Design-By-Contract in .NET

© 2003, Manfred Rätzmann

RÄTZMANN GmbH, Claudiusstr. 3, 10557 Berlin

mailto: m.raetzmann@raetzmann-gmbh.de

In vielen Programmiersprachen stehen ASSERT Anweisungen oder ähnliches zur Verfügung mit denen sich die Eingangsbedingungen einer Funktion oder einer Klassenmethode abprüfen lassen. Im .NET Framework stehen dafür die beiden Klassen *Debug* und *Trace* bereit, mit denen Design-By-Contract Konzepte realisiert werden können. Dieser Artikel stellt das Design-By-Contract Konzept und dessen Realisierung mit der *Debug* Klasse vor.

Dieser Artikel erschien erstmalig in Ausgabe 05.03 der Zeitschrift dot.net magazin

Das Konzept der automatischen Abprüfung von Vorbedingungen und Nachbedingungen einer Methode sowie von Bedingungen, die für jedes Objekt einer Klasse immer gelten müssen, ist als „Design-by-Contract“ aus der Sprache Eiffel bekannt. Dort ist es direkt in die Sprache integriert. Mit dem Schlüsselwort „require“ wird zu Beginn einer Methode eine Liste von Ausdrücken eingeleitet, die die Vorbedingungen (Preconditions) darstellen. Nach einem „ensure“ folgt am Ende der Methode eine Liste von Ausdrücken mit den Nachbedingungen (Post conditions). Mit „invariant“ lassen sich zusätzlich Bedingungen für alle Objekte einer Klasse angeben, die immer wahr sein müssen (Invariants). Der Eiffel Compiler stellt wahlweise sicher, dass alle Bedingungen, nur die Vorbedingungen, Vor- und Nachbedingungen oder keiner dieser Ausdrücke ausgewertet werden. Eine Beschreibung des „Design-by-Contract“ Konzepts in Eiffel finden Sie unter [ISEDbC].

Permanenter Regressionstest

In den meisten Programmiersprachen lassen sich zumindest die Vorbedingungen ohne große Probleme auswerten. Assert-Anweisungen oder simple If-Statements zu Beginn einer Methode stellen sicher, dass alle Voraussetzungen für den Start der Methode vorliegen. Die Abprüfung der Vorbedingungen (die Default-Einstellung beim Eiffel Compiler) reicht für viele Testzwecke aus. Wenn die Nachbedingungen einer Methode bzw. die Invarianten der Klasse nicht stimmen, so fällt dies spätestens dann auf, wenn sie als Vorbedingungen einer Folgemethode abgeprüft werden. Ein sorgfältiges Abprüfen der Vorbedingungen aller Funktionen oder Methoden kommt einem permanenten Regressionstest gleich. Testtreiber müssen dann nur noch dafür sorgen, dass alle zu testenden Funktionen aufgerufen werden. Die Rückgabewerte werden vom Testtreiber protokolliert oder ausgeworfene Exceptions werden abgefangen. Mehr über Regressionstests und andere Testverfahren finden Sie in [Rätzmann2002].

Das Abprüfen von Vorbedingungen zu Beginn einer Funktion hat den angenehmen Nebeneffekt, dass die Vorbedingungen der Funktion gültig dokumentiert sind. Textliche Beschreibungen von Vorbedingungen in Funktionsheadern oder als Constraints in einem Klassenmodell haben nämlich die Eigenschaft, dass sie nach einer Weile nicht mehr allzu zuverlässig sind. Wenn die Vorbedingungen über Asserts oder If-Anweisungen zu Beginn der Funktion abgeprüft werden, ist dies nicht nur Kommentar-Prosa sondern reales Programmverhalten – und lesbar ist es zumindest für Softwareentwickler immer noch.

Die .NET Debug Klasse

Die beiden Klassen *Debug* und *Trace* aus dem System.Diagnostics Namensraum unterscheiden sich im wesentlichen darin, dass der Compiler Aufrufe von Methoden der *Debug* Klasse nur im Debug-Build erstellt. In

Release-Builds wird die *Debug* Klasse vom Compiler ignoriert. Für die *Trace* Klasse hingegen wird auch im Release-Build Code erzeugt. Ansonsten stimmen beide Klassen in ihren öffentlichen Eigenschaften und Methoden überein. Alle öffentlichen Eigenschaften und Methoden sind statisch, gehören demnach zur Klasse und nicht zu einer einzelnen Instanz. Da beide Klassen versiegelt sind (sealed) und keine Ableitungen davon erzeugt werden können, gibt es auch keine geschützten Eigenschaften und Methoden.

Zum Abprüfen von Vorbedingungen, Nachbedingungen und invarianten Bedingungen entsprechend des Design-By-Contract Konzepts benutzen wir im Folgenden die *Debug.Assert* Methode. Diese Methode hat drei Überladungen, von denen wir hier die Variante *public static void Assert(bool,string)* benutzen.

Assert Anweisungen prüfen einen booleschen Ausdruck ab und geben eine Meldung aus, wenn dieser Ausdruck als FALSCH ausgewertet wird. Entsprechend können wir der *Debug.Assert* Methode einen booleschen Ausdruck übergeben und einen String, der als Meldung ausgegeben werden soll, wenn der Ausdruck als FALSCH ausgewertet wird. Im Beispiel 1 prüft der boolesche Ausdruck ab, dass die *PrimaryKey* Setzung eines *DataAccessObject* ungleich dem Leerstring ist. Anderenfalls wird eine Meldung ausgegeben, die besagt, dass der *PrimaryKey* nicht gesetzt ist.

```
Debug.Assert(DataAccessObject.PrimaryKey.Trim()!= "",
  "DataAccessObject.PrimaryKey nicht gesetzt!");
```

Beispiel 1: Aufruf der Debug.Assert Methode

Vorbedingungen

Prüfen Sie mit *Debug.Assert* Aufrufen alle Vorbedingungen, die für die jeweilige Methode erfüllt sein müssen. Mit „Vorbedingungen“ ist jedoch nicht die Schnittstellendefinition (Signatur) der aufgerufenen Methode gemeint. Die korrekte Übergabe der Parameter an eine öffentliche Methode sollte nicht mit *Debug.Assert* sondern mit *if*-Anweisungen überprüft werden. Die Überprüfung der übergebenen Parameter gehört zu den Pflichten einer jeden öffentlichen Methode und darf daher auch zur Laufzeit nicht außer Kraft gesetzt werden können, wie das bei Verwendung von *Debug.Assert* der Fall ist. Das gleiche gilt, wenn ein Objekt zur Durchführung einer bestimmten Methode in einem definierten Zustand sein muss. Wenn, als Beispiel, ein Beleg vollständig erfasst sein muss, bevor er gedruckt werden kann, darf man dies beim Aufruf der *drucken()* Methode nicht mit *Debug.Assert* abprüfen. Mit *Debug.Assert* prüfen Sie den Zustand des Systems ab, der von der Methode vorausgesetzt wird. Wenn das System bei Aufruf der Methode von diesem Zustand abweicht liegt auf jeden Fall ein Programmfehler vor. Beispiel 2 zeigt den Unterschied noch einmal an einer konkreten Situation auf.

```
public void saveBusinessObject(BusinessObject BO)
{
  if (BO == null)
    throw new ArgumentNullException("Business Object darf nicht null sein!");

  Debug.Assert(DataAccessObject != null,"DataAccessObject ist null!");
...
  DataAccessObject.save(BO);
...
}
```

Beispiel 2: Unterschiedliche Nutzung von Assert und if

Die Methode in Beispiel 2 erwartet einen Parameter vom Typ *BusinessObject*, der nicht *null* sein soll. Diese Bedingung, die zur Schnittstellendefinition der Methode gehört, wird mit einem *if*-Statement abgeprüft. Im Fehlerfall wird eine *Exception* ausgeworfen, auf die das aufrufende Programm reagieren muss. Damit die Methode problemlos ihre Aufgabe erfüllen kann, ist sie darauf angewiesen, dass das später benutzte *DataAccessObject* tatsächlich existiert und die Referenz darauf nicht *null* ist. Diese Bedingung gehört zu den Vorbedingungen der Methode und wird über *Debug.Assert* abgeprüft.

Vorbedingungen prüfen, heißt, alles abzuprüfen, was eine Funktion zum Leben braucht. Es heißt jedoch nicht, die Plausibilität von Setzungen zu überprüfen! Beispiel 3 zeigt die set- und get-Accessor Methoden der Eigenschaft DataAccessObject aus Beispiel 2.

```
public DataAccessBase DataAccessObject
{
  get {return _DataAccessObject;}
  set {_DataAccessObject = value;}
}
```
Beispiel 3: get- und set-Accessoren des DataAccessObject

Im set-Accessor wird absichtlich nicht geprüft, ob *value* eventuell *null* ist. Das heißt, die Eigenschaft *DataAccessObject* darf – aus welchen Gründen auch immer – zwischenzeitlich durchaus einmal auf *null* gesetzt werden. Eine Methode jedoch, die mit dem *DataAccessObject* arbeiten will, muss die Existenz eines solchen als eine ihrer Vorbedingungen abprüfen.

Nachbedingungen

Bei der Abprüfung von Nachbedingungen wird nicht mit *if* sondern immer *Debug.Assert* gearbeitet. Während eine Methode die korrekte Parameterübergabe immer abprüfen sollte, ist ein korrekter Rückgabewert oder genereller die Einhaltung der definierten Nachbedingungen ja der eigentliche Sinn der Methode. Alles, was davon abweicht, kann nur ein Programmfehler sein. Abprüfung der Nachbedingungen sind demnach immer eingebettete Tests, die nicht zur eigentlichen Aufgabe der Methode gehören. Solche Tests sollten in Release-Builds entfernt werden um die Performance nicht zu beeinträchtigen. Voraussetzung dafür ist natürlich, dass nur ein ausreichend getester Sourcecode für Release-Builds freigegeben wird. So sollten zum Beispiel die Pfade, die sich aus den möglichen Zustandsübergängen des Objektes ergeben, zu 100% durch entsprechende Testfälle abgedeckt sein. Mehr über die zustandsbasierte Pfadabdeckung und andere Testabdeckungsgrößen finden Sie in [Rätzmann2002].

Nachbedingungen werden vor dem Rücksprung aus der Methode abgeprüft. Wenn die Methode einen berechneten Wert zurück liefert, kann die Prüfung eventuell über eine Rückrechnung erfolgen. Bei einer Datenänderung kann eventuell der veränderte Datenstatus über eine zweite Methode oder über einen Kontrollzugriff überprüft werden. Beispiel 4 zeigt noch einmal die Methode aus Beispiel 2, diesmal jedoch ergänzt um die Abprüfung eine Nachbedingung.

```
public void saveBusinessObject(BusinessObject BO)
{

...
  DataAccessObject.save(BO);
  Debug.Assert(!BO.IsModified,"BusinessObject konnte nicht gespeichert werden!")
}
```
Beispiel 4: Das erfolgreiche Speichern wird überprüft

Interne Invarianten

In einer Sourcecodestruktur gibt es häufig Stellen, an denen eine Bedingung abgeprüft wird, aber die Alternativbedingung nicht. Wenn zum Beispiel aus der vorhergehenden Verarbeitung resultierend der Wert einer Variablen *i* nur gleich 1 oder 2 sein kann, erfolgen häufig Verzweigungen über *if* Statements wie:

```
if (i == 1)
{
   ... Verarbeitung 2
}
else //  i = 2
{
   ... Verarbeitung 2
{
```
Beispiel 5a: Das i = 2 gilt wird vorausgesetzt

Die Tatsache, dass im *else*-Zweig immer i = 2 gelten sollte, nennt man eine interne Invariante der Methode. Solche internen Invarianten lassen sich sehr einfach durch kurze Asserts abprüfen:

```
if (i == 1)
{
```

```
   ... Verarbeitung 1
}
else //  i = 2
}
   Debug.Assert(i == 2,"Verletzung der Invariante i = 2")
   ... Verarbeitung 2
{
```

Beispiel 5b: Interne Invariante wird abgeprüft

Kontrollfluss Invarianten

Auf ähnlich einfache Art können eigene Annahmen über den Kontrollfluss im Programm abgesichert werden. Statt die Annahmen zum Kontrollfluss lediglich als Kommentar zu hinterlegen (siehe Beispiel 6a) schreibt man an diese Stelle besser ein Assert Statement wie in Beispiel 6b dargestellt.

```
void tuWas() {
   for (…) {
      if (…)
         return;
   }
   // hier sollte das Programm nie hinkommen
}
```

Beispiel 6a: Kommentar zu Kontrollfluss

```
void tuWas() {
   for (…) {
      if (…)
         return;
   }
   Debug.Assert(false,"Hier sollte das Programm nie hin geraten");
}
```

Beispiel 6b: Kontrollfluss wird überprüft

Allerdings kommt uns hier die VS.NET Entwicklungsumgebung etwas in die Quere. Bei der Erstellung eines Programmteils mit der oben beschriebenen Abprüfung des Kontrollflusses gibt der C# Compiler die Warnung aus, dass dieser Code nicht erreicht werden kann. (Der VB-Compiler schweigt sich zu diesem Thema generell aus.) Solche Überprüfungen durch den Compiler sind zwar sehr zu begrüßen, im Fall der *Debug* Klasse sollte der Compiler von dieser Regel aber eine Ausnahme machen, um eigene Kontrollflussprüfungen zu ermöglichen (Verbesserungsvorschlag). Wenn im Beispiel 6a durch irgendwelche späteren Änderungen an der Sourcenstruktur das Programm nämlich anfängt, falsche Wege zu gehen, gibt es keine Warnungen des Compilers. Unser *Debug.Assert(false,"...")* würde diese Unregelmäßigkeit jedoch finden.

Leider steht nicht zu erwarten, dass mein Verbesserungsvorschlag kurzfristig realisiert wird. Deshalb müssen wir zunächst mit dem Warnhinweis auf unerreichbaren Code leben oder das *Debug.Assert* Statement in ein *#if DEBUG ... #endif* einkleiden. Dann wird zumindest der Release-Build diese Warnung nicht mehr enthalten.

Klassen Invarianten

Klassen Invarianten sind Bedingungen, die für jedes Objekt einer Klasse zu jeder Zeit gelten müssen. Klassen-Invarianten werden außerhalb Eiffel häufig durch Prüf-Methoden realisiert. Das sind Methoden der Klasse, die keine Verarbeitungsfunktion haben sondern lediglich den aktuellen Status der Klasse überprüfen. Eine solche Prüf-Methode gibt am besten einfach ein *true* zurück, wenn alles in Ordnung ist, ansonsten ein *false*.

Eine Buchungs-Klasse, deren Objekte stets in sich ausgeglichen sein müssen (in welcher Art auch immer), könnte diesen Zustand über ein entsprechende Prüf-Methode kontrollieren:

```
// Gibt true zurück, falls ausgeglichen
private bool ausgeglichen()
{
   ... <Prüfung der Ausgeglichenheit>
   if (<Alles OK>)
```

```
      return true;
   else
      return false;
}
```
Beispiel 7: Prüfmethode

Diese Methode kann als Klassen-Invariante angesehen werden. Alle anderen Methoden können durch ein *Debug.Assert(ausgeglichen(),"...");* vor und nach jeder Verarbeitung abprüfen, dass die invariante Bedingung eingehalten wird. Die *ausgeglichen()*-Methode wird zwar nur in Debug-Builds benötigt, eine vom DEBUG Flag bedingte Compilierung funktioniert aber leider nicht. Auch in Release-Builds würde *ausgeglichen()* vom Compiler gesucht, selbst wenn die Zeile *Debug.Assert(ausgeglichen(),"...")* nicht mit in den IL Code compiliert wird.

Nebeneffekte

Bei der Verwendung von Asserts muss unbedingt darauf geachtet werden, dass keine unbeabsichtigten Nebeneffekte auftreten. So darf die Methode *ausgeglichen()* im Beispiel 7 auf keinen Fall etwas am Status der Klasse ändern. Wenn Asserts aus der Endfassung des Programms in der Laufzeitumgebung entfernt werden, würde *ausgeglichen()* nicht mehr ausgeführt und der Status der Klasse wäre ein anderer als bei einem Durchlauf in der Entwicklungsumgebung.

Manchmal werden Nebeneffekte aber auch bewußt herbeigeführt, wenn zum Beispiel ein Vergleichswert für die Abprüfung einer Nachbedingung gebraucht wird, der ohne Asserts nicht mehr benötigt wird. Erstellen Sie dazu – ähnlich wie bei den Prüfmethoden weiter oben – eine Methode, die den Vergleichswert sichert. Diese Methoden kann mit dem Attribut „Conditional“ als optional gekennzeichnet werden. Voraussetzung dafür ist, dass die Methode nichts zurück liefert. Optionale Methoden bleiben zwar im Code, werden aber nur aufgerufen, wenn das hinter „Conditional“ angegebene Flag gesetzt ist. In C# könnte das so etwa aussehen:

```
string TmpString;
[Conditional("DEBUG")]
private void tmpCopy(string s)
{
   TmpString = s;
   return true;
}


public void tuWas()
{
   string MyString;
   MyString = "irgendwas";
   // String sichern (optional bei DEBUG)
   tmpCopy(MyString);
   // weitere Verarbeitung ...
   MyString = "was anderes";
   // Vergleich mit dem gesicherten String
   Debug.Assert(MyString == TmpString,"MyString wurde verändert!");
}
```
Beispiel 8: Assert Anweisung mit Nebeneffekt

Das temporäre Sichern des Strings in Beispiel 8 wird in einer optionalen Methode durchgeführt. Der Compiler wird dann bei einem Release-Build die Methode *tmpCopy()* zwar mit übersetzen, aber nicht aufrufen weil DEBUG dann nicht gesetzt ist.

Der Ernstfall

Was passiert denn nun aber wenn, wie in Beispiel 8, eine Behauptung (Assertion) nicht zutrifft? In diesem Fall gibt die Debug Klasse eine Meldung aus, wie in Abbildung 1 dargestellt.

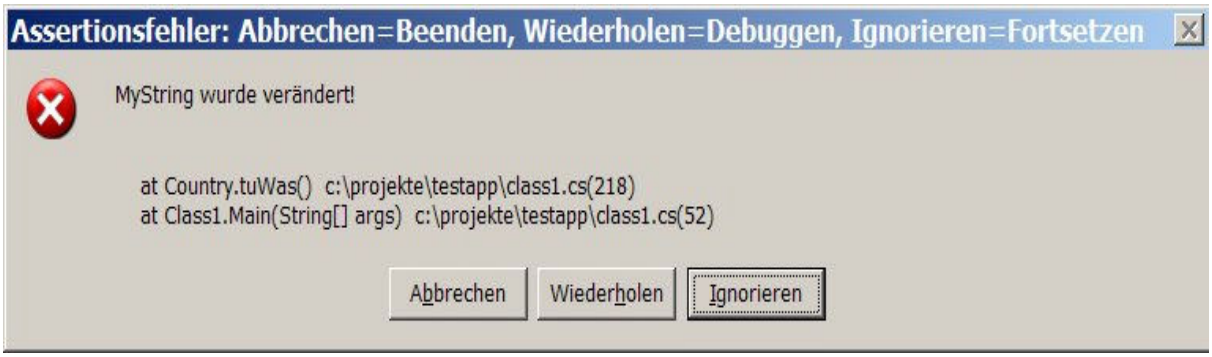

Abb. 1: Meldung bei nicht zutreffender Bedingung

Abbildung 1 zeigt die Ausgabe der Debug Klasse bei Aufruf der Assert Methode mit einer nicht zutreffenden Bedingung. Ausgegeben wird der gesamte Call-Stack, also nicht nur die Zeile, die die nicht zutreffende Bedingung enthält. Wie die Titelzeile des Meldungsfensters angibt können Sie an dieser Stelle das Programm abbrechen, in den Debugger wechseln oder die nicht zutreffende Bedingung ignorieren. Insbesondere der Wechsel in der Debugger bietet dem Entwickler in der Test & Tune Phase der Entwicklung eine wichtige Analysemöglichkeit. Wenn irgendwo in Ihrem Programm etwas nicht stimmt haben Sie so die Chance, unmittelbar darauf zu reagieren und die Situation im Debugger zu überprüfen. Voraussetzung dafür ist allerdings, dass Sie Ihr Programm mit der *Debug.Assert* Methode instrumentiert haben.

Wenn Sie Ihre Unit Tests allerdings nicht manuell sondern automatisiert mit Hilfe eines Testframeworks wie NUnit [NUnit] durchführen kann eine solche Ausgabe allerdings ziemlich störend sein. Vor allem dann, wenn schon einiges an Tests zusammen gekommen ist und der Durchlauf eine Weile dauert. Dann sollte keine Messagebox am Schirm erscheinen, die mit einem Buttonclick quitiert werden muss. Vielmehr sollten alle Meldungen protokolliert werden, außerdem wäre es schön, wenn beim Fehlschlagen einer Bedingung eine spezifische Ausnahme (Exception) ausgelöst wird. Dann könnte man im GUI Fenster von NUnit (oder anderen Testwerkzeugen) gleich erkennen, bei welchem Test eine Assertion fehlgeschlagen ist. Der Code in Beispiel 9 zeigt, was dafür zu tun ist.

```
// 1. einen eigenen TraceListener erstellen
public class myTraceListener : TextWriterTraceListener
{
   public myTraceListener(FileStream Output): base(Output){}
   public override void WriteLine(string Message)
   {
      base.WriteLine(Message);
      this.Flush();
      throw new ApplicationException("Assertion failed: "+Message);
   }
}

...
// 2. den DefaultTraceListener gegen den eigenen TraceListener austauschen
FileStream Testout =
  new FileStream("Testout.txt",FileMode.OpenOrCreate);
TextWriterTraceListener myWriter =
  new myTraceListener(Testout);
Debug.Listeners.Remove(Debug.Listeners[0]);
Debug.Listeners.Add(myWriter);
```

Beispiel 9: einen eigenen TraceListener installieren

Die Zuhörer

Die *Debug*-Klasse und die *Trace*-Klasse haben eine Eigenschaft „Listeners“ hinter der sich eine TraceListenerCollection verbirgt. Alle dort eingetragenen Listener werden bei einer fehlschlagenden Assertion mit der Meldung versorgt, die bei *Debug.Assert(<Bedingung>,<Meldung>)* angegeben wurde. Konkret heißt das, dass die WriteLine-Methode jedes Listener aufgerufen wird. Der Listener schreibt die Meldung dann entweder in eine Textdatei oder einen Stream (TextWriterTraceListener), in ein Windows-Ereignisprotokoll

(EventLogTraceListener) oder in das Output Window von Visual Studio .NET (DefaultTraceListener). Ein DefaultTraceListener gibt die Meldung einer fehlschlagenden Assertion zusätzlich zusammen mit der Aufruf-Liste in einer Messagebox aus.

Wenn wir also unsere Assert-Meldungen in einer Textdatei protokollieren und statt der Messagebox-Ausgabe eine Exception auswerfen wollen, müssen wir einen eigenen Listener schreiben und den vorhandenen DefaultTraceListener gegen unseren eigenen Listener austauschen. In Beispiel 9 wird zunächst eine eigene Listener-Klasse *myTraceListener* aus der TextWriterTraceListener Klasse abgeleitet. In Zeile 4 wird einer der Konstruktoren überschrieben, damit wir später beim Erstellen eines myTraceListener-Objektes den FileStream für die Ausgabe gleich mit übergeben können. Anschließend überschreiben wir noch die WriteLine-Methode indem wir zunächst die WriteLine-Methode der Basisklasse aufrufen und die neue Zeile mit *Flush()* aus dem internen Puffer in den Ausgabestrom schreiben. Danach wird dann die gewünschte Exception ausgeworfen.

Irgendwo in der Initialisierung der Applikation wird dann der DefaultTraceListener gegen eine Instanz des eigenen TraceListeners ausgetauscht. Dazu erstellen wir zunächst einen FileStream, der die Ausgabe in die Datei „Testout.txt“ schreibt. Danach wird der neue TraceListener erstellt, wobei der FileStream gleich an den Konstruktor übergeben wird. Als nächstes entfernen wir den DefaultTraceListener auf Position 0 der Listeners-Collection und fügen unseren neuen Listener der Listeners-Collection hinzu. Das war’s – von nun an verhält sich Debug.Assert so wie wir das für automatisierte Unit-Tests brauchen.

Zusammenfassung

Mit den Klassen *Debug* und *Trace* des .NET Frameworks können in Debug- und wahlweise auch Release-Builds alle Prüfmöglichkeiten realisiert werden, die das Design-By-Contract Konzept vorsieht. Durch die Möglichkeit, eigene Listener-Klassen zu erstellen, lassen sich solche eingebetteten Tests nahtlos in jede Testumgebung integrieren. Insbesondere können auch eingebettete Tests genau auf die Bedingungen beim automatisierten Unit-Test mit externen Testframeworks wie NUnit abgestimmt werden.

Literaturhinweise

[ISEDbC]
http://www.eiffel.com/doc/manuals/technology/contract/page.html
[NUnit]
http://nunit.org
[Rätzmann2002]
Manfred Rätzmann: Software-Testing, erschienen bei Galileo Computing, 2002, ISBN 3-89842-271-2