2 NUnit

2.1 Homepage
http://www.nunit.org

2.2 Lizenz
zlib-Lizenz (kompatibel zur „GNU General Public License“)

2.3 Untersuchte Version
NUnit 2.5.9.10348

2.4 Letzter Untersuchungszeitpunkt
07.02.2011

2.5 Kurzbeschreibung
NUnit ist ein in C# geschriebens Unit-Testing Programm. Mit ihm lassen sich Unit-Tests für alle .NET Sprachen[9] schreiben und auswerten. Das Tool ist zusätzlich in der Lage, Unit-Tests auszuwerten die mit anderen Frameworks kompiliert wurden[10].

2.6 Fazit
Als Unit-Testing Werkzeug ist NUnit sehr gut geeignet. Es sticht durch seine gute Dokumentation und sowie durch Funktionsumfang (Attribute, Asserts und Constraints) heraus. Dem Anwender steht damit ein mächtiges Unit-Testing Werkzeug zur Verfügung.

2.7 Einsatzgebiete
Das Werkzeug wird für .NET Programmiersprachen eingesetzt. Es bietet dem Entwickler die Möglichkeit seinen Quellcode mittels Unit-Tests zu prüfen.

2.8 Einsatzumgebungen
Das Programm besitzt eine grafische Oberfläche, kann aber auch als Kommandozeilen-Tool genutzt werden. Es bietet leider kein echtes Addin für Microsoft Visual Studio an, kann somit lediglich als externes Tool aufgerufen werden. Was auf der Projektseite nicht angegeben ist, ist, dass bei der Installation von Sharpdevelop[11] NUnit bereits integriert ist. Dabei gibt es lediglich die Konsolenversion.

2.9 Installation
Die benötigte Software kann mittels Microsoft Windows Installer[12] auf dem PC eingerichtet werden. Während der Installation, sind keine besonderen Einstellungen vorzunehmen. Lediglich die AGBs sind zu bestätigen und der Speicherort zu wählen.

Visual Studio lässt sich soweit einrichten, dass die NUnit Anwendung aus VS aufgerufen werden kann[13].

Um in ein C# Projekt das NUnit-Framework verwenden zu können, muss das Projekt auf die „nunit.framework.dll“-Bibliothek verweisen.

[9] C#, VB.NET, Managed C++, …
[10] (NUnit (version 2.4.7, .NET 2.0) und Microsoft Unit Testing (basic support, VS 2008)
[11] http://www.icsharpcode.net/opensource/sd/
[12] Dateien mit der Endung .msi
[13] http://www.matthimatiker.de/2009/04/nunit-in-visual-studio-integrieren/

- Visual Studio – Im Projektbrowser, Rechtsklick auf den Ordner „Verweis“ und auf „Verweis hinzufügen...“ klicken. Nun den Pfad zur Datei „nunit.framework.dll“ im Installationsverzeichnis auswählen.
- Sharpdevelop - Im Projektbrowser, Rechtsklick auf den Ordner „Referenzen“ und auf „Referenz hinzufügen“ klicken. Nun den Pfad zur Datei „nunit.framework.dll“ m Installationsverzeichnis auswählen.

2.10 Dokumentation

Im Installationsverzeichnis von NUnit befindet sich ein Ordner „doc“. Dieser beinhaltet die gesamte Dokumentation im HTML-Format. Sie ist identisch mit der Dokumentation der Webseite zur jeweils installierten Version. Man kann diese Anleitung über die Datei „index.html“ öffnen oder in der Anwendung über („Help“→ „Nunit Help...“) bzw. der Taste F1 zu ihr gelangen.

Auf der Homepage befinden sich noch weitere Informationen, Tutorials und Tipps zu NUnit, sowie die passende Anleitung zu jeder Software-Version. Sucht man bestimmte Attribute oder Constraints, vermisst man schnell eine Suchfunktion in der Dokumentation. Die Projektseite beinhaltet noch ein Wiki, welches aber schlecht gepflegt wird.

2.11 Wartung der Projektseite

Die Webseite ist gut strukturiert und liefert eine schnelle Übersicht. Es werden über den Blog der Seite Nachrichten und Informationen mitgeteilt. Der letzte Eintrag stammt allerdings aus dem Jahr 2009. Zusätzlich gibt es noch eine Diskussionsgruppe[14].

2.12 Nutzergruppen und Support

Der Support ist umfangreich, die Entwickler bieten einen Blog und ein Diskusions-Forum an. Des Weiteren wird außerdem ein Wiki, welches für Entwickler und Anwender bereitgestellt. Es sei noch die Möglichkeit erwähnt, sich am Bug Report oder an der Feature Request List zu beteiligen.

2.13 Intuitive Nutzbarkeit

Die grafische Benutzeroberfläche ist einfach gehalten. Daher lässt sich der Umgang mit dem Werkzeug schnell erlernen. Die Syntax der NUnit-Tests ist leicht zu verstehen, da auf eine gute Dokumentation zurückgegriffen werden kann.

2.14 Automatisierung

NUnit kann einen Test auf verschiedene Art und Weise Parameter übergeben. Die Testwerte können dabei generisch erzeugt werden oder auch aus externen Quellen (zum Beispiel XML-Dateien oder eine Datenbank) stammen.

Das Zusammenfassen / Gruppieren von Tests kann sowohl über das Anlegen von Kategorien (CategoryAttribute), als auch über das Schreiben einer Test-Suite erfolgen. Das Starten der Suite ist allerdings etwas aufwändiger.

2.15 Einführendes Beispiel

Als erstes benötigt man ein TestFixture. Dieses wird durch ein `[TestFixture]`-Attribut vor der Testklasse gekennzeichnet.

Anschließend können eine oder mehrere Testmethoden erstellt werden. Diese sind durch `[Test]` gekennzeichnet.

[14] http://groups.google.com/group/nunit-discuss

```
[TestFixture]
public class MitarbeiterTest{ // es soll die Klasse Mitarbeiter getestet werden
     [Test]
     public void Mitarbeiter() //Test Methode
     { /* ... */ }

     [Test]
     public void Gehalt()
     { /* ... */ }
}
```

Listing 10: TextFixture- und Test-Attribute unter NUnit

Es können vor und nach jedem Test Methoden ausgeführt werden, die zum Beispiel vor jedem Test das gleiche Objekt erzeugen, damit alle Testmethoden ein Objekt zu gleichen Bedingungen erhalten. Dabei handelt es sich um die Attribute `[SetUp]` und `[TearDown]`. Letzteres sorgt dafür, dass diese Funktion nach jeder Testmethode ausgeführt wird.

Falls es nötig sein sollte einen Test nicht ausführen zu wollen, weil er zum Beispiel noch nicht komplett ausgeschrieben ist, dann kann er mit dem Attribut `[Ignore]` übersprungen werden. Natürlich lässt sich auch eine TestFixture überspringen. Der optionale Kommentar wird dann beim Testen angezeigt.

```
[TestFixture]
public class MitarbeiterTest
{
     [SetUp]
     public void Init()
     { /* ... */ }

     [TearDown]
     public void Cleanup()
     { /* ... */ }

     [Test]
     public void Gehalt()
     { /* ... */ }
}
```

Listing 11: SetUp- und TearDown-Attribute unter NUnit

Wenn davon auszugehen ist beziehungsweise erwartet wird, dass eine Testmethode (zum Beispiel GehaltException()) eine Exception auslöst, muss diese Methode entsprechend gekennzeichnet werden, ansonsten gilt der Test als nicht bestanden. Ist die Exception nicht vom geforderten Typ, so gilt dies auch als nicht bestanden. Über das Attribut `[ExpectedException]` und dem Typ der Exception ist dies anzugeben.

Um nun einzelne Konditionen / Bedingungen zu prüfen, benötigt man die Assert-Klasse.

Mit Hilfe dieser kann zum Beispiel geprüft werden, ob das Gehalt eines Mitarbeiters größer, kleiner oder genau gleich einem bestimmten Betrag ist.

Bei fehlgeschlagener Assertion kommt es zu einem Fehler, was als fehlgeschlagener Test zu werten ist. Nach dieser Assertion, führt die Testmethode keine weiteren Assertions mehr aus (s.Abbildung 10). Daher ist es zu empfehlen am besten nur eine Assertion pro Test durchzuführen.

```
[Test]
public void Gehalt(){
     Assert.Greater(mitarbeiter.Gehalt, 20000); //i.o. Wenn Betrag>20.000
     Assert.Less(mitarbeiter.Gehalt, 100000); //i.o. Wenn Betrag<100.000
     Assert.AreEqual(mitarbeiter.Gehalt, 36000, "Verdient 36.000 euro");
}
```

Listing 12: Verschiedene Asserts unter NUnit

In Listing 12 werden `Greater`, `Less` und `Equal` als Constraints bezeichnet. Alle Asserts, Attribute und Constraints befinden sich im Framework namespace von NUnit.

Zusätzliche Beispiele befinden sich in den folgenden Quelldateien:

Mitarbeiter.cs, MitarbeiterTest.cs, Projekt.cs, ProjektTest.cs

2.16 Detaillierte Beschreibung

2.16.1 Gruppieren von Tests

2.16.1.1 SuiteAttribute

Mit der Suite lassen sich ausgewählte Tests zu Gruppen zusammenfassen. Suites lassen sich über die „nunit-consolse.exe“ mit dem Befehl „/fixture“ ausführen. Auf das Konzept der Suites soll an dieser Stelle nicht tiefer eingegangen werden, da sie nicht in der GUI angezeigt werden können.

```
private class AllTests
{
     [Suite]
     public static IEnumerable Suite{
          get{
               ArrayList suite = new ArrayList();
               suite.Add(new OneTestCase()); //einzelne Testmethoden
               suite.Add(new AssemblyTests());
               suite.Add(new NoNamespaceTestFixture());
               return suite;
          }
     }
}
```

Listing 13: Suite-Attribute unter NUnit (Quelle: www.nunit.org)

2.16.1.2 CategoryAttribute

Um eine Gruppierungsfunktion auch in der GUI zu nutzen, besteht die Möglichkeit, Tests und Testfixtures in Kategorien einzuteilen. Kategorien können dann einzeln oder auch als Gruppe ausgeführt werden.

Somit werden lediglich die Tests ausgeführt, die in einer ausgewählten, beziehungsweise aktiven Kategorie stehen[15].

Die Konsolenbefehle lauten „`/include`“ und „`/exclude`“.

[15] Wie die einzelnen Kategorien im NUnit-Programm (GUI) aktiviert, beziehungsweise deaktiviert werden ist im Kapitel 2.16.7 „Die grafische Benutzeroberfläche von NUnit“ (Seite 19) erläutert.

```
[Test]
[Category(„Mitarbeiter Gehalt“)]
public void Gehalt()
{
Assert.Greater(mitarbeiter.Gehalt, 20000); //i.o. Wenn Betrag>20.000
Assert.Less(mitarbeiter.Gehalt, 100000); //i.o. Wenn Betrag<100.000
Assert.AreEqual(mitarbeiter.Gehalt, 36000, "Verdient 36.000 euro");
}
```

Listing 14: CategoryAttribute unter NUnit

2.16.2 Parametrisierte Tests

NUnit kann einem Test auf verschiedene Art und Weise Parameter übergeben. Damit lassen sich Tests schnell mit vielen Testwerten ausführen. Ein weiterer Vorteil ist, dass die Daten extern gehalten werden können.

Folgende Tabelle soll als Übersicht der verschiedenen Parameterübergabenattribute und deren Möglichkeiten dienen.

	Komplette Test Fälle	**Daten für ein Argument**
Inline	TestCaseAttribute	RandomAttribute RangeAttribute ValuesAttribute
Separate	TestCaseSourceAttribute	ValueSourceAttribute

Tabelle 1: Übersicht über die Attribute der Parameterübergabe (Quelle: Dokumentation NUnit)

2.16.3 TestCaseSourceAttribute

Dieses Attribut kann der Testmethode auf verschiedenste Arten von Daten übermitteln. Dabei kann das Attribut in zwei Varianten verwendet werden:

- sourceType – Wird benötigt, falls die Testwerte in einer anderen Klasse stehen. `TestCaseSourceAttribute(Type sourceType, string sourceName);`
- sourceName – Ist der Name der Datenquelle (zb.: Instanz, Funktion, Property) `TestCaseSourceAttribute(string sourceName);`

```
static int[] EvenNumbers = new int[] { 2, 4, 6, 8 };

[Test, TestCaseSource("EvenNumbers")]
public void TestMethod(int num) //wird 4 mal aufgerufen
{
     Assert.IsTrue( num % 2 == 0 );
}
```

Listing 15: TestCaseSource unter NUnit (Quelle: nunit.org)

```
[Test, TestCaseSource("DivideCases")]
public void DivideTest(int n, int d, int q) //wird 3 mal aufgerufen
{
     Assert.AreEqual( q, n / d );
}

static object[] DivideCases =
{
     new object[] { 12, 3, 4 }, //Testwerte
     new object[] { 12, 2, 6 },
     new object[] { 12, 4, 3 }
};
```

Listing 16: Erweiterte TestCaseSource unter NUnit (Quelle: nunit.org)

2.16.4 RandomAttribute – Zufallszahlen

NUnit bietet dem Programmierer die Möglichkeit mit dem Random-Attribut Zufallszahlen zu generieren. Dabei gibt es drei unterschiedliche Funktionen mit unterschiedlichen Parametern.

Count – Wieviele Zahlen generiert werden sollen.

```
public Random( int count );
```

Min/Max – Unter und Obergrenze eines gültigen Wertebereiches

```
public Random( double min, double max, int count );
public Random( int min, int max, int count );
```

Listing 17 liefert insgesamt 15 Ausgaben. Zu jedem der drei x-Werte werden fünf Zufallszahlen zwischen -1,0 und 1,0 erzeugt.

```
[Test]
public void MyTest(
     [Values(1,2,3)] int x,
     [Random(-1.0, 1.0, 5)] double d)
     {
          [...]
     }
```

Listing 17: RandomAttribute unter NUnit

2.16.5 TestCaseAttribute – Inline Anweisung mit Parametern

Mit dem TestCase-Attribut gelingt es, alle Argumente für einen Test anzugeben. Falls einer der Testcases eine Exception auslöst, ist das Attribut ExpectedException zu verwenden.

```
[TestCase(12,3,4)]
[TestCase(12,2,6)]
[TestCase(12,0,9), ExpectedException(typeof(DivideByZeroException))]
public void DivideTest(int n, int d, int q) //wird 3 mal ausgeführt
{
     Assert.AreEqual( q, n / d );
}
```

Listing 18: TestCase-Attribut unter NUnit

2.16.6 Sequential- und CombinatorialAttribute

NUnit ist in der Lage Testfälle selber zu generieren. Dazu gehören unter anderem das `[Sequential]`- und das `[Combinatorial]`-Attribut. An Tabelle 2 ist zu sehen, dass Combinatorial alle Kombinationsmöglichkeiten liefert.

```
[Test, Sequential] // oder [Test, Combinatorial]
public void MyTest(
     [Values(1,2,3)] int x,
     [Values("A","B")] string s)
     {
          ...
     }
```

Listing 19: Sequential-Tests unter NUnit

	Sequential	Combinatorial
Ausgabe	MyTest(1, "A") MyTest(2, "B") MyTest(3, null)	MyTest(1, "A") MyTest(1, "B") MyTest(2, "A") MyTest(2, "B") MyTest(3, "A") MyTest(3, "B")

Tabelle 2: Ausgabe der beiden Attribute zum zu Listing 19

2.16.7 Die grafische Benutzeroberfläche von NUnit

Zu Anfang muss die Assembly-Datei des kompilierten Projektes in die Anwendung geladen werden. Die Anwendung ist auch in der Lage, mehrere Assemblies einzubinden.

Über die „Project“→ „Add Assembly“ wird der Dateipfad der Anwendung mitgeteilt. Im Nachfolgenden Beispiel handelt es sich um die „Mitarbeiterverwaltung.dll“ aus dem Ordner „..\Mitarbeiterverwaltung\bin\Debug\“.

Falls diese Einstellungen wieder verwendet werden sollen, kann ein Projekt angelegt und gespeichert werden. Die Testergebnisse lassen sich im XML-Format exportieren.

Um das Testen zu starten, muss auf den Button „Run“ geklickt werden. Anschließend gibt es drei Möglichkeiten, wie der Fortschrittsbalken gefärbt sein kann.

2.16.7.1 Gelb - Test Ignoriert

Tests die mit dem Attribut `ignore` versehen sind, werden übersprungen. Somit ist das Resultat des Tests unbekannt und der Fortschrittsbalken wird gelb gefärbt (auch wenn alle anderen Tests korrekt sind). Der betreffende Test wird zusätzlich mit einem gelben Fragezeichen, in der Baumansicht gekennzeichnet.

Um die angefügten Kommentare des `ignore` Attributs zu lesen, muss die Ansicht „Tests Not Run“ gewählt werden.

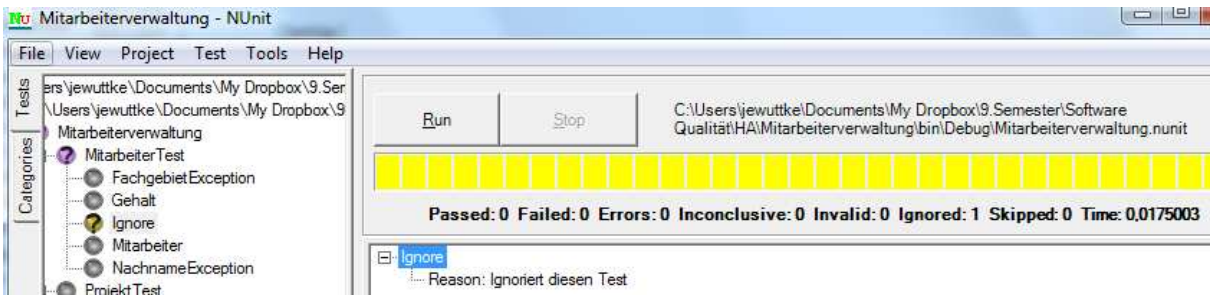

Abbildung 4: Ignore-Attribut mit Kommentar – [ignore(„Ignoriere diesen Test“)]

2.16.7.2 Rot – Test fehlgeschlagen

Entspricht das Ergebnis eines Tests, nicht dem gewünschten Wert, so wird dem Benutzer angezeigt, welche Assertion/Exception stattdessen erwartet wurde. Zusätzlich besteht die Möglichkeit zwischen den Ansichten „Display source code context“ und „Display actual stack trace“ zu auszuwählen.

- *Display source code context* – Zeigt die Zeile(n) im Quelltext an, die zu dem falschen Ergebnis geführt haben.
- *Display actual stack trace* – Liefert den Stack Inhalt

In Abbildung 5 ist zu sehen, wie die Assertion beim Vergleich der Fachgebiete des Mitarbeiters Jens das gesuchte Fachgebiet DESIGN nicht findet und somit „false“ zurück liefert. Damit wurde der Test als fehlgeschlagen markiert.

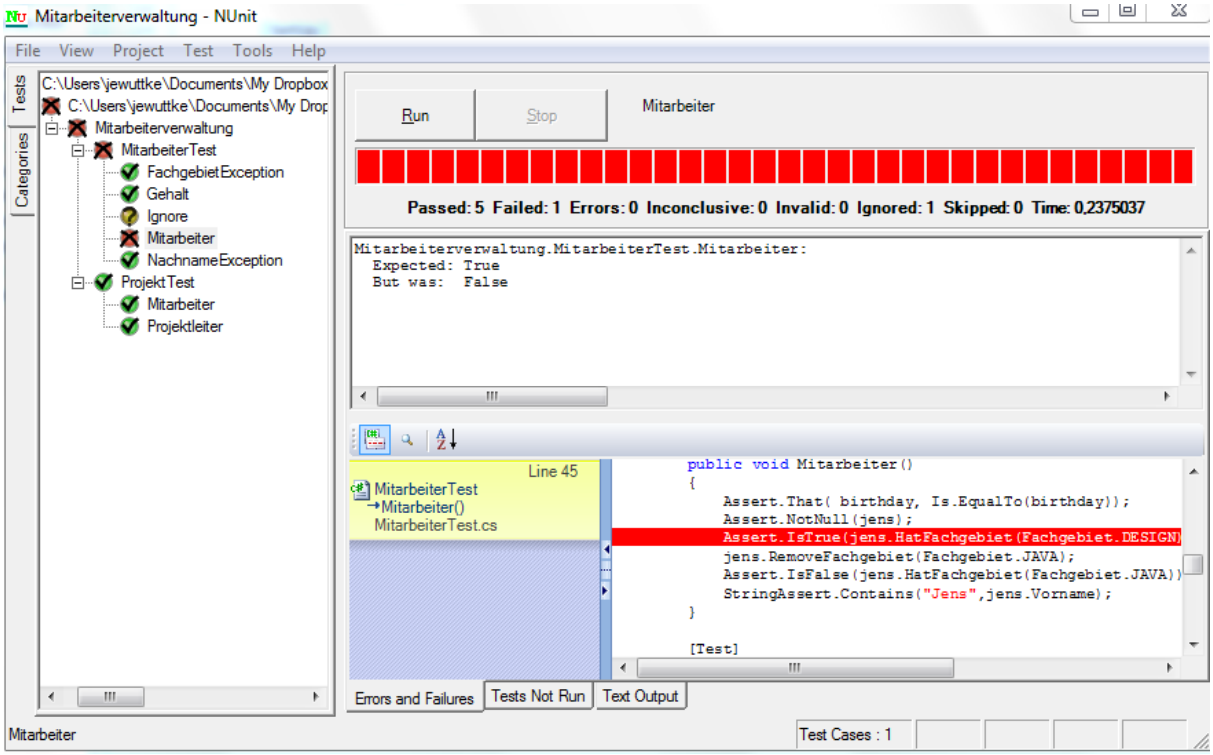

Abbildung 5: Test-Methode „Mitarbeiter“ – Fehlgeschlagen

2.16.7.3 Grün – Test erfolgreich

Das Testresultat des Projektes Mitarbeiterverwaltung, kann in Abbildung 6 betrachtet werden. Es besteht die Möglichkeit, ungewünschte Tests auszuschließen. Führt man einen Rechtsklick in der Baumansicht durch, so kann die Option „Show CheckBoxes“ aktiviert werden.

Deutlich zu sehen ist, dass der Test „GehaltException“ drei Mal und „MitarbeiterCombinatorial“ insgesamt neun Mal ausgeführt wurde.

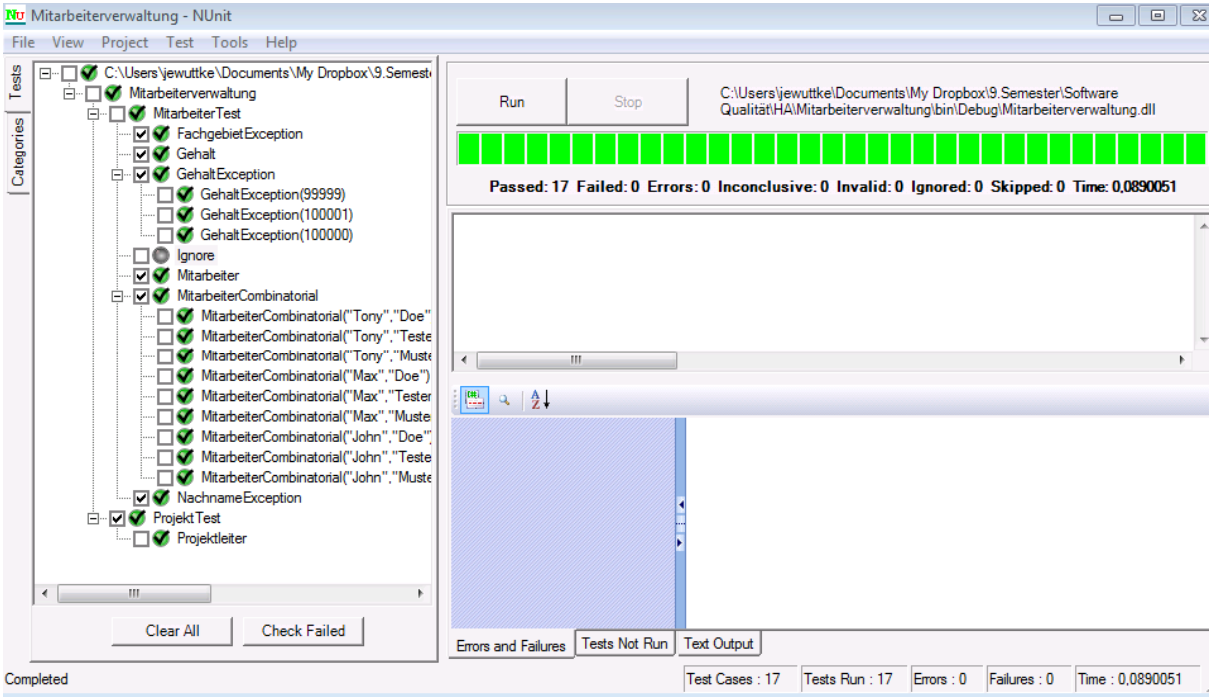

Abbildung 6: Erfolgreicher Test des Projekttests „Mitarbeiterverwaltung“

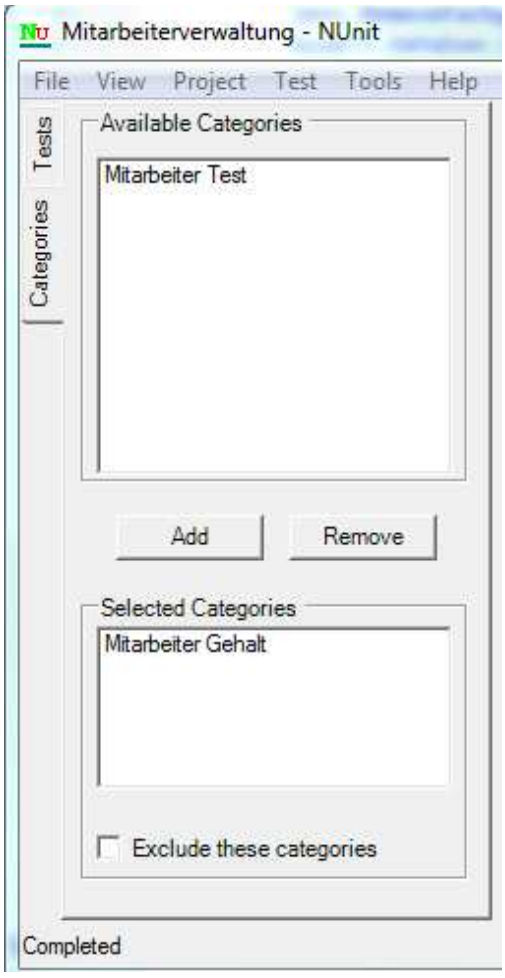

Abbildung 7:Aktivieren von Kategorien

2.16.8 Kategorien verwalten

Die mit dem Attribut `[Category]` versehenden Tests können in der Ansicht „Categories“ verwaltet werden.

Diese lassen sich über die Buttons „Add“ und „Remove“ ein- und ausschalten.

Zur Veranschaulichung, wurde nur die Kategorie „Mitarbeiter Gehalt“ aktiviert. Das Resultat ist, dass alle Tests mit der entsprechenden Kategorie aus dem Projekt Mitarbeiterverwaltung ausgeführt wurden.

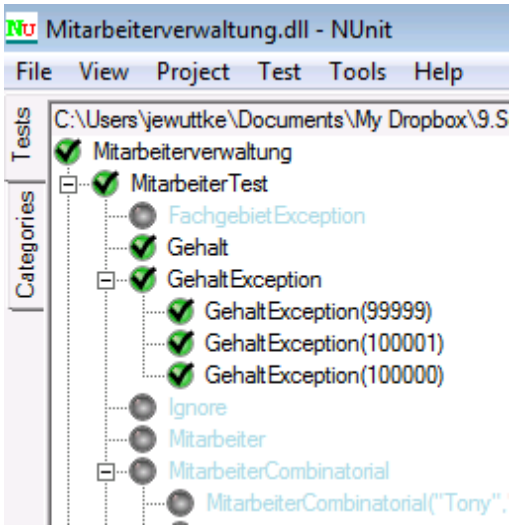

Abbildung 8: Ausführung mit aktiven Kategorien