Was lange währt ...

JUnit erscheint in Version 4

von klaus meffert

Lange hat die Entwicklergemeinde auf die Veröffentlichung von JUnit 4 gewartet. Bis zum 16. Februar diesen Jahres dauerte die Warterei, die nun ein Ende hat. Zuvor waren lediglich Vorab-Releases im Archivsystem (CVS) erhältlich, jetzt gibt es neben dem Haupt-Release gleich zwei Zusatzkandidaten. Diskussionen in einschlägigen Foren sowie das Konkurrenz-Framework TestNG [8] (mit vergleichbaren Eigenschaften wie JUnit 4) haben den Veröffentlichungsdruck verstärkt.

Kasten

Der Text ist ein leicht modifizierter Auszug aus dem im Mai im Software & Support Verlag erscheinenden Buch „JUnit Profi-Tipps – Bewährte Techniken und Antipatterns“.

Ende Kasten

Auch wenn inoffiziell der Release Candidate 2 [2] kursiert, der gegenüber dem ersten Kandidaten erkennbar weiterentwickelt und verändert wurde, lohnt sich ein Einblick in JUnit 4, zumal es jedem freisteht, aus den verfügbaren, noch aktuelleren Quelldateien des Archivsystems ein Kompilat zu erzeugen. Alleine schon die Tatsache, dass das neue JUnit 4 nun auf Java 5 basiert, macht eine Betrachtung interessant.

JUnit 4 basiert im Gegensatz zu früheren Releases auf Java 5 und ist nicht unter niedrigeren Java-Versionen lauffähig. Die neuen Java-Sprachkonstrukte Annotations, Generics und Varargs fanden Einzug in JUnit 4 (alle Neuerungen von Java 5 sind in [6] beschrieben).Wichtigste Neuerung aus Sicht von JUnit ist die Einführung der Annotations. Annotations sind in Java 5 eingeführte Sprachkonstrukte, die es erlauben, Metadaten im Programmtext anzubringen. Diese Metadaten werden von JUnit 4 ausgewertet. Doch auch strukturell hat sich einiges getan. Bisher enthielt JUnit nur Packages, die dem Namensraum *junit.** folgten. Die neue Version führt die Package-Hierarchie *org.junit.** ein. Dadurch entsteht eine gewisse Konfusion, denn der Entwickler muss nun überlegen, aus welchem Package er was importiert. Diese Doppelung der Namensräume hängt mit der Abwärtskompatibilität zusammen, die JUnit 4 anbietet. Dadurch ist es möglich, früher erstellte Testfälle mit der neuen JUnit-Version auszuführen.

Neuerungen

Die weiteren Ausführungen und Listings beziehen sich auf *Release Candidate 2* (verfügbar über [2] oder auf der Heft-CD), da sich in den letzten Tagen die verfügbare Distribution geändert hat und der Kandidat eine bessere Klassenstruktur hat. Natürlich erwartet die Entwicklergemeinde nach Einführung einer so lange herbeigesehnten Version auch Neuerungen und Erweiterungen gegenüber früheren Ständen. Den bisherigen Programmcode auf Java 5 zu heben, sorgt sicherlich in gewissem Umfang bereits dafür; die gleich beschriebenen Annotations sind wohl die gravierendste Veränderung, die daraus resultierte. Weiterhin haben monatelange konstruktive Diskussionen in spezifischen Foren und Newsgroups zwischen Entwicklern und JUnit-Machern und -Unterstützern dafür gesorgt, dass sich die Richtung hin zum neuen Release herauskristallisieren konnte. Auch die pragmatische Art von Kent Beck, einem der Erfinder von JUnit, hat ihren Teil zum bisherigen Ergebnis dazu beigetragen. Letzteres enthält einige Überraschungen, manche Neuerungen erscheinen auf den ersten Blick nicht unbedingt wünschenswert.

Gewöhnungsbedürftig und Diskussionen entfachend ist der Wegfall der Unterscheidung zwischen Abbrüchen (Failures) und Fehlern (Errors). In die erstgenannte Kategorie fielen Probleme, die nicht vorhergesehen sind und den Testlauf abbrechen. Fehler hingegen waren vorhergesehene Probleme, die das kontrollierte Fehlschlagen eines Testfalls aufgrund eines fehlerhaften Programmstücks bedingen. In JUnit 4 gibt es nur noch Abbrüche (oder nur noch Fehler, wie man mag).

Der neu eingeführte Namensraum *org.junit* (als Top-Level-Package) enthält Logiken, die mit dem Schreiben von Tests in Verbindung stehen. Die neuen Packages in diesem Namensraum waren bis vor Ende der Fertigstellung dieses Artikels Änderungen unterworfen. Momentan herrscht ein kleines Chaos (so existieren im offiziellen Release [1] sowohl die Packages *org.junit.internal.runners*, *org.junit.runner* als auch *org.junit.runners*) Hier deshalb die Beschreibung der Packages des Release Candidate 2:

- *notify*: Benachrichtigungen zu laufenden Tests
- *plan*: Strukturierung von geplanten Tests
- *runner*: Ausführung von Tests
- *runner.extensions*: Implementierungen populärer Testrunner
- *runner.internal*: Klassen, die internen Charakter haben
- *runner.internal.request*: dito

Die frühere Klassenhierarchie unterhalb von *junit.framework* wurde beibehalten bzw. erweitert.

Erstellen von JUnit 4-Testfällen

Die Klasse *Test* befindet sich sowohl im Package *org.junit* als auch in *junit.framework*. Sie sollte nicht aus beiden gleichzeitig importiert werden! Dies ist auch nicht notwendig, da ersteres Package die neue Annotation *Test* enthält und letzteres Package die nun für eigene JUnit 4-Tests nicht mehr benötigte Schnittstelle gleichen Namens.

Listing 1 zeigt die Implementierung einer Testklasse mit JUnit 4. Die Testklasse enthält einen Testfall, der durch die Annotation *Test* als solcher gekennzeichnet ist. Eine Annotation ist durch das vorangestellte @-Symbol gekennzeichnet. Eine Annotation repräsentiert eine Java-Klasse und wird auch in einer Klassendatei definiert, mit leicht unterschiedlicher Syntax. Weitere Details hierzu sind [6] zu entnehmen.

Listing 1

JUnit 4-Testfall

```
import static org.junit.Assert.*;
import org.junit.Test; //Annotation

public class MeineTestklasse {
  @Test public void einTest() {
   Manager manager = new Manager();
   int bonus = manager.getBonus() ;
   assertEquals(25,bonus);
  }
}
```

Ende Listing

Die Testklasse enthält nun keine *suite()*-Methode mehr, da die Annotation *Test* bereits genug Aussagekraft besitzt. Darüber hinaus erbt die Klasse nicht mehr von *junit.framework.TestCase*. Würde sie dies tun, wäre sie in dieser Form nicht mehr als JUnit 4-Testklasse ausführbar. Auffällig sind auch andere Aspekte, wie zum Beispiel die erste Import-Anweisung in Listing 1. Sie stellt einen statischen Import dar, eine mit Java 5 neu

hinzugekommene Möglichkeit. Somit ist es möglich, alle Methoden aus der statisch importierten Klasse, hier *org.junit.Assert*, so zu verwenden, als wären sie über eine Vererbungsbeziehung eingeführt worden. Die Vererbungsbeziehung zur Klasse *Assert* bestand früher indirekt durch den Zwang, jede Testklasse von *junit.framework.TestCase* abzuleiten, die wiederum von *Assert* abgeleitet war.

Die zweite Importanweisung im Listing dient der Importierung der Annotation *Test*. In Listing 1 ist dies die Methode mit dem Namen *einTest*. Der Name ist vollkommen willkürlich, alleine das Voranstellen der Annotation *Test* reicht zur Bestimmung deren Zwecks aus. Die eigentliche Testlogik wird genauso wie früher implementiert, an dieser Stelle gibt es keine Unterschiede. Übrigens sind die meisten *assertXXX*-Methoden (*assertSame, assertNull* usw.) gleich geblieben. Ausnahme sind die *assertEquals*-Implementierungen, bei denen der Vergleich von Arrays eingeführt wurde. Durch den neuen Java 5-Mechanismus namens Autoboxing konnten die Signaturen für primitive Typen wie *int* oder *byte* entfallen [6].

(De-)Initialisierungen

Annotations werden in JUnit 4 auch verwendet, um besondere Methoden zu kennzeichnen. Wer mit JUnit schon gearbeitet hat, dem sind die Methoden *setUp()* und *tearDown()* vertraut. Zu Recht als unschön haben viele Entwickler es empfunden, dass das Überschreiben einer geerbten *setUp*-Methode ohne deren expliziten Aufruf zu Seiteneffekten führen konnte. Dieser unerwünschte Nebeneffekt konnte nur mit gesondert zu verwendenden Prüfwerkzeugen entdeckt werden. Dies hat nun ein Ende. Anstatt in einer Methode namens *setUp* Initialisierungen (die vor jedem Testfall exekutiert werden) durchzuführen und eine Supermethode explizit aufzurufen, bleibt die Wahl des Methodennamens nun freigestellt und es gibt keine Supermethode im bisherigen Sinn mehr. Einzige Bedingung ist, die Methode mit der Annotation *Before* zu versehen. Für das bisherige *tearDown()*, das für Aufräumarbeiten nach jedem Testfall zuständig war, tritt die Annotation *After* ein. Komplett neu ist die Möglichkeit, vor Starten des ersten bzw. nach Beendigung des letzten Testfalls der Klasse Vorbelegungen bzw. abschließende Arbeiten auszuführen. Hierfür stehen die Annotations *BeforeClass* und *AfterClass*. Früher musste dafür getrickst werden, etwa durch Verwendung statischer Blöcke. In Listing 2 sind die Möglichkeiten der (De-)Initialisierung zusammenfassend dargestellt. Die Methode *init* wird sowohl vor *meinTest1* als auch vor *meinTest2* aufgerufen, wohingegen *initClass* ganz am Anfang nur einmal ausgeführt wird. Die Reihenfolge der Testfälle ist übrigens absichtlich nicht definiert. Das war auch in früheren JUnit-Versionen so, um alle Testfälle voneinander unabhängig zu machen. Analog zu *setUp()* verhält es sich mit *tearDown* (sowohl nach *meinTest1* als auch nach *meinTest2* aufgerufen) und *destroy* (ganz am Ende nach Ausführen des letzten Testfalls der Methode relevant).

Die Möglichkeit der klassenweiten (De-)Initialisierung sollte mit Vorsicht angewandt werden. Hier dürfen auf keinen Fall Daten vorbelegt werden, die ein Testfall modifiziert und ein folgender ausliest, weil sonst die Unabhängigkeit der Testfälle verletzt würde. Sinnvoll ist hier vielmehr der Aufbau bzw. Abbau einer Datenbankverbindung o.Ä., um unnötige Doppelinitialisierungen zu vermeiden.

Listing 2

Initialisierungen und Aufräumarbeiten

```
import org.junit.*;
public class MeinTest {
 private java.util.List list;
 private long startTime;
 // Initialisierungen ganz am Anfang
 @BeforeClass public void initClass() {
  startTime = java.util.Calendar.getTimeInMillis();
 }
 // Initialisierungen vor jedem Testfall
 @Before public void init() {
  list = new Vector();
```

```
  ...
 }
 // Aufräumarbeiten nach jedem Testfall
 @After public void tearDown() {
  list.clear();
 }
 // Aufräumarbeiten ganz am Ende
 @AfterClass public void destroy() {
  list = null;
 }
 @Test public void meinTest1() {...}
 @Test public void meinTest2() {...}
 ... // weitere Tests
}
```

Ende Listing

Ignorieren von Testfällen

Gelegentlich tritt der Fall ein, dass ein Testfall zwar (evtl. nur teilweise) implementiert, aber nicht ausgeführt werden soll. Die Gründe können vielschichtig sein. Beispielsweise kann sich eine getestete Funktionalität geändert haben und die Anpassung des Testfalls steht noch aus oder der Testfall dauert in der aktuellen Testphase zu lange etc. Bisher blieb nur der Weg, Programmzeilen auszukommentieren und sie somit für den Compiler ununterscheidbar von reinen Kommentaren zu machen. Nun bietet JUnit mit der Annotation *Ignore* ein sauberes Verfahren an. Diese Annotation wird einer Testmethode vorangestellt und bewirkt, dass der Testfall zwar registriert, aber nicht ausgeführt wird. Listing 3 zeigt die Verwendung von *Ignore*. Wie zu erkennen ist, existiert für die Annotation ein optionaler String-Parameter, der die Angabe eines Kommentars erlaubt. Dieser hat aber momentan keine tiefere Bedeutung, da er für ignorierte Testfälle nicht ausgegeben wird. Weiterhin im Listing zu sehen ist, dass mit *Ignore* und *Test* zwei Annotations für eine Methode angebracht wurden. Das sei nur als Hintergrundinformation erwähnt, da es nicht selbstverständlich erscheint, dass von Seiten Java 5 grundsätzlich beliebig viele Annotations pro Klasse oder Methode möglich sind.

Listing 3

Ignorieren von Testfällen

```
import static org.junit.Assert.*;
import org.junit.Test; //Annotation
import org.junit.Ignore; //Annotation
...
 @Ignore("Kommentar ist optional");
 @Test public void unfertigerTest() {
  Kunde kunde = new Kunde();
  int alter;
  //kunde.getAlter() ist noch nicht
  // ausimplementiert
  alter = kunde.getAlter();
  assertTrue(alter > 0);
 }
```

Ende Listing

Testen von Ausnahmen

Die Klasse *junit.framework.ExceptionTestCase* diente in früheren JUnit-Versionen der Prüfung, ob in einem Test eine erwartete Ausnahme (Exception) auch auftrat. Sie wird nun nicht mehr angeboten. Das hat seinen Grund: Ihre Verwendung war so umständlich, dass es bisher ratsam war, auf keinen Fall von ihr Gebrauch zu machen. Eine quasi proprietäre Möglichkeit, gegen erwartete Exceptions zu testen, bietet der optionale Parameter *expected* der Annotation *Test*. Ein Beispiel hierzu ist in Listing 4 dargestellt. Die erwartete Ausnahme ist dort mit *IllegalArgumentException* angegeben. Sie tritt auf, weil an Methode *subList* ein Startindex übergeben wird, der größer als der Endindex ist. Bisher mußte ein solcher Testfall mittels *try-catch*-Block und *fail()*-Befehl vor dem *catch* abgehandelt werden. Sollen Eigenschaften wie der Meldungstext eines Ausnahmeobjekts geprüft werden, ist allerdings auf die alte Vorgehensweise zurückzugreifen.

Listing 4

Testen von Ausnahmen

```
@Test (expected=IllegalArgumentException.class)
public void irgendeinTest() {
 List list = new Vector();
 // Folgendes generiert Exception
 List subList = list1.subList(1, 0);
 assertEquals(0, subList.size());
}
```

Ende Listing

Testen von Timeouts

Eine recht simple Möglichkeit, die Laufzeit eines Testfalls zu begrenzen, bietet der Parameter *timeout* der Annotation *Test*. Er ist genauso zu verwenden wie der eben genannte Parameter *expected*. Nach Überschreitung der angegebenen Zeit in Millisekunden schlägt der Test fehl. In Multitasking-Umgebungen, deren Hardware zudem differieren kann, ist dieses Vorgehen jedoch fragwürdig. Diese JUnit-Unterstützung zu nutzen, sollte also gut überlegt sein. Aus meiner Sicht wird es aus dem genannten Grund in Zukunft weitere Debatten dazu geben.

Parametrisierbare Testfälle

Ein parametrisierter Testfall erlaubt die kompakte Definition eines Tests, der mit verschiedenen Eingaben und zugehörigen Erwartungswerten wieder und wieder ausgeführt wird. Um solche Testfälle zu definieren, wurde zwei Klassen von JUnit 4 bereitgestellt. Erstens die Klasse *Parameterized* im Package *org.junit.runner.extensions*, zweitens die Annotation *Parameters*, die eine innere Klasse der ersten ist. Listing 7 zeigt die Umsetzung parametrisierter Testfälle in einer eigenen Testklasse. Die Testklasse wird mit *@RunWith(Parameterized.class)* annotiert. Weiterhin wird eine Methode, die Parameterwerte bereitstellt, mit *@Parameters* annotiert. Im Konstruktor der Klasse werden die Parameter gespeichert. Schlussendlich führt eine gewöhnliche Testmethode (Annotation *@Test*) die eigentliche Prüfung durch.

Testsuiten

JUnit 4 geht mit Testsuiten anders um als seine Vorgängerversionen. Die Listings 5 und 6 demonstrieren zwei Möglichkeiten, Testsuiten zu definieren. In beiden Listings wird die bisher noch nicht genannte Annotation *RunWith* verwendet. Der von ihr bereitgestellte Parameter erlaubt die Angabe einer Serviceklasse, die feststellt, welche Testklassen letztendlich ausgeführt werden sollen. Die angegebene Dienstklasse *AllTests* macht nichts weiter, als nach einer statischen Methode namens *suite* zu suchen und diese auszuführen. Das ist quasi die Emulation des Verhaltens aus früheren JUnit-Versionen. In Listing 6 wird ein anderes Verfahren angewandt. Hier lautet die Serviceklasse *Suite*. Sie sucht nach einer Annotation namens *SuiteClasses* und berücksichtigt die darüber spezifizierten Klassen bei der Suche nach Testfällen. Deshalb ist auch die letztgenannte Annotation im Listing 6 zu sehen. Ihr Parameter ist ein Array mit Elementen vom Typ *Class*. Es kann eine beliebige Anzahl an Klassen angegeben werden. Das bisher gewohnte und umständliche *addTest(TestKlasse.suite())* samt Implementierung einer entsprechenden *suite()*-Methode in der Testklasse entfällt.

Listing 5

Definieren einer TestSuite – Variante 1

```
import junit.framework.*;
import org.junit.runner.extensions.AllTests;
import org.junit.runner.RunWith;

@RunWith(org.junit.runner.extensions.AllTests.class);
public class MeineSuite {
```

```
  public static Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTest(MeinTest.suite());
    suite.addTest(WeitererTest.suite());
    return suite;
  }
}
```

Ende Listing

Listing 6

Definieren einer TestSuite – Variante 2

```
import org.junit.runner.RunWith;
import org.junit.runner.extensions.Suite.SuiteClasses;
import org.junit.runner.extensions.Suite;

@RunWith(Suite.class);
@Suite.SuiteClasses({MeinTest.class, WeitererTest.class})
public class MeineSuite {
}
```

Ende Listing

Listing 7

Parametrisierte Testfälle

```
import java.util.*;
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.junit.runner.extensions.Parameterized;
import org.junit.runner.extensions.Parameterized.Parameters;

@RunWith(Parameterized.class)
public class ParamTest {
 private int m_eingabe;
 private int m_erwartet;

 @Parameters
 public static Collection<Object> params() {
   return Arrays.asList(new Object[][] { {1,2}, {3,28}});
 }
 public ParamTest(int eingabe, int erwartet) {
   m_eingabe = eingabe;
   m_erwartet = erwartet;
 }
 @Test public void derTest() {
   assertEquals(m_erwartet, berechne(m_eingabe));
 }
 private int berechne(int x) {
   return (int)Math.pow(x, x) + 1;
 }
}
```

Ende Listing

Ausführen von Testfällen und Testsuiten

Im Unterschied zu früher wird nur noch ein Testrunner mitgeliefert, mit dem die Ausführung der Testfälle möglich ist. Er verbirgt sich hinter das Klasse *org.junit.runner.JUnitCore*, ein nicht unbedingt selbstsprechender Name. Dieser Testrunner erlaubt es, beliebig viele Testklassen hintereinander als Parameter anzugeben, etwa so: *java -cp .;junit.jar org.junit.runner.JUnitCore MeinTest WeitererTest*. Aufgrund der variablen Arrays (Varargs), die mit Java 5 eingeführt wurden, hat sich die Unterstützung beliebig vieler Übergabeparameter entsprechend simpel gestaltet. Als Parameter ist selbstverständlich auch der Name einer Klasse, die eine Testsuite definiert, möglich! Das Ergebnis wird nach Ausführung der Unit-Tests rein textuell ausgegeben. Im Ergebnisreport werden erfolgreiche Tests mit einem Punkt dargestellt, Fehler mit einem „E“ und ignorierte Tests mit einem „I“.

Die textuelle Ergebnisausgabe ist nicht unbedingt das gewohnte Bild. Einerseits wird es sicher weitere Testrunner geben (sei es direkt von JUnit oder erstellt von Dritten). Andererseits werden populäre Entwicklungsumgebungen wie Eclipse das neue JUnit 4 direkt unterstützen und die Ergebnisdarstellung wie gewohnt (oder evtl. besser) anbieten. Nach Angaben von David Saff, einem in die JUnit-Entwicklungen

involvierten Insider, wird Eclipse 3.2 M6 mit voraussichtlichem Veröffentlichungsdatum Ende März die neue JUnit-Version direkt unterstützen. Auch wird es in absehbarer Zeit eine direkte Unterstützung durch das populäre Build-Tool Ant geben.

Alte JUnit-Testfälle ausführen

Für Testsuiten wurde eben bereits beschrieben, wie sie an JUnit 4 angepasst werden können. Für ordinäre Testklassen ist es einfacher. Für diese muss lediglich die *suite()*-Methode adaptiert werden. Dabei hilft die extra dafür bereitgestellte Klasse *junit.framework.JUnit4TestAdapter*. Anstatt wie früher zu schreiben *return new TestSuite(MeinTest.class);*, heißt es jetzt: *return new JUnit4TestAdapter(MeinTest.class);*. Den Rest erledigt die Adapterklasse. Migrationsprobleme sollten damit kein Thema sein.

Fazit

JUnit 4 ist noch nicht vollständig ausgereift. Es ist noch ein gutes Stück des Weges. Auch die Javadoc-Kommentare lassen noch zu wünschen übrig. Vielleicht schon bei Erscheinen dieses Heftes wird der bisher nur inoffiziell erhältliche Release Candidate 2 offiziell verfügbar sein. Die Quellen im Archivsystem jedenfalls haben sich gegenüber diesem schon weiterentwickelt, Umstrukturierungen sind erkennbar. Was Testrunner sowie Unterstützung bei der Ausführung von JUnit 4-Tests durch Entwicklungsumgebungen angeht, ist die Situation ähnlich gelagert. Auch hier wird es bald die notwendigen Erweiterungen geben (etwa grafische Testrunner). Früher erstellte Unit Tests können einfach umgeschrieben werden, um mit der neuen JUnit-Version zu funktionieren. Es stellt sich jedoch die Frage, ob zum derzeitigen Zeitpunkt eine Umstellung von JUnit 3.x auf JUnit 4 lohnt, wenn bereits eine hinreichende Menge an Testklassen vorhanden ist. Zu groß scheint die Ungewißheit über Entwicklungen der nahen Zukunft hierfür, zu gering der (momentane) Nutzen. Bleibt abzuwarten, wie die JUnit-Welt zu einem späteren Zeitpunkt in diesem Jahr aussehen wird. Wer die neuen Sprachkonstrukte von Java 5 nutzen will, hat jedenfalls die Möglichkeit der Portierung. Abschließend sei die Prognose gewagt, dass diese Sprachkonstrukte, insbesondere Annotationen, einen fruchtbaren Boden für JUnit-Erweiterungen durch Dritte bereitstellen. Der Drittanbietermarkt wird dadurch – so die Prognose weiter – deutlich wachsen. Darauf ist JUnit auch angewiesen, denn die Philosophie lautete bisher, nur das wirklich in breiter Masse benötigte in die Kerndistribution zu integrieren. Und dieses Prinzip ist auch mit JUnit 4 nicht verletzt worden. Das JUnit-Buch [7] gibt Einblick in die neuesten Entwicklungen zu JUnit 4 und beleuchtet intensiv das Thema automatisierte Softwaretests mit JUnit.

Klaus Meffert arbeitet als Organisationsberater der Brandt & Partner GmbH in der Software-Entwicklung. Er arbeitet parallel an seiner Doktorarbeit zum selben Thema. Weiterhin engagiert er sich im Open-Source Bereich und als Fachautor.

Links & Literatur

[1] JUnit-Homepage: www.junit.org

[2] Download von JUnit 4 RC2: www.junit-buch.de

[3] Barbara Beenen: Rot-grünes Wechselspiel. Automatisiertes Testen mit JUnit – eine Einführung, in *Java Magazin* 5.2004

[4] Barbara Beenen: Bitte einsteigen. Komplexe Probleme meistern mit JUnit, in *Java Magazin* 6.2004

[5] Barbara Beenen: Alles dreht sich um den grünen Balken. Tools rund um JUnit, in *Java Magazin* 7.2004

[6] Java 5-Neuerungen: java.sun.com/j2se/1.5.0/docs/relnotes/features.html

[7] Klaus Meffert: JUnit Profi-Tipps. Bewährte Techniken und Antipatterns, Software & Support Verlag (erscheint im Mai 2006)
[8] TestNG-Homepage: http://testng.org