PROGRAMMEERTECHNIEKEN EN TESTEN–UNIT TESTEN

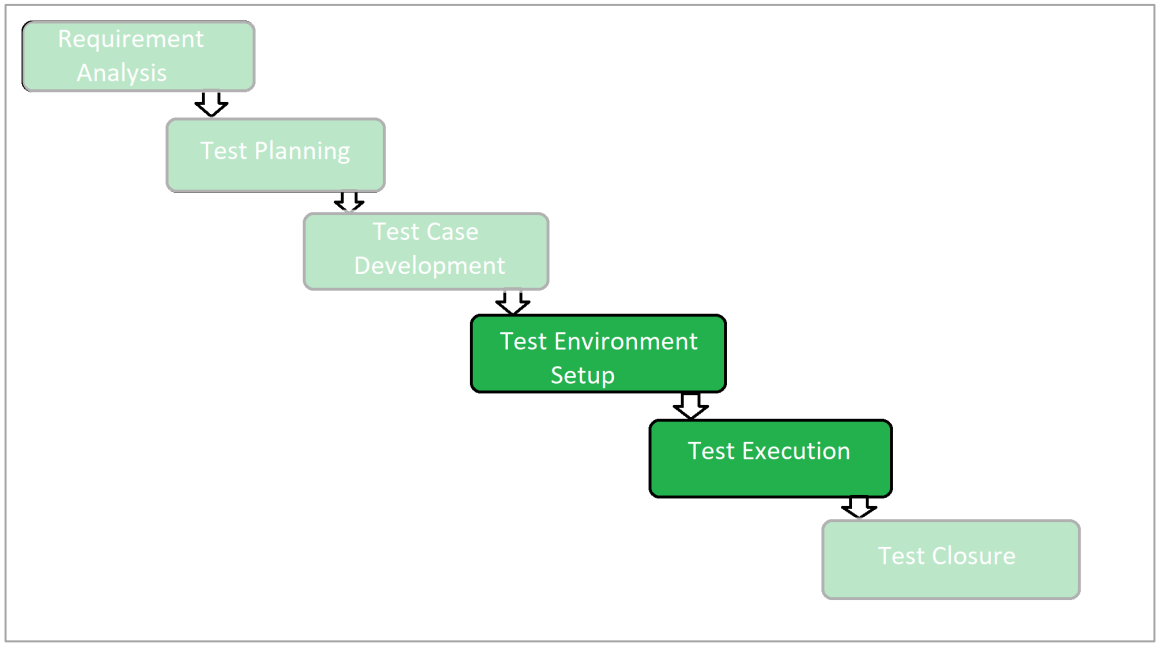
MATTHIAS DRUWÉ
SAM VAN BUGGENHOUT

Wat is Unit Testen?

- Een *Unit Test* is een **stuk code** (meestal een methode) dat een ander stuk code aanroept en de **correctheid** van een aantal **veronderstellingen** nagaat.
  Indien deze veronderstelling fout blijkt te zijn, is de test gefaald.
  Met een "unit" wordt een **methode** (of functie) bedoeld.
- Wat getest wordt, wordt ook het **"SUT"** (System Under Test) genoemd (of "CUT", Class Under Test)

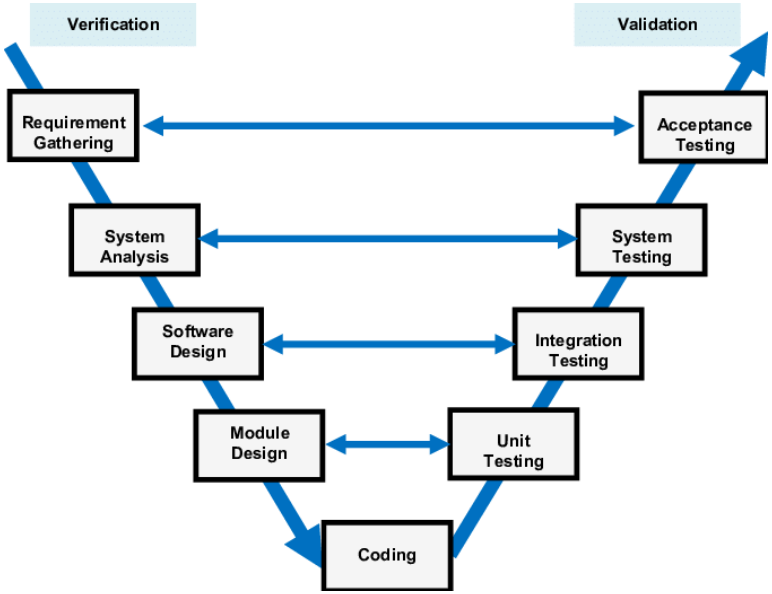
- Voorbeeld: *je schrijft een Calculator-klasse om wiskundige berekeningen uit te voeren. Je gaat ervan uit dat wanneer je de Som-methode van deze klasse aanroept met de argumenten 10 en 20, de return-waarde van de methode 30 is. Je schrijft een test waarin je dit verifieert. Indien de return-waarde NIET overeenkomt met de waarde 30, is de test gefaald.*

Unit Tests: situering



Unit Tests: situering

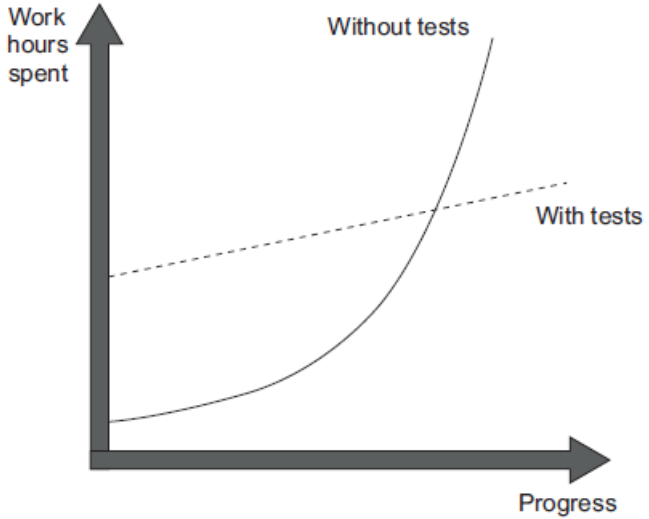
- Situering van Unit Testing in het **V-model**:



Bron: https://www.researchgate.net/figure/V-Model-life-cycle-for-the-automotive-software-testing_fig1_314665883

Doel van Unit Tests

- Unit Tests zorgen voor een **duurzame groei** van het software project.



Snelheid van ontwikkeling neemt snel af = *software entropy*

Kwaliteit van code gaat snel achteruit (bugs-fixes introduceren nieuwe bugs, code wordt complex, weinig gestructureerd, ...)

Bron: Khorikov, V. (2020). The goal of unit testing. In Unit Testing Principles, Practices, and Patterns (p. 6). Shelter Island, NY: Manning Publications.

Doel van Unit Tests

- Goede Unit Tests voorkomen dit probleem.
  Ze fungeren als een “vangnet” voor **regressies**.
- **Regressie** = wanneer een functionaliteit **niet meer werkt** zoals verwacht, na een bepaalde **gebeurtenis** (bv.: een wijziging aan de code, refactoring*, toevoegen van nieuwe functionaliteiten, …)
- Unit Testen zorgen ervoor dat **bestaande functionaliteiten blijven werken**, ook na het introduceren van nieuwe functionaliteiten of het refactoren van bestaande code.

** Refactoring = het herstructureren van bestaande code omwille van leesbaarheid, onderhoudbaarheid, … zonder de functionaliteit ervan te veranderen*

Wat is een “goede” Unit Test?

- **MAAR:** niet élke test is een *goede* Unit Test!
- Eigenschappen van goede Unit Tests:
  - **Geautomatiseerd** en kunnen **herhaald** worden (mogen **geen “state”** achterlaten!)
  - Moeten **eenvoudig geïmplementeerd** kunnen worden
  - Eens geschreven, moeten ze beschikbaar blijven voor **later gebruik**
  - Iedereen moet ze kunnen uitvoeren
  - Ze moeten kunnen uitgevoerd worden met een druk op een knop (**geen manuele interventies** nodig)
  - Ze moeten **snel** uitgevoerd kunnen worden

Wat is een “goede” Unit Test?

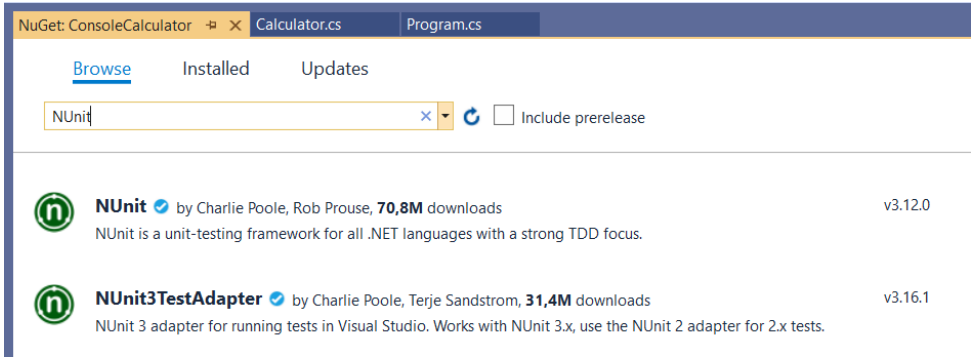
- Rekening houdend met deze eigenschappen, kunnen we de definitie van een Unit Test herformuleren:
  - Een unit test is een **geautomatiseerd** stukje code dat de methode/klasse aanroept die getest wordt en een aantal **veronderstellingen** (assumpties) **verifieert** over het **“logische gedrag”** van deze methode/klasse. Een unit test wordt (zo goed als) altijd geschreven met behulp van een **testing framework**. Het kan **snel geschreven en uitgevoerd** worden. Het is volledig **geautomatiseerd**, **betrouwbaar**, **leesbaar** en **onderhoudbaar**.
  - **Logische code** = een stuk code dat “logica” bevat (een if-statement, een iteratie-structuur, switch-case, berekening, …). Dus: géén properties/constructors/…

Testing Framework

- Er bestaan heel wat **Testing Frameworks** voor het schrijven van Uni Testen in C# (en andere talen binnen het .NET-platform)
- De meest gebruikte frameworks zijn:
  - MS Test
  - xUnit.NET
  - NUnit
- In deze module zullen we gebruik maken van het framework **NUnit**
- Waarom?
  - Zeer populair testing framework
  - Open Source
  - Goede integratie met Visual Studio
  - Stabiel en matuur framework
  - Goede documentatie

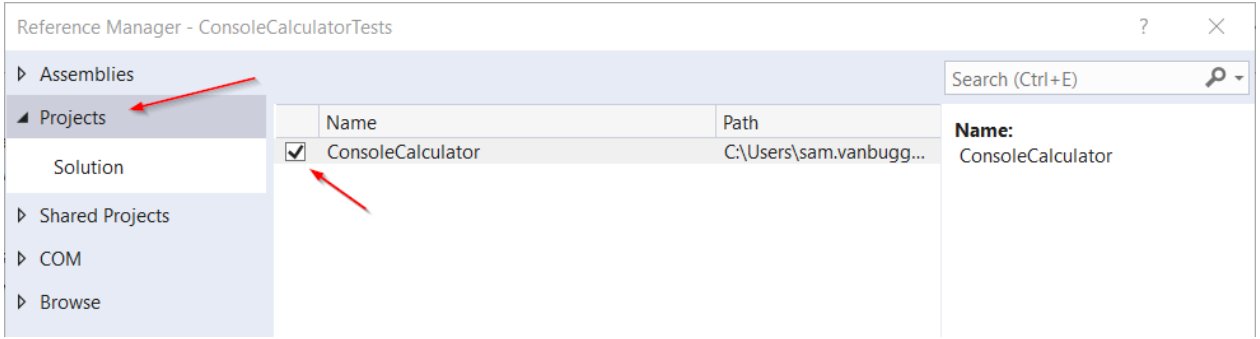
Testing Framework

- Zet je testen steeds in een **apart VS Project**!
- Stappen:
  1. Maak een nieuw project van het type **Class Library (.NET Framework)**
  2. **Rechtsklik** op het nieuwe project en kies **"Manage NuGet Packages…"**
  3. **Installeer de volgende packages:**
     - ✓ NUnit
     - ✓ NUnit Test Adapter



Testing Framework

- Maak voor elk project dat je test een bijhorend Test-project aan!
- Alvorens je de klasses van het project dat je wilt testen kan gebruiken, moet je eerst een **referentie leggen** vanuit het Test-project naar het project dat getest wordt
  - Rechtsklik op je Test-project
  - Kies: *Add* > *Reference*
  - Selecteer in het linker menu de optie "Projects" en vink het vakje aan vóór het project dat je wilt testen



Een eerste Unit Test

```
[TestFixture] 1
public class CalculatorTests
{
     [Test] 2
     public void Som_van_twee_getallen()
     {
          double eerste = 10;
          double tweede = 20;
          Calculator calculator = new Calculator();

          double resultaat = calculator.Som(eerste, tweede);

      3  Assert.AreEqual(30, resultaat);
     }
 }
```

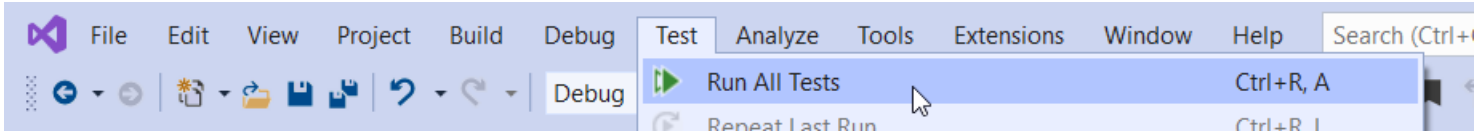
1) *TextFixture-attribuut* geeft aan dat deze klasse NUnit testen bevat

2) *Test-attribuut* geeft aan dat deze methode een unit test is en moet aangeroepen worden bij het uitvoeren van de tests

3) De *Assert-klasse* bevat een aantal static methoden om na te gaan of het werkelijke resultaat overeenkomt met het verwachte resultaat

Een eerste Unit Test

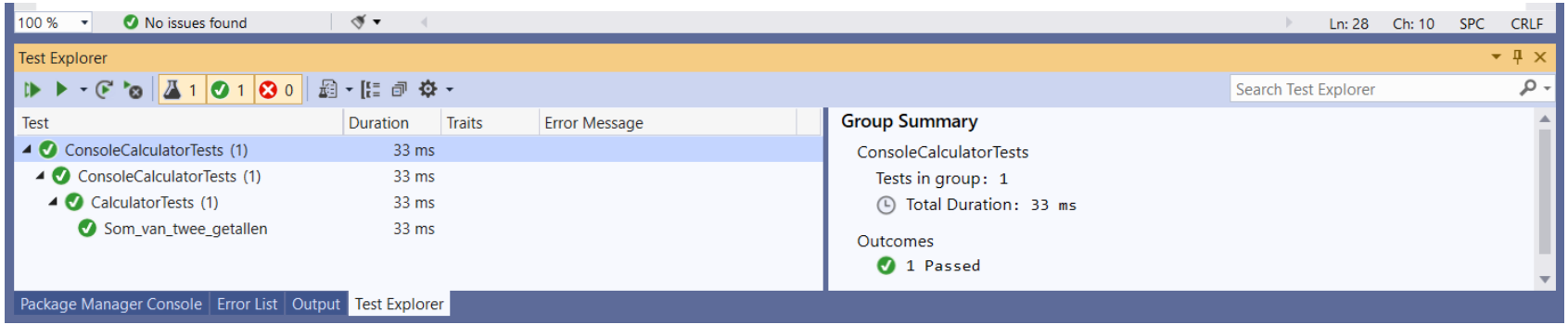
- Vervolgens kan je de unit tests **uitvoeren**
- **Alle testen** in het project uitvoeren:
  - Rechtsklik op het Test-project en selecteer de optie *"Run Tests"*
  - Of: *Test* > *Run All Tests* in de menubalk



- Tests binnen een **specifieke klasse** uitvoeren:
  - Rechtsklik op de gewenste klasse (in **Code Editor** of **Solution Explorer**), kies *"Run Test(s)"*

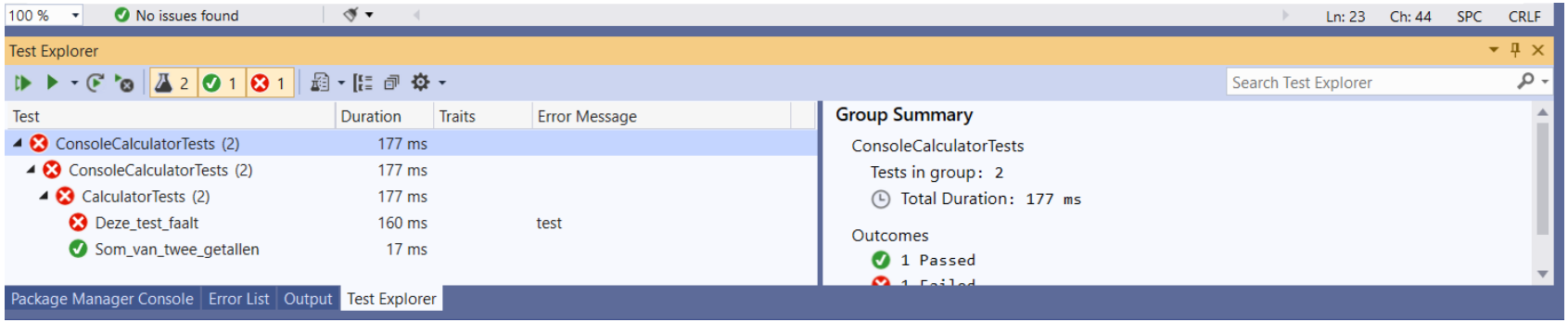
Een eerste Unit Test

- De testresultaten worden getoond in de **Test Explorer**:
  - Aan de linkerkant zie je welke testen uitgevoerd werden en of deze testen geslaagd zijn of niet. Je vindt er tevens terug hoe lang het duurde om de test uit te voeren.
  - Aan de rechterkant vind je een samenvatting van de tests die uitgevoerd werden en de resultaten van deze tests (aantal geslaagde/gefaalde testen, totale duur om de testen uit te voeren, ...)



Een eerste Unit Test

- Het icoontje dat vóór de naam van de testmethode getoond wordt, geeft aan of deze test geslaagd of gefaald is.
- Indien één van de testen binnen een testklasse faalt, krijgt de volledige klasse een rood icoontje om aan te geven dat minstens één test gefaald is.

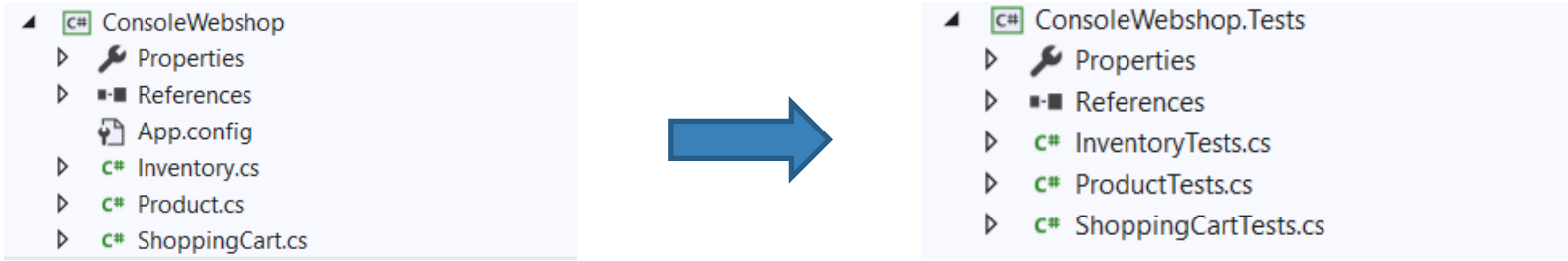


Opmerkingen/aandachtspunten

- Indien je een klasse vanuit een ander project (een andere *Assembly*) wilt gebruiken, moet je deze klasse **public** maken.
  *In de test uit het voorbeeld, dient de klasse Calculator dus **public** te zijn.*
- Zorg er tevens voor dat alle **testmethoden** public zijn!
- Streef naar een één-op-één-relatie tussen code-projecten en test-projecten
- Zorg voor een duidelijke naamgeving om dit verband duidelijk te maken:
  - Naam code-project: "*ConsoleCalculator*", naam test-project: "*ConsoleCalculator**.Tests***"
- Probeer een gelijkaardige conventie te gebruiken voor het benoemen van je testklasses.
  - Stel: class-under-test: "*Calculator*", testklasse: "*Calculator**Tests***"

Opmerkingen/aandachtspunten

- Voorbeeld naamgeving:



- Zorg voor een zinvolle naamgeving van je testmethodes!
  - *Benoem de methode alsof je het **scenario** aan iemand zou moeten uitleggen die **geen** programmeer-achtergrond heeft, maar **wél** kennis heeft van het domein van de applicatie (de "business")*
  - *Scheid woorden d.m.v. underscores '_' (**voor je tests** mag je dus afwijken van de standaard naamgevingsconventie binnen C#)*

Opmerkingen/aandachtspunten

- Voorbeelden van **slechte** naamgeving:
  - Test1(), Test2(), Test3(), … → **NIET DOEN!**
  - TestSum(), TestInventory(), TestAdd() → **wees concreter, *Sum* van? *Add* wat?**
  - IsDeliveryValid_InvalidDate_ReturnsFalse() → **zorg voor leesbaarheid!**

- Voorbeelden van **goede** naamgeving:
  - Sum_of_two_numbers()
  - Delivery_with_a_past_date_is_invalid()
  - Purchase_fails_when_not_enough_inventory()

Structuur van een Unit Test

- Structureer je testen volgens het **AAA-patroon**:

```
[Test]
public void Som_van_twee_getallen()
 {
    //Arrange
    double eerste = 10;
    double tweede = 20;
    Calculator calculator = new Calculator();

    //Act
    double resultaat = calculator.Som(eerste, tweede);

    //Assert
    Assert.AreEqual(30, resultaat);
 }
```

1. **A**rrange
2. **A**ct
3. **A**ssert

Structuur van een Unit Test

- In de ***Arrange-sectie*** breng je het *System Under Test* in de **toestand** ("state") die nodig is om de test uit te voeren (aanmaken van de nodige objecten, manipulatie van deze objecten, …) → baseer je op de **pre-condities** van de test case!
- In de ***Act-sectie*** voer je de teststappen uit (bv. door het aanroepen van methodes) en vang je eventueel het **resultaat** van deze acties op.
- In de ***Assert-sectie*** verifieer je het resultaat van de test (bv.: een **return-waarde**, welke **methoden** wel/niet aangeroepen werden en hoe vaak ze aangeroepen werden, de **finale toestand** van het SUT, …).

- Het AAA-patroon wordt ook het "*Given-When-Then*"-patroon genoemd

Assert-klasse in NUnit

- De ***Assert***-klasse van het NUnit-framework bevat een aantal handige voor het verifiëren van de resultaten van teststappen.
- Hieronder vind je een tabel met enkele veelgebruikte methoden *(oude syntax)*:

Methode(s)	Gebruik	Voorbeeld
`AreEqual()`	Gaat na of twee waarden aan elkaar gelijk zijn	`Assert.AreEqual(50, totaal);`
`IsTrue()/IsFalse()`	Gaat na of een waarde *true* of *false* is	`Assert.IsTrue(valid);`
`IsNull()/IsNotNull()`	Gaat na of een referentie een *null-reference* is	`Assert.IsNotNull(customer);`
`Greater()/GreaterOrEqual()/ Less()/LessOrEqual()`	Gaat na of een waarde *groter dan (of gelijk aan)/kleiner dan (of gelijk aan)* een andere waarde is	`Assert.Greater(customer.Age, 18);`
`AreSame()/AreNotSame()`	Gaat na of twee referenties naar het *zelfde object* verwijzen (reference-equality)	`Assert.AreSame(cust1, cust2);`

Een tweede voorbeeld

- *Context: we maken een programma voor het beheer van de inventaris van een webshop*
- **Test case:** *Aankoop lukt indien voldoende voorraad*
  - Beschrijving: indien de klant producten koopt die voldoende voorradig zijn, wordt de aankoop succesvol afgerond en wordt de inventaris bijgewerkt.
  - Pre-condities: er zijn 10 flessen shampoo beschikbaar in de inventaris
  - Stappen: de klant koopt 5 flessen shampoo
  - Resultaten (post-condities):
    - De aankoop is gelukt
    - Het aantal flessen shampoo van de inventaris werd met 5 verlaagd

Een tweede voorbeeld

```
[TestFixture]
class CustomerTests
 {
    [Test]
    public void Purchase_succeeds_when_enough_inventory()
     {
       //Arrange
       Store store = new Store();
       store.AddInventory(Product.Shampoo, 10); //Pre-conditie
       Customer customer = new Customer();

       //Act
       bool success = customer.Purchase(store, Product.Shampoo, 5); //Test-stap

       //Assert
       Assert.IsTrue(success); //Aankoop gelukt
       Assert.AreEqual(5, store.GetInventory(Product.Shampoo)); //Er blijven nog 5 flessen shampoo over
     }
 }
```

Unit Tests: richtlijnen

- Vermijd **meerdere** Arrange, Act, Assert-secties!
  - Indien je meerdere Act-secties hebt, gescheiden door Assert-secties (en eventueel Arrange-secties), wijst dit erop dat je **meerdere gedragingen** in één test aan het testen bent.
  - **Oplossing:** deel te test op in twee afzonderlijke testmethoden
- Vermijd **if-statements** in je testmethoden!
  - Testen zélf mogen **géén logica** bevatten
  - Indien nodig: splits je test op in twee afzonderlijke testen

Unit Tests: richtlijnen

- Opgelet voor **Act-secties** die langer zijn dan **één lijn** code!

```
[Test]
public void Purchase_succeeds_when_enough_inventory()
 {
    //Arrange
    Store store = new Store();
    store.AddInventory(Product.Shampoo, 10);
    Customer customer = new Customer();

    //Act
    bool success = customer.Purchase(store, Product.Shampoo, 5);
    store.RemoveInventory(Product.Shampoo, 5);

    //Assert
    Assert.IsTrue(success);
    Assert.AreEqual(5, store.GetInventory(Product.Shampoo));
 }
```

... Wat kan hier fout gaan?

Unit Tests: richtlijnen

- Indien men vergeet om de methode ***RemoveInventory(..)*** aan te roepen (met de correcte argumenten), treedt er een **inconsistentie** op.
  - Vanuit business-oogpunt, heeft een succesvolle aankoop **twee gevolgen**: de klant ontvangt het product en de inventaris wordt verminderd.
  - Beide gevolgen moeten steeds **gelijktijdig** optreden
  - **Dus:** beide acties moeten in **één methode** uitgevoerd worden! **(= encapsulatie)**
  - **Oplossing:** in de ***Purchase(..)***-methode van de klasse ***Customer*** moet de ***RemoveInventory(..)***-methode aangeroepen worden

Unit Tests: richtlijnen

- Sommige "best practices" beweren: *één Assert-statement per test*
  - DIT IS **NIET** CORRECT!
  - Met een unit test wordt één bepaald **"gedrag"** geverifieerd.
    Dit gedrag, kan **meerdere gevolgen** hebben die **allemaal** moeten geverifieerd worden.
    (zie voorbeeld Store en Inventory!)
- **Herbruikbaarheid:** indien je Arrange-secties in verschillende testen wilt hergebruiken, zet deze code dan **NIET** in de constructor van de klasse!

Unit Tests: richtlijnen

- Herbruikbaarheid: voorbeeld

```
[TestFixture]
class CustomerTests
 {
   private readonly Store _store;
   private readonly Customer _customer;

   public CustomerTests()
    {
      _store = new Store();
      _store.AddInventory(Product.Shampoo, 10);
      _customer = new Customer();
    }

    ...
}
```

```
[Test]
public void Purchase_succeeds_when_enough_inventory()
 {
    bool success = _customer.Purchase(_store, Product.Shampoo, 5);

    Assert.IsTrue(success);
    Assert.AreEqual(5, _store.GetInventory(Product.Shampoo));
 }

[Test]
public void Purchase_fails_when_not_enough_inventory()
 {
   bool success = _customer.Purchase(_store, Product.Shampoo, 15);

   Assert.IsFalse(success);
   Assert.AreEqual(10, _store.GetInventory(Product.Shampoo));
 }
```

Unit Tests: richtlijnen

- **Nadelen:**
  - **Hoge koppeling** tussen verschillende tests!
    - Bv.: wat gebeurt er indien je (per ongeluk) de initiële inventaris verhoogt naar 15?
  - Vermindert de **leesbaarheid**
    - Het "volledige plaatje" is niet meer zichtbaar

- **Aandachtspunten:**
  - Een wijziging aan de code van één test, mag geen invloed hebben op andere tests

    → vermijd gedeelde "state" (gebruik van instance-variabelen in tests!)

Unit Tests: richtlijnen

- Een betere oplossing: maak gebruik van **factory-methoden**:

```
private Store CreateStoreWithInventory(Product product, int quantity)
 {
   Store store = new Store();
   store.AddInventory(product, quantity);

   return store;
 }

private static Customer CreateCustomer()
 {
   return new Customer();
 }
```

Unit Tests: richtlijnen

- Gebruik van de **factory-methoden** in je tests:

```
[Test]
public void Purchase_succeeds_when_enough_inventory()
 {
   Store store = CreateStoreWithInventory(Product.Shampoo, 10);
   Customer sut = CreateCustomer();

   bool success = sut.Purchase(store, Product.Shampoo, 5);

   Assert.IsTrue(success);
   Assert.AreEqual(5, store.GetInventory(Product.Shampoo));
 }

[Test]
public void Purchase_fails_when_not_enough_inventory()
 {
   Store store = CreateStoreWithInventory(Product.Shampoo, 10);
   Customer sut = CreateCustomer();

   bool success = sut.Purchase(store, Product.Shampoo, 15);

   Assert.IsFalse(success);
   Assert.AreEqual(10, store.GetInventory(Product.Shampoo));
 }
```

NUnit: setup/teardown

- Soms kan het handig zijn om een aparte setup-methode te maken voor het uitvoeren van je testen. NUnit voorziet hiervoor de volgende **attributes**:
  - `OneTimeSetUp:` deze methode wordt **één keer** uitgevoerd, **vóór** het uitvoeren van de **tests** in deze klasse (bv.: opzetten van een databank, …)
  - `SetUp:` deze methode wordt **vóór elke test** binnen de klasse aangeroepen

- Denk eraan dat een unit test nóóit state mag achterlaten! NUnit voorziet tevens twee attributen waarmee je clean-up methoden kunt specificeren:
  - `OneTimeTearDown:` deze methode wordt **één keer** uitgevoerd, nadat **alle testen** uitgevoerd zijn (bv.: sluiten van database-connectie, opkuisen van tijdelijke bestanden, …)
  - `TearDown:` deze methode wordt uitgevoerd **ná elke test** binnen de klasse

NUnit: setup/teardown

- Voorbeeld **initialisatie** en **clean-up**:

```
[OneTimeSetUp]
public void ClassInit()
 {
    //wordt één keer uitgevoerd,
    //vóór het uitvoeren van de tests binnen deze klasse
 }

[SetUp]
public void TestInit()
 {
    //wordt uitgevoerd vóór het uitvoeren van elke test
 }
```

```
[TearDown]
public void TestCleanUp()
 {
    //wordt uitgevoerd ná het uitvoeren van elke test
 }

[OneTimeTearDown]
public void ClassCleanUp()
 {
    //wordt één keer uitgevoerd,
    //nadat alle testen binnen de klasse uitgevoerd werden
 }
```

NUnit: TestCase-attribuut

- In sommige gevallen, wil je eenzelfde unit test uitvoeren met verschillende testgegeven.
- *Voorbeeld: test een methode die nagaat of een persoon mag stemmen. De minimale leeftijd om te mogen stemmen is 18 jaar.*

```
[Test]
public void Person_can_vote_if_18_or_older()
 {
   Person sut = new Person();
   sut.Age = 18;

   bool canVote = sut.CanVote;

   Assert.IsTrue(canVote);
 }
```

Enkel 18 jaar, idem voor leeftijd > 18 (20, 25, 30, ...)

NUnit: TestCase-attribuut

- **Oplossing:** gebruik het ***TestCase***-attribuut
  - Voorzie **parameter(s)** in de header van je testmethode
  - Voeg voor elke waarde die je wilt testen een **TestCase-attribuut** toe, met als argument de **testwaarde**

```
[TestCase(18)]
[TestCase(20)]
[TestCase(25)]
[TestCase(30)]
public void Person_can_vote_if_18_or_older(int age)
 {
   Person sut = new Person();
   sut.Age = age;

   bool canVote = sut.CanVote;

   Assert.IsTrue(canVote);
 }
```

de test wordt vier keer uitgevoerd

NUnit: TestCase-attribuut

- De testcode voor het verifiëren of personen jonger dan 18 jaar niet mogen testen, is vrij gelijkaardig. Ook hier kunnen we gebruik maken van het ***TestCase*-attribuut** met een extra parameter: "ExpectedResult"
  - **Opgelet!** Schrijf geen ***Assert*** in de test zélf, maar **return** het **resultaat** van de test!

```
[TestCase(12, ExpectedResult = false)]
[TestCase(17, ExpectedResult = false)]
[TestCase(18, ExpectedResult = true)]
[TestCase(20, ExpectedResult = true)]
public bool Person_can_only_vote_if_18_or_older(int age)
 {
   Person sut = new Person();
   sut.Age = age;

   bool canVote = sut.CanVote;

   return canVote;
 }
```

Bronnen

- Khorikov, V. (2020). *Unit Testing Principles, Practices, and Patterns*. Shelter Island, NY: Manning Publications.
- Osherove, R. (2013). *The Art of Unit Testing*. Greenwich, CT: Manning Publications.