Softwaretests in Visual Studio 2010

Eine Anleitung für die Sprache C#

Alexander Schunk, Marcel Teuber, Henry Trobisch

Inhalt

1. Vorbereitung

Um Softwaretests in Visual Studio 2010 zu erstellen und durchzuführen, gehört zur Vorbereitung die Installation von Visual Studio 2010 Ultimate. Da in diesem How-To auch Testmöglichkeiten mit NUnit aufgezeigt werden, sollte diese Software ebenfalls mit installiert werden. Zur Installation benötigt man einfach nur den „msi“ Installer von www.nunit.org, hier genutzt in der Version 2.5.9. Da das Framework in der Vollinstallation nur 7MB benötigt wurde es vollständig auf der Festplatte installiert.

Beim ersten Start von Visual Studio wird erfragt, welche Ansicht der Benutzer haben möchte. Da die folgenden Beispiele alle in C# geschrieben sind, wird auch die C#-Ansicht ausgewählt. Die mitgelieferten Testmöglichkeiten sind in Visual Studio aber auch für andere Programmiersprachen gegeben.

2. Unit-Tests in Visual Studio

2.1 Erstellung von Unit-Tests

Alle in Visual Studio gebotenen Testmöglichkeiten, mit Ausnahme der Leistungsanalyse sowie NUnit, finden sich unter dem Menüpunkt „Test“. Hier wird eine Übersicht über alle auswählbaren Tests angezeigt. Bei der Erzeugung des ersten Tests, egal welcher Art. wird von Visual Studio automatisch ein zu benennendes Testprojekt erstellt. Dieses Testprojekt wiederrum beinhaltet Testlisten, eine solche wird mit dem Testprojekt zusammen erstellt, lediglich die Eingabe eines Namens sowie einer Beschreibung ist vonnöten. Diesen Testlisten, können beliebig viele Testklassen hinzugefügt werden, welche die eigentlichen Testmethoden enthalten. Die Struktur der Tests sieht daher wie folgt aus:

Testprojekt

- Testliste1
    - Testklasse1
        - Testmethode1
        - Testmethode2
    - Testklasse2
        - Testmethode1
- Testliste2
    - Testklasse3
        - Testmethode1
        - Testmethode2

Die für die Unit-Tests relevanten Möglichkeiten sind zum einen der „einfache Komponententest“, der „Komponententest“ sowie der „Komponententest-Assistent“.

Der einfache Komponententest erzeugt die nötige Grundstruktur für Komponententests, gibt allerdings keine Beispielstruktur vor und ist daher nur für Anwender sinnvoll, welche bereits Unit-Tests in Visual Studio erstellt haben und die entsprechend nötige Syntax kennen.

```
[TestClass]
  public class UnitTest1
  {
    [TestMethod]
    public void TestMethod1()
    {
    }
  }
```

Mit dem „Komponententest“ werden neben der Grundstruktur auch hilfreiche Kommentare sowie eine leere Methode für Quelltext erstellt, welcher vor den nachfolgenden Tests ausgeführt werden soll, erzeugt. Darüber hinaus wird eine leere Beispieltestmethode erstellt. Der Komponententest eignet sich somit dafür, gezielt Unit-Tests mit Beispielstruktur für bestimmte Klassen zu erstellen, welche dann nach eigenen Anforderungen gestaltet werden können.

```
[TestClass]
  public class UnitTest2
  {
    public UnitTest2()
    {
      //
      // TODO: Konstruktorlogik hier hinzufügen
      //
    }

    [TestMethod]
    public void TestMethod1()
    {
      //
      // TODO: Testlogik hier hinzufügen
      //
    }
  }
```

Der Komponententest-Assistent erstellt für selbst ausgewählte, bisher geschriebene Klassen und deren Methoden Testklassen mit Beispielcode, wobei jede Klasse einen eigenen Unit-Test erhält. Dieser Assistent eignet sich daher für Einsteiger, welche generell noch keine Unit-

Tests erstellt haben und jene, welche mit Visual Studio 2010 diesbezüglich noch keine Erfahrung gesammelt haben. Für die folgenden Beispiele wurde dieser Assistent gewählt.

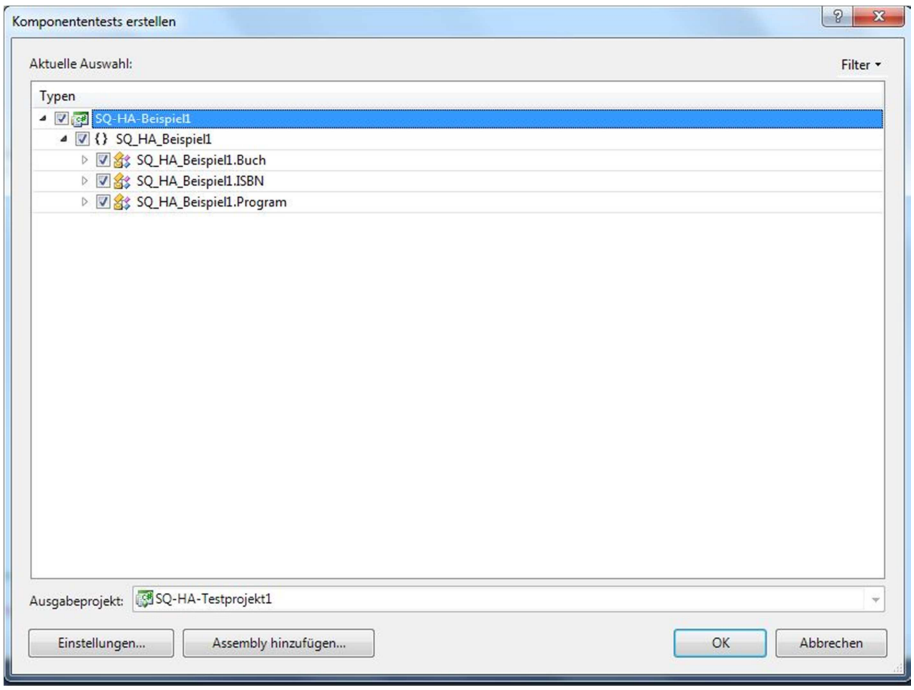


Abbildung 1

2.2 Schreiben von Unit-Tests

Wichtig ist, dass vor jedem Test [TestMethod()] steht, da sonst der Test in der Testliste nicht auftaucht und somit auch nicht ausgeführt werden kann.

```
[TestMethod()]
public void ISBNConstructorTest()
      {
         string ISBN = "978383481417"; // Pruefziffer weglassen, wird automatisch berechnet
         ISBN target = new ISBN(ISBN);
         String actual = target.Isbn13;
         Assert.AreEqual("9783834814173", actual);
      }
```

Des Weiteren bietet Visual Studio die Möglichkeit, vor und nach allen Tests sowie vor und nach einzelnen Tests Code auszuführen. Am Beispiel eines einzelnen Tests sieht das wie folgt aus:

```
public class ISBNTest
   {
      private ISBN target;

      [TestInitialize()]
```

```
public void ISBN10Eingabe()
{
   String isbNummer = "383481417"; // einzulesende ISBN10 Nummer
   target = new ISBN(isbNummer);
}

[TestMethod()]
[DeploymentItem("SQ-HA-Beispiel1.exe")]
public void ISBN10EingabeTest()
{
   target.berechnePruefziffer();
   String actual = target.Isbn13;
   Assert.AreEqual("9783834814173", actual);

}
```

Der vorher auszuführende Code kann einfache Textausgaben beinhalten, jedoch auch, wie im obigen Beispiel zu sehen, für die Erzeugung von Objekten und Übergabeparametern, um den Test zu strukturieren.

2.3 Ausführen von Unit-Tests

Mit der Erstellung der Unit-Tests können diese noch nicht ausgeführt werden. Hierfür müssen in Visual Studio Testlisten verwendet werden. Dazu wird zuallererst eine Testliste erstellt, wie im untenstehenden Bild zu sehen ist.

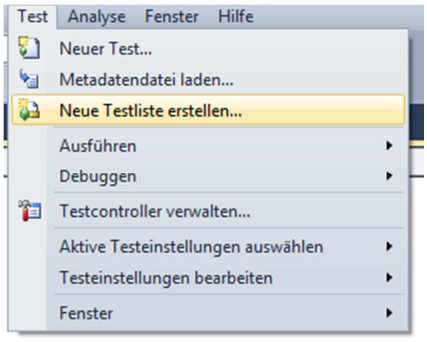


Abbildung 2

Hier wird ein Name für die Testliste erstellt sowie eine Beschreibung für die Testliste. Mit der Testlistenhierarchie können die Testlisten strukturiert werden. Bei Erstellung kann daher ausgewählt werden, wo diese Liste platziert werden soll.

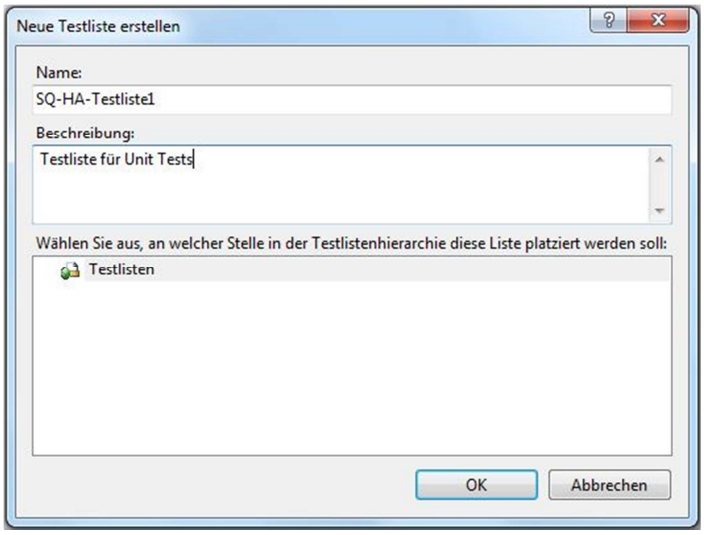


Abbildung 3

Nach Erstellung der Testliste werden nach einem Klick auf den „Aktualisieren“ Knopf alle bisher erstellten Tests in der Rubrik „Alle geladenen Tests“ aufgeführt. Diese können dann per Drag & Drop in die Testliste gezogen werden.

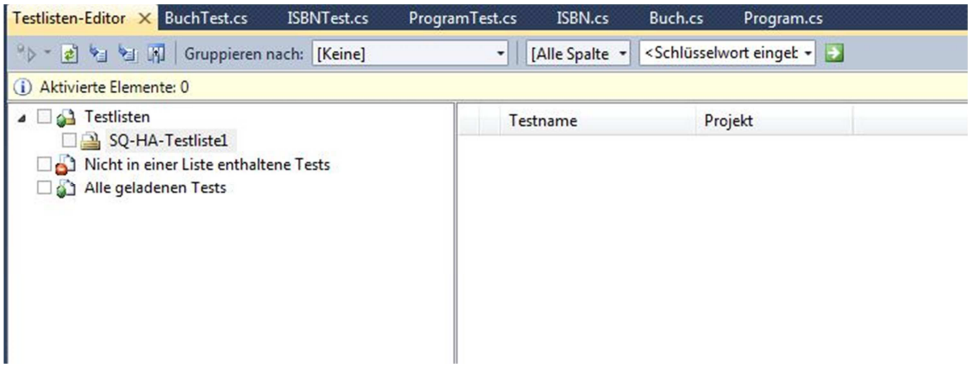


Abbildung 4

Um die Tests dann auszuführen, werden die relevanten Tests dann mit einem Haken versehen und anschließend auf „Aktivierte Tests durchführen“ geklickt. Nun werden alle Tests ausgeführt und das Ergebnis der Tests im Fenster „Testergebnisse“ angezeigt.

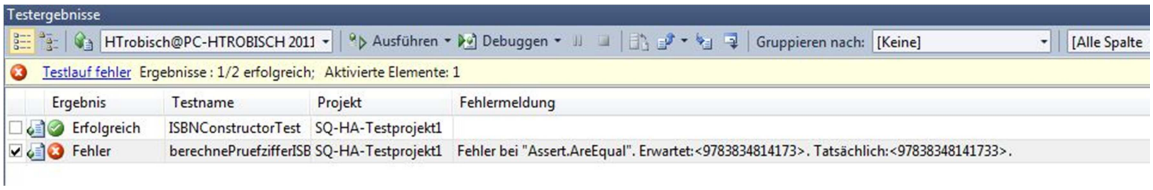


Abbildung 5

3. Unit-Tests mit NUnit

3.1 Erstellen von Unit-Tests

Um NUnit in Visual Studio 2010 nutzen zu können muss ein Testobjekt angelegt werden. Da NUnit ein externes Framework ist muss als Projekttyp „Klassenbibliothek“ gewählt werden.

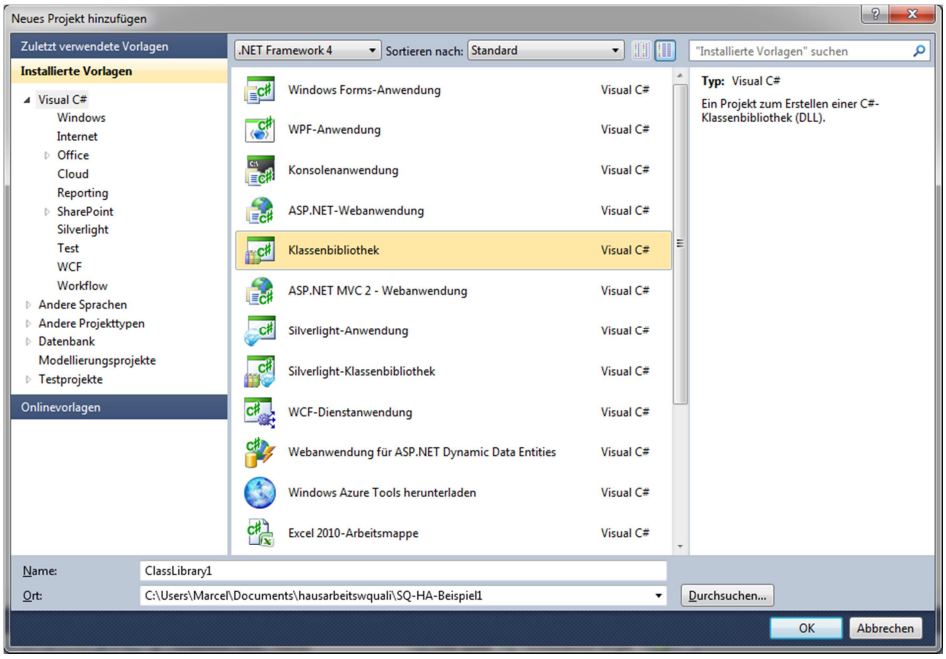


Abbildung 6

Als nächstes fügt man dem Projekt einen Verweis auf „nunit.framework“ und „nunit.mocks“ hinzu.

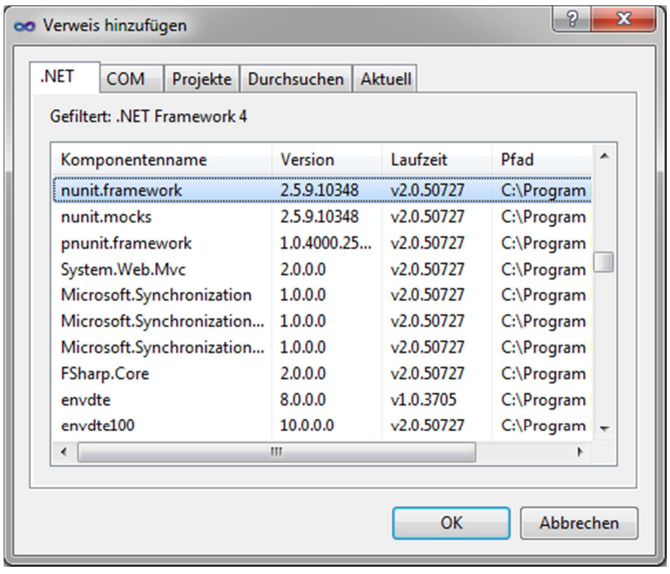


Abbildung 7

Um NUnit nun in Visual Studio nutzen zu können muss das Projekt angepasst werden. Dazu klickt man mit der rechten Maus auf das NUnit Projekt und geht auf Eigenschaften dort dann auf Debuggen und wählt bei Startaktion „Externes Programm starten“. Hier wählt man den Pfad zur „nunit.exe“ aus.

Sollte die Visual Studio 2010 als 32Bit Version installiert sein, muss zwingend die „nunit-x86.exe“ ausgewählt sein da es sonst nicht möglich ist NUnit auszuführen!

Unter Befehlszeilenargumente schreibt man die DLL den NUnit Projekts gefolgt mit /run. Und unter Arbeitsverzeichnis wählt man das Verzeichnis wo das Programm kompiliert werden soll.

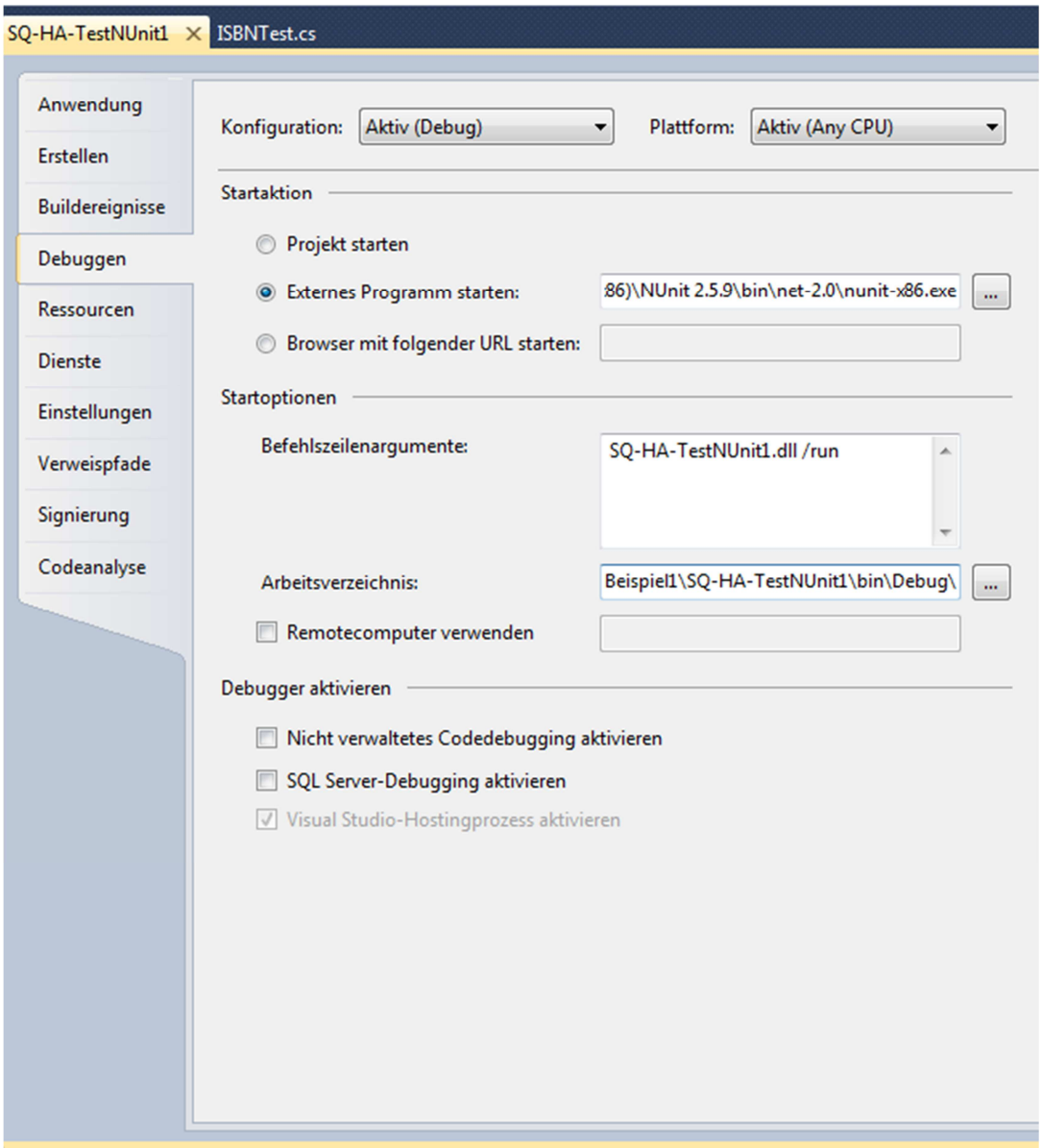


Abbildung 8

3.2 Ausführen von Unit-Tests

Durch diese Einstellung startet automatisch wenn das NUnit Projekt kompiliert der NUnit Test Runner:

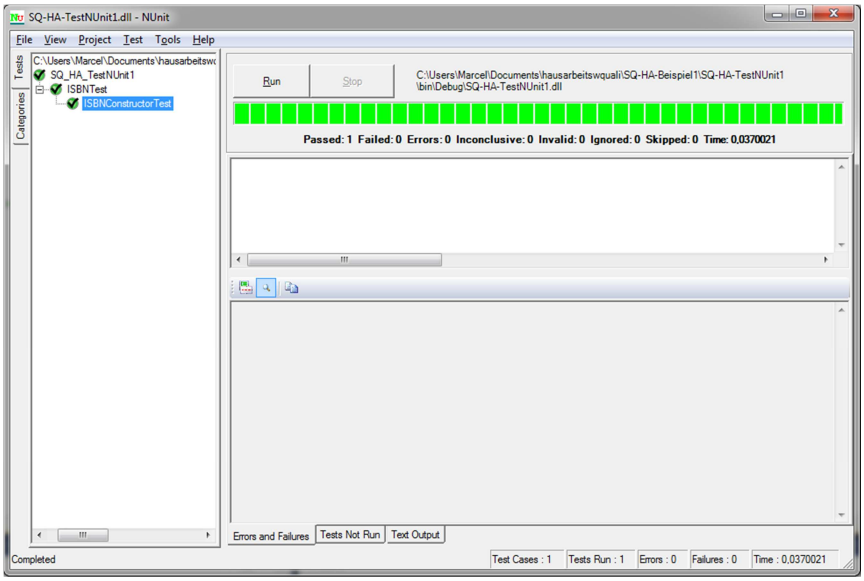


Abbildung 9

3.3 Anmerkungen zur Syntax

Vor einer Testklasse schreibt man „TestFixture“ um zu signalisieren das diese Klasse getestet werden soll. Vor einer Testfunktion wird dann nur der Vermerk „Test“ gesetzt wie z.B. für JUnit in Java. Abbildung 10 zeigt, wie dies innerhalb des Quelltextes aussieht.

```
1  using SQ_HA_Beispiel1;
2  using System;
3  using System.Collections.Generic;
4  using System.Linq;
5  using System.Text;
6  using NUnit.Framework;
7
8
9
10 namespace SQ_HA_TestNUnit1
11 {
12     [TestFixture]
13     public class ISBNTest
14     {
15         [Test]
16         public void ISBNConstructorTest()
17         {
18             string ISBNn = "978383481417";
19             ISBN target = new ISBN(ISBNn);
20             String actual = target.Isbn13;
21             Assert.AreEqual("9783834814173", actual);
22         }
23     }
24 }
25
```

Abbildung 10

4. Mocking mit NUnit

Da das Mocking in Visual Studio 2010 nicht unterstützt wird, wird dies nur anhand von NUnit gezeigt. Gemockt wird eine Suchfunktion die nicht implementiert wurde. Als Rückgabe wird einfach ein Eintrag aus der „Datenbestand“ Klasse zurückgegeben. Im SetUp Teil wird zunächst ein Mock Objekt erstellt das den Typ der zu Mockenden Klasse als Interface erwartet. In der Testmethode wird die aufzurufende Methode dem Mock Objekt hinzugefügt sowie der Rückgabewert und die Parameter mit angegeben die erwartet werden. Als nächstes wird eine „MockInstance“ dem Datenbestand Objekt als „Listensucher“ Klasse vorgetäuscht. Nun kann man auf die „MockInstance“ die Funktion aufrufen.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using SQ_HA_Beispiel1;
6  using NUnit.Framework;
7  using NUnit.Mocks;
8
9  namespace SQ_HA_TestNUnit1
10 {
11     [TestFixture]
12     public class MockingTest
13     {
14         private DynamicMock ListensucherMock;
15         private Buch buch1 = new Buch("978383481417", "Buch 1 Titel", "Buch 1 Beschreibung");
16         private Buch buch2 = new Buch("978383481416", "Buch 2 Titel", "Buch 2 Beschreibung");
17         private Buch buch3 = new Buch("978383481415", "Buch 3 Titel", "Buch 3 Beschreibung");
18
19         private List<Buch> liste;
20
21         [SetUp]
22         public void TestInit()
23         {
24             ListensucherMock = new DynamicMock(typeof(IListensucher));
25             liste = new List<Buch>();
26             liste.Add(buch1);
27         }
28
29         [Test]
30         public void Suchtest()
31         {
32             ListensucherMock.ExpectAndReturn("search", liste, 0, "978");
33             Datenbestand test = new Datenbestand();
34             test.sucherobj = (IListensucher)ListensucherMock.MockInstance;
35
36             Assert.AreEqual(1, (test.sucherobj.search(0, "978")).Count);
37         }
38     }
39 }
```

Abbildung 11

5. Codeabdeckungstests in Visual Studio

5.1 Einrichten der Codeabdeckung

Codeabdeckungstests sind auch standardmäßig in Visual Studio 2010 Ultimate enthalten, allerdings nicht aktiviert. Daher muss die entsprechende Option erst angestellt werden. Diese ist über den Menüpunkt Test in den Testeinstellungen zu finden.

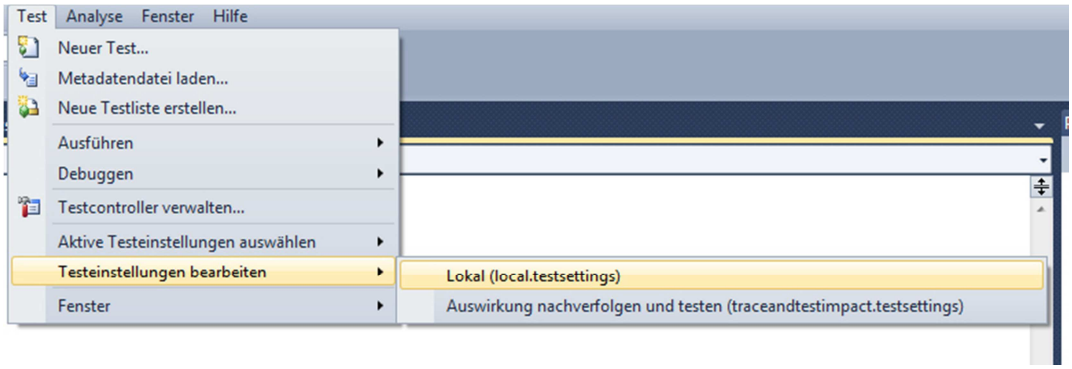


Abbildung 12

Im nun geöffneten Fenster kann unter dem Menüpunkt „Daten und Diagnose“ nun die Codeabdeckung aktiviert werden. Mit der Option „Konfigurieren“ wird noch angegeben, für welche Projekte die Codeabdeckungsanzeige verwendet werden soll. Diese Schritte müssen für jedes neue Projekt wiederholt werden, da diese Einstellungen nicht global in Visual Studio gespeichert werden.

5.2 Ausführen der Codeabdeckung

Die Codeabdeckung wird bei nun erneut ausgeführten Tests gleich mit angegeben. Dies geschieht sowohl mit einem eigenen Karteireiter bei den Testergebnissen, welcher die Abdeckung prozentual angibt. Auch wird nun der eigentliche Quellcode, sofern dieser nicht abgedeckt ist, rot unterlegt.

6. Leistungsanalyse in Visual Studio

Die Leistungsanalyse bezeichnet in Visual Studio die Ermittlung der am häufigsten verwendeten Methoden sowie die Diagnose von Leistungsproblemen des analysierten Programms. Als Voraussetzung für Leistungsanalyse in Visual Studio muss das zu analysierende Programm in ausführbarer Form vorliegen. Wenn dies der Fall ist, kann über den Menüpunkt Analyse eine neue Leistungsanalyse gestartet werden. Hier wird unterschieden zwischen CPU-Sampling, Instrumentation, .NET-Speicherbelegung sowie Parallelität.

6.1 CPU-Sampling

Mit der CPU-Sampling Leistungsanalyse, welche auch von Visual Studio als erste Leistungsanalyse empfohlen wird, kann die Prozessorauslastung des zu analysierenden Programms ermittelt werden. Hierbei ist nur die Auswahl des zu analysierenden Projektes nötig, dann startet bereits die Analyse. Nach der Durchführung zeigt Visual Studio die Analyse sowohl grafisch auf einem Auslastungs-Zeit-Diagramm an als auch in Listenform mit prozentualer Angabe, welch Funktion welche Auslastung verursacht, angezeigt. Die speicherintensivste Methode wird mit komplettem Aufrufpfad angezeigt, durch Klick auf jene Methode wird angezeigt, welche Zeile in dieser Methode die speicherintensivste ist.

Abbildung 13

Langsamster Pfad

Der speicherintensivste Aufrufpfad basierend auf Ausführungszeiten

Name	Inklusiv %	Exklusiv %
SQ_HA_Beispiel2.ProfileList.profile(int32,int32)	99,92	0,02
SQ_HA_Beispiel2.ProfileList.aufbauen()	99,90	0,04
SQ_HA_Beispiel2.DataHolder..ctor(int32)	99,86	29,93
System.String.Concat(object,object)	**59,50**	**59,50**
System.Collections.Generic.List`1.Add(!0)	**10,40**	**10,40**

Verknüpfte Ansichten: Aufrufstruktur Funktionen

Abbildung 14

6.2 Instumentation

Mittels der Instrumentationsanalyse zeigt Visual Studio an, wie oft welche Methoden aufgerufen wurden und wie hoch der zeitliche Aufwand jeder Methode ist. Auch hierfür ist lediglich die Angabe des Projektes nötig, welches analysiert werden soll. Ebenfalls werden hier die Ergebnisse der Analyse visuell und textuell dargestellt.

6.3 NET-Speicherbelegung

Die Analyse auf Speicherbelegung ermittelt, wie der Name bereits andeutet, die Speicherbelegung der einzelnen Objekte. Hier werden neben Objekten von selbst erstellten Klassen auch die grundlegenden Datentypen mit angezeigt, welche sich innerhalb der eigenen Klassen befinden. Für diese Analyse ist erneut nur die Angabe des Projektes nötig. Zu beachten ist, dass die Speicherbelegungsanalyse selbst bei kleinen Programmen bereits deutlich länger dauern kann als die beiden oben beschriebenen Analysemöglichkeiten. Dies liegt daran, dass alle erzeugten Objekte in eine Datei geschrieben werden, welche im Projektordner erstellt wird. Je nach Programm kann die so erzeugt Datei sehr umfangreich werden und der Projektordner sollte daher über entsprechend Speicherplatz verfügen.

6.4 Parallelität

Mit der Parallelität kann das Threadverhalten der Software analysiert werden. Auch die Ermittlung von Ressourcenkonflikten kann hiermit durchgeführt werden. Um diese Analyse auszuführen, muss mindestens eine der beiden Auswahlmöglichkeiten, Sammeln der Ressourcenkonfliktdaten bzw. Verhalten der Multithreadanwendung visualisieren, angewählt sein. Auch hier ist dann wieder lediglich die Auswahl des zu analysierenden Projektes nötig. Sollte die Visualisierung ebenfalls ausgewählt sein, so wird Visual Studio fragen, ob die Anwendungsicons der derzeitig laufenden Programme geladen werden sollen. Die Analyse wird aber unabhängig von dieser Auswahl durchgeführt. Die Analyse wird, je nach getroffener Auswahl, einige Zeit in Anspruch nehmen.

7. Oberflächentests in Visual Studio

Abbildung 15 zeigt die zu testende GUI für C# in Visual Studio. Die Oberfläche ist mittels der WPF- Toolbar, welche die GUI-Objekte enthält, erstellt und anhand der von Visual Studio bereitgestellten Test für UIs getestet worden. Es handelt sich hierbei um einen kleinen Taschenrechner der zwei Werte zur Eingabe verlangt und bei dem mittels Listenbox eine Rechenoperation ausgewählt werden kann. Ein Klick auf die Schaltfläche „Rechnen“ führt die Operation aus. Da hier gezielt nur „Integer“ Werte berechnet werden kann man bei Fließkommazahlen sowie Brüchen Fehler feststellen.

Abbildung 15

7.1 Aufzeichnen von Tests

Starten kann man die Aufzeichnung der Test in dem man im Menü unter Test -> Test der codierten UI einen neuen Test anlegt nun öffnet sich das Fenster in dem man seine Tests der GUI aufzeichnen kann.

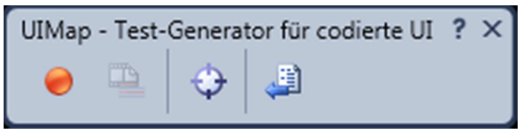


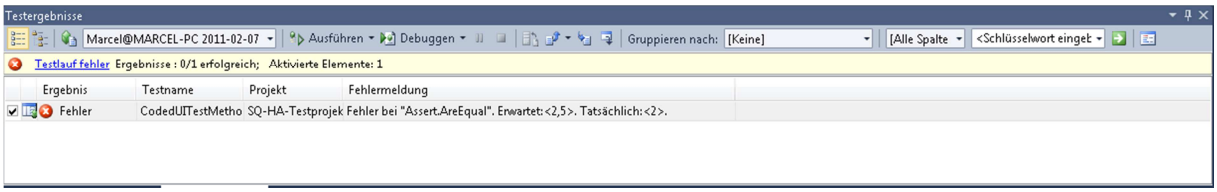
Abbildung 16

Durch Klick auf die rote Aufzeichnungstaste startet man die Aufzeichnung. Nun öffnet man seine GUI über den Ordner wo die „.exe“ liegt und gibt die Zahlenwerte ein und klickt auf rechnen. Nun kann man die Aufzeichnung beenden in dem man auf das Pause Symbol klickt was, dass Aufzeichnungssymbol vorher ersetzt hat. Die Schaltfläche ganz rechts in dem Fenster generiert aus dem Aufgezeichneten Ablauf Programmcode und beendet abschließend die Aufzeichnung. Abbildung 17 zeigt den generierten Programmcode.

```
103         public void geteilttest()
104         {
105             Variable Declarations
113
114             // Doppelklicken "Name" Textfeld
115             Mouse.DoubleClick(uINameEdit, new Point(56, 7));
116
117             // "5" in "textBox1" Textfeld eingeben
118             uITextBox1Edit.Text = this.geteilttestParams.UITextBox1EditText;
119
120             // "2" in "textBox2" Textfeld eingeben
121             uITextBox2Edit.Text = this.geteilttestParams.UITextBox2EditText;
122
123             // "/" in "comboBox1" Kombinationsfeld auswählen
124             uIComboBox1ComboBox.SelectedItem = this.geteilttestParams.UIComboBox1ComboBoxSelectedItem;
125
126             // Klicken "textBox3" Textfeld
127             Mouse.Click(uITextBox3Edit, new Point(22, 14));
128
129             // Klicken "Rechnen" Schaltfläche
130             Mouse.Click(uIRechnenButton, new Point(40, 9));
131
132             // Klicken "textBox3" Textfeld
133             Mouse.Click(uITextBox3Edit, new Point(38, 13));
134
135             Assert.AreEqual(2.5, Convert.ToDouble(uITextBox3Edit.Text));
136         }
```

Abbildung 17

Hinzugefügt wurde hier nur die Assert Anweisung um einen Vergleich mit dem richtigen Ergebnis zu erhalten 2,5 erwartet man bei 5/2 man erhält jedoch 2 was man dann unter dem Testlisteneditor ausgeführten Test sieht:



8. Auslastungstests in Visual Studio

8.1 Einrichten von Auslastungstests

Mit den sehr umfangreichen Auslastungstests ist es möglich, das erstellte Projekt auf Belastbarkeit in Bezug auf zugreifende Benutzer, ausgeführte Operationen, System- und Netzwerkbelastung sowie Reaktionszeit zu testen. Für die Auslastungstests werden Unit-Tests benötigt, mit denen dann die Auslastung simuliert wird. Zuerst kann die Reaktionszeit der Software für das Szenario angegeben werden, d.h. wie viel Zeit die Software hat, um zwischen den Tests zu reagieren.

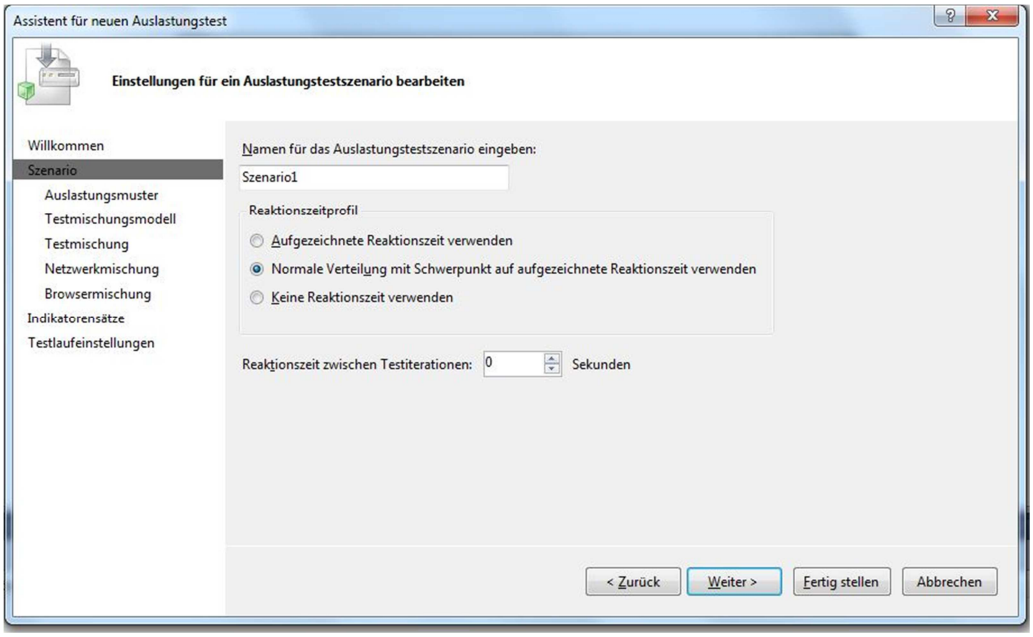


Abbildung 18

Im nächsten Schritt wird angegeben, wie viele zugreifende Benutzer simuliert werden sollen. Hier kann die Angabe entweder konstant sein, oder im Laufe der Testdauer schrittweise nach eigenen Angaben erhöht werden. Zu beachten ist, dass mit der Installation von Visual Studio 2010 maximal 250 Nutzer simuliert werden können. Für die Simulation von mehr als 250 Benutzern müssen virtuelle Benutzerpakete erworben werden. Sollte die Anzahl an Benutzern ohne ein solches Paket die 250 überschreiten, wird der Test mit einem Hinweis auf eine Lizenzverletzung abgebrochen, sobald die Anzahl an simulierten Nutzern den Lizenzwert überschreitet.

Mit dem Testmischungsmodell kann festgelegt werden, auf welcher Grundlage die verschiedenen Tests aufgerufen werden sollen. Die Vorgehensweise jeder Auswahl wird mit einer Grafik sowie einem Infotext erklärt. Ebenfalls im Infotext enthalten ist eine Empfehlung, für welche Art Software eine solche Testmischung geeignet ist bzw. welchen Schwerpunkt diese hat.

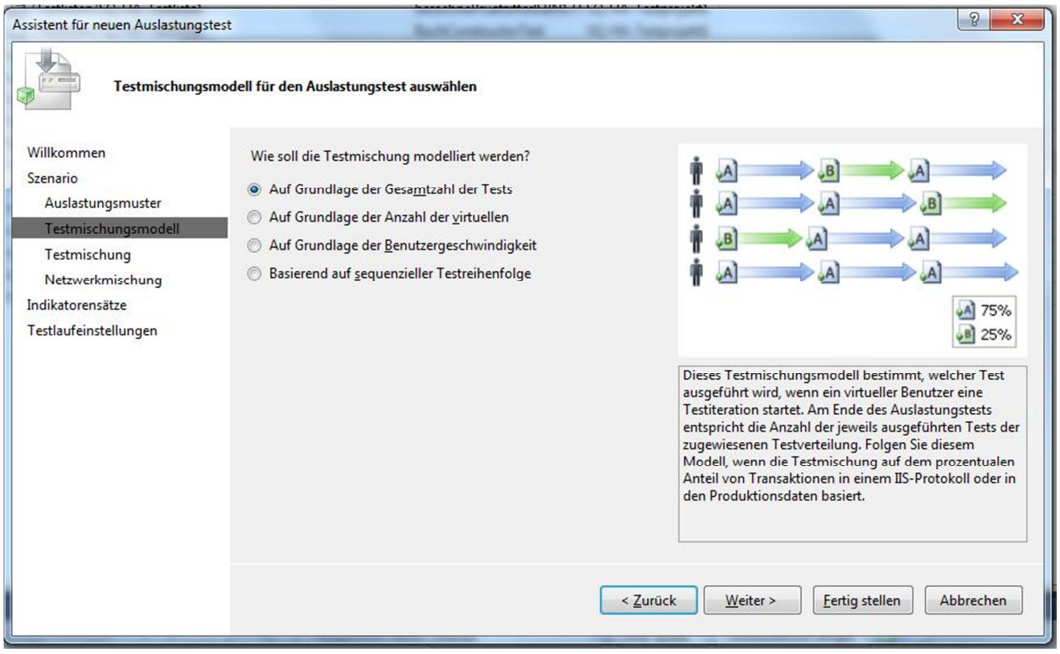


Abbildung 19

Die Testmischung ist abhängig von dem vorher gewählten Modell. Entweder kann hier prozentual bestimmt werden, welche Tests wie oft aufgerufen werden, oder die Gewichtung geschieht über die Angabe von Aufrufen pro Benutzer pro Stunde. Hier wird beim ersten Aufruf kein Test aufgeführt, diese müssen erst noch hinzugefügt werden aus dem Pool der verfügbaren Tests.

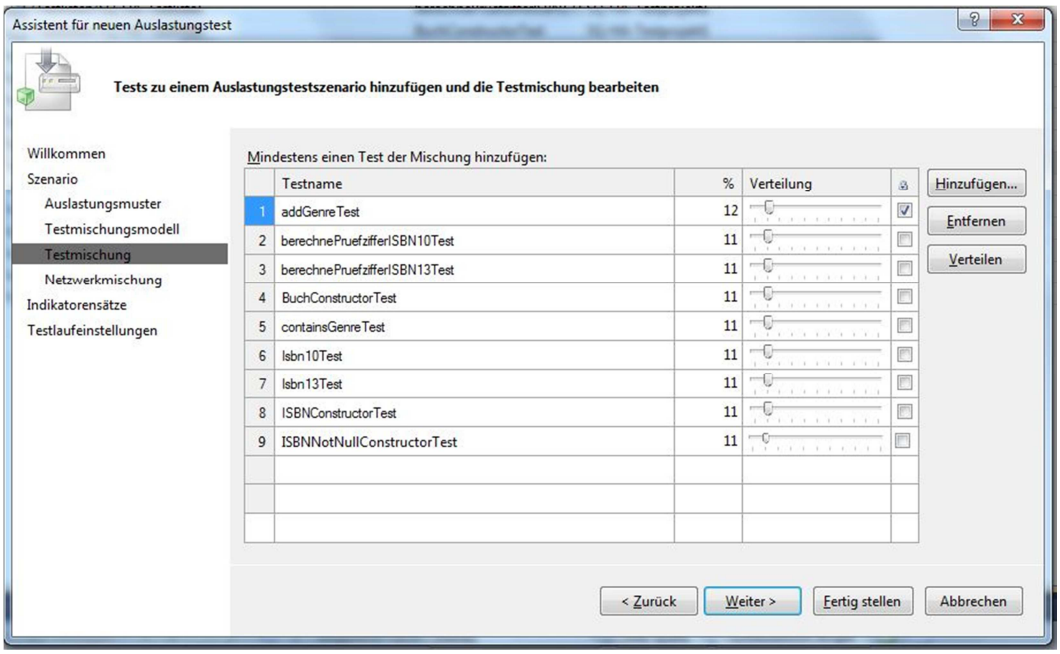


Abbildung 20

Bei der Angabe der Netzwerkmischung kann festgelegt werden, aus welchen Netzwerken ein Zugriff auf die Tests simuliert werden soll. Zu beachten ist, dass lediglich bei reinen Webanwendungen eine Ausführung des Tests mit mehreren angegebenen Netzwerkschnittstellen möglich ist. Bei allen anderen Anwendungstypen ist ein Hinzufügen von Netzwerkschnittstellen zwar möglich, der Lasttest wird dann aber nicht von Visual Studio ausgeführt und es wird eine diesbezügliche Meldung ausgegeben. Des Weiteren werden für die Simulation von Schnittstellen, über die der ausführende PC nicht verfügt, nachträglich Treiber installiert, was eine kurze Unterbrechung der gesamten Netzwerkkommunikation nach sich zieht.

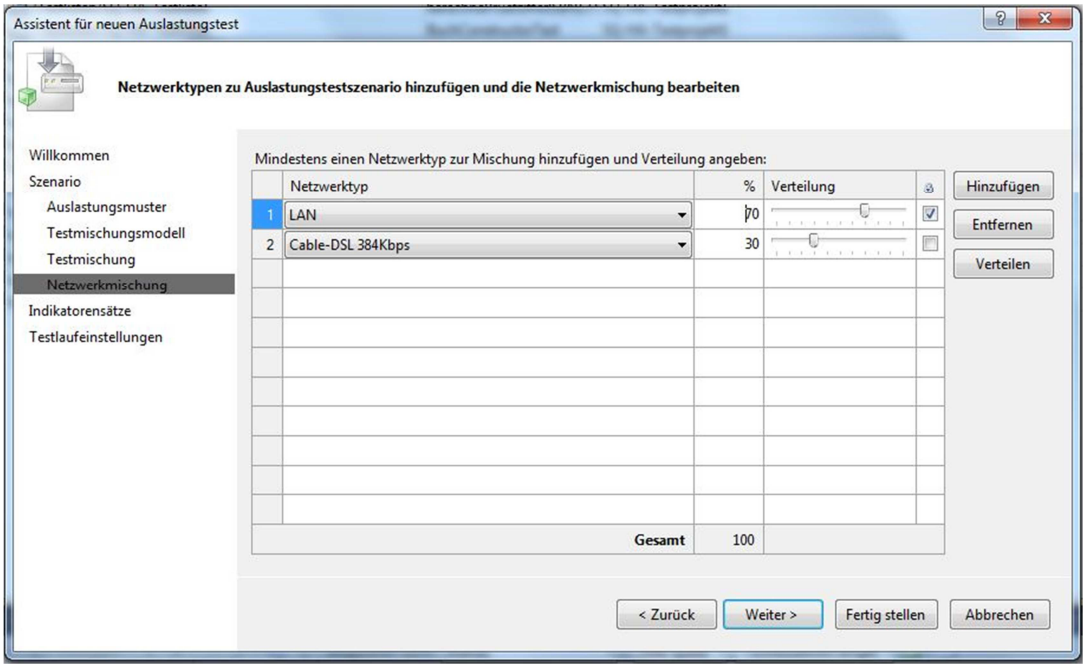


Abbildung 21

Mittels der Indikatorensätze können die Tests auf mehreren Rechnern ausgeführt werden. Dazu wird die Installation von „Visual Studio 2010 Agents“ auf den zu testenden Rechnern vorausgesetzt, welches separat erhältlich ist. In Ermangelung dieser Zusatzsoftware kann an dieser Stelle auf die Möglichkeiten mit Lasttests auf mehreren Rechnern nur begrenzt eingegangen werden.

Bei den Indikatorsätzen werden alle Rechner aufgeführt, welche den Agenten-Dienst installiert haben. Hier kann zum einen festgelegt werden, welcher Rechner welche Art von Programm simulieren soll, also ob es sich um ein SQL-Zugriff oder eine Anwendung handelt. Auch kann hier eine Benutzergewichtung vorgenommen werden, d.h. jedem Rechner wird ein prozentualer Anteil an zu simulierenden Benutzern zugewiesen.

In den Testlaufeinstellungen kann festgelegt werden, wie lange ein Lasttest laufen soll. Dies geschieht entweder über eine zeitliche Angabe, in welcher so viele Tests wie möglich ausgeführt werden. Alternativ kann eine feste Anzahl an Tests eingestellt werden.

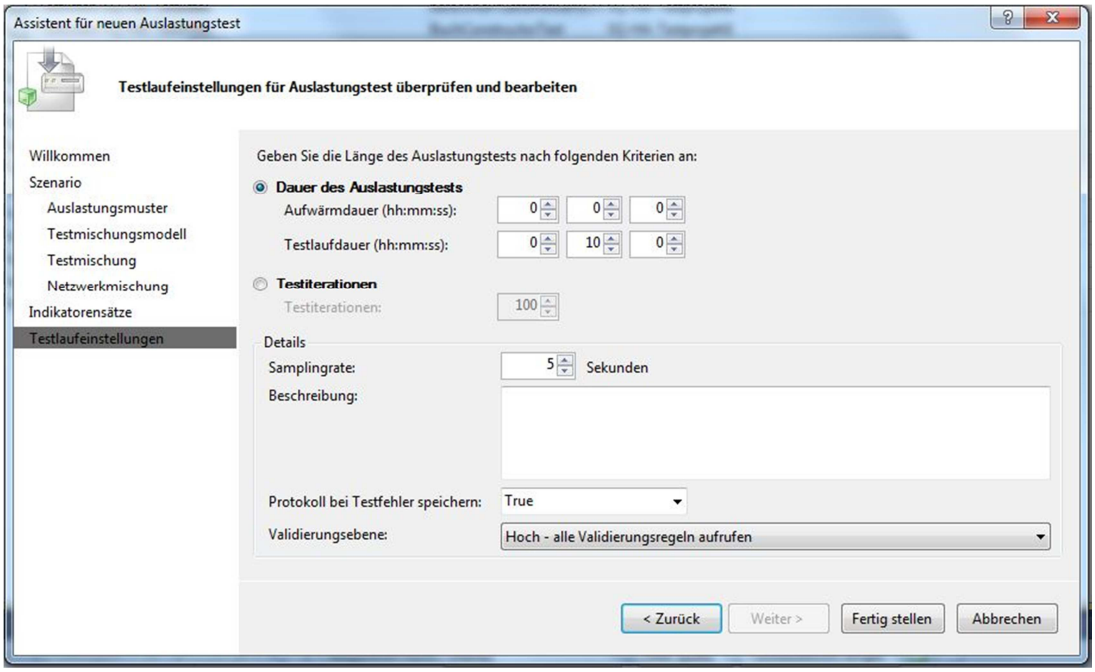


Abbildung 22

8.2 Ausführen von Auslastungstests

Nach Abschluss des Auslastungstest zeigt die eingeblendete Übersicht neben allgemeinen Testlaufinformationen wie die Start- und Endzeit auch Informationen an über die langsamsten Tests. Auch wird angezeigt, wie oft jeder Test ausgeführt wurde und wie viele Fehler bei diesem Test aufgetreten sind. Des Weiteren werden die benötigten Prozessorressourcen der getesteten Systeme angezeigt sowie eine allgemeine Fehlerliste, welche neben Fehlern innerhalb der einzelnen Tests auch Ausnahmefehler an, z.B. wenn ein Rechner während des Tests nicht mehr verfügbar ist.

Generell ist anzumerken, dass Auslastungstest sich hauptsächlich für Multiuseranwendungen und verteilte Anwendungen eignen. Für kleinere Programme eignen sich die anderen vorgestellten Testmöglichkeiten besser.

9. Abbildungsverzeichnis