Kurzanleitung JUnit

JUnit ist ein Testframework zum Testen von Java-Code. Es eignet sich besonders gut zum automatisierten Testen und ist ein wichtiges Hilfsmittel des Extreme Programmings, bei dem der Test-First-Ansatz angewendet wird. JUnit ist aber auch zum Testen von bereits bestehenden Code geeignet. Zu beachten ist, dass der Programmcode ggf. umstrukturiert werden muss, damit er mit JUnit getestet werden kann.

Diese Anleitung bezieht sich auf JUnit 4, das z.Zt. in der Version 4.4 vorliegt. Eine der wichtigsten Neuerungen gegenüber früheren Versionen (3.8 oder älter) ist die Nutzung von Annotationen. JUnit-Tests, die der früheren Spezifikation genügen, können auch weiterhin mit der neuen Version ausgeführt werden.

1 Grundlagen

Um JUnit verwenden zu können, muss `junit.jar` als Bibliothek verfügbar sein. Die Schnittstellenberscheibung ist in [1] zu finden.

1.1 Begriffsdefinitionen

Testklasse/Testcode Als Testklasse wird eine Klasse bezeichnet, die Methoden zum Testen von Code enthält. Die einzige Bedingung besteht darin, dass sie durch einen öffentlichen Default-Konstruktor instanzierbar ist. In der Testklasse sind mit Hilfe der JUnit-Annotationen Testmethoden gekennzeichnet.

Testmethode Testmethoden bezeichnen Methoden, die durch Annotationen als solche gekennzeichnet sind. Jede Methode, die das Sichtbarkeitsattribute „public“ besitzt, keine Parameter verlangt und als Return-Typ void liefert, darf als Testmethode gekennzeichnet werden.

Testfall Ein Testfall ist zunächst ein bestimme Vorgehensweise, wie ein Teil einer Software getestet werden soll, zu ihm gehören Testwerte und das erwartete Ergebnis. Abhängig von der Komplexizität der zu testenden Funktionalität können Testfälle der selben Art zu einer Testmethode zusammen gefasst werden.

Testframework JUnit 4 wird als Testframework bezeichnet.

1.2 Empfehlungen

In jeder beliebigen Klasse können Testfälle definiert werden. Aus Gründen der Übersicht ist zu empfehlen, Testcode getrennt zu halten und die Testmethoden in eine oder mehrere separate Klassen auszulagern. Der Name der Klasse kann frei gewählt werden. Eine verbreitete Konvention ist, den Namen aus dem Namen der zu testenden Klasse und dem Postfix „`Test`“ zusammen zusetzen.

Die Testklasse sollte für jede Methode, in der etwas zu testen ist, mindestens eine Testmethode enthalten. Neben der Funktionalität sind auch die Vorbedingungen zu prüfen. Die Testmethoden überprüfen jeweils nur einen atomaren Teil, so dass eine Methode auch durch verschiedene Tests getestet werden sollte. Die Vorbedingungen sollten in der Reihenfolge, in der sie im zu testenden Code definiert sind, geprüft werden. Es erleichtert später die Suche nach der Ursache für fehlgeschlagene Tests.

Keinen Testcode zum Testen des Compilers schreiben!

2 Verwendung

Testmethoden werden durch die JUnit-Annotation `@Test` gekennzeichnet. Zum Überprüfen liefert das Testframework die Klasse `org.junit.Assert`. Tritt eine Abweichung auf, wird ein `java.lang.AssertionError` oder ein davon abgeleiteter Fehler geworfen.

2.1 Einfache Testklasse

Die Verwendung soll zunächst durch ein einfaches Beispiel verdeutlicht werden.

```
1  import java.util.*;
2  import org.junit.Test;
3  import static org.junit.Assert.*;
4
5  public class CollectionTest {
6
7    @Test
8    public void sortedSet(){
9      Set<String> s = new TreeSet<String>();
10     s.add("Bär");
11     s.add("Aal");
12
13     Iterator<String> iter = s.iterator();
14     assertEquals("Aal", iter.next());
15     assertEquals("Bär", iter.next());
16   }
17 }
```

Listing 1: einfache Testklasse

Listing 1 zeigt ein kurzes Beispiel. Zuerst werden die benötigten Klassen importiert. In Zeile 3 wird die Technik der statischen Imports, die seit Java 1.5 möglich sind, genutzt. Sie bewirkt, dass alle statischen Methoden der Klasse Assert lokal verfügbar sind.

Die Methode `sortedSet()` prüft, ob die Klasse `TreeSet` aus dem Package `java.util` Strings korrekt sortiert. Dazu werden zwei Strings in einen `TreeSet` eingefügt, anschließend per `Iterator` der Reihe nach aus der `Collection` gelesen und mit assertEqauls auf die erwarteten Werte überprüft.

2.1.1 JUnit mit Eclipse ausführen

Das Ausführen des Test aus Eclipse heraus ist sehr komfortabel. Die Testklasse wird per Rechtsklick selektiert. Im Kontextmenü muss nun der Punkt „Run As -> JUnit Test“ ausgewählt werden. Eclipse öffnet automatisch die JUnit-Ansicht und zeigt Erfolg oder Misserfolg durch grüne bzw. rote Balken an.

Abbildung 1 auf der nächsten Seite zeigt den Erfolg des in Listing 1 definierten Tests an. Die Implementierung des TreeSets funktioniert wie erwartet: Strings, die in das Set abgespeichert werden, legt es sortiert von A bis Z ab.

Der Test in Abbildung 2 ist hingegen fehl geschlagen, da nun als erwartete Werte die umgekehrte Reihenfolge angegeben wurde, also zuerst „Bär“ und dann „Aal“. Es wird ein `java.lang.AssertionError` geworfen. Wie im unteren Teil der Abbildung zu erkennen, steht dort die Info welcher Fehler aufgetreten ist, jeweils mit dem erwarteten und dem tatsächlich erhaltenen Wert. Außerdem wird die Klasse samt Zeilennummer, in der der Fehler aufgetreten ist, ausgegeben.

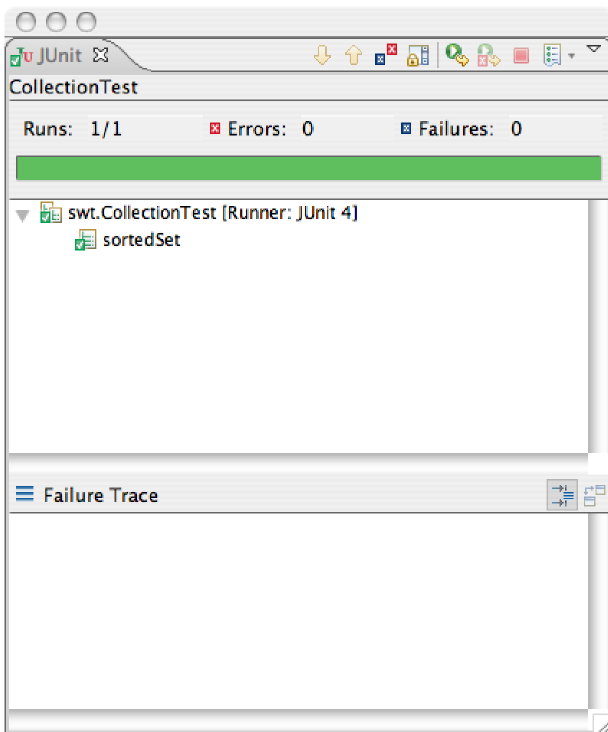

Abbildung 1: JUnit mit Eclipse im Erfolgsfall

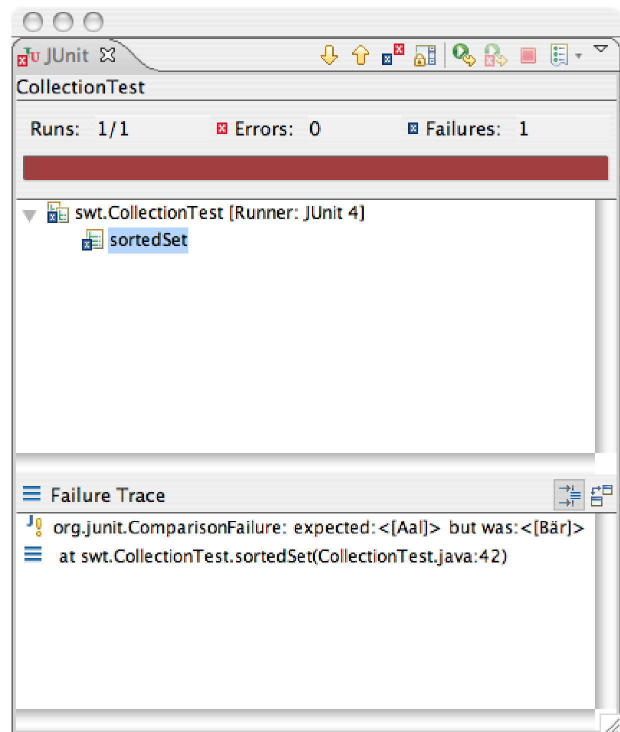

Abbildung 2: JUnit mit Eclipse im Fehlerfall

2.1.2 JUnit mit der Kommandozeile ausführen

Selbstverständlich können JUnit-Tests auch ausserhalb von Eclipse aufgerufen werden, sowohl über die Kommandozeile als auch mit Hilfe von Ant.

Für das Testen über die Kommandozeile ist es nötig die JUnit-Bibliothek zum Classpath hinzuzufügen und danach folgendes auszuführen:

`$ java org.junit.runner.JUnitCore` *Pfad_und_Name_der_Testklassen*

2.2 Das Werfen von Exceptions testen

Neben der „normalen“ Funktionalität sollte beim Test auch die korrekte Fehlerbehandlung getestet werden. JUnit unterstützt das gezielte Abfangen von Exceptions. In Listing 2 auf der nächsten Seite ist eine Beispieltest zu finden.

```
import java.util.ArrayList;
import java.util.List;
import org.junit.Test;

public class ExceptionTest {

    @Test(expected = IndexOutOfBoundsException.class)
    public void exceptionList() {

        List<String> l = new ArrayList<String>();
        l.add("Bär");
        l.add("Aal");

        l.get(2);
    }

}
```

Listing 2: Das Werfen einer Exception testen.

In Zeile 7 wird an die Annotation `@Test` der Zusatz `(expected = IndexOutOfBoundsException.class)` angehangen. Er besagt, dass das Auftreten einer Exception erwartet wird. Die Exception tritt in Zeile 14 auf, da der Index um 1 überschritten wird. In diesem Fall wird eine `RuntimeException`, also eine unchecked Exception, erwartet. Wird auf eine checked Exception geprüft, muss die Testmethodendeklaration mit dem Zusatz `throws ...Exception` versehen werden.

2.3 Einen definierten Startzustand herstellen

Häufig kommt es vor, dass viele oder sogar alle Testfälle einer Testklasse eine identische Umgebung benötigen. Werden zum Beispiel die Methoden der Klasse `List` getestet, so benötigt jeder Testfall mindestens ein Objekt vom Typ `List`. Auch ist es für viele Fälle sehr sinnvoll, wenn die List-Instanz bereits einige Objekte enthält.

Um diese Vorbereitungen nicht in jeder Testmethode erneut hinschreiben zu müssen, bietet JUnit die Annotation `@Before`. Sie wird vor eine Methode geschrieben und sorgt dafür, dass diese Methode vor *jedem* Testfall aufgerufen wird. Listing 3 zeigt ein einfaches Beispiel.

```
import java.util.ArrayList;
import java.util.List;
import org.junit.Test;
import org.junit.Before;

import static org.junit.Assert.*;

public class ListTest {

    private List<String> list1;
    private List<String> list2;

    @Before
    public void setUp() {
        list1 = new ArrayList<String>();
        list1.add("William DeWorde");
        list1.add("Herbert Starkimarm");


```

```
20          list2 = new ArrayList<String>();
21          list2.add("Samuel Mumm");
22      }
23
24      @Test
25      public void remove() {
26          list1.clear();
27          assertEquals(0, list1.size());
28      }
29
30      @Test
31      public void addAll() {
32          list1.addAll(list2);
33          assertEquals(3, list1.size());
34      }
35  }
```

Listing 3: Einen definierten Zustand für alle Testfälle erzeugen.

Ab Zeile 14 wird hier die Methode `setUp` definiert. Sie instanziiert zwei Listen und füllt bereits einige Wert in die Listen ein. Die Listen werden als private Klassenvariable in den Zeilen 10 und 11 deklariert, so dass sie aus allen Testmethoden zugegriffen werden können. Dadurch, dass die `setUp`-Methode mit der Annotation `@Before` gekennzeichnet ist, wird sie vor jeder Testmethode aufgerufen und erzeugt so vor jedem Test die beiden Listen mit dem entsprechenden Inhalt.

2.4 Mehrere Testklassen gleichzeitig ausführen

Um die Funktionalität einer Klasse zu testen, reicht es aus, die Testklassen separat ablaufen zu lassen. Möchte man ein ganzes Projekt auf korrekte Funktion testen, können sie zu einer Testsuite zusammengefasst werden.

```
1   import org.junit.runners.Suite;
2   import org.junit.runners.Suite.SuiteClasses;
3   import org.junit.runner.RunWith;
4
5   @RunWith(Suite.class)
6   @SuiteClasses( {
7     CollectionTest.class,
8     StringBuilderTest.class
9   })
10  public class TestSuite {
11
12  }
```

Listing 4: TestSuite

Das Listing 4 zeigt die Klasse `TestSuite`. Sie besitzt zwei Annotationen: `@RunWith` meldet JUnit, dass es zum Ausführen dieser Klasse die Klasse `org.junit.runners.Suite` verwenden soll und nicht den integrierten Standard-Test-Runner. `@SuiteClasses` ordnet dieser Suite konkrete zu testende Klassen zu. Sie werden als Komma-separierte Liste in geschweiften Klammern, also als Array, erwartet.

Das Ausführen der Testsuite folgt dem selbem Schema wie ein separater Test (siehe Abschnitt „Einfache Testklasse“). Alle als SuiteClasses definierten Klassen werden nacheinander ausgeführt.

3 Kurz-Referenz

Dieser Abschnitt enthält eine Auswahl der in JUnit 4 existierenden Annotationen und Assert-Methoden. Diese Referenz ist nicht als vollständig anzusehen. Sie enthält lediglich die wichtigsten Punkte zum schnellen Nachschlagen.

3.1 Annotationen für Methoden

Annotation	Beschreibung
`@Test`	Kennzeichnung als Testfall
`@Test(expected=`...*Exception*`.class)`	Kennzeichnung als Testfall mit der Festlegung, dass der Test nur erfolgreich ist, wenn die geforderte Exception auftritt.
`@Test(timeout =`...*ms*`)`	Kennzeichnung als Testfall mit der Festlegung, dass der Test erfolgreich ist, wenn die geforderte Zeit in ms nicht überschritten wird.
`@Before`	Ausführung vor jedem Aufruf einer Testmethode zum Aufbau einer definierten Testumgebung.
`@After`	Ausführung nach jedem Aufruf einer Testmethode zur Erledigung von Aufräumarbeiten.
`@BeforeClass`	Eine derartig gekennzeichnete Methode muss statisch definiert sein. Diese Annotation dient der Kennzeichnung zur einmaligen Ausführung vor dem Aufruf aller anderen Testmethoden.
`@AfterClass`	Eine derartig gekennzeichnete Methode muss statisch definiert sein. Diese Annotation dient der Kennzeichnung zur einmaligen Ausführung nach dem Aufruf aller anderen Testmethoden.
`@Ignore("`*Kommentar*`")`	Methode wird temporär nicht ausgeführt. Es sollte unbedingt ein Kommentar mit übergeben werden, dieser wird im Protokoll des Testlaufs ausgegeben.

3.2 Annotationen für Klassen

Annotation	Beschreibung
`@RunWith(... .class)`	Angabe der Klasse mit der diese Klasse ausgeführt werden soll. Dies ist nötig, wenn ein anderer Runner als der Standard-Runner verwendet werden soll.
`@Suite.SuiteClasses({... .class, ...})`	Definition der Klassen, die zu dieser Suite gehören

3.3 Auswahl an Assert-Methoden

Die folgende Tabelle zeigt eine Auswahl der Methoden, die sich in der Klasse org.junit.Assert befinden. Die Details und eine vollständige Liste ist in [1] zu finden.

Alle Methoden werfen im Fehlerfall einen `java.lang.AssertionError`, der ggf. noch Details zum Fehler enthält. Jede Assert-Methode existiert in zwei Ausführungen: in der Form wie in der Tabelle gelistet und zusätzlich mit dem ersten Parameter `String message`, der weitere Informationen zum Fehler enthalten kann. Die Message wird durch den AssertionError ausgegeben.

Assert-Methode	**Beschreibung**
`assertEquals(Object exp, Object act)`	Überprüft anhand der equals-Methode, des jeweiligen Objektes, ob zwei Objekte identisch sind. Auch für float, double und Arrays möglich.
`assertEquals(float exp, float act, float delta)`	Überprüft, ob zwei floats bis auf die Differenz delta identisch sind. Auch für double möglich.
`assertFalse(boolean condition)`	Prüft, ob die Bedingung falsch ist.
`assertNotNull(Object object)`	Überprüft, ob das Objekt ungleich null ist.
`assertNotSame(Object unexp, Object act)`	Prüft, ob zwei Referenzen auf unterschiedliche Objekte verweisen.
`assertNull(Object object)`	Überprüft, ob das Objekt gleich null ist.
`assertSame(Object exp, Object act)`	Prüft, ob zwei Referenzen auf identische Objekte verweisen.
`assertTrue(boolean condition)`	Prüft, ob die Bedingung wahr ist.
`fail(String message)`	Keine Prüfmethode. Lässt einen Test explizit fehlschlagen.

Da es sich bei den Assert-Methoden um statische Methoden der Klasse `Assert` handelt, ist der Aufruf etwas länglich:

```
1 Assert.assertNotNull(o);
```

Dies ist insbesondere bei der häufigen Verwendung der Assert-Methoden in den Testmethoden eher unschön. Durch die Angabe der folgenden Import-Anweisung kann der Klassenname beim Aufruf der Assert-Methode entfallen und dadurch der Testcode deutlich kompakter geschrieben werden:

```
1 import static org.junit.Assert.*;
```

Literatur

[1] JUnit Entwickler: Kent Beck, Erich Gamma & et al. *JUnit Api*. URL `http://junit.sourceforge.net/javadoc_40`.

Autor: Nadja Krümmel, Malte Ried, Institut für SoftwareArchitektur.