Autor: Thomas Reinwart

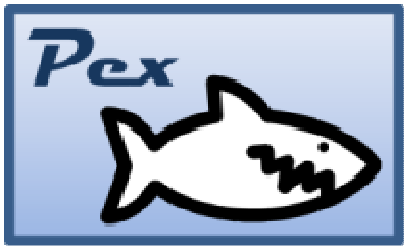
2009-03-11

office@reinwart.com

Pex - automatisches white box testen mit .net

Inhalt

Pex – Automatisches White Box Testen in .net



Ein Teil der Sicherstellung des Qualitätslevels bei der Entwicklung von Programmen wird mit Unit Tests bei der Entwicklung selber abgedeckt. Mit Unit Tests lässt sich das Verhalten von Methoden in Komponenten testen. Das passiert entweder parallel zur Entwicklung (XP - Extreme programming und TDD test driven development) oder im Nachhinein vom Ersteller des zu testenden Codes. Der Test selber ist eine parameterlose Methode, der die zu testende Methode (kann Übergabeparameter haben) aufruft, das Ergebnis bzw. Verhalten der Methode mit Assert auswertet. (Einem Vergleich von erwartetem Ergebnis und dem Testergebnis)

Folgendes Problem ergibt sich aber: Wie stelle ich sicher, dass ausreichend und vor allem sinnvolle Tests erstellt werden? Mittels Code Coverage (kann mit externen Tools oder im Visual Studio selber ermittelt werden) lässt sich die Abdeckung, für welche Teile des Sourcecodes bereits Unit Tests vorhanden sind, feststellen. Ein Code Coverage Wert von 80% ist akzeptabel, 100% sind optimal. Aufgrund von komplexen Zusammenhängen ist es oft nicht einfach, alle notwendigen Varianten der Tests zu erkennen. Zudem ist es aufwendig, alle möglichen Varianten der Übergabeparameter der zu testenden Methode im Unit Test zu implementieren. Vom Entwickler wurden meist eine oder zwei Tests erstellt, die er für den Zeitpunkt der Implementierung für sinnvoll erachtet hat. (Dass er keine Lust hatte weitere Tests zu schreiben schließe ich jetzt mal aus).

Bei einem Projekt schaut das dann so aus: Da der erstellte Code durch die Codecoverage einen positiven Abdeckungsgrad erhalten hat, wurde das Modul von keinem beanstandet und die Komponente released.

Monate später ergibt sich in diesem positiv unit getesteten Modul ein Fehler. Wie konnte das passieren? Genau dieser aufgetretene Fall der übergebenen Parameter wurde bei der Testerstellung einfach nicht bedacht und nie aufgerufen. Ganz nach dem XP Ansatz wird nun der Unittest erstellt, mit dem nun diesen Fehler reproduziert wird. Anschießend wird der Code gefixt, bis der erstellte Unit Test ein positives (grünes) Ergebnis zeigt.

D.h. ich benötige von Anfang an sinnvolle Tests, die alle Varianten abdecken. Dafür gibt es eine Möglichkeit.

1. Pex – automatisches white box testen mit .net

PEX steht für "**P**rogramming **EX**plorations" und ist ein Projekt von Microsoft Research. Es ermöglicht die automatische Generierung von Whitebox Unit Tests. PEX besteht aus einem GUI, das in Visual Studio integriert ist. Ebenso können Command Line Befehle genutzt werden.

Eine sehr ausführliche Dokumentation, Tuturial und Samples gibt es online auf der Homepage des Herstellers.

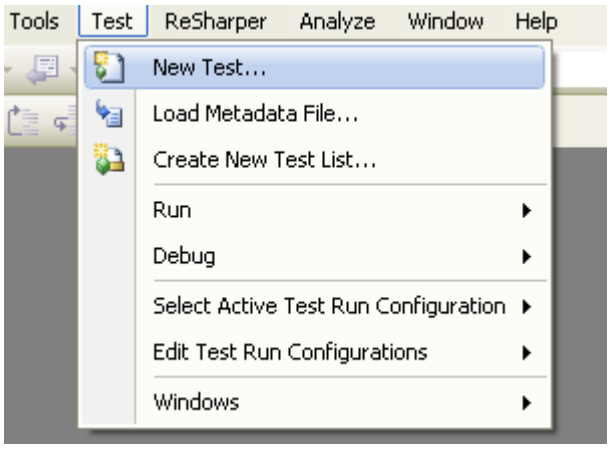
Ein „Whitebox Test" generiert Test Cases indem es den Program Code analysiert. Es sucht dabei nach verschiedenen Fehlerquellen. Als Übergabeparameter an die zu testende Methode werden auch jene Werte übergeben, an die man mit Sicherheit nicht gedacht hat. PEX sucht systematisch nach möglichen Fehlern, und erstellt Tests dafür. Das Ergebnis sind relativ wenige aber dafür sinnvolle Unit Tests.

1.1 Bisherige Möglichkeiten Unit Test zu erstellen

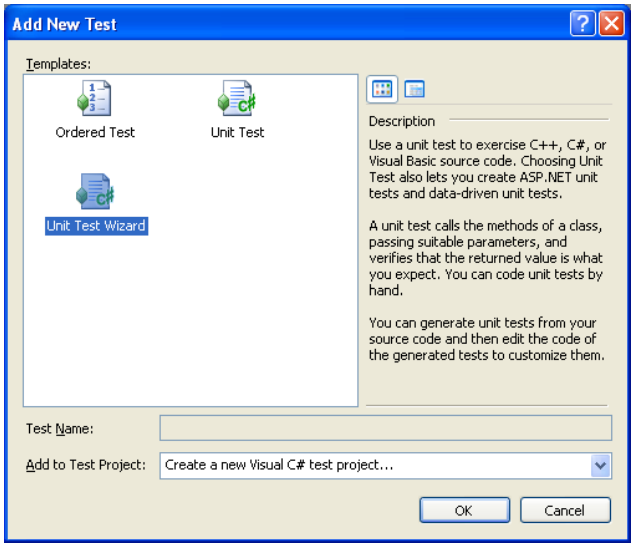
Wie wurden die Unit Tests bisher erstellt:

Visual Studio 2003/2005 und externe Unit Test Tools (z.B. Nunit) : alle Test wurden manuell selber codiert. Zur Ermittlung der Unit Test Abdeckung wurde z.B. Ncover benutzt.

Visual Studio 2008: Schon besser - hiermit lässt sich das Gerüst für einen einzelnen Unit Tests erstellen, indem man auf der zu testenden Methode im Contextmenü „Generate Unit Test" klickt. Die Test Abdeckung ist im Studio zu erkennen.



Menü Test: new Test



Legt ein neues Testprojekt in der sln an.

Test Klasse ins Projekt hinzufügen: Context Menü „Create Unit Test“

```
public class Tools
{
    public static string ConvertStrings(string one, string two)
    {
        string result = one[5].To
        result += two.Substring(0
        return result;
    }
}
```

Bsp eines erzeugten Unit Test Gerüstes:

```
    /// <summary>
///A test for ConvertStrings
///</summary>
[TestMethod()]
public void ConvertStringsTest()
{
    string one = string.Empty; // TODO: Initialize to an appropriate value
    string two = string.Empty; // TODO: Initialize to an appropriate value
    string expected = string.Empty; // TODO: Initialize to an appropriate value
    string actual;
    actual = Tools.ConvertStrings(one, two);
    Assert.AreEqual(expected, actual);
    Assert.Inconclusive("Verify the correctness of this test method.");
}
```

1.2 XP und Pex

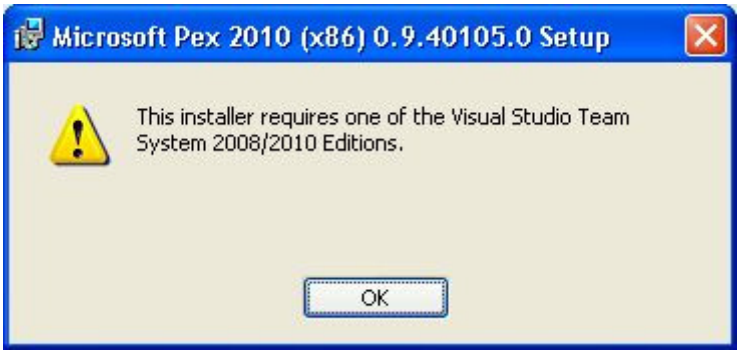
Wie passt nun XP (Exteme programming) und Pex zusammen?

XP definiert nicht, wie die Unit Test zu schreiben sind. Die bisherigen Unit Test Frameworks (OpenSource Nunit bzw. Visual Studio selber) führen die Unit Test bloß aus, es gab bisher nicht die Möglichkeit automatisiert wirksame Tests zu generieren.

1.3 Installation von Pex

- Windows Vista 32bit / 64bit
  Other versions of Windows (XP, W2K3, W2K8) should work but are untested.
  On x64, only 32-bit (Wow64) processes are supported.
- .NET Framework 2.0, 3.0, 3.5, 4.0

PEX Installations- Voraussetzung:



Pex kann nicht in der Standard Edition sondern nur in der Team Edition von Visual Studio installiert werden.

Lizenz:

DevLabs Pre-Release

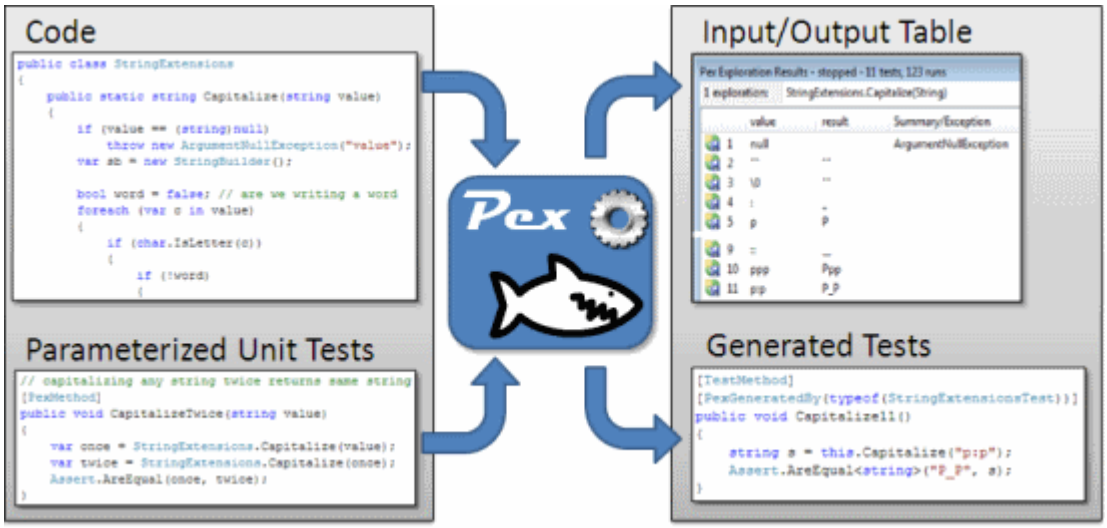
License: Microsoft Pre-Release Software License,
Pre-Release License Agreement, Commercial Use Allowed.

Academic Release

1.4 Funktionsweise

Pex erzeugt von der Anzahl her wenige aber dafür wirksame Unit Tests. Wirksam in der Hinsicht, das eine hohe Code coverage abgedeckt wird. Zudem kann jeder generierte Test in einen Test Case gespeichert werden um diesen debuggen zu können.

Pex führt den zu testenden Code mehrfach mit unterschiedlichen Parametern aus. Je nachdem wie gut der eigene Code ist, kann es nun auch zu Effekten kommen. Etwa wenn dahinter eine Datenbank oder Filessystem genutzt wird. Daher Pex nicht gegen ein Echtsystem laufen lassen sondern im Test Environment bleiben.



Grafik von http://research.microsoft.com/en-us/projects/Pex/

1.5 Unit Test Frameworks

Die Methode `ConvertStrings` aus der Klasse Tools soll getestet werden:

```
namespace PexSample
{
    public class Tools
    {
        public static string ConvertStrings(string one, string two)
        {
            string result = one[2].ToString();
            result += two.Substring(0, 3);
            return result;
        }
    }
}
```

Überblick über die unterschiedlichen Unit Test Framework um diesen Code zu testen:

Microsoft Visual Studio

Projektreferenz:	Microsoft.VisualStudio.QualityTools.UnitTestFramework
Namespace:	using `Microsoft.VisualStudio.TestTools.UnitTesting;`
Attribut Testklasse:	`[TestClass]`
Attribute Testklasse Konstructor:	`[ClassInitialize()],[TestInitialize()]`
Attribute Testklasse DeKonstructor:	`[ClassCleanup()],[TestCleanup()]`
Attribute Testmethode:	`[TestMethod]`

Bsp Code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace PexSample.VisualStudio.Test
{
    [TestClass]
    public class TestVisualStudioPexSample
    {
        [TestMethod]
        public void _1_Test()
        {
            string result = PexSample.Tools.ConvertStrings("Hello", "world");
            Assert.IsNotNull(result, "Result is empty.");
        }
    }
}
```

OpenSource NUnit

Projektreferenz:	nunit.framework.dll
Namespace:	`using NUnit.Framework;`
Attribut Testklasse:	`[TestFixture]`
Attribute Testklasse Konstructor:	`[SetUp] [TestFixtureSetUp]`
Attribute Testklasse DeKonstructor:	`[TearDown] [TestFixtureTearDown]`
Attribute Testmethode:	`[Test]`

Bsp Code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using NUnit.Framework;


namespace PexSample.Nunit.Test
{
    [TestFixture]
    public class ToolsTest
    {
        [Test]
        public void _1_Test()
        {
            string result = PexSample.Tools.ConvertStrings("Hello", "world");
            Assert.IsNotNull(result, "Result is empty.");
        }
    }
}
```

2. Pex Anwenden

2.1. Sample 1

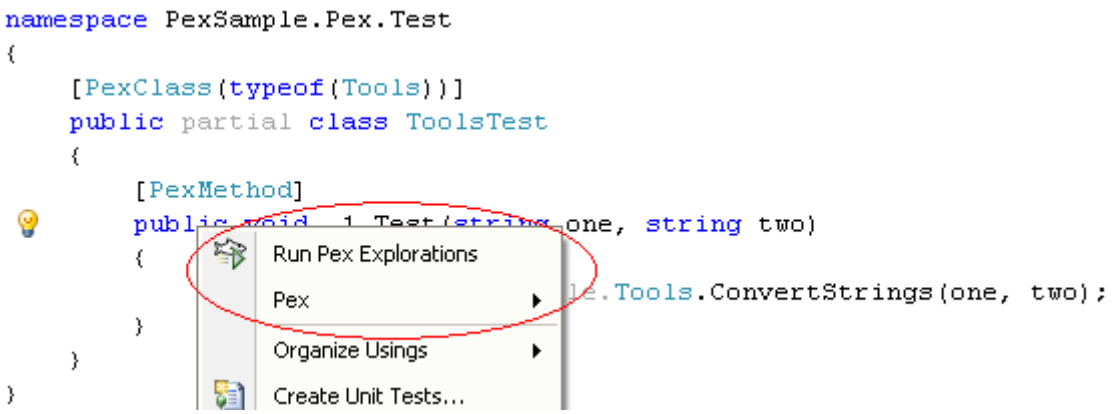
Der Unterschied zu den herkömmlichen Unit Tests ist neben den Referenzen und Attributen die Möglichkeit, beim Unit Test auch Parameter zu übergeben.

Projektreferenz:	Microsoft.Pex.Framework.dll einbinden in das Testprojekt einbinden.
Namespace:	`using Microsoft.Pex.Framework;`
Attribut Testklasse:	`[PexClass (typeof(TestClass))]`
Attribute Testmethode:	`[PexMethod]`

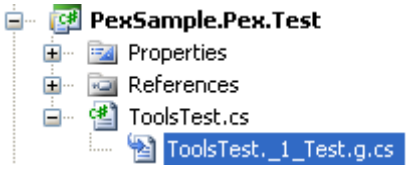
Bsp Code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Pex.Framework;

namespace PexSample.Pex.Test
{
    [PexClass(typeof(Tools))]
    public partial class ToolsTest
    {
        [PexMethod]
        public void _1_Test(string one, string two)
        {
            string result = PexSample.Tools.ConvertStrings(one, two);
        }
    }
}
```

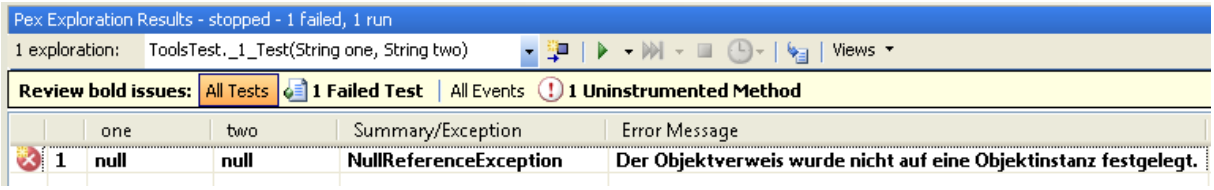
Im Contextmenü der Testmethode findet man *Run Pex Explorations* um damit die Tests zu starten.



Die `ConvertStrings` Methode wird analysiert, die Tests werden nun automatisch erstellt. Die erstellten Tests setzen auf denen von VisualStudio.TestTools auf. D.h. in das Testprojekt muss auch die Refernz auf *Microsoft.VisualStudio.QualityTools.UnitTestFramework* eingebunden sein.



Es wurde ein Fehlerfall entdeckt:



Pex hat erkannt, das die Parameterüber null, null einen Fehler in ConvertStrings liefert.

Die erstellte Klasse:

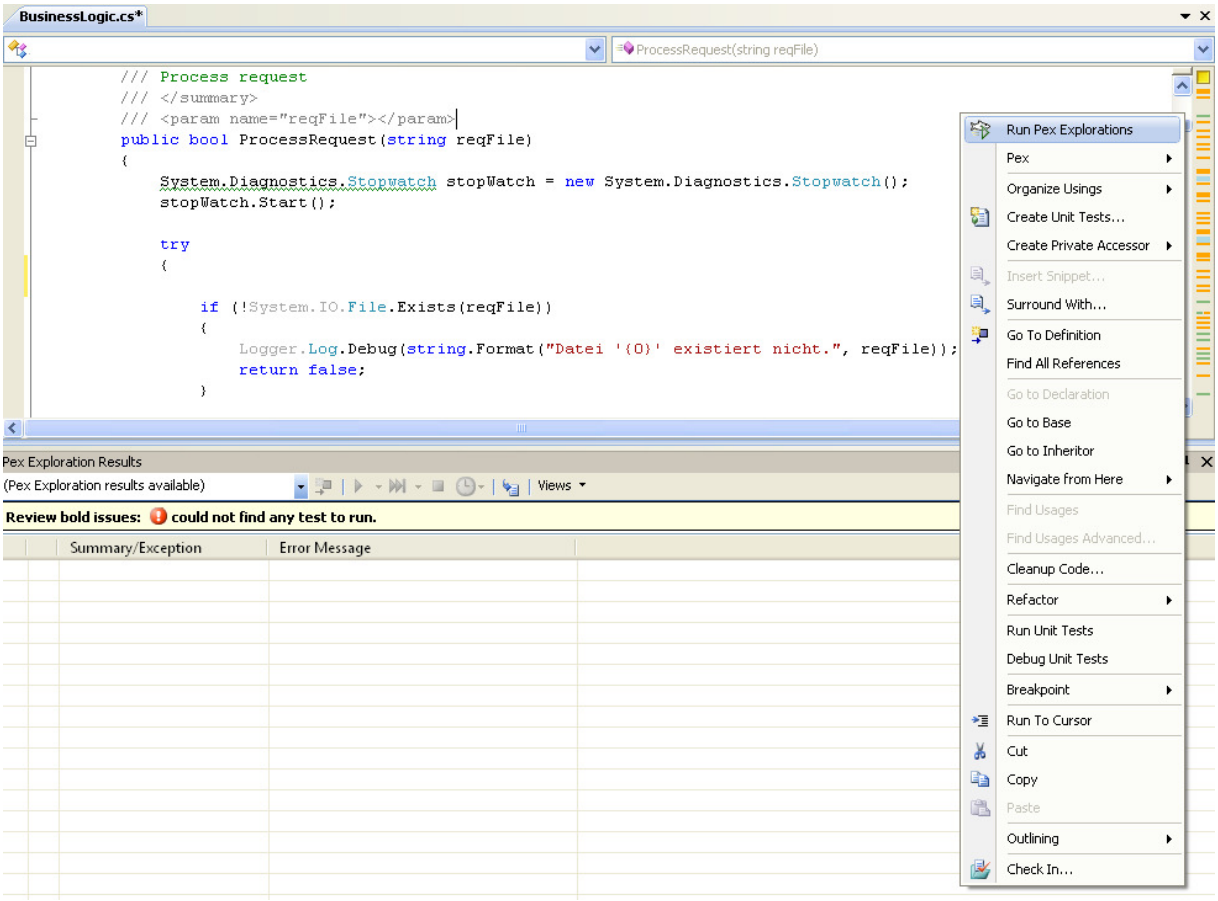
```
// <copyright file="ToolsTest._1_Test.g.cs" company="MyCompany">Copyright © by ThisProject
2009</copyright>
// <auto-generated>
// This file contains automatically generated unit tests.
// Do NOT modify this file manually.
//
// When Pex is invoked again,
// it might remove or update any previously generated unit tests.
//
// If the contents of this file becomes outdated, e.g. if it does not
// compile anymore, you may delete this file and invoke Pex again.
// </auto-generated>
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using Microsoft.Pex.Framework.Generated;

namespace PexSample.Pex.Test
{
    public partial class ToolsTest
    {
        [TestMethod]
        [PexGeneratedBy(typeof(ToolsTest))]
        [PexRaisedException(typeof(NullReferenceException))]
        public void _1_Test01()
        {
            this._1_Test((string)null, (string)null);
        }
    }
}
```

2.2. Sample 2

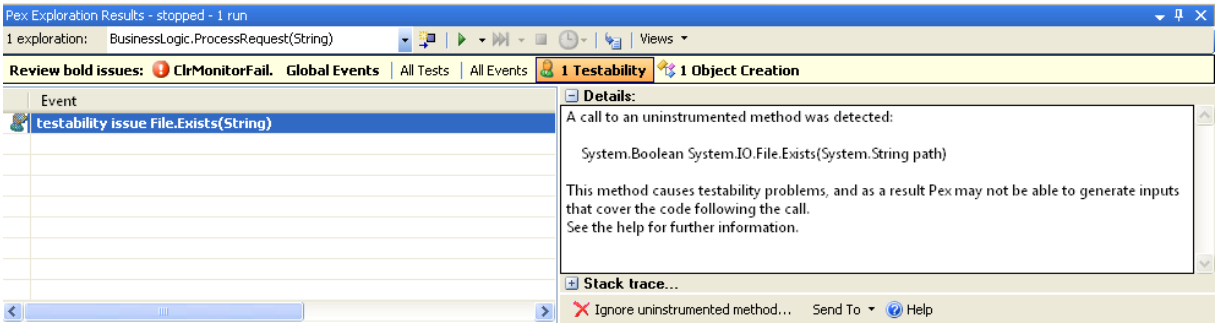
Was bringt mir das in der Praxis - Beispiel des Einsatzes der PEX Exploration im bestehenden Code:

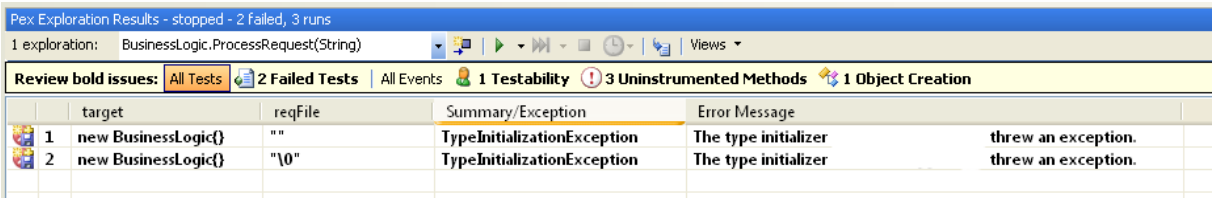
Bei dieser Methode einer Klasse wird eine Datei als Parameter übergeben. Es gibt die Abfrage *File.Exists* im Code, auf den ersten Blick sieht der Code sauber aus. Die bestehenden Unit Test, die Mock Files als Parameter verwenden, sind alle fehlerfrei. Was erkennt PEX, gibt es doch noch Fehler im Code?

Auf der Methode selber starte ich im Contextmenü „Run Pex Explorations“ und warte gespannt einige Sekunden.

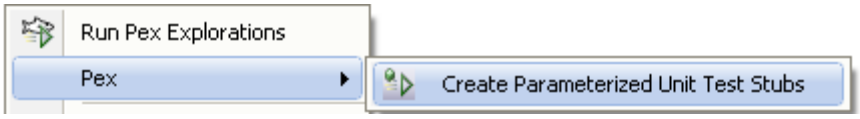


PEX hat einen Fehler gefunden:





Im Output sehe ich, das der bloße Aufruf mit einem leeren Parameter (null) zu einem Fehler führt, nämlich bei der Überprüfung File Exists. Um diesen Test in meine selbsterstellten Unit Tests gemäß dem XP Ansatz (Unit Test für den Fehler schreiben, dann beheben) aufzunehmen, kann ich mir den Test generieren lassen, den Fehler beheben und somit sicher stellen, dass er kein weiteres Mal auftritt. Oder ich erkenne den Fehler nicht sofort und muss ohnehin debuggen, was ich erst im generierten Test kann. (Create parameterized unit test stubs)



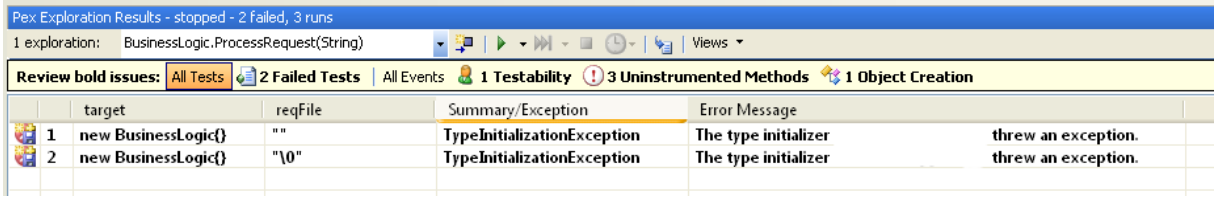
Der von PEX erstellte Code:

```
/// <summary>This class contains parameterized unit tests for BusinessLogic</summary>

  [TestClass]
  [PexClass(typeof(BusinessLogic))]
  [PexAllowedExceptionFromTypeUnderTest(typeof(ArgumentException), AcceptExceptionSubtypes = true)]
  [PexAllowedExceptionFromTypeUnderTest(typeof(InvalidOperationException))]
  public partial class BusinessLogicTest
  {
    /// <summary>Test stub for ProcessRequest(String)</summary>
    [PexMethod]
    public bool ProcessRequest([PexAssumeUnderTest]BusinessLogic target, string reqFile)
    {
      // TODO: add assertions to method BusinessLogicTest.ProcessRequest(BusinessLogic, String)
      bool result = target.ProcessRequest(reqFile);
      return result;
    }
  }
```

Der Fehler wird nun in der Methode „*ProcessRequest*“ selber behoben.

```
public bool ProcessRequest(string reqFile)
    {
      System.Diagnostics.Stopwatch stopWatch = new System.Diagnostics.Stopwatch();
      stopWatch.Start();

      try
      {
        // Bugfix
        if (string.IsNullOrEmpty(reqFile))
        {
          Logger.Log.Debug("Es ist keine Datei für die Verarbeitung angegeben.");
          return false;
        }

        if (!System.IO.File.Exists(reqFile))
        {
          Logger.Log.Debug(string.Format("Datei '{0}' existiert nicht.", reqFile));
          return false;
    }
```

Wie testet PEX?



Um nicht alle möglichen Zeichen (65535) durchzutesten (das würde zu lange dauern), testet PEX jeweils ein Zeichen eines bestimmten Typs. (Also Klein, Gross usw.) Wenn hier Fehler auftreten, wird der Wert der den Fehler verursacht, mit der Art der Exception aufgelistet. In den Details ist dann die Stelle im Code zu erkennen.

3. Links

http://research.microsoft.com/en-us/projects/Pex/

http://research.microsoft.com/en-us/people/jhalleux/

4. Fazit

Mit dieser Technik wird das Testen wesentlich effektiver und die Komponenten sind viel besser getestet. Der Vorteil zu den selbsterstellten Unit Tests ist, dass Pex eine Codeanalyse des zu testenden Codes vornimmt und wirksame Tests generiert, also beispielsweise mit den Randbereichen der Übergabeparameter. Somit ist der erstellte Code automatisch wesentlich besser und schneller getestet, als dies bisher manuell möglich ist.