Unit Testing in C#

Inhoud

- Basis van unit testing
- NUnit framework en vergelijking met xUnit
- Test Doubles
- Manuele Test Doubles
- Test Doubles met een mocking framework (NSubstitute)
- TDD (Test Driven Development)
- Beste praktijken bij het schrijven van Unit Tests

Unit Tests

- Hoe effectieve unit tests schrijven in C# met een framework zoals bv Nunit en xUnit
  (gebruik van mocking en andere technieken)

Waarom Unit Testing?

- Kwaliteitsgarantie van de code
- Om betrouwbare en onderhoudbare software te schrijven
- Wordt verwacht van de doorsnee programmeur

Inleiding Unit Test

1. Wat is een unit test
2. Welke soorten unit test frameworks bestaan er
3. Hoe schrijf je een unit test
4. Naamconventies
5. Hoe run en debug je unit tests
6. Wat zijn de voordelen bij het schrijven van unit tests
7. Wie schrijft unit tests en wanneer

Wat is een Unit Test?

Een **unit test is een functie/methode** die een andere functie/methode (unit) aanroept en het resultaat van deze aangeroepen functie controleert op juistheid.

Wat is een SUT?

Een **“systeem onder test” (SUT)** is een instantie van een object die de units bevat die zullen worden getest.

Wat is Integratie Testing?

Integratie testing is het testen van een werk-eenheid, **zonder dat er volledige controle is over al zijn afhankelijkheden.**

Door deze te testen met het behoud van:
één of meer van zijn afhankelijkheden
bv tijd, network, database, threads,…

Unit Testing Frameworks

- MSTest
- NUnit
- xUnit.NET

Wat is Integratie Testing?

Er is **weinig verschil tussen** unit testing frameworks.

Ze werken op gelijkaardige wijze: zie cheatsheets Nunit en xUnit

Unit Testing Frameworks

- NUnit is het oudste framework
- MSTest is geIntegreerd in VS.
- xUnit.net is tegenwoordig popular en er is online veel documentatie en tutorials over te vinden

Naamconventies

- Naamgeving van test projecten bij conventie: **[ProjectUnderTest].Tests**
  Bijvoorbeeld: TransactionModel.Tests.
- Naamgeving van Test classen bij conventie: [ClassNameUnderTest]Tests
  Bijvoorbeeld: **CalculatorTests** of **CustomerTests**.

Naamgevingsconventies

- “void **ShouldAddTwoNumbers**()”
- “void **Sum_ShouldAddTwoNumbers**”

Naamgevingsconventies

Naamgevingsconventie: “UnitUnderTest_Scenario_ExpectedOutcome”.

Bijvoorbeeld: “void **ParsePort_COM1_Returns1**()”.

Naamgevingsconventies

1. “**ReturnsValue**”
   Bijvoorbeeld: “ReturnsZero” of “ReturnsFalse.”
2. “**ChangesState**” of “**SetState**”
   Bijvoorbeeld: “SetNumberToZero”.
3. “**CallsDependency**”
   Bijvoorbeeld: “CallsProcessingGateway.”

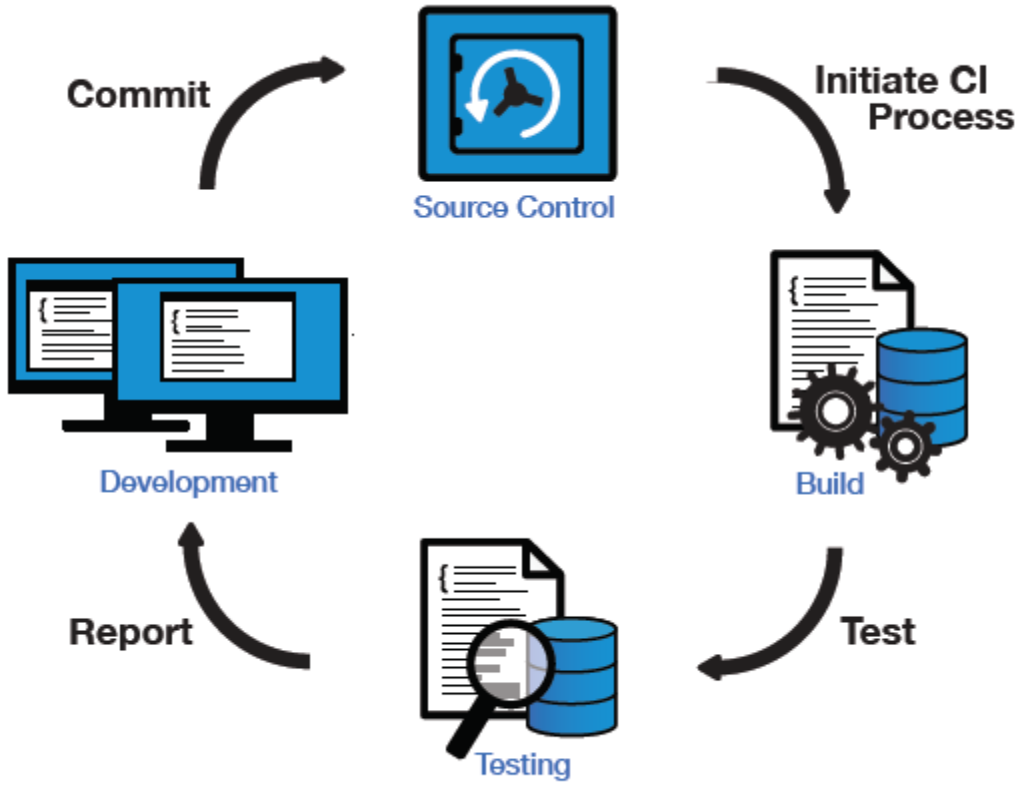
Voordelen van unit tests

- Betrouwbaarheid: zekerheid dat de geteste code correct werkt
- Bescherming tegeven “slechte” weizigingen
- Echte documentatie van hoe een system werkt

Voordelen

- Productiever development team:
  - Meer tijd om op nieuwe features te werken
  - Mogelijkheid om code te wijzigen en te refactoren zonder risico dat er iets ‘gebroken’ wordt
  - developers worden ‘s nachts niet meer opgebeld
- Complexe systemen met ingebouwde tests zijn transparanter (tests zijn een soort documentatie)
- Minder problemen wanneer de software in productie staat
- Verlaagt algemeen de bedrijfskosten

Werkwijze


Commit
Source Control
Initiate CI Process
Development
Build
Report
Testing
Test

Overwegingen bij Unit Testing

1. Testers zijn meestal enkel verantwoordelijk voor het schrijven en uitvoeren van manuele testen
2. Unit testen zijn veel sneller dan manuele testen, er is onmiddelijke feedback
3. Unit Tests creëren een veiligheidsnet en er kunnen zonder risico wijzigingen in de code worden doorgevoerd
4. Schrijf geen unit tests voor “Hello World” apps
5. Unit Testen schrijven is één van de taken van een software ontwikkelaar

Nunit test framework

Inhoud

1. ‘**A**rrange-**A**ct-**A**ssert’ pattern bij Unit tests
2. Een Nunit Test project aanmaken in Visual Studio
3. Assertions schrijven in NUnit
4. Hoe kan je unit tests laten uitvoeren
5. Gebruik van **SetUp** en **TearDown** attributen bij NUnit
6. Parametrisering van unittesten
7. Groeperen en negeren van bepaalde unit testen
8. Berekening van de unit-testing code-dekking en het belang hiervan

Arrange-Act-Assert patroon (AAA)

Bij Unit tests word het AAA-patroon toegepast

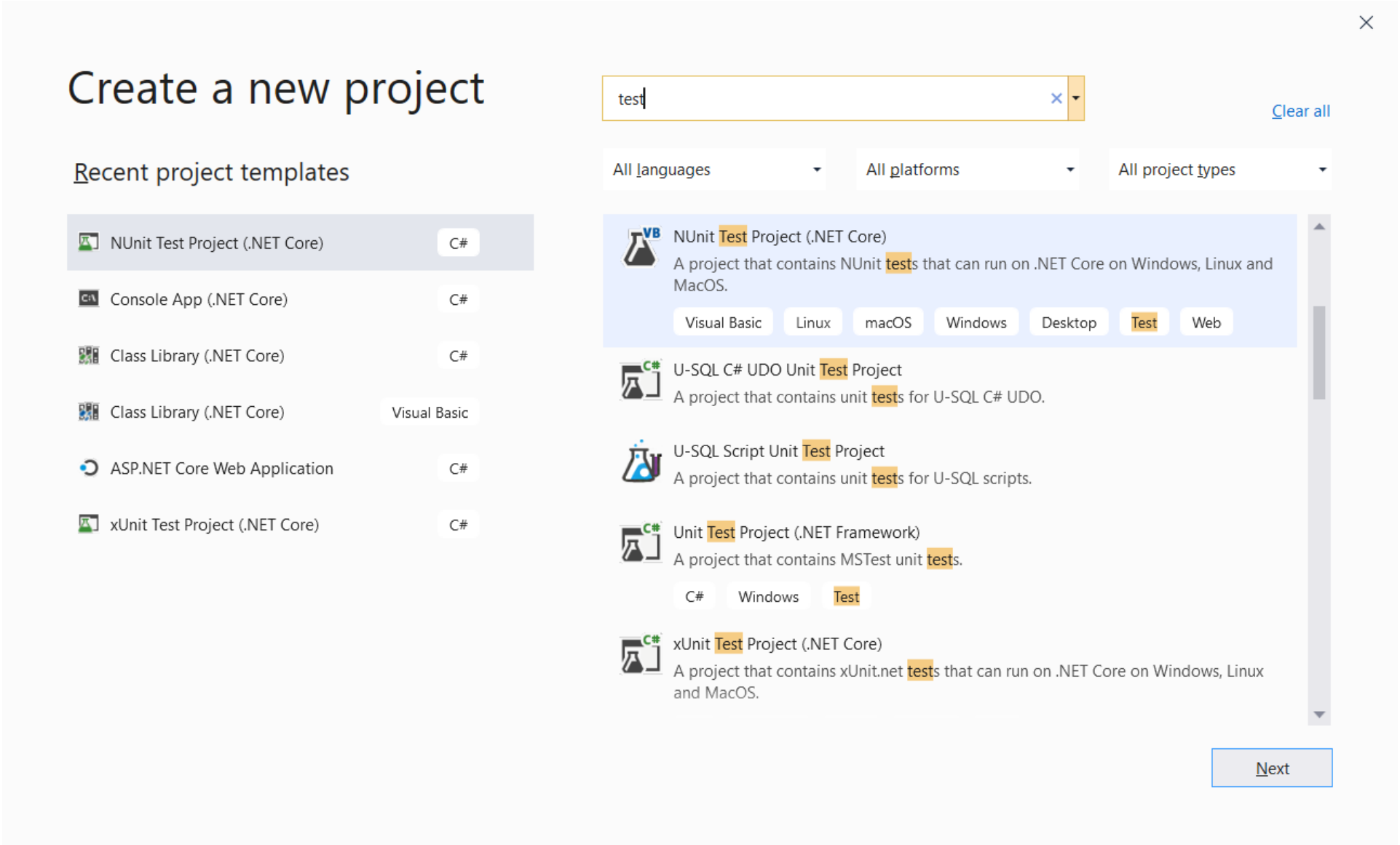
1. **Arrange**: Initialisatie van de nodige objecten en variabelen voor de unit test
2. **Act**: Aanroep van de methode die moet worden getest, waarbij de nodige parameters worden meegegeven.
3. **Assert**: Verificatie van de verwachte resultaten

Voorbeeld Arrange-Act-Assert patroon

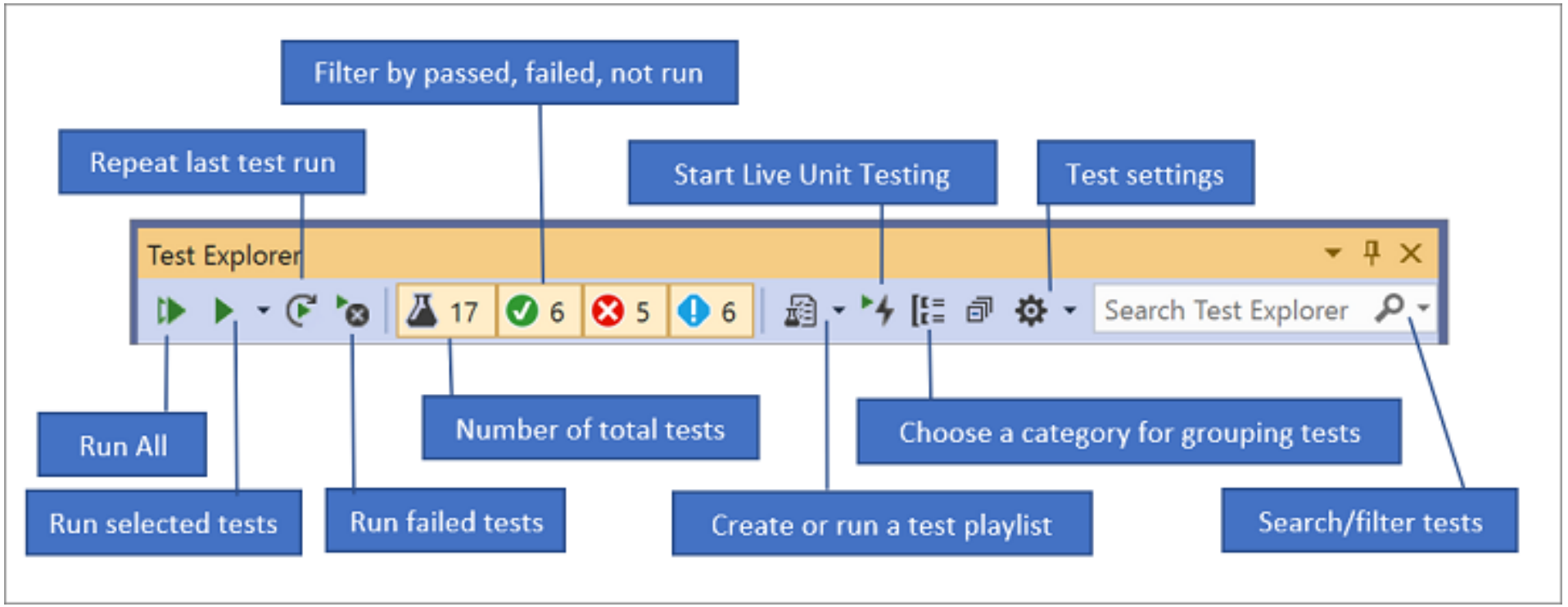
```
[Test]
public void IsDead_KillCharacter_ReturnsTrue()
{
    //arrange
    Character c = new Character(Type.Elf);
    //act
    c.Damage(500);
    //assert
    Assert.That(c.IsDead, Is.True);
}
```

“AAA” of “3A” patroon

Aanmaken van Test project in Visual Studio 2019



Test Explorer - Test project in Visual Studio 2019



Bron: https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2019

Assertion Model bij Unit Test

- **Assertion Model** – is de kern van elk unit testing framework
- Assertions bij Unit Testing zijn Boolean expressies die false teruggeven indien de test gefaald is, en true indien de test succesvol is.
- NUnit voorziet hiervoor een **constrained-based** model:

```
Assert.That<T>(T actual, IResolveConstraint expression);
```

https://www.tutorialspoint.com/software_testing_dictionary/assertion_testing.htm

Soorten Constraints

Logische constraints:

- Is
- Has
- Does

Soorten Constraints

2nd niveau constraints:

- All
- Not
- Some

Assertions en Contraints

“All” constraint

```
string[] array = new string[] { "abc", "bad", "dba" };
Assert.That(array, Is.All.Contains("b"));

int[] array = new int[] { 1, 2, 3, 4, 5 };
Assert.That(array, Has.All.GreaterThan(0));
```

Assertions en Contraints

“Not” constraint:

```
Assert.That(array, Is.Not.Length.EqualTo(4));
Assert.That(@"C:\tmp.txt", Does.Not.Exist);
Assert.That(42, Is.Not.Null);
Assert.That(42, Is.Not.True);
Assert.That(42, Is.Not.False);
Assert.That(2.5, Is.Not.NaN);
Assert.That(2 + 2, Is.Not.EqualTo(3));
```

Assertions en Contraints

“Does” constraint:

```
string phrase = "Are you OK?";
Assert.That(phrase, Does.EndWith("!"));
Assert.That(phrase, Does.Not.EndWith("?"));
Assert.That(phrase, Does.Not.Contain("goodbye"));
```

Assertions en Contraints

“Has” constraint:

```
object[] strings = new object[] { "abc", "bad", "cab", "bad", "dad" };
Assert.That(strings, Has.Some.StartsWith("ba"));

object[] doubles = new object[] { 0.99, 2.1, 3.0, 4.05 };
Assert.That(doubles, Has.Some.EqualTo(1.0).Within(.05));
```

Assertions en Contraints

“Or & And” compound constraints:

```
Assert.That(5, Is.LessThan(1).Or.GreaterThan(10));
Assert.That(5, Is.LessThan(10).And.GreaterThan(1));
```

Demo Unit test voor SerialPortParser class

Oefening Unit test voor DegreeConverter class

1. Maak een nieuw Visual Studio Class Library (.Net Core) project met naam Business

Maak in het project Business de class DegreeConverter met deze 2 methodes:

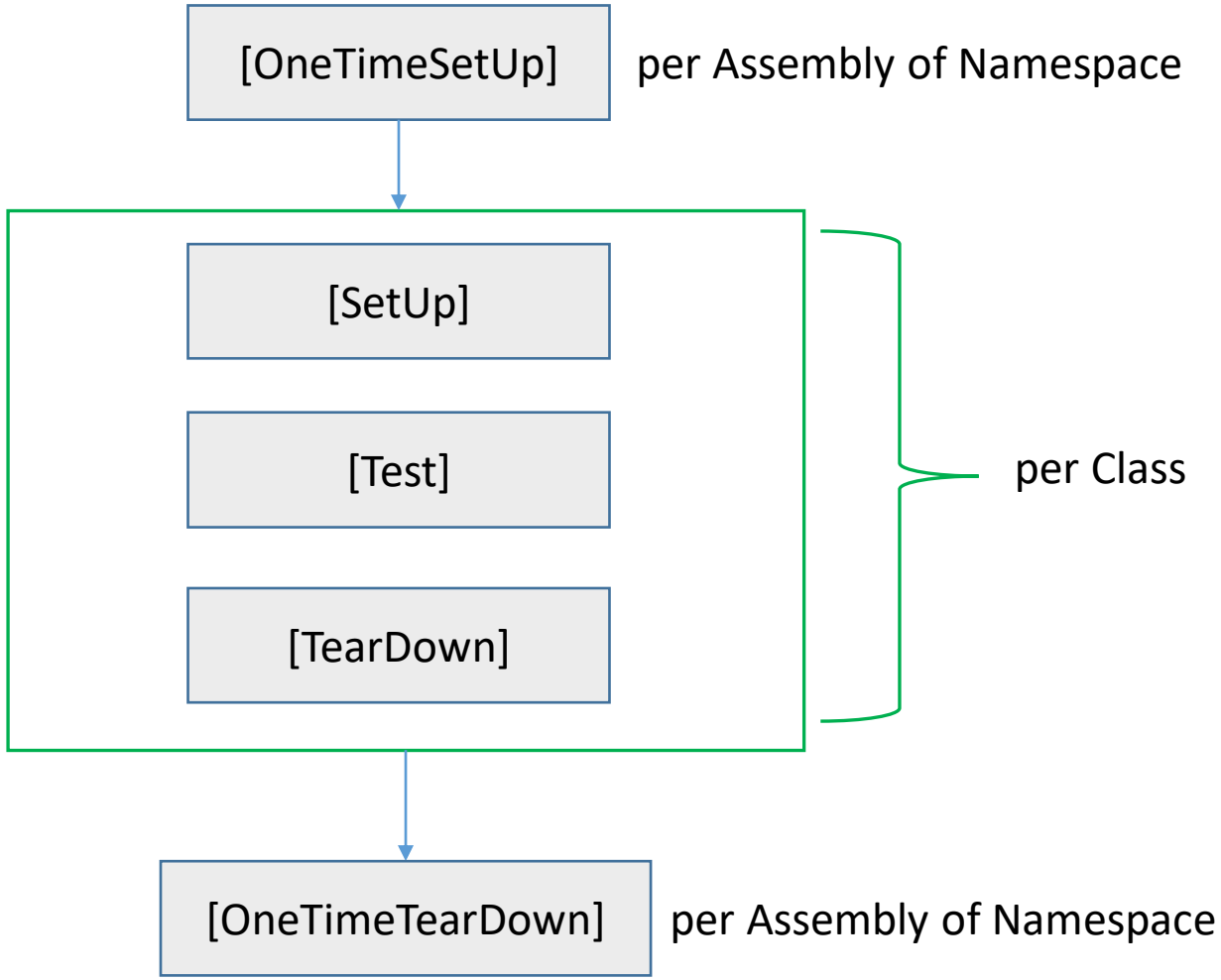
```
namespace Business{
    public class DegreeConverter    {
        public double ToFahrenheit(double celcius)        {
            return (celcius * 9 / 5);
        }

        public double ToCelsius(double fahrenheit)        {
            return (32 * fahrenheit - 32) * 5 / 9;
        }

    }
}
```

2. maak een Business.Tests Nunit Test project in dezelfde solution
3. Voeg een Test class toe met naam DegreeConverterTests
4. Voeg een methode void ToFahrenheit_ZeroCelcius_Returns32(): Deze controleert of dat ToFahrenheit(0) methode van DegreeConverter de waarde 32 teruggeeft
5. Voeg een methode void ToCelcius_1Fahrenheit_Returns0(): Deze controleert of dat ToCelcius(1) methode van DegreeConverter de waarde 0 teruggeeft

verloop uitvoering van unit test



Overzicht unit test mogelijkheden

Open CharacterTests class in de Business.Tests project
En neem de Nunit Cheat_cheat.pdf erbij
We overlopen de Test voorbeelden

Open daarna de Business.Tests.SetupAndTearDown class
Je kan hier de attributen vinden:
[SetUp]
[TearDown]
En een voorbeeld van geparametriseerde test met deze attributen:
[TestCase(100, 45)]
[TestCase(80, 65)]

Geparametriseerde unit test

geparametriseerde test kan gebeuren met [TestCase] attribuut
bv

```
[TestCase(100, 45)]
[TestCase(80, 65)]
```

Of `[TestCaseSource(typeof(classname))]`
Waarbij classname IEnumerable implementeert

Voorbeeld van geparametriseerde unit test

```
[TestCase(100, 45)]
[TestCase(80, 65)]
[TestCaseSource(typeof(DamageSource))]
[Category("Slow")]
public void Health_Damage_ReturnsCorrectValue(int damage, int expectedHealth)
{
    _character.Damage(damage);
    Assert.That(_character.Health, Is.EqualTo(expectedHealth));
}

public class DamageSource : IEnumerable
{
    public IEnumerator GetEnumerator()
    {
        yield return new int[] {100, 45};
        yield return new int[] {80, 65};
    }
}
```

Groeperen en negeren van bepaalde unit tests

Groeperen van Unit Tests kan via de attribuut [Category(«categorieNaam»)]

Een test kan (tijdelijk) genegeerd worden via het attribuut [Ignore(«reden voor negeren»)]

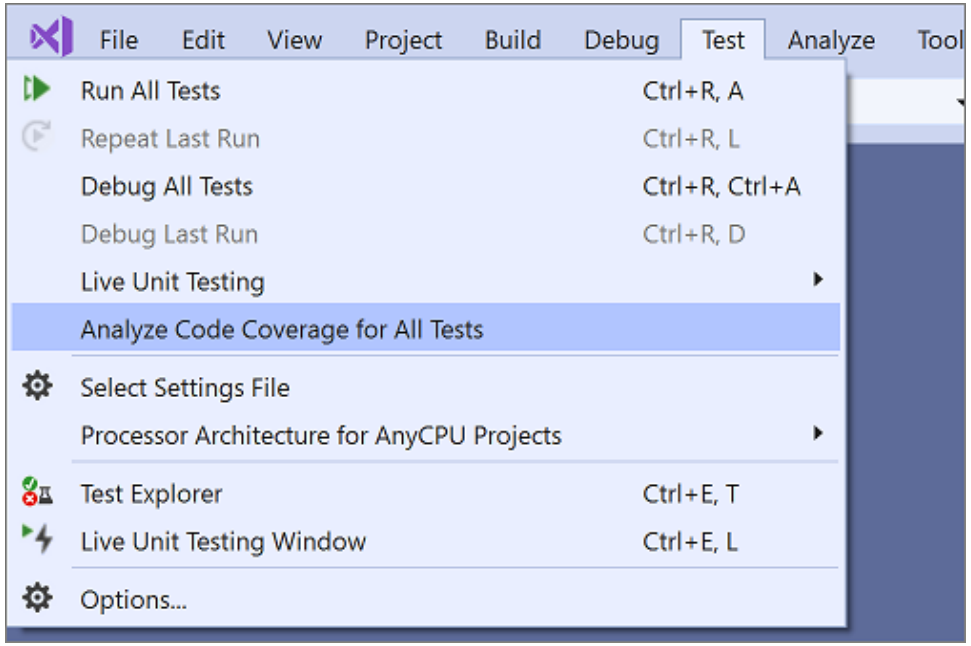
Voorbeeld:

```
[Test]
[Ignore("reason")]
public void IsDead_KillCharacter_ReturnsTrue()
{
    _character.Damage(500);
    Assert.That(_character.IsDead, Is.True);
}

[Test]
[Category("Slow")]
public void IsDead_DefaultCharacter_ReturnsFalse()
{
    Assert.IsFalse(_character.IsDead);
}
```

Unit test Code-dekking (code-coverage)

Minstens 80% (streefdoel) van de software zou gedekt moeten zijn door unit tests
Om betrouwbare en kwaliteitsvolle software te kunnen bereiken.
In Visual Studio (helaas enkel in de Enterprise Edition) in de kan dit berekend worden via een ingebouwde tool:



https://docs.microsoft.com/en-us/visualstudio/test/using-code-coverage-to-determine-how-much-code-is-being-tested?view=vs-2019

Samenvatting

1. Het AAA patroon bij unit tests
2. Gebruik van Assert.That bij NUnit.
   Welke zijn de belangrijkste constraints en hoe worden ze gebruikt om Assertions te schrijven
3. Hoe kunnen unit tests worden uitgevoerd
4. Gebruik en nut van de attributen SetUp en TearDown
5. Hoe unit testen parametriseren en toepassen van DRY (don’t repeat yourself) principe
6. Hoe unit testen groeperen en negeren
7. Hoe de unit test –code dekking van een project berekenen

Oefening Unit Test

Maak in het Business project een nieuwe class FizzBuzz aan

```
namespace Business
{
    public class FizzBuzz
    {
        public static string Ask(int number) {
            if (number % 3 == 0 && number % 5 == 0)
                return "FizzBuzz";
            if (number % 3 == 0)
                return "Fizz";
            if (number % 5 == 0)
                return "Buzz";
            return "";
        }
    }
}
```

Maak een Unit Test onder het project Business.Test voor de methode Ask

Oefening Unit Test (vervolg)

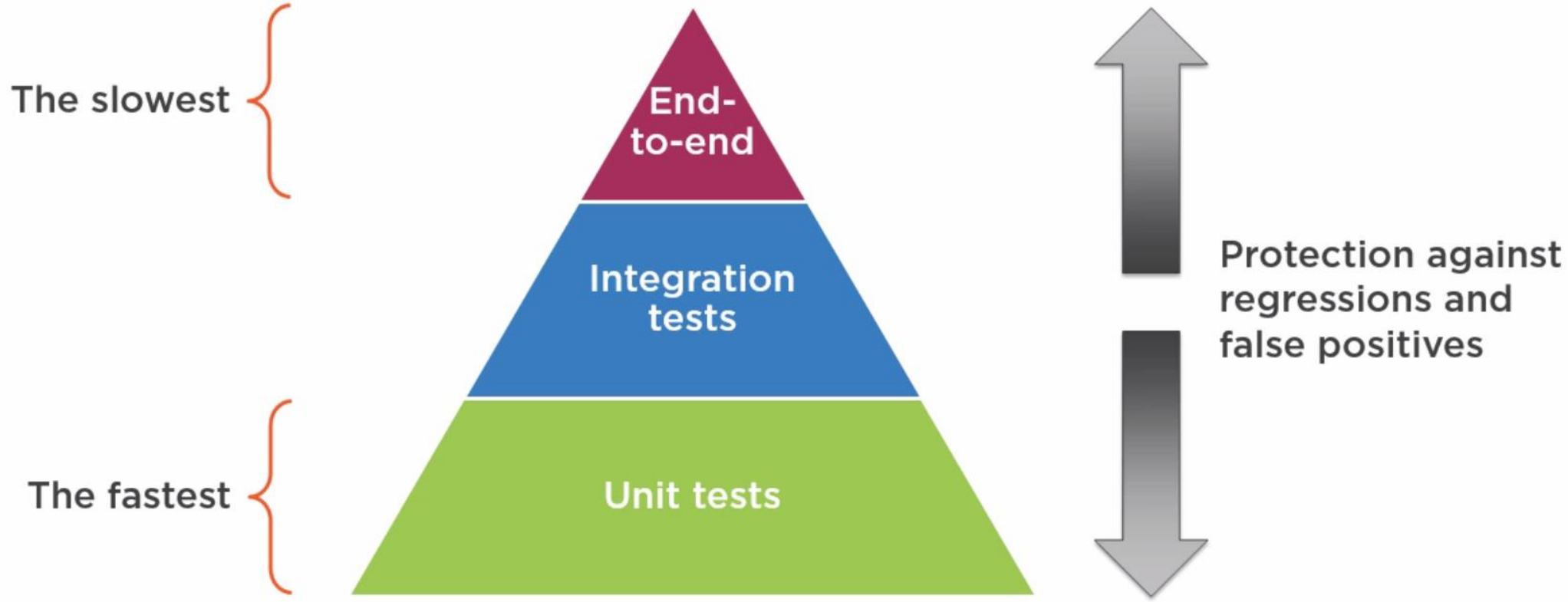
Maak een Unit Test onder het project Business.Test voor de methode Ask
De methode Ask zou moeten worden getest op deze functionaliteit:
Indien input parameter (int number)

deelbaar is door 3 -> "Fizz"
deelbaar is door 5 -> “Buzz“
deelbaar is door 3 en 5 -> “FizzBuzz“
Alle andere gevallen -> ""

Gebruik een geparametriseerde unit test (met waarden deelbaar door 3, 5, 3 en 5,.. kleiner of gelijk aan 30 die de methode voldoende testen

Beste praktijken bij Unit Testing

Wat maakt een Test waardevol?



Integration Tests

Dekking van unit tests

Voor minstens 80% van de code in een project zouden unit tests moeten bestaan om kwaliteitsgarantie te bieden voor de software.

Triviale Code: niet testen

- **Schrijf geen unit tests voor getters en setters. Deze worden indirect getest d.m.v. de unit tests van andere methoden.**
- **Schrijf geen unit tests voor triviale code of one-line methoden/functies. Deze worden indirect getest d.m.v. de unit tests van andere methoden.**

Test ‘Single Concern’

‘One-Assert-Per-Test’ principe:

Test enkel één aspect of gedrag per Unit test

Kenmerken van een goede Unit Test

- **Betrouwbaarheid**
- **Onderhoudbaarheid**
- **Leesbaarheid**

Wat te vermijden

- Vermijd control flow operatoren (controlestructuren) in unit testen
- Vermijd Duplicaties
- Zet geen Test Doubles op in de [SetUp] methode
- Vermijd tests die in bepaalde volgorde moeten worden uitgevoerd
  - creëert tijdelijke koppeling tussen unit tests
  - Je moet bij elke toevoeging van een unit test nadenken in welke volgorde welke test zou moeten worden uitgevoerd.
  - Testen die in een bepaalde volgorde moeten worden uitgevoerd, runnen trager
  - Testen die in een bepaalde volgorde moeten worden uitgevoerd zijn moeilijk te onderhouden

Wat te vermijden

- Over-specificatie te te vermijden, bv:
  - Meer ‘asserts’ geven dan enkel de interne status in een object die wordt getest.
  - Gebruik van meer dan één mock in een enkele test.
  - Gebruik maken van zowel stubs en mocks in een enkele test.
  - ‘Asserts’ geven van specifieke aanroepen of met exacte waarden indien dit niet noodzakelijk is

Streefdoel bij unit test

Onafhankelijkheid en isolatie

Conclusies

1. Unit tests geven geen garantie voor een succesvol software project
2. Beperk je bij het schrijven waardevolle unit testen.
3. Singletons en static classen maken het onmogelijk om unit tests te schrijven die ervan afhankelijk zijn, aangezien het moeilijk is om de statische afhankelijkheden te vervangen of imiteren
4. Voor kwaliteitsgarantie van de code in een project zou voor minstens 80% van de code unit tests moeten bestaan
5. Test slechts één logische eenheid in één enkele unit test.

Referenties

https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2019

https://www.youtube.com/watch?v=hsfVPPYoc9o

https://jakeydocs.readthedocs.io/en/latest/mvc/controllers/testing.html

https://raaaimund.github.io/tech/2019/05/07/aspnet-core-unit-testing-moq/

https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-with-nunit

https://wakeupandcode.com/unit-testing-in-asp-net-core/