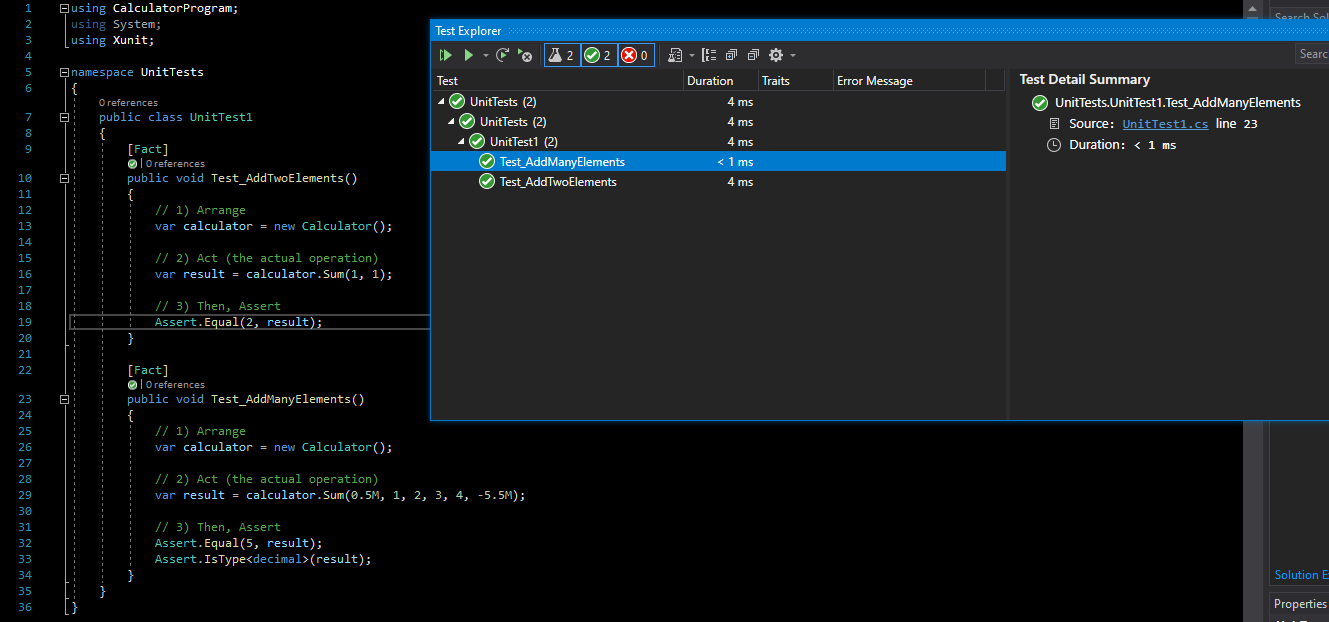
Hierna gaan we onze eerste test maken namelijk Test\_AddTwoElements. Hier beginnen we met een calculator aan te maken vervolgens maken we een variabele result en zetten we deze gelijk aan de waarde dat we terugkrijgen uit de methode Sum van de Caclulator klasse. We geven hierbij de nummers 1 en 1 mee. Ten laatste kijken we met Assert of de variabele resultaat gelijk is aan 2.



Vervolgens gaan we nog een test maken vooraleer we gaan kijken of deze testen lukken en die test is Test\_AddManyElements. We beginnen hier ook weer met een calculator aan te maken. Vervolgens gaan we weer een variabele result aanmaken en nu geven we meerdere nummers mee. Dan gaan we met een Assert weer kijken of result gelijk is aan 5 en gaan we met een Assert.IsType ook kijken of result wel een decimal getal is.



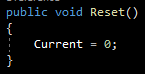
Vervolgens voeren we de testen uit en kunt u zien dat deze 2 testen goed gelukt zijn.



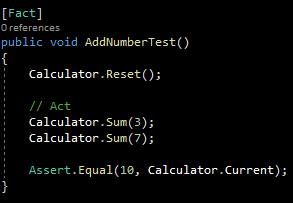
## Reset

Nu gaan weer een test schrijven AddNumberTest waar we een nieuwe method Reset gaan gebruiken en ook nummer per nummer gaan optellen.

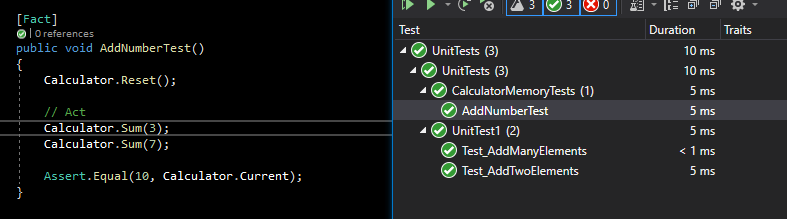
Eerst de methode reset welke heel simpel is hier gaan we de gedeclareerde waarde Current gelijkzetten aan 0.



Dan onze test, we gaan beginne met eerst Reset op te vragen zodat Calculator.Current nu 0 is. Vervolgens gaan we de methode sum 2 keer uitvoeren, een keer met het getal 3 en daarna met het getal 7. Ten laatste gaan we weer kijken met een Assert of Caclulator.Current nu gelijk is aan 10.



Hieronder ziet u dat als de test wordt uitgevoerd dat deze slaagt.

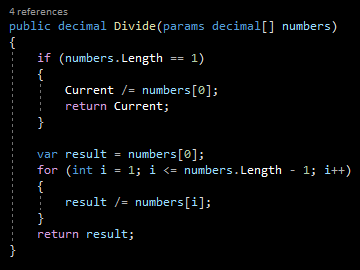


## Delen

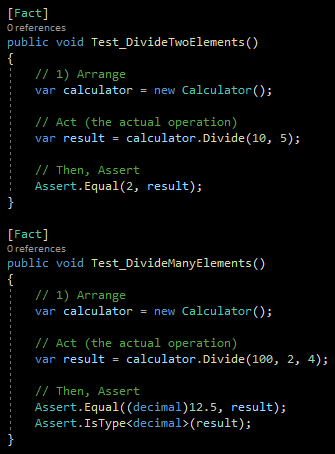
Hierna hebben we nog wat testen uitgevoerd waar we gingen delen. De testen bij het delen verloopte op dezelfde manier als het optellen aleen moesten we hier een nieuwe methode voor maken namelijk Divide.

Bij Divide geven we ook weer decimale waarden mee. Eerst gaan we hier bekijken of de lengte van de array met nummers gelijk is aan 1. Als dit zo is dan gaan we het nummer gebruiken om Current te delen door dat getal sinds we anders geen getal hebben om door te delen. Hier returnen we dan ook Current.

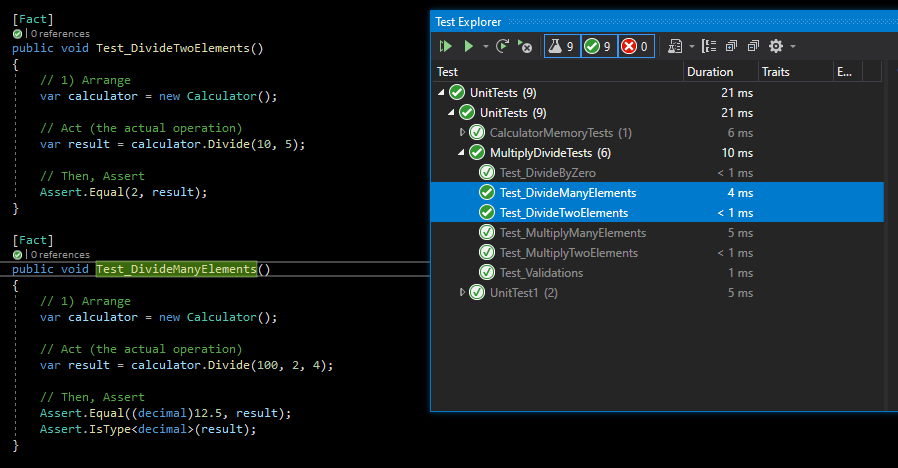
Als dit niet zo is maken we een variabnele result en zetten we deze gelijk aan het eerste nummer in de array. Vervolgens gaan we met een for loop door alle nummers behalve de eerste. Hier delen we dan result met het tweede getal en vervolgens result met het derde enzovoort tot we alle nummers doorlopen hebben. Ten slotte geven we dan result terug.



Hieronder ziet u dan de twee testen die hetzelfde proces doorlopen als toevoegen behalve dat hier de methode Divide wordt gebruikt.



Hieronder kan u dan zien dat de twee testen van delen goed lukken.

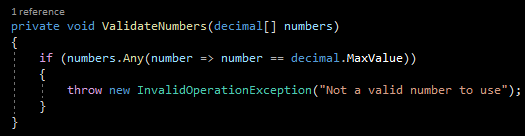


## Vermenigvuldigen

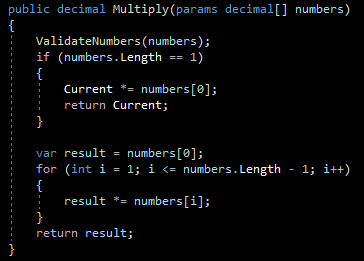
Hierna hebben we nog tests gedaan met vermenigvuldigen. Deze verlopen weer hetzelfde als delen en optellen maar natuurlijk hebben we hier weer een nieuwe methode genaamd Multiply nodig.

Bij deze methode gaan we natuurlijk eerst weer een array van decimalen waarden binnenkrijgen. Vervolgens gaan we nog een nieuwe methode gebruiken namelijk ValidateNumbers.

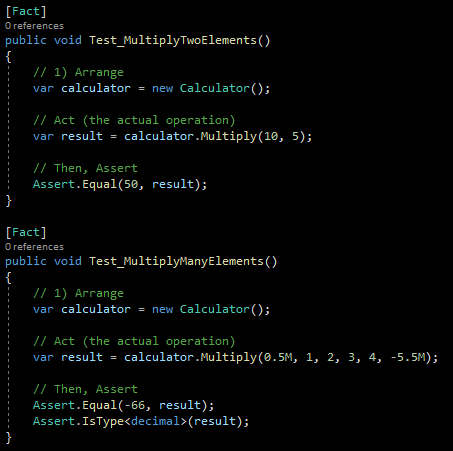
Hier kijken we met een if statemant of een van de nummers in de array de maxValue van een decimaal getal is. Als dit zo is het natuurlijk geen goed getal en gaan we een Exception terugsturen met de boodschap “Not a valid number to use.”.



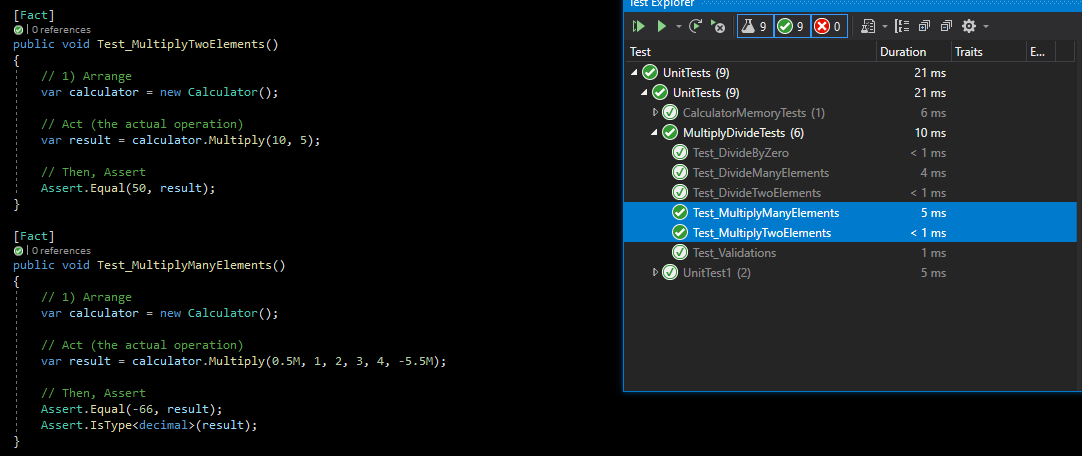
Als de nummers dan gevalideerd zijn wordt er gekeken of er maar één nummer in de array zit als dit zo is gaan we weer Current gebruiken en het getal van de array hiermee vermenigvuldigen en dan Current terugsturen. Als dit niet zo is dan zullen we weer hetzelfde doen als bij delen alleen veranderd het toestandsteken hier van / naar \*.



Hieronder kunt u dan de twee testen zien die we gaan uitvoeren. Deze zijn hetzelfde als delen en optellen alleen gebruiken we hier de methode Multiply.



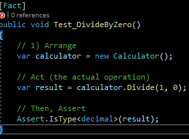
Hieronder kunt u dan zien dat deze twee tests tot een goed einde komen.



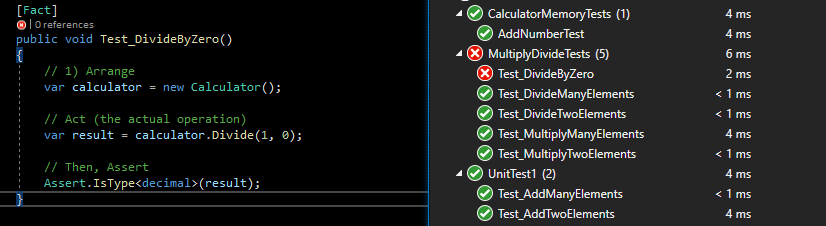
## ZeroException

Vervolgens gaan we een test schrijven waar we gaan delen met 0. Dit word gedaan om te leren werken met Exceptions in testing.

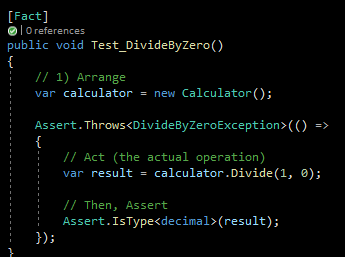
We maken hiervoor de test Test\_DivideByZero voor. Hierin gaan we hetzelfde doen als divide maar gaan we als tweede parameter een 0 meegeven.



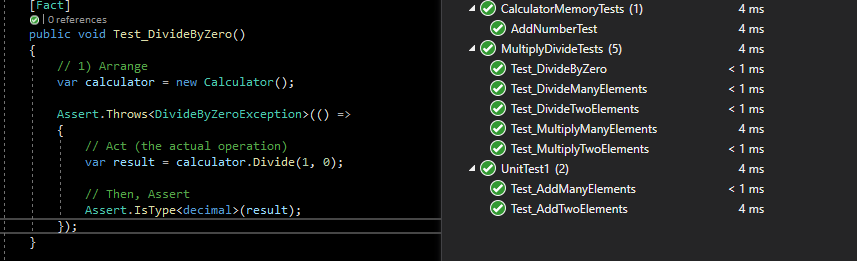
Als we deze test dan runnen dan gaat deze test falen en gooit het systeem een DivideByZeroException.



Om dit op te lossen gaan we onze Act en Assert in een Asset.Throws<DivideByZertoException zetten zodat als het systeem deze error gooit dat de assert die opvangt en dan zegt dat dit te verwachten was en dat de test eigenlijk gelukt is sinds het doel een DivvideByZeroException was.



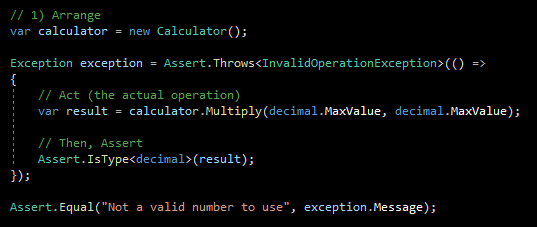
Hieronder kan u dan zien dat de test nu wel tot een goed einde komt.



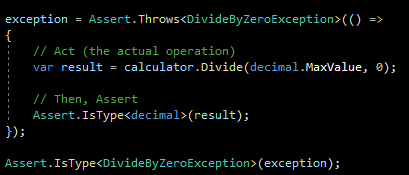
## Exceptions in 1 test

Hierna wordt er geleerd hoe je in plaats van exceptions per test te schrijven dat je de gegevens ook kan valideren in een test. Daarmee weet je dan of er geen expections gaan komen in de andere tests.

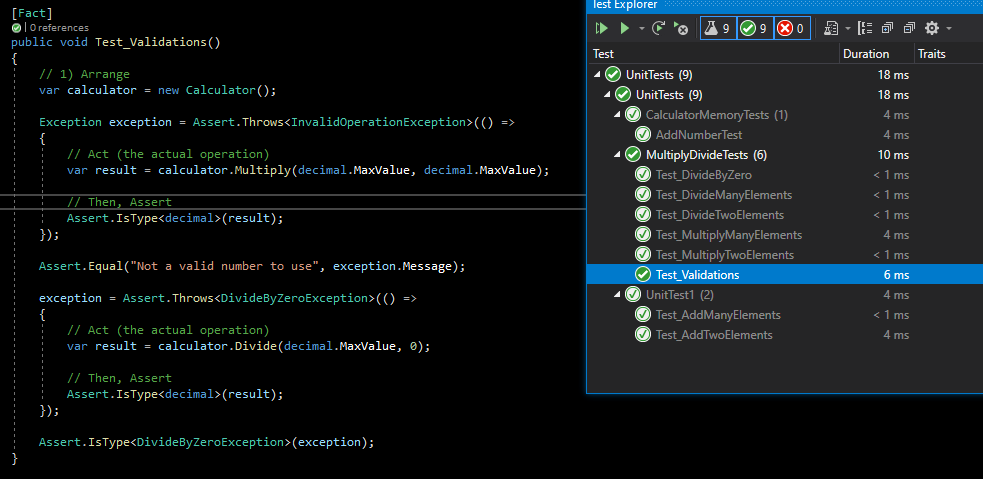
Dit wordt gedaan in de Test\_Validations. Hier zijn 2 exceptions de worden bekeken. Eerst wordt bekeken of de waarden bij het vermenigvuldigen wel correct zijn zoals ook in de test Multiply wordt gedaanmet de methode ValidateNumbers.



En vervolgens nog een exception namelijk de DivideByZeroException die we daarnet ook hebben laten zien.



Als de test dan worden uitgevoerd zal u zien dat de twee exceptions dus hun exceptrions gooien en dat zo de testen natuurlijk op een goed einde worden gebracht.

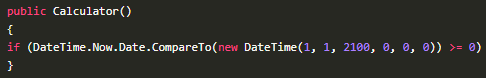


## Mock Data

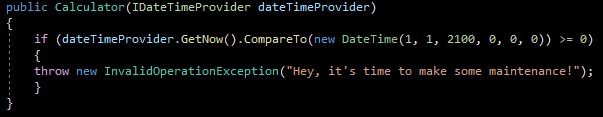
Mock-gegevens zijn nepgegevens die kunstmatig in een stukje software worden ingevoegd. Mocken is in wezen het op gecontroleerde wijze simuleren van het gedrag van echte gegevens.

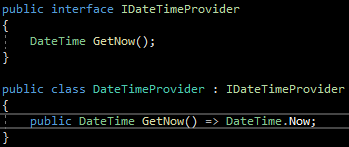
We gaan hier een voorbeeld gebruiken om het nut en de werking te laten zien. Het voorbeeld dat we gaan gebruiken is dat we op een bepaalde datum namelijk juist voor 2100 een message gaan weergeven dat er maintanance moet gebeuren.

De code nodig is deze hieronder maar hier zal je moeten wachten tot 2099 voor dat dit wordt uitgevoerd. Dit is natuurlijk een groot probleem om te testen.



Om mocks te gebruiken gaan we deze code hervormen zodat we de interface IDateProvider gaan gebruiken. We gebruiken hier dan de GetNow van datetimeprovider. In datetimeprovider gaan we die GetNow zetten dat dit DateTime.now is.





# Bronnen

https://openclassrooms.com/en/courses/5670586-test-your-net-applications/5945960-why-is-testing-important