

## **JUnit & Friends**

(Link zum Artikel: <http://www.it-republik.de/jaxenter/artikel/2528>)

### **Ein Überblick über das Test-Framework und sinnvolle Ergänzungen**

Text: Martin Bisanz

Das Ganze ist bekanntlich mehr als die Summe seiner Teile. Aber wenn schon die Teile eines Softwaresystems nicht so sind, wie sie sein sollen, wird die Summe kaum besser sein. Ein wichtiges Mittel, um die Qualität von Softwarebausteinen zu erhöhen, sind Unit-Tests.

Unit- bzw. Komponententests sind ein grundlegender Bestandteil für die Qualitätssicherung von Software. Sie testen isoliert die Funktionalität einzelner Bausteine – in Java zumeist einzelner Klassen – bevor diese zu einem Gesamtsystem zusammengesetzt werden. Sie bilden die erste Teststufe vor Integrations- und Systemtests und können somit schon unmittelbar während der Implementierungsphase durchgeführt werden. Dadurch ermöglichen sie ein frühes Feedback und eine kostengünstige Beseitigung von Fehlern.

Charakteristisch für Unit-Tests ist, dass Komponenten isoliert von anderen Teilen des Systems getestet werden (siehe: [A. Spillner, T. Linz: Basiswissen Softwaretest, dpunkt Verlag, 2004](#) [1]). So soll sichergestellt werden, dass ausschließlich die von der Komponente bereitgestellte Funktionalität geprüft wird. Die Wechselwirkung mit anderen Komponenten hingegen ist Gegenstand des Integrationstests.

In Java hat sich [JUnit](#) [2] als De-facto-Standard für komponentenbasiertes Testen etabliert. Das u.a. von Erich Gamma und Kent Beck entwickelte Framework ermöglicht das automatisierte Unit-Testen von Java-Programmen. Es ist schnell und einfach einzusetzen, die Tests werden komplett in Java geschrieben, und alle gängigen Entwicklungsumgebungen bieten Unterstützung für die Ausführung und Auswertung der Testläufe. So ist der Reibungsverlust für Java-Entwickler beim Einsatz von JUnit gering.

Bereits seit 2006 ist JUnit 4 verfügbar (aktuell ist 4.5 vom August dieses Jahres), das sich gegenüber der immer noch weit verbreiteten 3er-Version vor allen Dingen durch die konsequente Verwendung von Annotationen auszeichnet. Auch wurden einige recht unflexible Strukturvorgaben der Vorgängerversion aufgehoben – die Testfallklasse muss beispielsweise nicht mehr von TestCase abgeleitet werden, die Methodennamen nicht mehr mit *test* beginnen.

Die aktuelle Version rückt damit näher heran an [TestNG](#) [3], den Pionier in Sachen annotationsbasiertem Testen. Auf den ersten Blick sehen beide Frameworks auch sehr ähnlich aus, haben aber letztendlich unterschiedliche Schwerpunkte: JUnit fokussiert auf Unit-Tests, während TestNG mehr in Richtung Integrationstests geht. Beispielsweise können in TestNG Abhängigkeiten zwischen Testfällen definiert werden. Somit werden abhängige Testfälle übersprungen, wenn einer fehlschlägt.

## JUnit Basics

Das Grundprinzip von JUnit ist simpel: Ein JUnit-Test besteht aus einer Anzahl von Testmethoden, die unabhängig voneinander ausgeführt werden. Ein Test kann nur entweder erfolgreich oder nicht erfolgreich verlaufen – schlägt eine Testmethode fehl, so scheitert der gesamte Test. Daneben gibt es noch vor- und nachbereitende Methoden, die jeweils vor und nach den Testmethoden ausgeführt werden und für die diese einen definierten Zustand herstellen.

Einen beispielhaften JUnit-Test zeigt Listing 1. Gegenstand ist ein einfacher Service, dessen Interface in Listing 2 zu sehen ist. Der Service liefert in Abhängigkeit von den Parametern *type* und *value* ein Resultat als String zurück. Die Testklasse besteht aus drei Methoden, die sowohl Positiv- als auch Negativtests realisieren. Das Beispiel soll nur zur Veranschaulichung dienen, in einem realen Test würde man noch weitere Testmethoden ergänzen.

### Listing 1

```
1. package com.prodyna.jm;
2.
3. import static org.junit.Assert.assertEquals;
4.
5. import org.junit.Before;
6. import org.junit.Test;
7.
8. public class SimpleServiceImplTest {
9.
10. private SimpleServiceImpl service;
11.
12. @Before
13. public void prepare() {
14. service = new SimpleServiceImpl();
15. }
16.
17. @Test
18. public void testInvokeSuccess() throws Exception {
19. String type = "TYPE_1";
20. int value = 13;
21. String expected = SimpleService.SUCCESS;
22. String actual = service.invoke(type, value);
23. assertEquals(expected, actual);
24. }
25.
26. @Test
27. public void testInvokeInvalid() throws Exception {
28. String type = "TYPE_1";
29. int value = 12;
30. String expected = SimpleService.ERROR;
```

```

31. String actual = service.invoke(type, value);
32. assertEquals(expected, actual);
33. }
34.
35. @Test(expected = UnknownTypeException.class)
36. public void testInvokeUnknownType() throws Exception {
37. String type = "TYPE_2";
38. int value = 13;
39. service.invoke(type, value);
40. }
41. }
42.

```

## Listing 2

```

1. package com.prodyna.jm;
2.
3. public interface SimpleService {
4.
5. public static final String SUCCESS = "SUCCESS";
6. public static final String ERROR = "ERROR";
7.
8. public String invoke(String type, int value)
9. throws UnknownTypeException;
10.
11. }
12.

```

Jede Testmethode ist mit *@Test* annotiert und damit als solche gekennzeichnet. Die *prepare* Methode wird per *@Before*-Annotation vor jeder Testmethode ausgeführt und sorgt für die Initialisierung des Testobjekts. Daneben lässt sich per *@BeforeClass*-Annotation eine Methode einmalig vor Ablauf aller Testmethoden ausführen. Analog existieren *@After*- und *@AfterClass*-Annotationen für die Ausführung nach jeder oder allen Testmethoden.

Die Methode *testInvokeUnknownType* erwartet zudem, dass eine *UnknownTypeException* geworfen wird. Der Test schlägt folglich fehl, wenn keine solche Exception gefangen werden kann.

## Abhängigkeiten

In den seltensten Fällen funktionieren Klassen völlig unabhängig von einander, vielmehr benötigen sie zur Laufzeit andere Objekte, deren Methoden zur Umsetzung der eigenen Funktionalität aufgerufen werden. Um ein Objekt einer solchen Klasse isoliert testen zu können, müssen alle ihre Abhängigkeiten von der Testumgebung simuliert werden. Häufig verwendet man so genannte Mock-Objekte als Platzhalter für die echten Abhängigkeiten eines

zu testenden Objekts. Die echten Abhängigkeiten haben häufig selbst wieder Abhängigkeiten, sodass eine Kette entstehen kann, oder sie sind in der Testumgebung nicht lauffähig (z.B. weil sie die Existenz einer Datenbankverbindung voraussetzen). Mock-Objekte implementieren dagegen nur die Schnittstellen, auf die das zu testende Objekt zugreift, und liefern die zum Testfall passenden Ergebnisse zurück. Sie sollten möglichst einfach sein und selbst nur so viel Programmlogik enthalten wie unbedingt notwendig.

Die Verwendung von Mock-Objekten ist nicht immer problemlos möglich und hängt entscheidend davon ab, wie das Objekt seine Abhängigkeiten erhält. Wenn etwa ein Objekt der Klasse A ein Objekt der Klasse B benötigt, kann A:

- Eine Instanz von B mittels *new* erzeugen
- Eine statische Methode verwenden, die eine Instanz von B zurückliefert
- B nicht selbst instanziiieren, sondern von außen setzen lassen

Der letzte Ansatz entspricht dem *Dependency-Injection*-Entwurfsmuster, auf dem Frameworks wie Spring oder Guice aufbauen. Es fördert eine weitgehende Entkopplung zwischen Klassen (A muss nicht wissen, wie es B erzeugt, sondern B nur verwenden) und erleichtert die Verwendung von Mock-Objekten. Die ersten beiden Ansätze sind grundsätzlich schwieriger mit dem Mock-Konzept zu vereinbaren, obwohl es auch hier einige Lösungsansätze gibt. Ein Design, das die Erzeugung und Bereitstellung von Abhängigkeiten von der eigentlichen Funktionalität einer Klasse trennt, erhöht aber in jedem Fall die Testbarkeit.

#### Mocken ...

Im Fall des *SimpleService* wäre es etwa denkbar, dass abhängig vom Parameter *type* ein anderer *ServiceHelper* zum Einsatz kommt, an den der Aufruf delegiert wird. Der Lookup des *ServiceHelpers* könnte über einen *ServiceHelperLocator* erfolgen. Listing 3 zeigt eine entsprechende Implementierung.

#### Listing 3

```
1. package com.prodyna.jm;
2.
3. public class SimpleServiceImpl implements SimpleService {
4.
5.     private ServiceHelperLocator locator;
6.
7.     public String invoke(String type, int value) throws
        UnknownTypeException {
8.         ServiceHelper helper = locator.locate(type);
9.         return helper.invoke(value);
10.    }
11.
12.    public void setLocator(ServiceHelperLocator locator) {
13.        this.locator = locator;
14.    }
15. }
```

16.

Für dieses Szenario würde der Test aus Listing 1 nicht mehr funktionieren – der Service benötigt den *ServiceHelperLocator*, und ein *invoke* ohne vorheriges *setServiceLocator* führt in allen drei Testmethoden zu einer *NullPointerException*. Also wird für den Test ein *ServiceHelperLocator* benötigt – in der Regel ein Mock-Objekt.

In manchen Fällen lässt sich ein Mock-Objekt einfach selbst implementieren, etwa durch eine anonyme Klasse. Allerdings ist diese Vorgehensweise potenziell fehleranfällig, artet schnell in Schreiarbeit aus und verschlechtert die Lesbarkeit, besonders wenn das Mock-Interface groß ist und der Test nur einen kleinen Teil des Interfaces verwendet. Zudem lässt sich nicht ohne weiteres kontrollieren, ob das Mock-Objekt auf die richtige Art und Weise verwendet wird.

#### ... oder mocken lassen

Abhilfe schaffen Mock-Frameworks, die das Erzeugen von Mock-Objekten vereinfachen und zusätzliche Kontrolle bei der Verwendung von Mock-Objekten bereitstellen. Ein Vertreter ist [EasyMock](#) [4]. Es erzeugt Mock-Objekte für beliebige Java-Interfaces zur Laufzeit, ohne dass eine Implementierung für die Interfaces geschrieben werden muss. Es unterstützt Rückgabewerte und Exceptions für die Methoden des Interfaces. Zudem ermöglicht es, die Anzahl der Methodenaufrufe zu überprüfen, sodass Tests im Fall unerwarteter Methodenaufrufe fehlschlagen.

Listing 4 zeigt einen JUnit-Test für den *SimpleService* mit EasyMock. Es wird über statische Methoden der Klasse *EasyMock* verwendet, die vom Testfall per *import static* importiert werden. Die *prepare*-Methode erzeugt zusätzlich zur *SimpleService* Instanz mit *createMock* ein Mock-Objekt für das *ServiceHelperLocator*-Interface.

#### Listing 4

```
1. package com.prodyna.jm;
2.
3. import static org.easymock.EasyMock.createMock;
4. import static org.easymock.EasyMock.expect;
5. import static org.easymock.EasyMock.replay;
6. import static org.easymock.EasyMock.verify;
7. import static org.junit.Assert.assertEquals;
8.
9. import org.junit.Before;
10. import org.junit.Test;
11.
12. public class SimpleServiceImplTestWithEasyMock {
13.
14.     private SimpleServiceImpl service;
15.     private ServiceHelperLocator mockLocator;
16.
17.     @Before
18.     public void prepare() {
```

```
19. service = new SimpleServiceImpl();
20. // create mock for ServiceHelperLocator interface
21. mockLocator = createMock(ServiceHelperLocator.class);
22. service.setLocator(mockLocator);
23. }
24.
25. @Test
26. public void testInvokeSuccess() throws Exception {
27. // prepare test data
28. String type = "TYPE_1";
29. int value = 13;
30. String expected = SimpleService.SUCCESS;
31. ServiceHelper helper = createMock(ServiceHelper.class);
32. // record expected behavior
33. expect(mockLocator.locate(type)).andReturn(helper);
34. expect(helper.invoke(value)).andReturn(expected);
35. // replay mocks and invoke service
36. replay(mockLocator, helper);
37. String actual = service.invoke(type, value);
38. // check result
39. verify(mockLocator, helper);
40. assertEquals(expected, actual);
41. }
42.
43. @Test
44. public void testInvokeInvalid() throws Exception {
45. // prepare test data
46. String type = "TYPE_1";
47. int value = 12;
48. String expected = SimpleService.ERROR;
49. ServiceHelper helper = createMock(ServiceHelper.class);
50. // record expected behavior
51. expect(mockLocator.locate(type)).andReturn(helper);
52. expect(helper.invoke(value)).andReturn(expected);
53. // replay mocks and invoke service
54. replay(mockLocator, helper);
55. String actual = service.invoke(type, value);
56. // check result
57. verify(mockLocator, helper);
58. assertEquals(expected, actual);
59. }
60.
61. @Test(expected = UnknownTypeException.class)
62. public void testInvokeUnknownType() throws Exception {
63. // prepare test data
64. String type = "TYPE_2";
65. int value = 13;
66. UnknownTypeException exception = new UnknownTypeException();
67. // record expected behavior
68. expect(mockLocator.locate(type)).andThrow(exception);
```

```

69. // replay mocks and invoke service
70. replay(mockLocator);
71. service.invoke(type, value);
72. verify(mockLocator);
73. }
74. }
75.

```

Ein so erzeugtes Mock-Objekt befindet sich anfangs im Aufnahmezustand, in dem es das erwartete Verhalten aufzeichnet. Um vom Aufnahme- in den Wiedergabezustand zu wechseln, muss die Methode *replay* aufgerufen werden, die eine beliebige Anzahl von (Mock-)Objekten als Parameter akzeptiert. Vor dem Wechsel in den Wiedergabezustand wird aufgezeichnet. Bei den dafür verwendeten Methoden *expect*, *andReturn* und *andThrow* sind die Namen Programm: *expect* zeichnet einen erwarteten Methodenaufruf auf, *andReturn* speichert das dazugehörige Rückgabeobjekt, *andThrow* eine dazugehörige Exception. Methoden vom Rückgabotyp *void* können im Aufnahmezustand direkt ohne *expect* aufgerufen werden.

Werden während der Wiedergabe Methoden des Mock-Objekts aufgerufen, die vorher nicht aufgezeichnet wurden, schlägt der Test mit einem *AssertionError* fehl. Umgekehrt muss überprüft werden, ob alle aufgezeichneten Methoden auch tatsächlich aufgerufen wurden. Dafür sorgt der Aufruf von *verify* nach dem Serviceaufruf.

Neben dieser Grundfunktionalität bietet EasyMock noch einige nützliche Zusatzfähigkeiten. So lassen sich mit *times*, *atLeastOnce* und *anyTimes* unterschiedlich viele Aufrufe einer Methode aufzeichnen. Soll die Reihenfolge der Methodenaufrufe eines Mock-Objekts bei der Wiedergabe mit der Aufzeichnung übereinstimmen, kann mit *createStrictMock* ein "strenges" Mock-Objekt erzeugt werden, das diese Überprüfung vornimmt. Mit *checkOrder* kann diese Überprüfung sogar auch zeitweise deaktiviert werden. Ein *createNiceMock* erzeugt hingegen ein Mock-Objekt, das auch unerwartete Methodenaufrufe toleriert.

EasyMock vergleicht die Parameter der erwarteten Methodenaufrufe mit *equals*, falls es sich um Objekte handelt. Alternativ kann man aus einer Vielzahl anderer so genannter Argument-Matcher wählen, die bestimmen, unter welchen Bedingungen die Argumente auf die Erwartung passen. Beispiele sind *isNull* für null, *notNull* für alles außer null, *isA* für ein beliebiges Objekt einer Klasse oder *anyObject* für ein beliebiges Objekt. Mit *and*, *or* und *not* lassen sich die einzelnen Argument-Matcher zu komplexen Ausdrücken zusammenbauen. Auch eigene Matcher-Implementierungen sind möglich.

Zu EasyMock existiert die Erweiterung *ClassExtension*, die es ermöglicht, auch Mock-Objekte für Klassen anstelle von Interfaces zu erzeugen. Dies ist besonders nützlich, wenn etwa Klassen aus vorhandenen Bibliotheken verwendet werden und sich Interfaces nicht ohne weiteres extrahieren lassen. Die *ClassExtension* stellt ihre Funktionalität über statische Methoden in *org.easymock.classextension.EasyMock* bereit. Um sie zu verwenden, muss also nur das Import-Statement entsprechend angepasst werden. Es wird auch ein partielles Mocking unterstützt. Dabei wird nur ein Teil der Methoden einer Klasse überschrieben, der Rest bleibt unverändert. Methoden, die *private* oder *final* sind, können nicht durch eine Mock-Implementierung ersetzt werden.

## Erweiterter Freundeskreis

Ähnlich wie EasyMock funktionieren [jMock](#) [5] und [rMock](#) [6]. [jMockit](#) [7] verwendet Java-Bytecode-Instrumentalisierung, um Klassen zur Laufzeit zu redefinieren und so durch Mock-Implementierungen zu ersetzen. Dadurch lassen sich auch in der zu testenden Klasse per *new* Operator erzeugte Objekte austauschen.

Neben den Mock-Frameworks gibt es zahlreiche andere Projekte, die JUnit um verschiedene Aspekte ergänzen. Teilweise verschiebt sich dadurch der Einsatzbereich in Richtung Integrations- und Systemtest. Datenbanknaher Code etwa lässt sich mit [DbUnit](#) [8] testen. Die JUnit-Erweiterung testet den Code nicht in Isolation, sondern in Zusammenspiel mit einer Datenbank. Dabei sorgt sie dafür, dass sich die Datenbank für den Test in einem definierten Zustand befindet. Nach dem Testlauf kann so überprüft werden, ob sich der Zustand der Datenbank entsprechend den Erwartungen geändert hat.

[HttpUnit](#) [9] kann zum Testen einer Webschnittstelle verwendet werden. Es simuliert das Browserverhalten und ermöglicht das Untersuchen der zurückgelieferten Webseiten. Prinzipiell funktioniert es unabhängig von JUnit, lässt sich aber leicht in JUnit-Tests integrieren. Es enthält [ServletUnit](#), welches einen Servlet-Container simuliert. Mit [ServletUnit](#) lassen sich Requests an das Servlet absetzen und die Interaktion mit dem Servlet-API untersuchen.

Einen Schritt weiter geht [Cactus](#) [10]. Das Framework ermöglicht die Ausführung von JUnit-Tests innerhalb eines Servlet-/EJB-Containers. Dadurch soll sichergestellt werden, dass die getesteten Komponenten in Interaktion mit ihrer tatsächlichen Laufzeitumgebung funktionieren.

Als zusätzliche Maßnahme zur Verbesserung der Codequalität bieten sich Code-Review-Werkzeuge an. Bekannte Vertreter sind [PMD](#) [11], [CheckStyle](#) [12] und [FindBugs](#) [13]. Diesen Tools ist gemeinsam, dass sie den Quellcode statisch analysieren und problematische Codeabschnitte auflisten. Das Spektrum der Analysen reicht von einfachen Überprüfungen von Namens- und Programmierkonventionen bis hin zur Erkennung von potenziell fehlerhaftem oder dupliziertem Code. Ein solches Tool kann sicherstellen, dass der Code gewissen Qualitätsanforderungen entspricht, noch bevor er (z.B. in Form eines Unit-Tests) zur Ausführung kommt.

## Unit-Tests und XP

Das Aufkommen agiler Prozesse wie Extreme Programming (XP) hat stark zur Verbreitung von Unit-Tests und des JUnit-Frameworks beigetragen. Unit-Tests sind neben testgetriebener Entwicklung und Refactoring ein zentraler Baustein des XP-Prozesses. In der testgetriebenen Entwicklung werden Unit-Tests vor der eigentlichen Implementierung geschrieben. Sie bilden so gewissermaßen einen ausführbaren Teil der Spezifikation, gegen den sich die Implementierung testen lässt. Jede Klasse muss ihre Unit-Tests vollständig bestehen, bevor sie etwa in ein Quellcode-Repository aufgenommen werden darf. So ist sichergestellt, dass zu jedem Zeitpunkt die Implementierung ihre Spezifikation (den Test) erfüllt. Bei XP wird vorausgesetzt, dass sich sowohl Spezifikation als auch Implementierung häufig ändern können. Der Code muss für diesen Zweck regelmäßig durch Refactorings „aufgeräumt“ werden. Auch hierfür sind funktionierende und umfassende Unit-Tests unabdingbar. Anhand der Tests kann sich der Entwickler vergewissern, dass das Refactoring einer Klasse nichts an deren Funktionalität verändert hat.

## Fazit

Unit-Tests sind sinnvoll, um die Qualität einer Software schon auf Klassenebene sicherzustellen. Mit JUnit steht ein ausgereiftes und weit verbreitetes Framework für Unit-Tests zur Verfügung, das sich zudem durch zahlreiche Erweiterungen universell einsetzen lässt. In Verbindung mit anderen Qualitätsmaßnahmen lässt sich so eine recht hohe Codequalität mit überschaubarem Aufwand sicherstellen.

*Martin Bisanz ist IT Consultant der PRODYNA AG in Frankfurt am Main und arbeitet als Software-Engineer in Java-EE-basierten Projekten.*

## Links & Literatur

1. [A. Spillner, T. Linz: Basiswissen Softwaretest, dpunkt Verlag, 2004](#)
2. <http://junit.sourceforge.net/>
3. <http://testng.org/>
4. <http://www.easymock.org/>
5. <http://www.jmock.org/>
6. <http://rmock.sourceforge.net/>
7. <https://jmockit.dev.java.net/>
8. <http://www.dbunit.org/>
9. <http://httpunit.sourceforge.net/>
10. <http://jakarta.apache.org/cactus/>
11. <http://pmd.sourceforge.net/>
12. <http://checkstyle.sourceforge.net/>
13. <http://findbugs.sourceforge.net/>