

Unit Tests

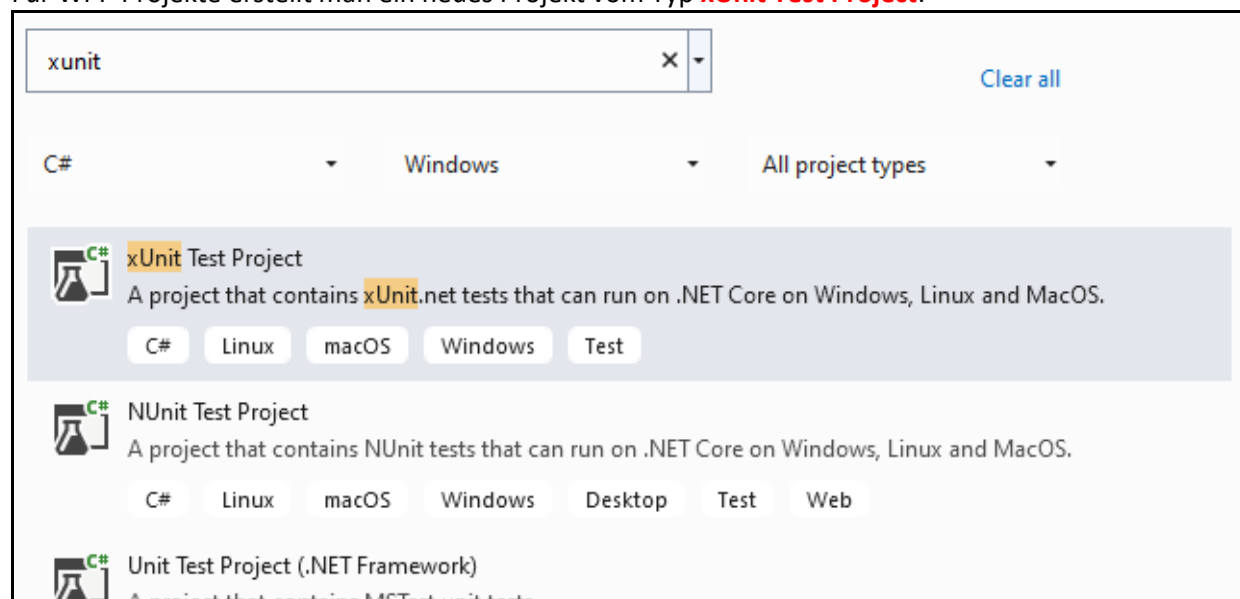
XUnit + FluentAssertions

1 PROJEKT ERSTELLEN	3
1.1 NuGet	3
1.1.1 Vorinstalliert	3
1.1.2 Update	3
1.1.3 Zusätzliche Pakete	4
1.2 .csproj file	4
1.3 Referenzen	4
2 EINFÜHRUNG	5
2.1 TestInitialize	5
2.2 TearDown	5
2.3 Hello World	5
2.4 Test ausführen	6
2.4.1 Test Explorer	6
2.4.2 Im Source Code	6
2.4.3 Konsole	7
2.4.4 watch	7
3 ALLGEMEINE TESTS	8
3.1 Zu testende Klassen	8
3.2 Einfache Tests	8
3.2.1 Konstruktor	8
3.2.2 [Fact]	8
3.2.3 Assert Exceptions	9
3.2.4 [Theory]	9
3.2.5 MemberData	10
3.3 Eigene ErrorMessage	10
3.4 Output	10
3.5 Ergebnisse	11
4 ZUSAMMENFASSUNG	13
4.1 XUnit Cheat Sheet	13
4.2 Vergleich xUnit/nUnit/MSTests	13
5 RACE CONDITIONS	14
5.1 Standard	14
5.2 Collection	14
5.3 xunit.runner.json	14
6 FLUENT ASSERTIONS	15
6.1 Dokumentation	15
6.2 Beispiel	16
6.3 Kategorien	16
6.3.1 Strings	17
6.3.2 Collections	17
6.3.3 Numerics	18
6.4 AssertionScopes	19
7 MVVM	20
7.1 Konstruktor	20

7.2 Einfacher Test	20
7.3 Events	21
7.3.1 Ohne FluentAssertions	21
7.3.2 Mit FluentAssertions	21
8 DATENBANK	22
9 DEPENDENCY INJECTION	24
9.1 ServiceProvider	24
9.2 Verwendung	24
9.3 Service	24
9.4 Testen	25
10 MOCKING	26
10.1 Interface	26
10.2 Mock-Injection	26
10.3 Mock verwenden	26

1 Projekt erstellen

Für WPF-Projekte erstellt man ein neues Projekt vom Typ **xUnit Test Project**:



Im Wesentlichen entsteht dadurch eine normale Klassenbibliothek für .NET Core, bei der schon einige Referenzen automatisch dabei sind.

1.1 NuGet

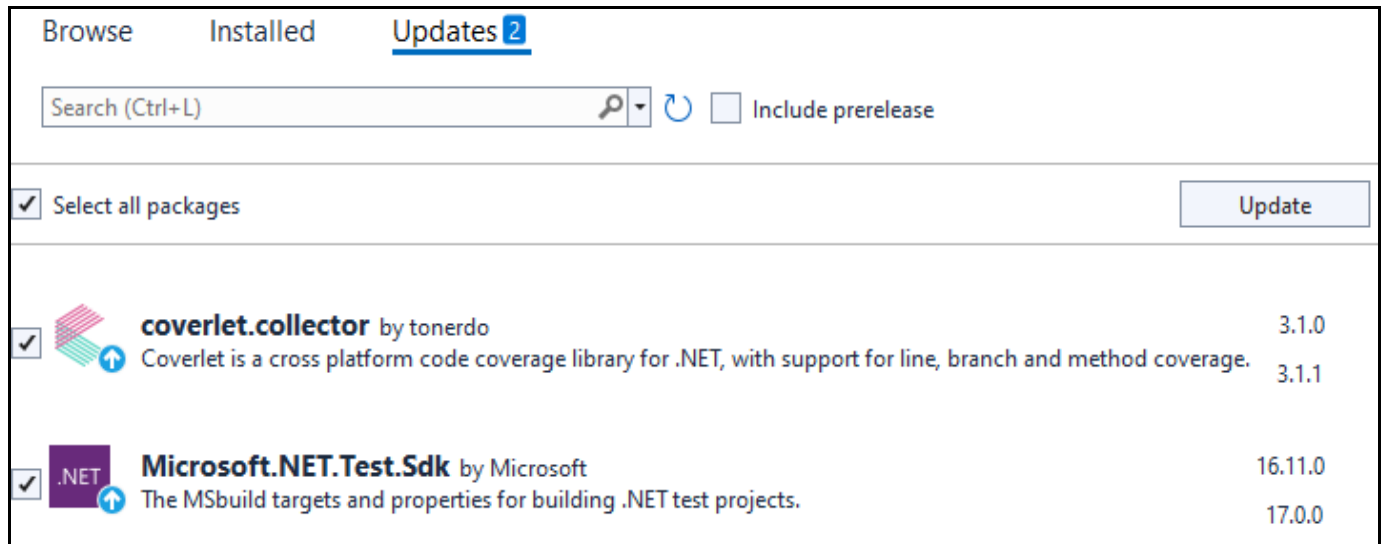
1.1.1 Vorinstalliert

Folgende Pakete sind aufgrund obiger Projektvorlage automatisch dabei:

Browse <u>Installed</u> Updates 2		
Search (Ctrl+L) <input type="checkbox"/> Include prerelease		
	coverlet.collector by tonerdo Coverlet is a cross platform code coverage library for .NET, with support for line, branch and method coverage.	3.1.0 3.1.1
	Microsoft.NET.Test.Sdk by Microsoft The MSbuild targets and properties for building .NET test projects.	16.11.0 17.0.0
	xunit by James Newkirk,Brad Wilson xUnit.net is a developer testing framework, built to support Test Driven Development.	2.4.1
	xunit.runner.visualstudio by .NET Foundation and Contributors Visual Studio 2017 15.9+ Test Explorer runner for the xUnit.net framework. Capable of running xUnit.net v1.9.2 and v2.0+ tests. Supports .NET 2.0 or later, .NET Core 2.1 or later, and Universal Windows 10.0.16299 or later.	2.4.3

1.1.2 Update

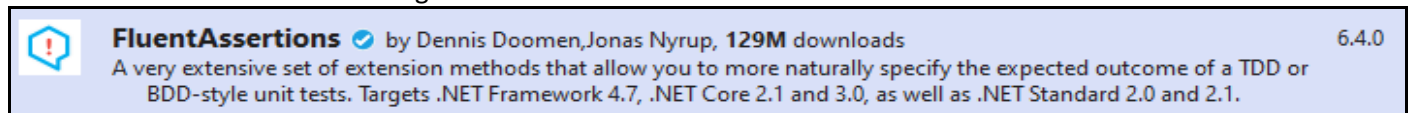
Diese sollten auf die neuesten Versionen aktualisiert werden (Tab „Updates“, Versionen Stand 2022-01-31).



1.1.3 Zusätzliche Pakete

Da vor allem die Angabe von benutzerdefinierten Fehlermeldungen nicht ganz optimal ist, bietet sich die Verwendung von Fluent Assertions (<https://fluentassertions.com/>) an.

Man braucht nur ein Paket mit Nuget zu installieren:



1.2 .csproj file

Das fertige Project-File sollte dann so aussehen:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <TargetFramework>net6</TargetFramework>
    <IsPackable>false</IsPackable>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="FluentAssertions" Version="6.4.0" />
    <PackageReference Include="Microsoft.NET.Test.Sdk" Version="17.0.0" />
    <PackageReference Include="xunit" Version="2.4.1" />
    <PackageReference Include="xunit.runner.visualstudio" Version="2.4.3">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
    <PackageReference Include="coverlet.collector" Version="3.1.1">
      <IncludeAssets>runtime; build; native; contentfiles; analyzers; buildtransitive</IncludeAssets>
      <PrivateAssets>all</PrivateAssets>
    </PackageReference>
  </ItemGroup>
</Project>
```

1.3 Referenzen

Wie immer darf man nicht vergessen, etwaige zusätzliche Referenzen dem Projekt hinzuzufügen. Um also Datenbankabfragen zu testen muss **EntityFrameworkCore** für das Testprojekt installiert werden.

Außerdem muss der richtige **ConnectionString** konfiguriert werden.

Testet man ein ViewModel, braucht man auch die Referenz auf **MvvmTools** sowie als TargetFramework **net6-windows** (siehe später).

2 Einführung

Eine Testklasse in xUnit ist eine ganz normale Klasse, sie muss nicht extra markiert werden (sie muss jedoch public sein). Der Name ist grundsätzlich egal, er sollte aber idealerweise sprechend sein. Lediglich die Testmethoden müssen mit Attributen gekennzeichnet werden:

- **[Fact]**: Normale Testmethode mit fixen Werten
- **[Theory]**: Testmethode mit Parametern. Die Parameter werden durch **[InlineData (...)]** übergeben

Man kann seine Tests also einfach auf mehrere Testklassen aufteilen. Einzige Voraussetzung ist eine using-Direktive:

```
using Xunit;
```

Damit kann man Klassenmethoden beliebiger Assemblies testen, also Libraries, WPF-Apps, Webserver,...

Die Tests innerhalb einer Klasse werden sequentiell ausgeführt, die Tests verschiedener Klassen jedoch parallel, falls man das nicht explizit verhindert (siehe weiter unten).

2.1 TestInitialize

Ein explizites TestInitialize gibt es nicht. Anstelle erstellt man im Konstruktor jene Objekte, die für alle Testmethoden dieser Klasse gelten.

2.2 TearDown

„Aufräumarbeiten“ könnte man in der Methode **Dispose()** erledigen. Dieser Teil ist aber oft nicht notwendig und fehlt daher.

```
public void Dispose()
{
    _db.Database.CloseConnection();
    GC.SuppressFinalize(this);
}
```

Voraussetzung: Die Testklasse implementiert das Interface **IDisposable**.

```
public abstract class TestBase : IDisposable
```

2.3 Hello World

Ein einfachster Test sieht so aus:

```
using Xunit;

namespace XunitDemo
{
    public class UnitTest1
    {
        [Fact]
        public void Test1()
        {
        }
    }
}
```

Es wird empfohlen, die Tests mit **T01_xxx** usw. durchnummerieren sowie im Namen der Testmethode zu beschreiben, was getestet wird.

```

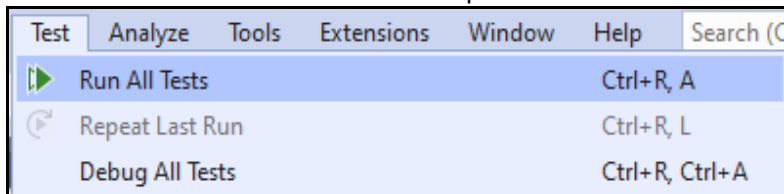
public class DummyTests
{
    [Fact]
    public void T01_TestDummy()
    {
        int expected = 66;
        int actual = 66;
        Assert.Equal(expected, actual);
    }
}

```

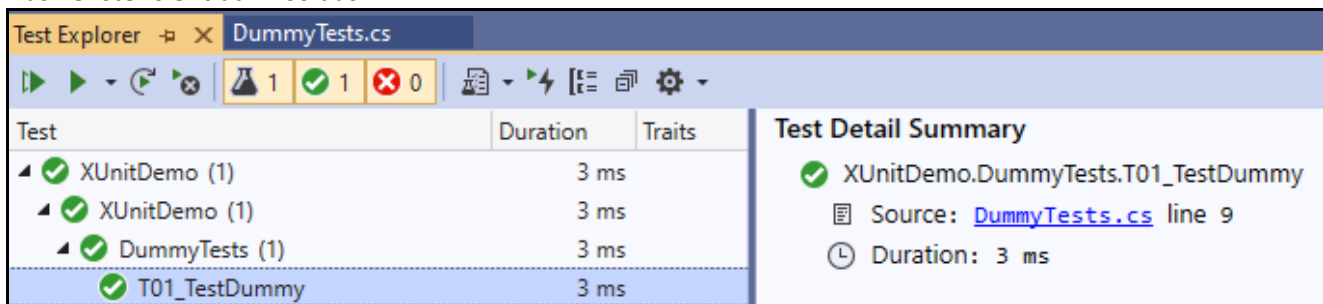
2.4 Test ausführen

2.4.1 Test Explorer

Unittests startet man über den Test Explorer. Diesen kann man mit **Test → All Tests** starten:



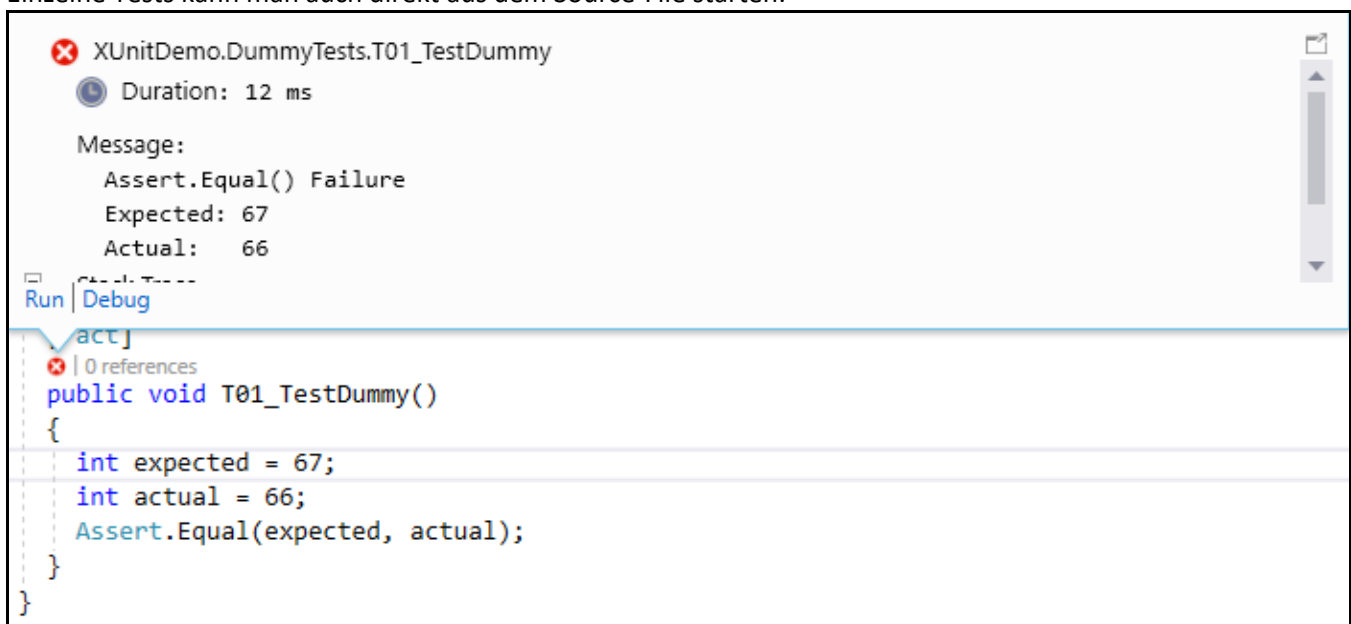
Das Fenster sieht dann so aus:



Dort kann man die Tests dann auch starten.

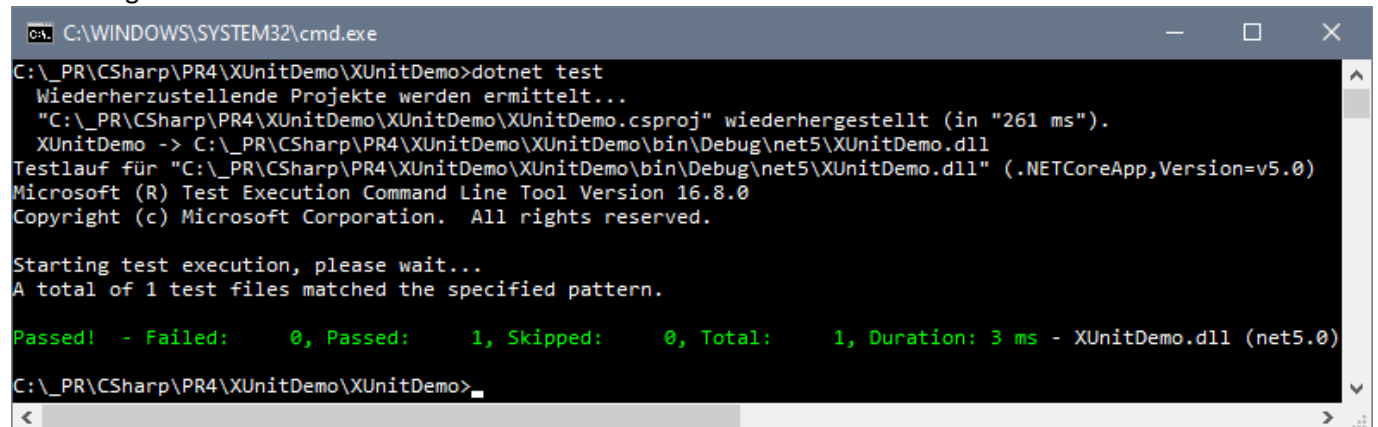
2.4.2 Im Source Code

Einzelne Tests kann man auch direkt aus dem Source-File starten:



2.4.3 Konsole

Um von der Konsole zu testen muss man sich in das Verzeichnis des UnitTests stellen und den Befehl **dotnet test** eingeben.

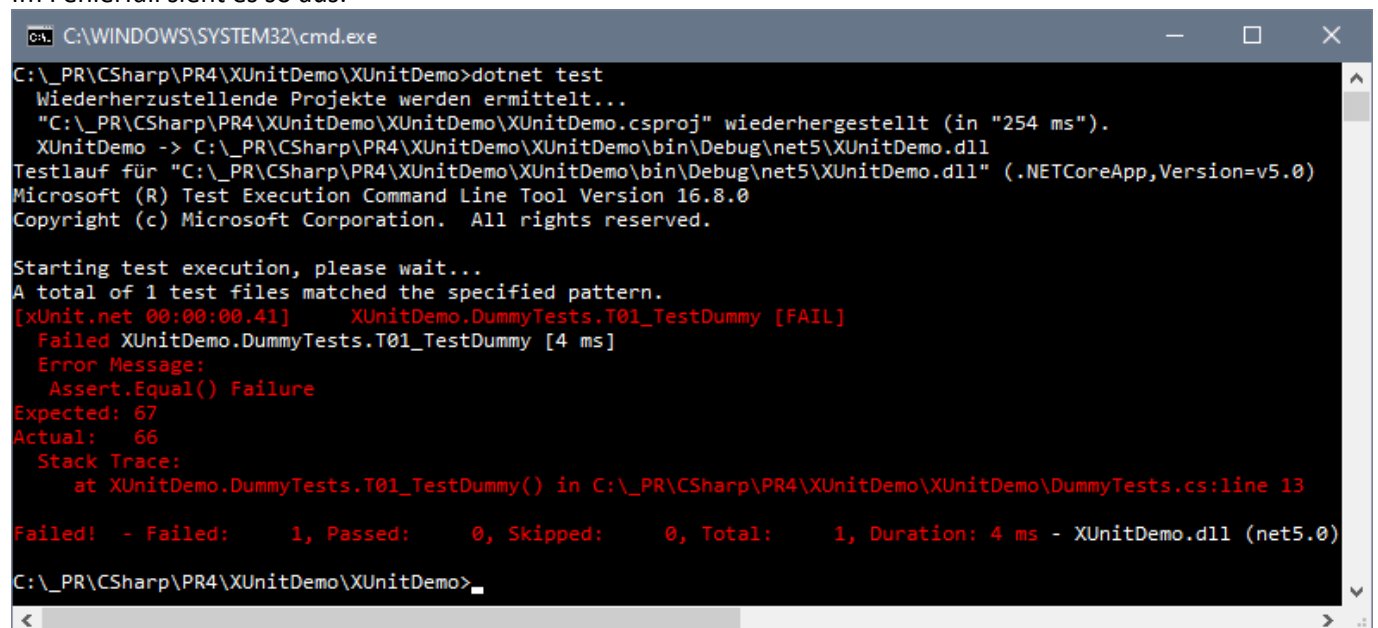


```
C:\WINDOWS\SYSTEM32\cmd.exe
C:\_PR\CSharp\PR4\XUnitDemo\XUnitDemo>dotnet test
Wiederherzustellende Projekte werden ermittelt...
"C:\_PR\CSharp\PR4\XUnitDemo\XUnitDemo\XUnitDemo.csproj" wiederhergestellt (in "261 ms").
XUnitDemo -> C:\_PR\CSharp\PR4\XUnitDemo\XUnitDemo\bin\Debug\net5\XUnitDemo.dll
Testlauf für "C:\_PR\CSharp\PR4\XUnitDemo\XUnitDemo\bin\Debug\net5\XUnitDemo.dll" (.NETCoreApp,Version=v5.0)
Microsoft (R) Test Execution Command Line Tool Version 16.8.0
Copyright (c) Microsoft Corporation. All rights reserved.

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.

Passed! - Failed:    0, Passed:    1, Skipped:    0, Total:    1, Duration: 3 ms - XUnitDemo.dll (net5.0)
C:\_PR\CSharp\PR4\XUnitDemo\XUnitDemo>
```

Im Fehlerfall sieht es so aus:



```
C:\WINDOWS\SYSTEM32\cmd.exe
C:\_PR\CSharp\PR4\XUnitDemo\XUnitDemo>dotnet test
Wiederherzustellende Projekte werden ermittelt...
"C:\_PR\CSharp\PR4\XUnitDemo\XUnitDemo\XUnitDemo.csproj" wiederhergestellt (in "254 ms").
XUnitDemo -> C:\_PR\CSharp\PR4\XUnitDemo\XUnitDemo\bin\Debug\net5\XUnitDemo.dll
Testlauf für "C:\_PR\CSharp\PR4\XUnitDemo\XUnitDemo\bin\Debug\net5\XUnitDemo.dll" (.NETCoreApp,Version=v5.0)
Microsoft (R) Test Execution Command Line Tool Version 16.8.0
Copyright (c) Microsoft Corporation. All rights reserved.

Starting test execution, please wait...
A total of 1 test files matched the specified pattern.
[xUnit.net 00:00:00.41] XUnitDemo.DummyTests.T01_TestDummy [FAIL]
  Failed XUnitDemo.DummyTests.T01_TestDummy [4 ms]
  Error Message:
    Assert.Equal() Failure
Expected: 67
Actual:   66
  Stack Trace:
    at XUnitDemo.DummyTests.T01_TestDummy() in C:\_PR\CSharp\PR4\XUnitDemo\XUnitDemo\DummyTests.cs:line 13

Failed! - Failed:    1, Passed:    0, Skipped:    0, Total:    1, Duration: 4 ms - XUnitDemo.dll (net5.0)
C:\_PR\CSharp\PR4\XUnitDemo\XUnitDemo>
```

2.4.4 watch

Startet man mit **dotnet watch test**, werden die Tests automatisch ausgeführt, sobald sich eine Datei ändert. Das gilt sowohl für den Unittest selbst, als auch für die zu testenden Source-Files.

3 Allgemeine Tests

3.1 Zu testende Klassen

Für die unten beschriebenen einfachen Tests werden folgende Klassen verwendet (entnommen und bearbeitet von <https://raygun.com/blog/unit-testing-frameworks-c/>).

```
public class Rabbit
{
    public void Dodge() => IsDodging = true;
    public void Hit() => IsDead = true;
    public void Miss() => IsDodging = false;
    public bool IsDodging { get; private set; }
    public bool IsDead { get; private set; }
}
```

```
public class Gun
{
    private int _ammo = 3;
    public void Recharge() => _ammo = 3;
    public bool HasAmmo => _ammo > 0;
    public void FireAt(Rabbit rabbit)
    {
        if (!HasAmmo) return;
        if (rabbit.IsDodging) rabbit.Miss();
        else rabbit.Hit();
        _ammo--;
    }
}
```

Die entsprechende Library dann wie oben erwähnt zum Projekt hinzufügen.

3.2 Einfache Tests

Mit obigen Klassen sollen einige einfache Testmethoden vorgestellt werden.

3.2.1 Konstruktor

Bei jedem Test soll ein Objekt vom Typ Rabbit sowie Gun erzeugt werden. Daher bietet sich an, diese Variable im Konstruktor zu erzeugen.

```
private readonly Rabbit rabbit;
private readonly Gun gun;
```

```
public ShootTests()
{
    rabbit = new Rabbit();
    gun = new Gun();
}
```

Achtung: es darf nur einen Konstruktor geben.

3.2.2 [Fact]

Mit **[Fact]** dekorierte Testmethoden haben keine Parameter und verhalten sich somit immer gleich. Eine einfache Testmethode könnte so aussehen:

```
[Fact]
public void T01_TryShootRabbit()
{
    gun.FireAt(rabbit);

    Assert.True(rabbit.IsDead);
    Assert.True(gun.HasAmmo);
}
```

```
[Fact]
public void T02_TryShootDodgingRabbit()
{
    rabbit.Dodge();
    gun.FireAt(rabbit);

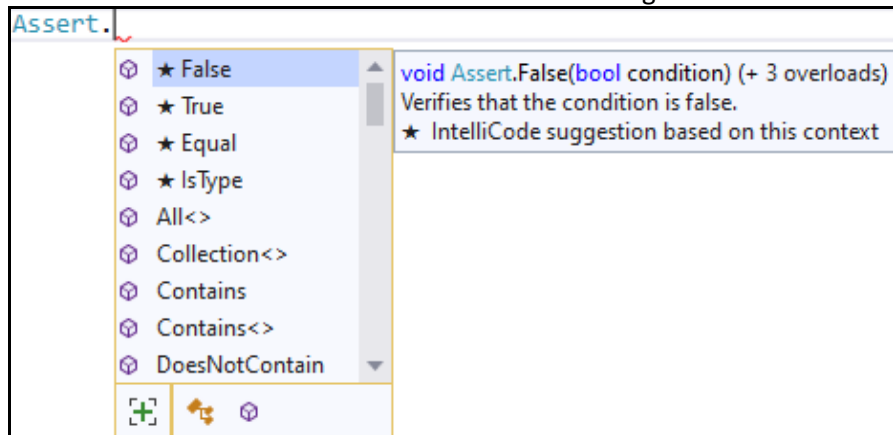
    rabbit.Dodge();
    gun.FireAt(rabbit);

    rabbit.Dodge();
    gun.FireAt(rabbit);

    Assert.False(rabbit.IsDead);
    Assert.False(gun.HasAmmo);
}
```


Die einzelnen Überprüfungen erfolgen mit Assert.

Aktuell scheint es dafür keine API-Dokumentation zu geben. Aber die Methodennamen sind sprechend:



3.2.3 Assert Exceptions

Möchte man sicherstellen, dass eine Methode eine Exception wirft, kann man das mit **Assert.Throws<>** überprüfen. Der Code-Teil, der eine Exception werfen soll, wird als Delegate übergeben:

```
[Fact]
public void T03_TryMakingHeapsOfGunsWithException()
{
    var guns = new Gun[5];
    Assert.Throws<IndexOutOfRangeException>( () => guns[5].FireAt(rabbit));
}
```

3.2.4 [Theory]

Testmethoden mit Parametern müssen mit **[Theory]** dekoriert werden. Die einzelnen Parameter werden mit **[InlineData()]** übergeben.

Möchte man z.B. testen, dass bei 1,2,3 bzw. 4 abgefeuerten Schüssen das richtige Ergebnis geliefert wird, könnte man das so programmieren:

```
[Theory]
[InlineData(1)]
[InlineData(2)]
[InlineData(3)]
[InlineData(4)]
public void T04_FireMultipleTimes(int firecount)
{
    for (int i = 0; i < fireCount; i++)
    {
        gun.FireAt(rabbit);
    }

    if (fireCount >= 3) Assert.False(gun.HasAmmo);
    else Assert.True(gun.HasAmmo);
}
```

Test Name	Data Points	Execution Time
ShootTests (9)	9	9 ms
T01_TryShootRabbit	1	2 ms
T02_TryShootDodgingRabbit	1	< 1 ms
T03_TryMakingHeapsOfGunsWithEx...	1	3 ms
T04_FireMultipleTimes (4)	4	4 ms
T04_FireMultipleTimes(fireCount: 1)	1	< 1 ms
T04_FireMultipleTimes(fireCount: 2)	1	< 1 ms
T04_FireMultipleTimes(fireCount: 3)	1	4 ms
T04_FireMultipleTimes(fireCount: 4)	1	< 1 ms

Man ist dabei nicht auf nur einen Parameter beschränkt:

```
[Theory]
[InlineData(true, false)]
[InlineData(false, true)]
public void T05_RabbitDodges(bool didDodge, bool shouldBeDead)
{
    if (didDodge) rabbit.Dodge();

    gun.FireAt(rabbit);

    if (shouldBeDead) Assert.True(rabbit.IsDead);
    else Assert.False(rabbit.IsDead);
}
```

3.2.5 MemberData

Will man komplexere Objekte als Parameter an eine Testmethode übergeben, muss man anstelle von InlineDate das Attribut MemberData verwenden.

```
[Theory]
[MemberData(nameof(TestRabbits))]
public void T06_TestWithObject(Rabbit rabbit)
{
    _output.WriteLine($"Rabbit dodge = {rabbit.IsDodging}");
    Assert.True(_rabbit.IsDodging);
}

public static List<object[]> TestRabbits => new()
{
    new object[] { new Rabbit { IsDodging = true } },
    new object[] { new Rabbit { IsDodging = false } },
};
```

Man erzeugt eine statische Methode, die die Aufruf-Parameter als object[] zurückgibt. In obigem Fall wird die Test-Methode zwei Mal aufgerufen (weil TestRabbits eine Liste mit 2 Einträgen zurückgibt), wobei jedes Mal ein Rabbit-Objekt als Parameter verwendet wird (weil das object[] aus einem Rabbit-Objekt besteht).

3.3 Eigene ErrorMessage

Bei einem fehlgeschlagenen Test wird von xUnit automatisch eine Fehlermeldung generiert. Möchte man den Fehlertext selber bestimmen, geht das bei Assert.Equal() leider nicht. Man kann aber den Umweg über **Assert.True()** gehen.

Anstelle von

```
Assert.Equal(expectedNrCategories, actualNrCategories);
```

schreibt man dann:

```
Assert.True(expectedNrCategories == actualNrCategories,
    $"NrCategories: expected {expectedNrCategories} was {actualNrCategories}");
```

Bessere Lösung: **Fluent Assertions** (siehe unten).

3.4 Output

Meines Wissens gibt es keine Möglichkeit, den Output der zu testenden Methoden anzuzeigen. Man kann aber innerhalb der Testmethode Output erzeugen. Dazu muss man sich ein Objekt vom Typ **ITestOutputHelper** per Dependency Injection übergeben lassen:

```
private readonly Gun gun;  
private readonly ITestOutputHelper output;  
  
public ShootTests(ITestOutputHelper output)  
{  
    this.output = output;  
    rabbit = new Rabbit();  
    gun = new Gun();  
}
```

Die Verwendung ist dann wie gewohnt:

```
public void T04_FireMultipleTimes(int fireCount)  
{  
    output.WriteLine($"Fire {fireCount} times");  
    var rabbit = new Rabbit();
```

Den Output kann man dann in einem eigenen Fenster öffnen:

Test Detail Summary

✓ XUnitDemo.ShootTests.T04_FireMultipleTimes(fireCount: 3)

Source: [ShootTests.cs](#) line 60

Duration: 3 ms

Standard Output:
Fire 3 times

Achtung: bei **dotnet test** wird der Output nur dann angezeigt, wenn der Test fehlschlägt!

3.5 Ergebnisse

Mit obigen Testmethoden entsteht im Fenster Unit Test Session folgendes Ergebnis:

Test Explorer

Run All | Run... | Playlist: All Tests

XUnitDemo (9 tests)

- ✓ UnitTestDemoProject (9) 37 ms
 - ✓ UnitTestDemoProject (9) 37 ms
 - ✓ MainTest (9) 37 ms
 - ✓ T01_TryShootRabbit 1 ms
 - ✓ T02_TryShootDodgingRabbit 1 ms
 - ✓ T03_TryMakingHeapsOfGunsWithException 1 ms
 - ✓ T04_FireMultipleTimes (4) 32 ms
 - ✓ T04_FireMultipleTimes(fireCount: 1) 1 ms
 - ✓ T04_FireMultipleTimes(fireCount: 2) 29 ms
 - ✓ T04_FireMultipleTimes(fireCount: 3) 1 ms
 - ✓ T04_FireMultipleTimes(fireCount: 4) 1 ms
 - ✓ T05_RabbitDodges (2) 2 ms

Änder man die Klasse Gun.cs derart um, dass z.B. **ammo** -- auskommentiert wird, wirkt sich das so aus:

The screenshot shows the Test Explorer window with a tree view of tests. The test `T02_TryShootDodgingRabbit` is selected and highlighted in blue. To its right, a detailed view of the test failure is displayed.

Test Explorer Tree View:

- XUnitDemo (9 tests) 3 failed
 - ❌ UnitTestDemoProject (9) 42 ms
 - ❌ UnitTestDemoProject (9) 42 ms
 - ❌ MainTest (9) 42 ms
 - ✅ T01_TryShootRabbit 1 ms
 - ❌ **T02_TryShootDodgingRabbit 1 ms**
 - ✅ T03_TryMakingHeapsOfGunsWithException 1 ms
 - ❌ T04_FireMultipleTimes (4) 37 ms
 - ✅ T04_FireMultipleTimes(fireCount: 1) 1 ms
 - ✅ T04_FireMultipleTimes(fireCount: 2) 33 ms
 - ❌ T04_FireMultipleTimes(fireCount: 3) 1 ms
 - ❌ T04_FireMultipleTimes(fireCount: 4) 2 ms
 - ▶️ ✅ T05_RabbitDodges (2) 2 ms

Test Failure Details:

UnitTestDemoProject.MainTest.T02_TryShoo... [Copy All](#)
Source: [MainTest.cs line 34](#)

❌ UnitTestDemoProject.MainTest.T02_TryShootDodgingRabbit

Message: Assert.False() Failure
Expected: False
Actual: True
Elapsed time: 0:00:00,001

Stack Trace:
Assert.False(Nullable`1 condition, String userMessage)
Assert.False(Boolean condition)
[MainTest.T02_TryShootDodgingRabbit\(\)](#)

4 Zusammenfassung

4.1 XUnit Cheat Sheet

Es gibt auch einen Cheat-Sheet für Xunit: <https://lukewickstead.wordpress.com/2013/01/16/xunit-cheat-sheet/>
 Üblicherweise ist der erste Wert der erwartete, der zweite der aktuelle.

```
Assert.Contains("n", "FNZ", StringComparison.CurrentCultureIgnoreCase);
Assert.Contains("a", new List<String> { "A", "B" }, StringComparer.CurrentCultureIgnoreCase);

Assert.DoesNotContain("n", "FNZ", StringComparison.CurrentCulture);
Assert.DoesNotContain("a", new List<String> { "A", "B" }, StringComparer.CurrentCulture);

Assert.Empty(new List<String>());
Assert.NotEmpty(new List<String> { "A", "B" });

Assert.Equal(1, 1);
Assert.Equal(1.13, 1.12, 1); // Precisions Num DP
Assert.Equal(new List<String> { "A", "B" }, new List<String> { "a", "b" },
    StringComparer.CurrentCultureIgnoreCase);
Assert.Equal(GetFoo(1, "A Name"), GetFoo(1, "a name"), new FooComparer());

Assert.NotEqual(1, 2);
Assert.NotEqual(new List<String> { "A", "B" }, new List<String> { "a", "b" },
    StringComparer.CurrentCulture);

Assert.False(false);
Assert.NotNull(false);
Assert.Null(null);
Assert.True(true);

Assert.InRange(1, 0, 10);
Assert.NotInRange(-1, 0, 10);

Assert.IsType(Type.GetType("System.Int32"), 1);
Assert.IsNotType(Type.GetType("System.Double"), 1);

var foo = new object();
var moo = new object();

Assert.Same(foo, foo);
Assert.NotSame(foo, moo);

Assert.Throws<Exception>(() => { throw new Exception(); });
Assert.True(true);
```

4.2 Vergleich xUnit/nUnit/MSTests

Es gibt mehrere Testframeworks, die prinzipiell sehr ähnlich sind und sich hauptsächlich in der Notation unterscheiden. Folgender Kurzvergleich wurde von <https://xunit.github.io/docs/comparisons.html#note1> entnommen:

NUnit 3.x	MSTest 15.x	xUnit.net 2.x
[Test]	[TestMethod]	[Fact]
[TestFixture]	[TestClass]	n/a
Assert.That	[ExpectedException]	Assert.Throws
Record.Exception		Record.Exception
[SetUp]	[TestInitialize]	Constructor
[TearDown]	[TestCleanup]	IDisposable.Dispose
[OneTimeSetUp]	[ClassInitialize]	IClassFixture<T>
[OneTimeTearDown]	[ClassCleanup]	IClassFixture<T>
n/a	n/a	ICollectionFixture<T>
[Ignore("reason")]	[Ignore]	[Fact(Skip="reason")]
[Property]	[TestProperty]	[Trait]
[Theory]	[DataSource]	[Theory]
		[XxxData]

5 Race Conditions

5.1 Standard

Innerhalb einer Klasse laufen die Tests sequentiell, jene unterschiedlicher Klassen jedoch parallel.

Möchte man also Tests parallel ausführen, muss man diese nur in unterschiedliche Testklassen platzieren.

5.2 Collection

Möchte man umgekehrt die parallele Ausführung verhindern, muss man die entsprechenden Testklassen mit Collection markieren, wobei derselbe Name benutzt werden muss:

```
[Collection("Sequential")]
public class ShootTests
{
    private readonly Rabbit rabbit;
    private readonly Gun gun;
```

```
[Collection("Sequential")]
public class DummyTests
{
    [Fact]
    public void T01_TestDummy()
```

Siehe auch: <https://xunit.net/docs/running-tests-in-parallel#parallelism-in-test-frameworks>

5.3 xunit.runner.json

Möchte man alle Tests sequentiell ausführen (weil sie z.B. Änderungen in der Datenbank vornehmen, die sich nicht gegenseitig beeinflussen dürfen), muss man die Datei xunit.runner.json erstellen mit folgendem Eintrag:

```
{
  "parallelizeTestCollections": false
}
```

Dann nicht vergessen, für diese Datei „Copy if newer“ einzustellen – das führt zu folgendem Eintrag im .csproj File:

```
<ItemGroup>
  <None Update="xunit.runner.json">
    <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
  </None>
</ItemGroup>
```

Details siehe: <https://xunit.net/docs/configuration-files>

parallelizeTestCollections Set this to **true** if the assembly is willing to run tests inside this assembly in parallel against each other. Tests in the same test collection will be run sequentially against each other, but tests in different test collections will be run in parallel against each other. Set this to **false** to disable all parallelization within this test assembly.

[Runners v2.1+]

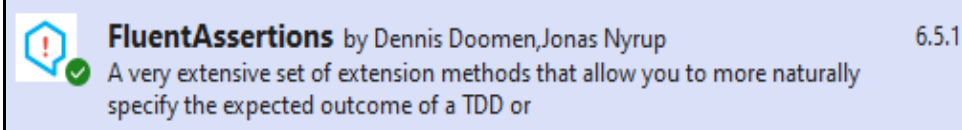
JSON schema type: **boolean**

Default value: **true**

6 Fluent Assertions

Da vor allem die Angabe von benutzerdefinierten Fehlermeldungen nicht ganz optimal ist, bietet sich die Verwendung von Fluent Assertions (<https://fluentassertions.com/>) an.

Voraussetzung ist das eingangs erwähnte Paket FluentAssertions:



Die Seite bietet eine umfangreiche Dokumentation mit vielen Beispielen. Daher ist die Verwendung praktisch selbsterklärend.

Die Unittests können damit so aufgebaut werden, dass man sie wie einen vollständigen englischen Satz lesen kann.

6.1 Dokumentation

Die Funktionen sind anhand von vielen Beispielen gut dokumentiert: <https://fluentassertions.com/introduction>

Getting started

Fluent Assertions is a set of .NET extension methods that allow you to more naturally specify the expected outcome of a TDD or BDD-style unit test. This enables a simple intuitive syntax that all starts with the following `using` statement:

```
using FluentAssertions;
```

This brings a lot of extension methods into the current scope. For example, to verify that a string begins, ends and contains a particular phrase.

```
string actual = "ABCDEFGHI";  
actual.Should().StartWith("AB").And.EndWith("HI").And.Contain("EF").And.HaveLength(9);
```

To verify that all elements of a collection match a predicate and that it contains a specified number of elements.

```
IEnumerable numbers = new[] { 1, 2, 3 };  
  
numbers.Should().OnlyContain(n => n > 0);  
numbers.Should().HaveCount(4, "because we thought we put four items in the collection");
```

The nice thing about the second failing example is that it will throw an exception with the message

"Expected numbers to contain 4 item(s) because we thought we put four items in the collection, but found 3."

6.2 Beispiel

Ein einfaches Beispiel könnte so aussehen:

```
[Fact]
public void T01_TestCollectionFluent()
{
    var expected = new List<int> { 2, 4, 6, 8, 10 };
    var actual = new List<int> { 2, 4, 6, 7, 10 };
    actual.Should().BeEquivalentTo(expected, because: "we expected only even numbers");
}
```

Hinweis: der zweite Parameter ist „because“, es bietet sich aber an, diesen trotzdem zu benennen, weil damit ein syntaktisch richtig lesbarer englischer Satz wird.

Fehlermeldung:

Test Detail Summary

✖ XUnitDemo.TestsWithFluentAssertions.T01_TestCollectionFluent

📄 Source: [TestsWithFluentAssertions.cs](#) line 10

🕒 Duration: 105 ms

Message:

Expected item[3] to be 8 because we expected only even numbers, but found 7.

6.3 Kategorien

Die einzelnen Test-Kategorien sind ebenfalls sehr gut beschrieben:

Getting Started

Detecting Test Frameworks

Subject Identification

Assertion Scopes

Basic Assertions

Nullable Types

Booleans

Strings

Numeric Types & IComparable

Dates & Times

Collections

Dictionaries

Guids

Enums

Exceptions

Object graph comparison

Event Monitoring

Type, Method, and Property assertions

Assembly References

XML

Execution Time

Meistens reicht schon die Übersichtsseite, um die wichtigsten Funktionen zu verstehen.

6.3.1 Strings

„Obviously you’ll find all the methods you would expect for string assertions“.

```
theString = "This is a String";
theString.Should().Be("This is a String");
theString.Should().NotBe("This is another String");
theString.Should().BeEquivalentTo("THIS IS A STRING");
theString.Should().NotBeEquivalentTo("THIS IS ANOTHER STRING");

theString.Should().BeOneOf(
    "That is a String",
    "This is a String",
);

theString.Should().Contain("is a");
theString.Should().Contain("is a", Exactly.Once());
theString.Should().Contain("is a", AtLeast.Twice());
theString.Should().Contain("is a", MoreThan.Thrice());
theString.Should().Contain("is a", AtMost.Times(5));
theString.Should().Contain("is a", LessThan.Twice());
theString.Should().ContainAll("should", "contain", "all", "of", "these");
theString.Should().ContainAny("any", "of", "these", "will", "do");
theString.Should().NotContain("is a");
theString.Should().NotContainAll("can", "contain", "some", "but", "not", "all");
theString.Should().NotContainAny("can't", "contain", "any", "of", "these");
theString.Should().ContainEquivalentOf("WE DONT CARE ABOUT THE CASING");
theString.Should().ContainEquivalentOf("WE DONT CARE ABOUT THE CASING", Exactly.Once());
theString.Should().ContainEquivalentOf("WE DONT CARE ABOUT THE CASING", AtLeast.Twice());
theString.Should().ContainEquivalentOf("WE DONT CARE ABOUT THE CASING", MoreThan.Thrice());
theString.Should().ContainEquivalentOf("WE DONT CARE ABOUT THE CASING", AtMost.Times(5));
theString.Should().ContainEquivalentOf("WE DONT CARE ABOUT THE CASING", LessThan.Twice());
theString.Should().NotContainEquivalentOf("HeRe ThE CaSiNg Is IgNoReD As WeLl");

theString.Should().StartWith("This");
theString.Should().NotStartWith("This");
theString.Should().StartWithEquivalent("this");
theString.Should().NotStartWithEquivalentOf("this");

theString.Should().EndWith("a String");
theString.Should().NotEndWith("a String");
```

6.3.2 Collections

„Most, if not all, are so self-explanatory that we’ll just list them here“:

```
IEnumerable collection = new[] { 1, 2, 5, 8 };

collection.Should().NotBeEmpty()
    .And.HaveCount(4)
    .And.ContainInOrder(new[] { 2, 5 })
    .And.ContainItemsAssignableTo<int>();

collection.Should().Equal(new List<int> { 1, 2, 5, 8 });
collection.Should().Equal(1, 2, 5, 8);
collection.Should().NotEqual(8, 2, 3, 5);
collection.Should().BeEquivalentTo(8, 2, 1, 5);
collection.Should().NotBeEquivalentTo(new[] { 8, 2, 3, 5 });

collection.Should().HaveCount(c => c > 3)
    .And.OnlyHaveUniqueItems();

collection.Should().HaveCountGreaterThan(3);
collection.Should().HaveCountGreaterOrEqualTo(4);
collection.Should().HaveCountLessOrEqualTo(4);
collection.Should().HaveCountLessThan(5);
collection.Should().NotHaveCount(3);
collection.Should().HaveSameCount(new[] { 6, 2, 0, 5 });
collection.Should().NotHaveSameCount(new[] { 6, 2, 0 });

collection.Should().StartWith(1);
collection.Should().StartWith(new[] { 1, 2 });
collection.Should().EndWith(8);
collection.Should().EndWith(new[] { 5, 8 });

collection.Should().BeSubsetOf(new[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, });

collection.Should().ContainSingle();
collection.Should().ContainSingle(x => x > 3);
collection.Should().Contain(8)
    .And.HaveElementAt(2, 5)
    .And.NotBeSubsetOf(new[] { 11, 56 });

collection.Should().Contain(x => x > 3);
```

6.3.3 Numerics

Hier gibt es die erwarteten Funktionen. Aufpassen muss man bei float bzw. double, weil man da nicht auf Gleichheit überprüfen darf.

```
[Fact]
public void T04_TestNumericEqual()
{
    double pi = Math.PI;
    //pi.Should().Be(3.1415927);
    pi.Should().BeApproximately(3.1415, 0.0001);
}
```

6.4 AssertionScopes

Normalerweise bricht ein UnitTest ab, sobald die erste Assertion verletzt wurde. Alle weiteren Tests werden nicht mehr ausgeführt. Durch Zusammenfassung in einen **AssertionScope** werden alle sich darin befindenden Überprüfungen ausgeführt und am Ende alle Fehlermeldungen gemeinsam ausgegeben.

```
[Fact]
public void T02_TestWithoutAssertionScope()
{
    5.Should().Be(10);
    "Actual".Should().Be("Expected");
}
```

```
[Fact]
public void T03_TestWithAssertionScope()
{
    using (new AssertionScope())
    {
        5.Should().Be(10);
        "Actual".Should().Be("Expected");
    }
}
```

Im ersten Fall wird nach dem ersten Fehler abgebrochen:

Test Detail Summary

✘ XUnitDemo.TestsWithFluentAssertions.T02_TestWithoutAssertionScope

Source: [TestsWithFluentAssertions.cs](#) line 19

Duration: 96 ms

Message:

Expected value to be 10, but found 5.

Stack Trace:

Mit AssertionScope wird auch bei Fehlschlagen der ersten Überprüfung die zweite ebenfalls noch ausgeführt:

Test Detail Summary

✘ XUnitDemo.TestsWithFluentAssertions.T03_TestWithAssertionScope

Source: [TestsWithFluentAssertions.cs](#) line 26

Duration: 100 ms

Message:

Expected value to be 10, but found 5.

Expected string to be "Expected" with a length of 8, but "Actual" has a length of 6, di

Stack Trace:

7 MVVM

Mit MVVM sind jetzt GUI-Tests sehr einfach, weil man ja nur Properties überprüfen muss. Sind die Daten/Properties im ViewModel richtig, werden diese auch sicher richtig in der GUI angezeigt (vorausgesetzt das Binding wurde richtig konfiguriert).

```
public class MainViewModel : ObservableObject
{
    private readonly NorthwindContext db;

    public MainViewModel(NorthwindContext db) { this.db = db; }

    private int selectedCategoryId;

    public int SelectedCategoryId
    {
        get { return selectedCategoryId; }
        set
        {
            selectedCategoryId = value;
            Products = db.Products
                .Where(x => x.Category.CategoryId == selectedCategoryId)
                .OrderBy(x => x.ProductName)
                .ToList();
            RaisePropertyChangedEvent(nameof(SelectedCategoryId));
        }
    }

    private List<Product> products;
    public List<Product> Products
    {
        get => products;
        set { products = value; RaisePropertyChangedEvent(nameof(Products)); }
    }
}
```

7.1 Konstruktor

Im Konstruktor wird das ViewModel und auch die Datenbank erzeugt.

```
private MainViewModel viewModel;

public MvvmTests()
{
    var db = new NorthwindDb.NorthwindContext();
    viewModel = new MainViewModel(db);
}
```

7.2 Einfacher Test

Es werden ganz normal die Properties zugewiesen und danach der neue Zustand des ViewModels überprüft.

```
[Fact]
public void T01_ProductsOfCategory()
{
    viewModel.SelectedCategoryId = 7;
    var expectedProductNames = new List<string>
    {
        "Longlife Tofu",
        "Manjimup Dried Apples",
        "Rössle Sauerkraut",
        "Tofu",
        "Uncle Bob's Organic Dried Pears",
    };
    var actualProductNames = viewModel.Products.Select(x => x.ProductName);
    actualProductNames.Should().BeEquivalentTo(expectedProductNames);
}
```

7.3 Events

Oft möchte man überprüfen, ob bei einer bestimmten Aktion auch im Anschluss das entsprechende PropertyChanged-Event ausgelöst wurde.

7.3.1 Ohne FluentAssertions

Etwas umständlich ist es nur mit den Sprachmitteln von xUnit. Dazu setzt man ein Flag zu Beginn auf false, registriert sich auf das Event und setzt das Flag auf true, wenn das Event eintrifft.

```
bool eventWasRaised = false;
string eventNameExpected = "Products";
viewModel.PropertyChanged += (_, e) =>
{
    if (e.PropertyName == eventNameExpected) eventWasRaised = true;
};
viewModel.SelectedCategoryId = 7;
Assert.True(eventWasRaised, $"Event <{eventNameExpected}> was not raised");
var expectedProductNames = new List<string>
```

7.3.2 Mit FluentAssertions

Durch FluentAssertions wird man auch hier besser unterstützt. Man muss sich für das zu überwachende Objekt einen sogenannten Monitor erstellen. Dieser stellt dann Methoden zur Verfügung, mit denen das Auftreten eines bestimmten Events überprüft werden kann.

```
using var monitor = viewModel.Monitor();
viewModel.SelectedCategoryId = 7;
monitor.Should().Raise("PropertyChanged")
    .WithArgs<PropertyChangedEventArgs>(e => e.PropertyName == "Products");
var expectedProductNames = new List<string>
```

Da im Bereich MVVM das Event PropertyChanged eine wesentliche Rolle spielt, wurde auch dieser Code noch vereinfacht:

```
using var monitor = viewModel.Monitor();
viewModel.SelectedCategoryId = 7;
monitor.Should().RaisePropertyChangeFor(x => x.Products);
var expectedProductNames = new List<string>
```

8 Datenbank

Problematisch sind noch Tests, bei denen zu Testzwecken neue Daten in die Datenbank eingetragen werden. Dies würde alle nachfolgenden Tests zum Scheitern bringen, weil da die erwarteten Werte möglicherweise nicht mehr stimmen.

Dazu bietet sich an, sich eine Datenbank zu konservieren und diese dann vor jedem Test in ein Temp-Directory zu kopieren, die Tests auszuführen und dann die Datenbank wieder zu löschen.

Der Code dazu könnte etwa so aussehen (die Start-Datenbank liegt im Beispiel unten auf D:\Temp):

```
private readonly MainViewModel viewModel;
private NorthwindContext db;
private string fileTemporaryDb;

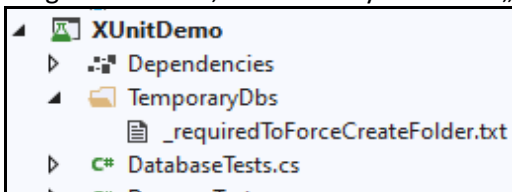
public DatabaseTests()
{
    InitDatabase();
    viewModel = new MainViewModel(db);
}

public void Dispose()
{
    db.Database.CloseConnection();
}

private void InitDatabase()
{
    string sourceDb = @"D:\Temp\Northwnd.mdf";
    string exeDirectory = Path.GetDirectoryName(Assembly.GetEntryAssembly().Location);
    fileTemporaryDb = Path.Combine(exeDirectory, "TemporaryDbs", "Northwnd.mdf");
    if (File.Exists(fileTemporaryDb)) File.Delete(fileTemporaryDb);
    File.Copy(sourceDb, fileTemporaryDb);

    string conectionString =
        $"server=(LocalDB)\mssqllocaldb;attachdbfilename={fileTemporaryDb};integrated security=True";
    var options = new DbContextOptionsBuilder<NorthwindContext>()
        .UseSqlServer(conectionString)
        .Options;
    db = new NorthwindContext(options);
    db.Database.OpenConnection();
}
```

Falls man ein lokales Sub-Directory benutzen will, muss man sicherstellen, dass dieses auch erstellt wird. Eine Möglichkeit wäre, eine Dummy-Datei mit „Copy if newer“ darin zu platzieren:



Damit kann man dann von einem konstanten Anzahl von z.B. Produkten mit CategoryId 7 ausgehen:

```
[Fact]
public void T01_ProductsOfCategory()
{
    int nrProductsBefore = 5;
    int categoryId = 7;
    viewModel.SelectedCategoryId = categoryId;
    viewModel.SelectedSupplier = db.Suppliers.Single(x => x.SupplierId == 1);
    viewModel.ProductName = "New Dummy Product";
    viewModel.ProductPrice = 666;
    viewModel.InsertProduct();
    int nrProductsAfterInsert = db.Products.Count(x => x.CategoryId == categoryId);
    nrProductsAfterInsert.Should().Be(nrProductsBefore + 1,
        because: "a new product should be added");
}
```


9 Dependency Injection

Unabhängig davon, ob eine Applikation mit Dependency Injection programmiert wurde oder nicht, kann man für Tests einen ServiceProvider aufbauen und diesen für UnitTests einsetzen.

9.1 ServiceProvider

Das funktioniert praktisch wie bei einer normalen Applikation, jedoch mit dem Unterschied, dass man die ServiceCollection selbst aufbaut und damit den ServiceProvider erstellt:

```
private readonly ServiceProvider _serviceProvider;
public TestsWithDependencyInjection()
{
    var services = new ServiceCollection();
    string connectionString = "Server=(LocalDB)\\mssqllocaldb;attachdbfilename=D:\\Temp\\No
    services.AddDbContext<NorthwindContext>(x => x.UseSqlServer(connectionString))
        .AddSingleton<MainWindow>()
        .AddSingleton<MainViewModel>();
    _serviceProvider = services.BuildServiceProvider();
}
```

9.2 Verwendung

Wie bei Dependency Injection gewohnt, müssen in diesem Fall dann auch alle Objekte über den ServiceProvider angefordert werden:

```
[Fact]
public void TestMvvm()
{
    var viewModel = _serviceProvider.GetRequiredService<MainViewModel>();
    viewModel.SelectedCategoryId = 7;
    var expectedProductNames = new List<string>{...};
    var actualProductNames = viewModel.Products.Select(x => x.ProductName);
    actualProductNames.Should().BeEquivalentTo(expectedProductNames);
}
```

9.3 Service

Damit kann man jetzt auch weitere Services verwenden, z.B. zum Kapseln einzelner Datenbankabfragen:

```
public class DbHelperService
{
    private readonly NorthwindContext _db;

    public DbHelperService(NorthwindContext db) => _db = db;

    public List<string> GetProductNamesBySupplierId(int supplierId)
    {
        return _db.Products
            .Where(x => x.Supplier.SupplierId == supplierId)
            .Select(x => x.ProductName)
            .OrderBy(x => x)
            .ToList();
    }
}
```

Dieses Service soll im ViewModel verwendet werden:


```
private readonly NorthwindContext _db;
private readonly DbHelperService _dbHelperService;
public MainViewModel(NorthwindContext db, DbHelperService dbHelperService)
{
    _db = db;
    _dbHelperService = dbHelperService;
}
```

```
private int _selectedSupplierId;
public int SelectedSupplierId
{
    get => _selectedSupplierId;
    set
    {
        _selectedSupplierId = value;
        SupplierProductNames = _dbHelperService.GetProductNamesBySupplierId(value);
        NotifyPropertyChanged(nameof(SelectedSupplierId));
    }
}
private List<string> _supplierProductNames;
public List<string> SupplierProductNames{...}
```

9.4 Testen

Dieses Service muss man jetzt nur registrieren, und schon kann damit getestet werden:

```
public TestsWithDependencyInjection()
{
    var services = new ServiceCollection();
    string connectionString = "Server=(LocalDB)\\mssqllocaldb;attachdbfilename=D:\\
services.AddDbContext<NorthwindContext>(x => x.UseSqlServer(connectionString))
    .AddSingleton<MainWindow>()
    .AddScoped<DbHelperService>()
    .AddSingleton<MainViewModel>();
    _serviceProvider = services.BuildServiceProvider();
}
```

```
[Fact]
public void T2_TestSupplierProductNames()
{
    var viewModel = _serviceProvider.GetRequiredService<MainViewModel>();
    viewModel.SelectedSupplierId = 4;
    var expectedProductNames = new List<string> {
        "Mishi Kobe Niku",
        "Ikura",
        "Longlife Tofu",
    };
    viewModel.SupplierProductNames.Should().BeEquivalentTo(expectedProductNames);
}
```

10 Mocking

Das kann man jetzt dazu verwenden, um zum Testen ein Service mit vordefinierten „Antworten“ zu benutzen. Dazu muss man Dependency Injection noch so umbauen, dass die „Best Practice“-Variante verwendet wird, nämlich ein Interface mit einer bestimmten Implementierung zu registrieren.

10.1 Interface

```
public interface IDbHelperService
{
    List<string> GetProductNamesBySupplierId(int supplierId);
}

public class DbHelperService : IDbHelperService
{
    private readonly NorthwindContext _db;
    public DbHelperService(NorthwindContext db) => _db = db;
```

Wichtig: die Konsumenten des Service müssen jetzt das Interface anfordern, und nicht die implementierende Klasse!

```
public class MainViewModel : ObservableObject
{
    private readonly NorthwindContext _db;
    private readonly IDbHelperService _dbHelperService;

    public MainViewModel(NorthwindContext db, IDbHelperService dbHelperService)
    {
        _db = db;
        _dbHelperService = dbHelperService;
    }
}
```

10.2 Mock-Injection

Man kann sich jetzt z.B. eine Klasse erstellen, die vordefinierte Werte produziert:

```
public class MockDbHelperService : IDbHelperService
{
    public List<string> GetProductNamesBySupplierId(int supplierId)
    {
        return supplierId switch
        {
            4 => new List<string> { "Mishi Kobe Niku", "Ikura", "Longlife Tofu" },
            1 => new List<string> { "Hansi", "Quaxi" },
            _ => new()
        };
    }
}
```

10.3 Mock verwenden

Mit dem Standard-Service schlägt jetzt folgender Test fehl:

```
viewModel.SelectedSupplierId = 1;
viewModel.SupplierProductNames.Should().BeEquivalentTo(new List<string> { "Hansi", "Quaxi" });
```

Injectet man die Mock-Klasse, sind die Tests erfolgreich:

```
services.AddDbContext<NorthwindContext>(x => x.UseSqlServer(connectionString))
    .AddSingleton<MainWindow>()
    // .AddScoped<IDbHelperService, DbHelperService>()
    .AddScoped<IDbHelperService, MockDbHelperService>()
    .AddSingleton<MainViewModel>();
```