

NUnit und xUnit.net im Vergleich

NUnit versus xUnit.net

Ende Mai 2008 wurde die erste stabile Version von xUnit.net veröffentlicht. Das neue Testtool bringt lange Zeit vermisste Testfunktionen mit. Allerdings hat xUnit einige, bislang hoch geschätzte Funktionen von NUnit nicht übernommen. Nicht in jedem Fall lohnt sich deshalb ein Umstieg auf xUnit. Dieser Artikel hilft bei der Auswahl der richtigen Testplattform.

Auf einen Blick



Gregor Biswanger ist Geschäftsführer von Web-enliven. Das Unternehmen bietet Beratung zur Softwarearchitektur im Bereich der Anwendungsentwicklung unter .NET. Biswanger ist auch Speaker und CLIPLer der Ingolstädter .NET Developers Group.

Inhalt

- Vor- und Nachteile von NUnit.
- Vor- und Nachteile von xUnit.net.

Grundlagen

Testwerkzeug NUnit,
 dnpCode.A0405Components

dnpCode
 A0901xUnit

Das Schreiben von Testklassen für Projekte hat einen hohen Stellenwert bei der Entwicklung. Selbst die Softwarearchitektur verlangt das Fertigstellen von Testklassen, bevor es an die Programmierung der eigentlichen Anwendung geht. Nun stellen sich zwei Fragen:

- Was genau ist wichtig für eine Testplattform?
- Wie flexibel muss die Testplattform anwendbar sein?

Auf der Suche nach Antworten auf diese Fragen werden hier die Open-Source-Projekte NUnit und xUnit.net unter die Lupe genommen und ihre Vor- und Nachteile abgewogen.

NUnit

Die immer noch bei Weitem bekannteste Testplattform unter .NET heißt NUnit [1]. Wie eine Handvoll anderer .NET-Open-Source-Projekte wurde NUnit ursprünglich von Java auf C# portiert. Durch das einfache Implementieren eigener Tests mithilfe von Attributen konnte es sich schnell bewähren. Die zu erstellende Klasse für Tests muss lediglich eine Referenz auf die Assembly *nunit.framework.dll* enthalten.

Die Testklasse bekommt ein Attribut mit dem Verweis *[TestFixture]* (Testvorrichtung). Die zu testenden Methoden werden einfach durch das Attribut *[Test]* gekennzeichnet.

Damit die Testmethoden auch gefunden werden können, müssen sie als *public* markiert sein. NUnit basiert auf *Assert* (Behauptung), womit die Testergebnisse verglichen werden. Listing 1 zeigt ein Beispiel für einen einfachen NUnit-Test.

Vorteile von NUnit

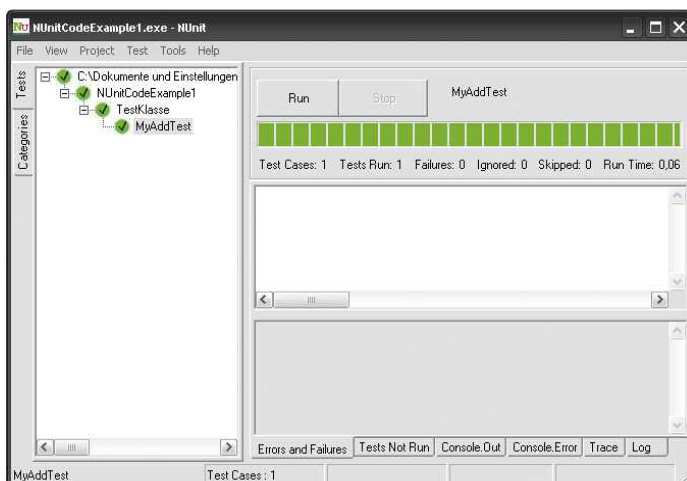
Viele Plug-ins für Visual Studio bieten eine zusätzliche Unterstützung von NUnit-Tests, die zum Beispiel automatisch eine Testvorlage aktueller Klassen generieren oder eigene Oberflächen zum Ausführen der Tests bereitstellen. Nach der Installation stellt NUnit zwei unterschiedliche Oberflächen zur Verfügung: eine Konsole für die manuelle Befehlseingabe und eine grafische Oberfläche.

Die grafische Oberfläche zeigt sich sehr umfangreich und macht einen guten Eindruck. Das Laden der Tests von Projekten geschieht über den Menüeintrag *File | Open | Project*. Die grafische Oberfläche erkennt alle Testklassen automatisch. Allesamt werden sie übersichtlich und strukturiert angezeigt. Der Start des Testlaufs erfolgt über den Button *Run*. Die Testergebnisse werden daraufhin farblich markiert angezeigt. Grün steht dabei für einen erfolgreichen Testdurchlauf, Rot signalisiert dagegen einen fehlgeschlagenen Test, siehe Abbildung 1.

NUnit stellt zusätzliche Hilfen zum Testen bereit, wie beispielsweise die Attribute *[SetUp]* (Testaufbau) und *[TearDown]* (Testabbau). Vor jeder Testmethode wird die *[SetUp]*-Methode durchgeführt, danach abschließend die *[TearDown]*-Methode.

Man kann hiermit Objekte instanzieren, frische Testdaten vorbereiten und nach dem Test auch wieder beseitigen. Somit bekommt jede Testmethode ihre eigenen Voraussetzungen und ist von der Reihenfolge der anderen Testläufe unabhängig. In Listing 2 finden Sie ein Beispiel für einen NUnit-Test mit Einsatz von *SetUp* und *TearDown*.

Weitere Attribute wie *[Platform("NET-1.0")]* können das Testen auf unterschiedlichen .NET-Framework-Versionen simulieren. Die in einem Betriebssystem eingestellte Sprache lässt sich



[Abb. 1] Die grafische Oberfläche von NUnit.

Listing 1

Beispiel für einen einfachen NUnit-Test.

```
using NUnit.Framework;
namespace NUnitCodeExample1
{
    [TestFixture] public class
    TestKlasse
    {
        [Test] public void MyAddTest()
        {
            var m = new MyMath();
            Assert.AreEqual(m.Add(2,3),5);
        }
    }
}
```

Mithilfe des Attributs *[SetCulture("en-GB")]* simulieren. Die in NUnit vorhandenen Funktionen und Optionen sind sehr mächtig, wie Sie unter [2] im Detail nachvollziehen können.

Nachteile von NUnit

Die Nachteile von NUnit offenbaren sich bei Testdurchläufen, die mit vorgefertigten Daten ausgeführt werden müssen, denn diese benötigen eine integrierte Quelle. Das bedeutet, dass über die Datenbankschicht bereits Daten abgefragt und den Testmethoden übermittelt werden müssen.

Leider stellt sich heraus, dass dies bei einer hohen Anzahl an Tests nicht besonders schnell ist und man deswegen auch gern manuelle, fest codierte Werte einbindet. Sollte dennoch ein direkter Zugriff auf eine Datenbank stattfinden, kommt das nächste Kontra gegen NUnit:

Im Testlauf auftretende Fehler können Datenmüll verursachen, der dann in die Datenbank geschrieben wird. Eine Unterstützung für Transaktionen, um solche Fehler wieder rückgängig machen zu können, wäre wünschenswert, ist aber nicht vorhanden. Abhilfe lässt sich hier über selbst gestrickte Lösungen schaffen, beispielsweise unter Einsatz von Methoden der Namensräume *System.Transactions* [3] oder *System.EnterpriseServices* [4].

	A	B	C	D
1	Int	Int	Int	
2		3	2	5
3		5	5	10
4		8	8	16
5				
6				

[Abb. 2] Excel-Datei mit Testdaten.

Listing 2

NUnit-Test mit SetUp und TearDown.

```
using NUnit.Framework;

namespace NUnitCodeExample2
{
    [TestFixture]
    public class TestKlasse
    {
        private Foo theFoo;

        [SetUp] public void Start()
        {
            theFoo = new Foo();
        }

        [Test] public void TestMethode()
        {
            Assert.IsNotNull(theFoo);
        }

        [TearDown] public void End()
        {
            theFoo.Dispose();
        }
    }
}
```

Listing 3

Ein einfacher xUnit-Test.

```
using Xunit;

namespace xUnitCodeExample1
{
    public class TestKlasse
    {
        [Fact] public void MyAddTest()
        {
            var m = new MyMath();
            Assert.Equal(m.Add(3, 2), 5);
        }
    }
}
```

Viele Testmöglichkeiten von NUnit sind zudem sehr oberflächlich, und es fehlt somit an detaillierten Ergebnissen. Ein Beispiel ist das Überprüfen bestimmter Ausnahmen (Exceptions) von Testmethoden. Mithilfe des Attributs *[ExpectedException]* [5] wird festgelegt, welche Exception in der zu testenden Methode ausgelöst werden sollte. Alternativ kann die erwartete Fehlermeldung vorgegeben werden. Dennoch findet diese Überprüfung für die komplette Testmethode statt. Optionen zum Festlegen, welches Objekt diese Exception auslöste, fehlen.

Listing 4

xUnit-Test mit Inline-Data.

```
using Xunit;
using XunitExt;

namespace xUnitCodeExample2
{
    public class TestKlasse
    {
        [Theory]
        [InlineData(3, 2, 5)]
        [InlineData(5, 5, 10)]
        [InlineData(8, 8, 16)]

        public void MyAddTest(int a,
            int b, int result)
        {
            var m = new MyMath();
            Assert.Equal(m.Add(a, b), result);
        }
    }
}
```

Listing 5

xUnit-Test mit Property-Data.

```
using System.Collections.Generic;
using Xunit;
using XunitExt;

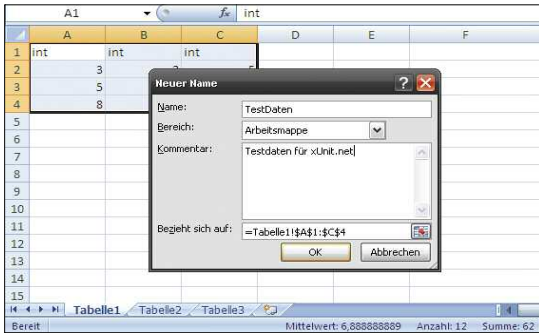
namespace xUnitCodeExample3
{
    public class TestKlasse
    {
        public static IEnumerable<object[]>
        MeineTestDaten
        {
            get
            {
                yield return new object[] { 3, 2, 5 };
                yield return new object[] { 5, 5, 10 };
                yield return new object[] { 8, 8, 16 };
            }
        }

        [Theory]
        [PropertyData("MeineTestDaten")]
        public void MyAddTest(int a, int b,
            int result)
        {
            var m = new MyMath();
            Assert.Equal(m.Add(a, b), result);
        }
    }
}
```

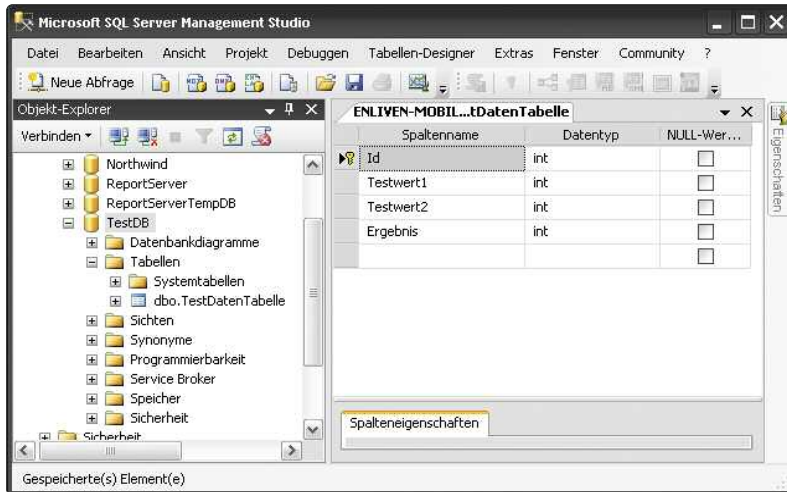
xUnit.net

Am 28. Mai 2008 wurde die erste stabile Version von xUnit.net auf Codeplex veröffentlicht [6]. Anders als NUnit ist xUnit.net von Anfang an ausdrücklich für das .NET

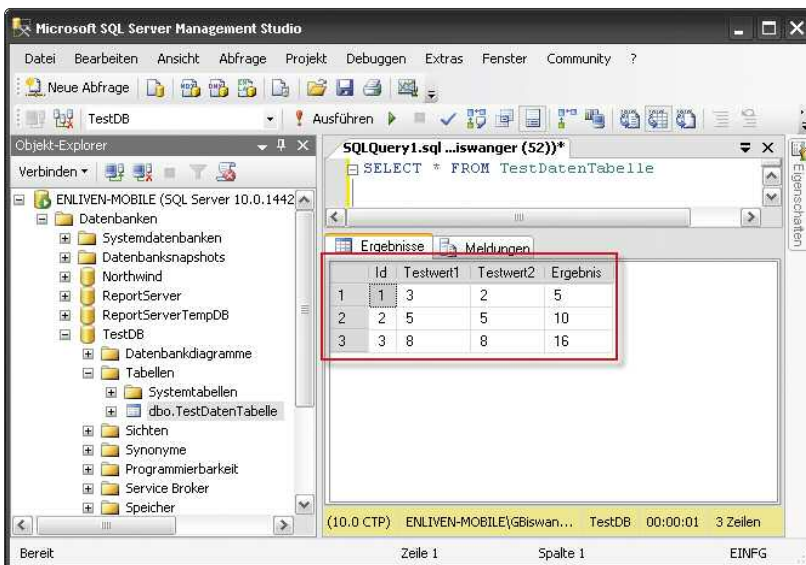
PRAXIS _NUnit und xUnit.net im Vergleich



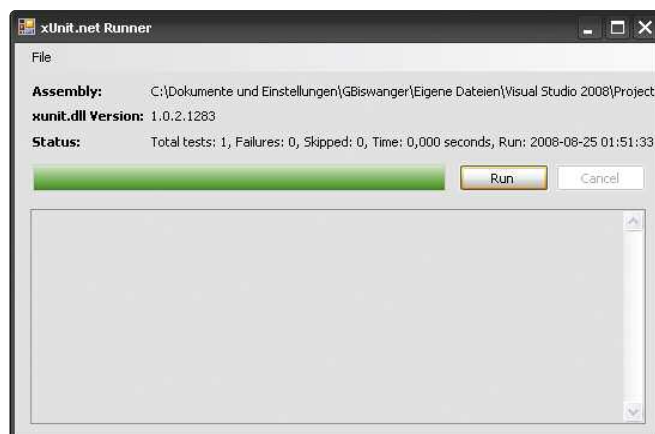
[Abb. 3] Neuer Name im Namens-Manager.



[Abb. 4] Aufbau der Testtabelle im SQL Server 2008.



[Abb. 5] Die in der Testtabelle enthaltenen Daten.



[Abb. 6] Die grafische Oberfläche von xUnit.net.

Framework konzipiert worden. Allein das ist schon ein guter Grund, sich näher mit xUnit.net zu befassen.

Die Testklassen benötigen eine Referenz auf die Assembly *xunit.dll*. Abweichend von NUnit bekommt die Testklasse kein Attribut zugewiesen, stattdessen werden die Testmethoden mit dem Attribut *Fact* gekennzeichnet. Auch bei xUnit.net müssen die Testmethoden als *public* markiert sein, damit der Zugriff auf sie klappt.

xUnit.net basiert wie NUnit auf *Assert* (Behauptung), womit die Testergebnisse verglichen werden können. Listing 3 zeigt einen einfachen Test mit xUnit.net.

Vorteile von xUnit.net

Obwohl xUnit.net noch recht jung ist, verfügt es bereits über eine Erweiterung: Mit einer zusätzlichen Referenz auf die Assembly *xunitExt.dll* werden neue Features zur Verfügung gestellt.

Eines der Features der Erweiterung wird über das Attribut *[Theory]* (Theorie) eingeleitet und erlaubt Datentests. Damit können Testmethoden mit Daten aus Excel-Dateien, Access-Dateien, Inline-Data, Property-Data oder einem SQL-Server geladen und ausgeführt werden. Listing 4 zeigt ein Beispiel für einen xUnit-Test mit Inline-Data und Listing 5 einen xUnit-Test mit Property-Data.

Besonders interessant ist das Auslesen von Excel-Dateien. Dafür muss die gewünschte Datei im Excel-2003-Format vorliegen. Abbildung 2 zeigt drei Felder in einer Excel-Tabelle. Damit die tabellarischen Daten als Datenbank simuliert werden können, müssen die belegten Felder markiert sein. Daraufhin wird durch die Tastenkombination [Strg]+[F3] der Excel-Namens-Manager geöffnet und nach einem Klick auf den Button *Neu...* der Name der gewünschten Datenbank angegeben, siehe Abbildung 3. Der Sourcecode zum Testen der Daten aus der Excel-Datei ist in Listing 6 zu finden.

Besonders reizvoll an xUnit.net ist die SQL-Server-Unterstützung. Abbildung 4 zeigt die *TestDatenTabelle* der Datenbank *TestDB* auf einem SQL Server 2008, und in Abbildung 5 sind die in der Tabelle gespeicherten Werte zu sehen.

Als Daten-Provider dient hier das Attribut *SqlServerData*. Durch eine ungekürzte SQL-Abfrage können die Spalten direkt an die Parameter der Testmethode übermittelt werden, wie es Listing 7 zeigt.

Durch die Erweiterungen der *xunitExt.dll* muss die Datenbankschicht keine fertigen Funktionen bieten. Die Datenquellen kön-

Listing 6

xUnit-Test mit Excel-Daten.

```
using Xunit;
using XunitExt;
namespace xUnitCodeExample4
{
    public class TestKlasse
    {
        [Theory]
        [ExcelData(@"ExcelTestDaten.xls", "select * from TestDaten")]

        public void MyAddTest(int a, int b, int result)
        {
            var m = new MyMath();
            Assert.Equal(m.Add(a, b), result);
        }
    }
}
```

Listing 7

xUnit-Test mit SQL-Server-Daten.

```
using Xunit;
using XunitExt;
namespace xUnitCodeExample5
{
    public class TestKlasse
    {
        [Theory]
        [SqlServerData("(local)", "TestDB",
            "SELECT Testwert1, Testwert2, Ergebnis FROM TestDatenTabelle")]

        public void MyAddTest(int a, int b, int result)
        {
            var m = new MyMath();
            Assert.Equal(m.Add(a, b), result);
        }
    }
}
```

nen von unterschiedlichen Testklassen benutzt werden.

Mit dem Attribut *[AutoRollback]* werden Datenzugriffe durch die eingebaute Transaktion ausgeführt. Sollte nun beim Testen etwas schief laufen, bekommen die Datenquellen auf keinen Fall unvollständige Daten übermittelt. Nicht nur mit dem Einsatz der Extension leistet xUnit.net gute Dienste.

Das detaillierte Überprüfen von Exceptions ist mittels *Assert.Throws* möglich. Mithilfe dieser Funktion kann sichergestellt werden, dass die Exception direkt vom getesteten Objekt ausgelöst wird.

```
Assert.Throws<ArgumentNullException>(() =>
    kunde.Name = null);
```

Für ASP.NET-Entwickler besteht zudem ein besonders attraktiver Vorteil: Beim Gebrauch des ASP.NET MVC Frameworks (*System.Web.Mvc*) kann der Controller direkt überprüft werden [7].

Nachteile von xUnit.net

Bei der Installation von xUnit.net werden bestehende Tools wie beispielsweise ReSharper [8] und TestDriven.NET [9] automatisch integriert. Wer keine Tools verwendet, ist jedoch auf die Konsole oder die schmalbrüstige grafische Oberfläche von xUnit angewiesen – siehe Abbildung 6 –, die erfahrungsgemäß auch noch zu oft ohne jegliche Meldung abstürzt. Doch auch wenn man diesen Punkt einmal außer Acht lässt, kam

mit der Veröffentlichung von xUnit eine Schreckensbotschaft für die Entwicklergemeinschaft: Die xUnit.net-Entwickler haben *[SetUp]* und *[TearDown]* aus der Unterstützung entnommen. Folglich kann nicht mehr für jede Testrunde eine eigene Instanzierung von Objekten durchgeführt werden. Auch die Möglichkeit zum Bereinigen am Schluss der jeweiligen Tests ist damit nicht mehr vorhanden.

Die umfangreiche Extension mag zwar trösten, doch ist leider ein weiterer Nachteil entstanden: Die Datenunterstützung ist auf ODBC aufgebaut, wodurch 64-Bit-Betriebssysteme ausgeschlossen werden. Auch die Unterstützung von CruiseControl.NET [10] (Build-Server) wird sofort uninteressant, wenn dieses auf ein 64-Bit-System aufsetzt.

Fazit

NUnit bietet bereits eine sehr umfangreiche Bedienoberfläche. Durch das Vorher- *[SetUp]* und Nachher-Prinzip *[TearDown]* können Tests sehr flexibel gestaltet werden. xUnit.net wäre ein hervorragender Nachfolger, wenn diese Funktionen nicht fehlen würden.

Leider ist der Einsatz von xUnit.net ohne unterstützende Plug-ins auch nicht korrekt durchführbar. Die grafische Oberfläche ist viel zu dünn. Und die charmante Extension entfaltet alle ihre Vorteile nur in einem 32-Bit-Umfeld. Dass xUnit.net aber in kommenden Versionen ein vollwertiger Nachfolger für NUnit wird, ist sehr gut vorstellbar. **[bl]**

- [1] NUnit-Download-Adresse, [↗ dnpLink SL0901xUnit1](#)
- [2] NUnit-Attribute, [↗ dnpLink SL0901xUnit2](#)
- [3] Golo Roden, Transactions.Erklärt(),
Interne Arbeitsweise des Namensraums
System.Transactions, dotnetpro 4/2008, S. 12 ff.
- [4] Marcel Gnoth, Einer für alle, alle für einen –
verteilte Transaktionen, dotnetpro 6/2003,
Seite 90 ff.
- [5] NUnit, ExpectedExceptionAttribute,
[↗ dnpLink SL0901xUnit3](#)
- [6] xUnit.net auf Codeplex,
[↗ dnpLink SL0901xUnit4](#)
- [7] xUnit.net für ASP.NET-MVC-Entwickler,
[↗ dnpLink SL0901xUnit5](#)
- [8] Visual Studio Add-in ReSharper 4.1,
[↗ dnpLink SL0901xUnit6](#)
- [9] Visual Studio Add-in TestDriven,
[↗ dnpLink SL0901xUnit7](#)
- [10] CruiseControl.NET, [↗ dnpLink SL0901xUnit8](#)
- [11] Attribute von xUnit.net,
[↗ dnpLink SL0901xUnit9](#)