



Odissee
DE CO-HOGESCHOOL

Unit testing



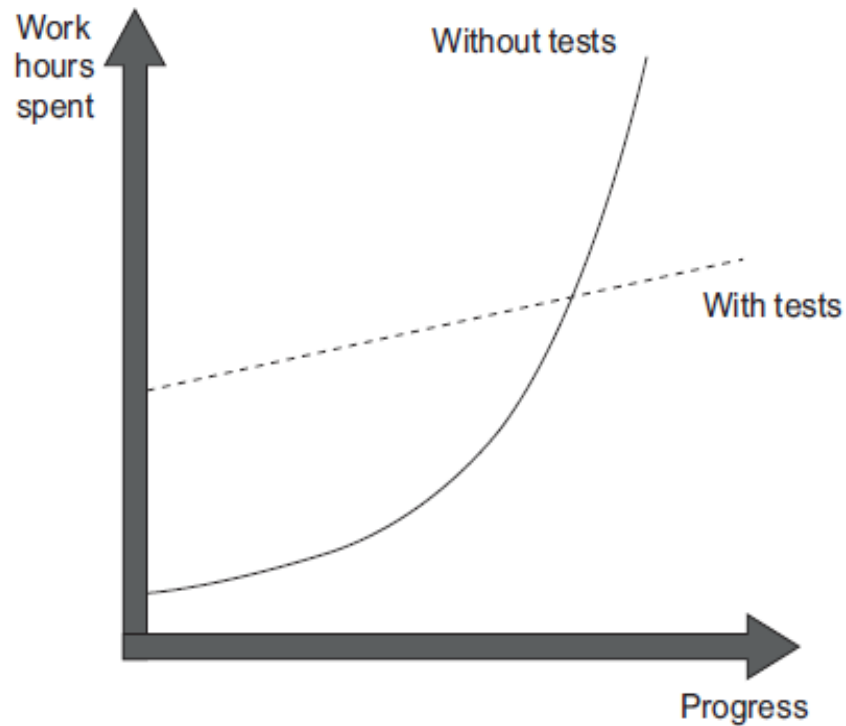
Jens Baetens

Wat is unit testing?

- 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. Vaak wordt met een “unit” 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.*

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, ...)

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

Doel van unit tests

- ▣ Schaalbaarheid verhogen
 - ▬ In het product maar ook in het aantal mensen die eraan werken
- ▣ Verbeteren van het design
 - ▬ Verminderen methode parameters
 - ▬ Voorkomen mega-methodes
 - ▬ Global State vermijden
 - ▬ Minder/geen afhankelijkheden -> single point of responsibility
 - ▬ Minder/geen zij-effecten
- ▣ Drempel om code aan te passen verlagen



Doel van unit tests

- ▣ Opleveren van een stabiele product
 - Unit tests leiden tot minder verrassingen of last-minute bugs
- ▣ Automatisch testen van de basis maakt tijd vrij voor complexere testing
- ▣ Vorm van code documentatie
 - Unit test beschrijft een bepaald gedrag in een bepaalde situatie
 - Unit tests komen overeen met de gewenste specificaties van het programma

Wat is een goede Unit test

Wat is een goede Unit test?

- ▣ Geautomatiseerd en herhaalbaar
- ▣ Laten geen state achter
- ▣ Eenvoudig te implementeren
- ▣ Beschikbaar voor later gebruik
- ▣ Iedereen kan ze uitvoeren
- ▣ Geen manuele interventie
- ▣ Snel
- ▣ Consistent



Wat is een goede Unit test?

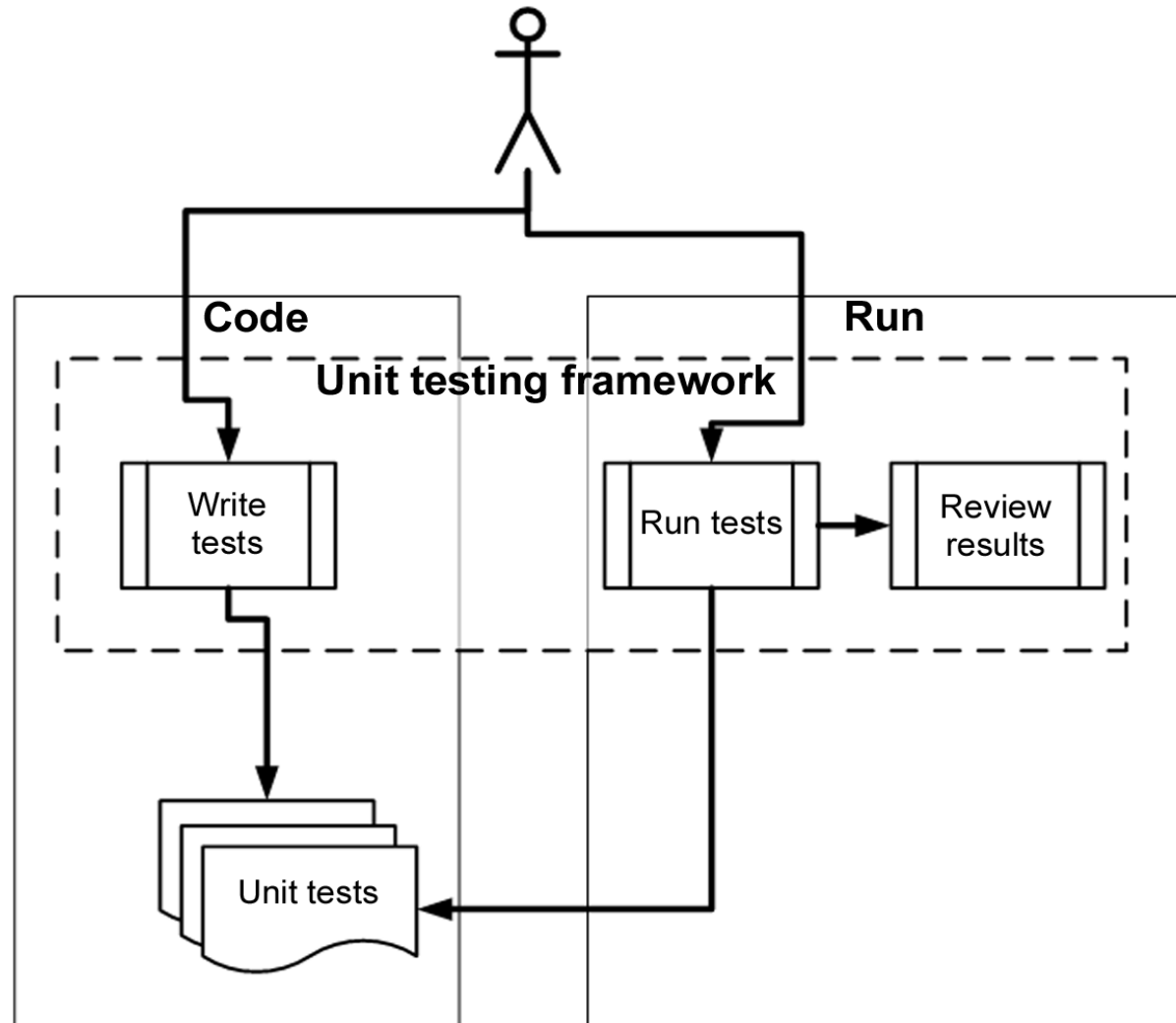
■ Geherformuleerde definitie

- Een unit test is een geautomatiseerd stukje code dat de methode/klasse aanroept die getest wordt en een aantal veronderstellingen (assumptions) 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 iteratiestructuur, switch-case, berekening, ...). Dus: géén properties/constructors/...

Testing framework

Testing framework





Testing framework

- ▣ Gestructureerde manier om tests te schrijven op basis van
 - Basis klassen of interfaces om van over te erven
 - Attributen om aan te geven welke code test-code is
 - Assertion-klassen om objecten/resultaten te verifiëren

Testing framework

▣ Test runner

- ▬ Automatisch uitvoeren van 1 of meerdere testen
- ▬ Stappen:
 - Zoek de uit te voeren testen in de code
 - Voer de testen uit (automatisch)
 - Hou de status bij tijdens het uitvoeren
 - Rapporteer de resultaten van de uitgevoerde testen
- ▬ Kan via command line gestart worden

Testing framework

▣ Resultaten van een test run

- Aantal tests uitgevoerd en niet uitgevoerd
- Hoeveel tests gefaald/geslaagd
- Welke tests gefaald
- Waarom faalden ze?
- Zelf voorziene output
- Waar faalden de tests
- Stack trace van exceptions waardoor de test faalde

Testing frameworks voor C#

▣ Meerdere mogelijkheden voor C#

- MS Test
- xUnit.Net
- NUnit

▣ Waarom NUnit?

- Zeer popular
- Open Source
- Goede integratie met Visual Studio
- Stabiel
- Goede documentatie





Hoe testen schrijven met NUnit

Creating project

Configure your new project

Console App (.NET Framework) C# Windows Console

Project name

Les1

Location

G:\My Drive\PTT\Leerstof\Lessen\Code\

Solution name ⓘ

Les1

☐ Place solution and project in the same location

Framework

.NET Framework 4.7.2

Microsoft Visual Studio

Creating project...

Cancel

Hoe testen schrijven?

- Zet je testen steeds in een apart VS Project!
- Stappen:
 - Maak een nieuw project aan: Class Library (.Net Framework)
 - Rechtsklik op het nieuwe project en kies Manage NuGet Packages...
 - Installeer de volgende packages
 - NUnit
 - NUnit Test Adapter
 - Microsoft.NET.Test.Sdk

Solution Explorer

Search Solution Explorer (Ctrl+;)

Solution 'Les1' (2 of 2 projects)

- Les1
 - Properties
 - References
 - App.config
 - Program.cs

Build

Rebuild

Clean

Analyze and Code Cleanup

Pack

Publish...

Scope to This

Change View To

New Solution Explorer View

Edit Project File

Build Dependencies

Add

Manage NuGet Packages...

Manage User Secrets

Remove Unused References...

Sync Namespaces

Set as Startup Project

Debug

Git

Cut Ctrl+X



Remove Del

Rename F2

Unload Project

Browse Installed Updates

NUit x ↻ ☐ Include prerelease

	NUit ✓ by Charlie Poole, Rob Prouse, 155M downloads NUit is a unit-testing framework for all .NET languages with a strong TDD focus.	3.13.2
	NUit3TestAdapter ✓ by Charlie Poole, Terje Sandstrom, 98.2M downloads NUit3 adapter for running tests in Visual Studio and DotNet. Works with NUit 3.x, use the NUit 2 adapter for 2.x tests.	4.2.0

Link leggen tussen het project en het testproject

- ▣ Rechtsklik op je test project
- ▣ Add -> Referene
- ▣ Selecteer Projects en klik op het project dat getest wordt

- New Item... Ctrl+Shift+A
- Existing Item... Shift+Alt+A
- New Folder
- Machine Learning Model...
- Project Reference...**
- Shared Project Reference...
- COM Reference...
- Service Reference...
- Connected Service
- Class...
- New EditorConfig

- Show in Test Explorer
- Scope to This
- Change View To ▶
- New Solution Explorer View
- Edit Project File
- Build Dependencies ▶
- Add ▶**
- Manage NuGet Packages...
- Manage User Secrets
- Remove Unused References...
- Sync Namespaces
- Set as Startup Project
- Debug ▶
- Git ▶
- Cut Ctrl+X
- Remove Del
- Rename F2
- Unload Project

Reference Manager - Les1Tests

▲ Projects


Solution

- ▶ Shared Projects
- ▶ COM
- ▶ Browse

	Name	Path
<input checked="" type="checkbox"/>	Les1	G:\My Drive\PTT\Leers...



Eerste unit test

- 
- ▣ Maak een classe aan **Calculator**
 - ▣ Maak een functie aan in deze classe:
 - Naam is **Sum**
 - Twee parameters: firstNumber en secondNumber
 - Returned de som van de twee getallen
 - ▣ Let erop dat de klasse **public** is want anders wordt de klasse niet gevonden



Test nu deze functie

```
[TestFixture]
0 references
public class CalculatorTests
{
    [Test]
    0 references
    public void Sum_GivenTwoPositiveIntegers_ReturnCorrectSum()
    {
        // Arrange
        Calculator sut = new Calculator();
        int number = 10;
        int number2 = 20;

        // Act
        int result = sut.Sum(number, number2);

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

▣ TestFixture

- Geeft aan dat de klasse NUnit testen bevat

▣ Test

- Deze methode is een test

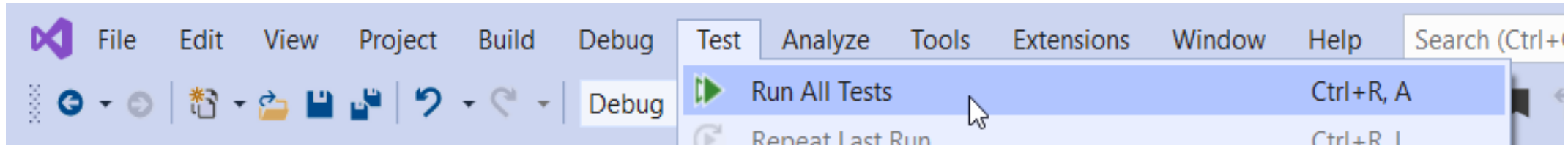
▣ Assert

- Functionaliteiten om resultaten te vergelijken

Uitvoeren van de test

▣ Alle testen uitvoeren:

- Rechtsklik op het testproject en selecteer run all tests
- Of via de menubalk bovenaan



▣ Specifieke tests uitvoeren

- Rechtsklik op het testproject en klik op run tests

Testresultaten

- Getoond in de Test Explorer venster
 - ▬ Aan de linkerkant zie je welke testen uitgevoerd werden, of ze geslaagd zijn en de runtime
 - ▬ Rechts zie je een samenvatting van de uitgevoerde tests

Test Explorer

Search Test Explorer (Ctrl+E)

Test	Duration	Traits	Error Message
✓ Les1Tests (1)	14 ms		
✓ Les1Tests (1)	14 ms		
✓ CalculatorTests (1)	14 ms		
✓ Sum_GivenTwoPositiveIntegers_...	14 ms		

Test Detail Summary

- ✓ Sum_GivenTwoPositiveIntegers_ReturnCorrectSum
 - Source: [CalculatorTests.cs](#) line 15
 - Duration: 14 ms

- ▣ Vinkje is geslaagd, rood kruisje is gefaald

Test Explorer

2 1 1

Search Test Explorer (Ctrl+E)

Test	Duration	Traits	Error Message
Les1Tests (2)	119 ms		
Les1Tests (2)	119 ms		
CalculatorTests (2)	119 ms		
Sum_GivenOnePositiveOneNeg...	119 ms		Expected: 10 But was: 30
Sum_GivenTwoPositiveIntegers_...	< 1 ms		

Group Summary

Les1Tests

Tests in group: 2

⌚ Total Duration: 119 ms

Outcomes

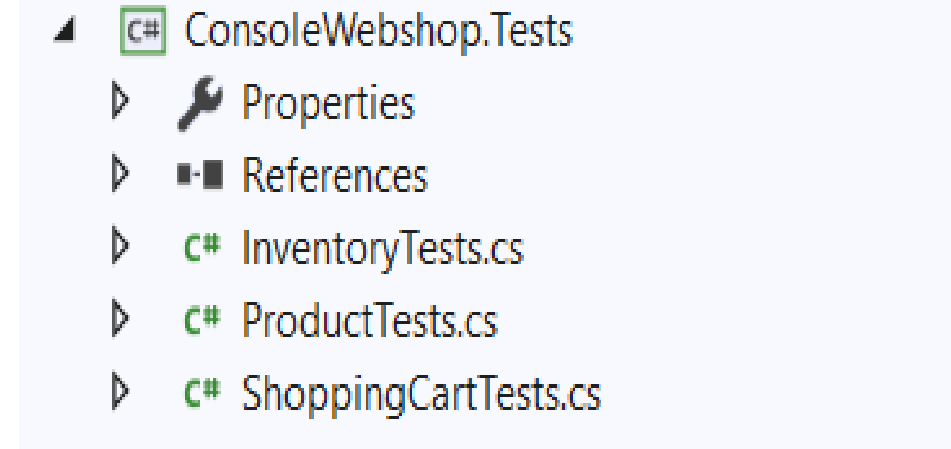
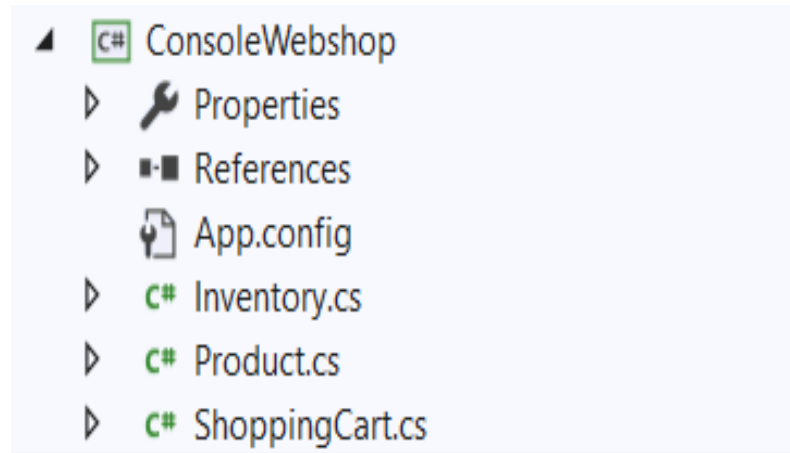
✓ 1 Passed

✗ 1 Failed

Aandachtspunten

- ▣ Je kan enkel **public** classen testen
- ▣ Ook je testmethoden moeten **public** zijn
- ▣ Streef naar een test-project voor elk code-project
- ▣ Naamgeving!
 - Code-project Calculator -> Testproject: Calculator.Tests
 - Klasse Calculator -> Testklasse: CalculatorTests
 - Ook van je functies

Naamgeving



Naamgeving

- ▣ Ook de naamgeving van je test-methoden is belangrijk
- ▣ Beschrijft wat er getest wordt: belangrijk als documentatie
- ▣ Structuur:
 - Naam de van de methode die getest wordt
 - Scenario dat er getest wordt
 - Verwachte output

Naamgeving

▣ Slechte naamgeving

- Test1(), Test2(), Test3(), ...
- TestSum(), TestInventory(), TestProduct(), ...

▣ Goede naamgeving

- Sum_TwoPositiveNumber_ReturnsCorrectSum()
- Divide_DenominatorIsZero_ExceptionThrown()
- ...

Structuur van de test

■ AAA-patroon

- Arrange
 - Opzetten beginsituatie
- Act
 - Voer de functie uit
- Assert
 - Controleer de resultaten

■ Given – When – Then

```
[TestFixture]
0 references
public class CalculatorTests
{
    [Test]
    0 references
    public void Sum_GivenTwoPositiveIntegers_ReturnCorrectSum()
    {
        // Arrange
        Calculator sut = new Calculator();
        int number = 10;
        int number2 = 20;

        // Act
        int result = sut.Sum(number, number2);

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

Assert

▣ 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);

■ Constraint syntax

Methode(s)	Gebruik	Voorbeeld
Is.EqualTo()	Gaat na of twee waarden aan elkaar gelijk zijn	<code>Assert.That(total, Is.EqualTo(5));</code>
Is.True/Is.False	Gaat na of een waarde true of false is	<code>Assert.That(valid, Is.True);</code>
Is.Null/Is.Not.Null	Gaat na of een referentie een null-reference is	<code>Assert.That(customer, Is.Null);</code>
Is.GreaterThan()/Is.GreaterOrEqualThan()/Is.LessThan()/Is.LessOrEqualThan()	Gaat na of een waarde groter dan (of gelijk aan)/kleiner dan (of gelijk aan) een andere waarde is	<code>Assert.That(customer.Age, Is.GreaterThan(18));</code>
Is.SameAs()/Is.Not.SameAs()	Gaat na of twee referenties naar het zelfde object verwijzen (reference-equality)	<code>Assert.That(cust1, Is.SameAs(cust2));</code>

Oefening

- ▣ Schrijf een unit test dat de deling van twee integers berekend
 - Bijvoorbeeld 12 gedeeld door 3
- ▣ Na het schrijven van de test
 - Controleer of de test faalt
- ▣ Schrijf nu de code om de test te doen slagen

- ▣ Indien gelukt: pas de code aan zodat het het dichtsbijge geheel getal returned ipv altijd naar beneden af te ronden.
 - Faalt je test nu? Indien nee, schrijf een test die wel faalt

Ander voorbeeld

- ▣ Context: We maken een programma voor het beheer van de inventaris van een webshop
- ▣ Test Case: Aankoop lukt indien er voldoende voorraad is.
 - Pre-condities: 10 flessen shampoo in voorraad
 - Stappen: klant koop 5 flessen
 - Resultaat: Aankoop is gelukt en voorraad met 5 verlaagd

Schrijf een test hiervoor

▣ Volgende klassen

▸ Store

- Met een functie AddInventory om een aantal van een product toe te voegen aan de voorraad
- Met een functie GetInventory om de voorraad van een product te controleren

▸ Customer

- Purchase functie met als argumenten een store, een product en een hoeveelheid

▸ Product

- Enum met de verschillende types

[Test]

0 | 0 references

```
public void Purchase_WithEnoughInventory_ReturnsTrueStockChanged()
{
    //Arrange
    store = new Store();
    store.AddInventory(Product.Shampoo, 10);
    sut = new Customer();
    Dictionary<Product, int> expectedBasket = new Dictionary<Product, int>() {
        { Product.Shampoo, 5 }
    };

    //Act
    bool result = sut.Purchase(store, Product.Shampoo, 5);

    //Assert
    Assert.IsTrue(result);
    Assert.That(store.GetInventory(Product.Shampoo), Is.EqualTo(5));
    Assert.That(sut.Basket, Is.EqualTo( expectedBasket));
}
```



Richtlijnen

Richtlijnen

- ▣ Vermijd meerdere Arrange, Act, Assert secties
 - Indicatie dat je teveel test in 1 test -> Splitsen van de test
- ▣ Vermijd if-statements in je test
 - Geen logica in test
 - Splits indien nodig

Richtlijnen

- Opgelet voor act-secties langer dan 1 lijn

```
[Test]
0 | 0 references
public void Purchase_WithEnoughInventory_ReturnsTrueStockChanged()
{
    //Arrange
    store = new Store();
    store.AddInventory(Product.Shampoo, 10);
    sut = new Customer();
    Dictionary<Product, int> expectedBasket = new Dictionary<Product, int>() {
        { Product.Shampoo, 5 }
    };

    //Act
    bool result = sut.Purchase(store, Product.Shampoo, 5);
    store.RemoveInventory(Product.Shampoo, 5);

    //Assert
    Assert.IsTrue(result);
    Assert.That(store.GetInventory(Product.Shampoo), Is.EqualTo(5));
    Assert.That(sut.Basket, Is.EqualTo(expectedBasket));
}
```

Richtlijnen

- ▣ Indien men later in de code de tweede lijn vergeet, is er een inconsistentie
 - Twee lijn = twee gevolgen
 - Moeten gelijktijdig zijn dus in 1 methode = encapsulatie
 - Oplossing: voer de tweede lijn in de purchase methode toe

Richtlijnen

- ▣ Met een unit-test wordt 1 bepaald gedrag geverifieerd
 - ▬ Meerdere gevolgen mogelijk
 - ▬ Alle gevolgen geverifieerd
- ▣ Herbruikbaarheid
 - ▬ Indien je Arrange secties in verschillende testen wilt hergebruiken: zet deze dan niet in de constructor van de klasse -> maar 1 keer gedaan



Richtlijnen

▣ Herbruikbaarheid voorbeeld

```
private readonly Store store;  
private readonly Customer sut;
```

```
0 references  
public CustomerTests()  
{  
    store = new Store();  
    store.AddInventory(Product.Shampoo, 10);  
    sut = new Customer();  
}
```

```
[Test]  
0 | 0 references  
public void Purchase_WithEnoughInventory_Succeeds()  
{  
    //Arrange  
    Dictionary<Product, int> expectedBasket = new Dictionary<Product, int>() {  
        { Product.Shampoo, 5 }  
    };  
  
    //Act  
    bool result = sut.Purchase(store, Product.Shampoo, 5);  
  
    //Assert  
    Assert.IsTrue(result);  
    Assert.That(store.GetInventory(Product.Shampoo), Is.EqualTo(5));  
    Assert.That(sut.Basket, Is.EqualTo( expectedBasket));  
}
```

```
[Test]  
0 | 0 references  
public void Purchase_WithoutEnoughInventory_Fails()  
{  
    //Arrange  
  
    //Act  
    bool result = sut.Purchase(store, Product.Shampoo, 15);  
  
    //Assert  
    Assert.IsTrue(false);  
    Assert.That(store.GetInventory(Product.Shampoo), Is.EqualTo(10));  
    Assert.IsFalse(sut.Basket.ContainsKey(Product.Shampoo));  
}
```

Richtlijnen

▣ Nadelen:

- Hoge koppeling tussen verschillende test
 - Code van de ene test beïnvloedt de beginsituatie van een andere test
- Verminderd de leesbaarheid
 - Je moet alle testen lezen ipv 1 test

▣ Aandachtspunten

- Een wijziging aan een test mag **geen** invloed hebben op een andere test
- Vermijd gedeelde state



Oplossing herbruikbaarheid

▣ Factory methoden

0 references

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

    return store;
}
```

0 references

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

Gebruik factory methods

```
[Test]
0 | 0 references
public void Purchase_WithEnoughInventory_Succeeds()
{
    //Arrange
    store = createStoreWithInventory(Product.Shampoo, 10);
    sut = createCustomer();
    Dictionary<Product, int> expectedBasket = new Dictionary<Product, int>() {
        { Product.Shampoo, 5 }
    };

    //Act
    bool result = sut.Purchase(store, Product.Shampoo, 5);

    //Assert
    Assert.IsTrue(result);
    Assert.That(store.GetInventory(Product.Shampoo), Is.EqualTo(5));
    Assert.That(sut.Basket, Is.EqualTo( expectedBasket));
}
```


NUnit: Setup en Teardown

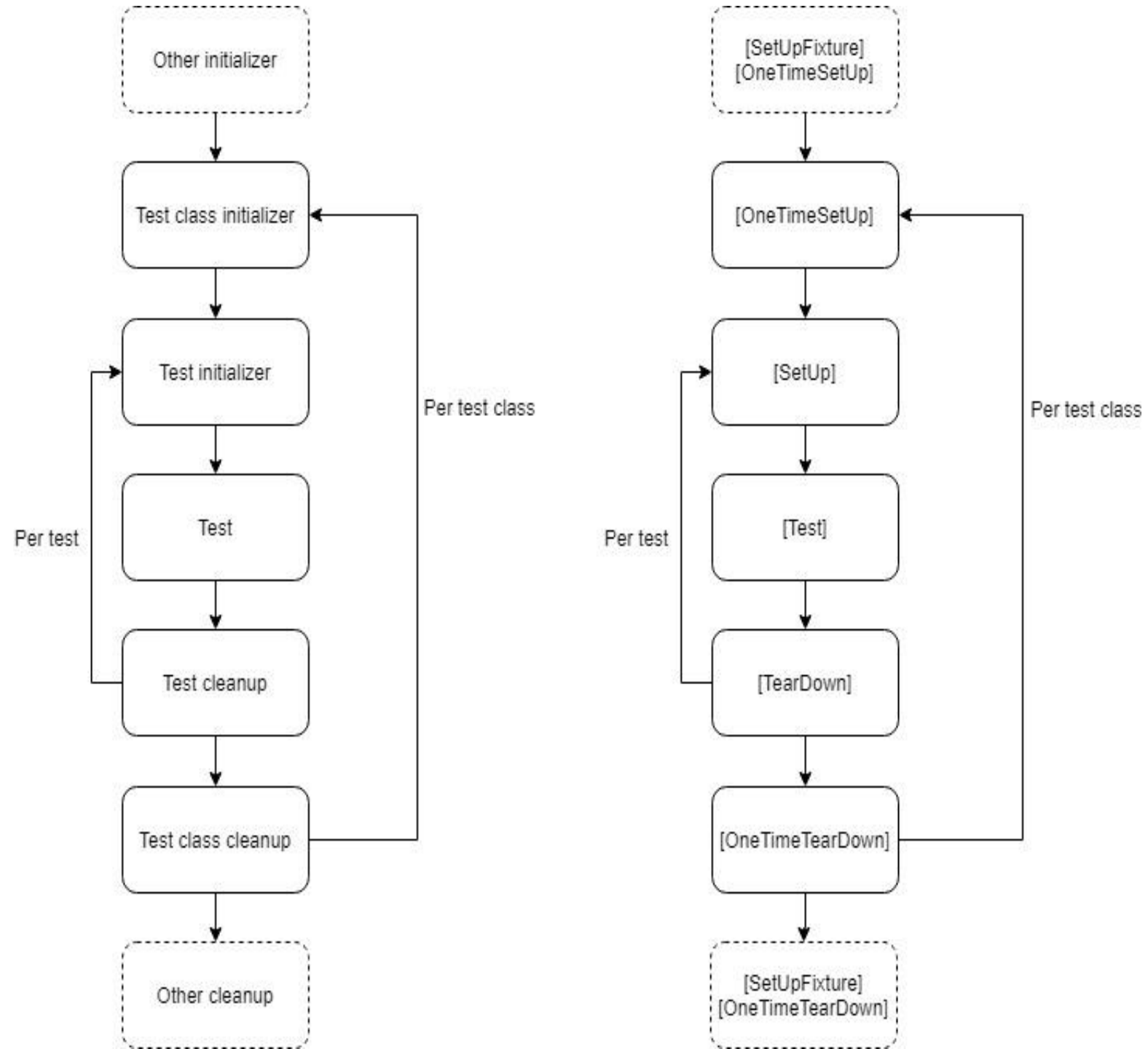
- ▣ Aparte setup methode kan handig zijn
 - ▬ [OneTimeSetUp]: 1 keer uitgevoerd, voor alle testen in deze klasse
 - Opzetten databank, ...
 - ▬ [SetUp]: voor elke test opgeroepen
- ▣ Een test mag geen state achterlaten dus ook teardown methoden voor opkuis
 - ▬ [OneTimeTearDown]: 1 keer uitgevoerd na alle testen
 - Sluiten databank connective, verwijderen tijdelijke bestanden, ...
 - ▬ [TearDown]: Uitgevoerd na elke test binnen de klasse
- ▣ Vaak is het gebruik van TearDown echter een teken dat je iets aanspreekt dat niet de bedoeling is

Setup/Teardown voorbeeld

```
1  [OneTimeSetUp]
2  public void ClassInit()
3  {
4      //wordt één keer uitgevoerd,
5      //vóór het uitvoeren van de tests binnen deze klasse
6  }
7
8  [SetUp]
9  public void TestInit()
10 {
11     //wordt uitgevoerd vóór het uitvoeren van elke test
12 }
13
```

```
1  [TearDown]
2  public void TestCleanUp()
3  {
4      //wordt uitgevoerd ná het uitvoeren van elke test
5  }
6
7  [OneTimeTearDown]
8  public void ClassCleanUp()
9  {
10     //wordt één keer uitgevoerd,
11     //nadat alle testen binnen de klasse uitgevoerd werden
12 }
13
```

Test Life Cycle



Setup

- ▣ De setup klasse is ook een alternatief voor het veelvuldig gebruikmaken van een factory klasse

```
[SetUp]
```

```
0 references
```

```
public void Initialize()
{
    store = new Store();
    store.AddInventory(Product.Shampoo, 10);
    sut = new Customer();
}
```

TestCase

- ▣ Dezelfde Test uitvoeren met verschillende data
- ▣ Bijvoorbeeld: test een methode die nagaat of een persoon mag stemmen.
 - ▬ Minimale leeftijd hiervoor is 18 jaar.

```
[Test]
0 | 0 references
public void CanVote_PersonIsAtLeast18_AllowedToVote()
{
    //Arrange
    Person person = new Person();
    person.Age = 18;

    //Act
    bool result = person.CanVote();

    //Assert
    Assert.IsTrue(result);
}
```

TestCase

▣ TestCase attribuut

- Parameter in de header
- TestCase met argument de te testen leeftijden

```
[TestCase(18)]  
[TestCase(20)]  
[TestCase(100)]  
0 | 0 references  
public void CanVote_PersonIsAtLeast18_AllowedToVote(int age)  
{  
    //Arrange  
    Person person = new Person();  
    person.Age = age;  
  
    //Act  
    bool result = person.CanVote();  
  
    //Assert  
    Assert.That(result, Is.EqualTo(true));  
}
```

TestCase

- Ook de output kan meegegeven worden
 - Extra parameter: *ExpectedResult*
 - *Geen assert in de test zelf maar return het resultaat van de test*
 - *Let ook op het returntype in de definitie van de methode*

```
[TestCase(18, ExpectedResult = true)]  
[TestCase(20, ExpectedResult = true)]  
[TestCase(100, ExpectedResult = true)]  
[TestCase(8, ExpectedResult = false)]  
[TestCase(17, ExpectedResult = false)]  
  
0 references  
public bool CanVote_PersonIsOfVariousAges_AllowedToVoteIfAtLeast18(int age)  
{  
    //Arrange  
    Person person = new Person();  
    person.Age = age;  
  
    //Act  
    bool result = person.CanVote();  
  
    //Assert  
    return result;  
}
```



Tips

Tips voor betere tests

1. Denk aan documentatie *→ leesbaarheid v.d. code*
2. Isoleer je tests *→ Naomgeving*
3. Hou het vlak / geen logica *(geen ifs)*
4. Mock wat je niet kan controleren (zien we later)
5. Vermijd for-lussen *(in de assert)*
6. Test de applicatie alsof je een gebruiker bent
7. Vermijd implementatie details (zien we later)

Tips voor betere tests

8. Verkies integratie testen (hier nog niet aan de orde)
9. Altijd alles groen voor mergen → *Testen voor code toe te voegen*
 - Kan afgedwongen worden door CI systemen zoals gitlab
10. Schrijf testen voor vertrouwen en niet voor metrieken
 - Code coverage is een indicatie geen doel