# 1 Einleitung

Die rasanten Fortschritte in der Softwareentwicklung in den letzten Jahren, wie auch im technologischen Umfeld insgesamt, sind unübersehbar. Die Zeiten, in denen man mit kleinem Aufwand auf dem Laufenden bleiben konnte, sind vorbei. Immer neue Werkzeuge, Standards und Bibliotheken erleichtern zwar die Arbeit des Softwareentwicklers. Er sieht sich jedoch mit einer Vielzahl von unterschiedlichen Anforderungen konfrontiert, ganz abgesehen davon, dass es gilt, auf dem aktuellen Stand zu bleiben. Eine Anwendung muss, besonders im Zeitalter des Internets, sicher sein. Sie sollte möglichst einfach zu bedienen sein, mit den gängigen Betriebssystemen zusammenarbeiten und eine hohe Funktionalität bereitstellen. Der Anwender erwartet mehr von einem Programm, denn auch er hat den Fortschritt registriert, der weiterhin andere Bereiche des Lebens berührt. Die höchste Erwartung an ein Produkt wie ein Computerprogramm aber ist dessen Fehlerfreiheit. Denn ein Programm, welches Fehler enthält, kann diese weder durch besondere Funktionalität noch durch einfache Bedienbarkeit kompensieren. Die Bedeutung fehlerfreier Software wird besonders durch das Beispiel der 1996 aufgrund eines Softwarefehlers abgestürzten Ariane 5 Rakete der ESA deutlich.

Fehlerfreiheit lässt sich in zwei Gruppen unterteilen: Die Korrektheit fachlicher Abläufe und die Korrektheit technischer Umsetzung. Daraus entstehen die Fragen: *Soll das Programm tun, was es tut?* und *Tut das Programm, was es tun soll?* Die erste Frage stellt sich, wenn eine Validierung durchgeführt werden soll, die zweite im Rahmen der Verifizierung eines Produkts. Während fachliche Anforderungen auf höherer Ebene – oft in Absprache mit Interessenvertretern wie Kunden des Produkts – geklärt werden müssen, kann die Umsetzung technischer Anforderungen weit gehend mit Hilfe automatisierter Softwaretests überprüft werden.

Dieses Buch widmet sich der Verifizierung von Software mit Hilfe von JUnit auf der Plattform Java, die eine objektorientierte Programmiersprache zur Verfügung stellt und seit einigen Jahren eine weite Verbreitung genießt. Das populäre Framework JUnit ist speziell für Java zu haben. Es stellt alle Grundfunktionen zum Erstellen von Tests auf Methodenebene zur Verfügung. Solche Tests werden auch Unit Tests genannt. Es ist wichtig festzustellen: Wer programmieren kann, kann auch Unit Tests erstellen, und die helfen deutlich bei der Reduzierung von Softwarefehlern, das zeigt meine Erfahrung vieler Jahre der Softwareentwicklung.

JUnit ist der wohl bekannteste Vertreter der so genannten *nUnit* Frameworks. Das *n* in *nUnit* steht für einen beliebigen Buchstaben und soll die Vielfalt der Frameworks gleicher Art für verschiedene Programmiersprachen andeuten. Auch die erst vor kurzem erschienene neue Version, JUnit 4, wird in diesem Buch angemessen gewürdigt.

## 1.1 Unit Tests und weitere Spielarten

Spricht man von JUnit, kommt unweigerlich der Begriff der Unit Tests ins Bewusstsein. Denn JUnit, so wird es häufig propagiert, sei ein Framework zur Durchführung solcher Tests. Was aber sind Unit Tests? Eine tiefer gehende Diskussion soll auf das *Kapitel 2* verschoben werden.

---

**Definition von Unit Tests**

Die Meinungen über die Definition von Unit Tests gehen stark auseinander: In seinem Weblog-Eintrag[1] mit dem Titel *A Set of Unit Testing Rules* legt der Autor Michael Feathers, fest, dass ein Test kein Unit Test ist, wenn er beispielsweise mit einer Datenbank oder über ein Netzwerk kommuniziert oder das Dateisystem anspricht. Diese Meinung wird von anderen kritisiert[2].

Ich meine, es ist letztendlich nicht entscheidend, was die akademische Definition von Unit Tests ist. Vielmehr kommt es darauf an, welche Tests sich mit Hilfe eines Testframeworks umsetzen lassen. Kann ein Test mit Mitteln, die ein solches Framework bereitstellt, implementiert werden, ordnen wir ihn der Einfachheit und Praktikabilität halber den Unit Tests zu. Gleiches gilt für Tests, die durch Hinzuziehen von JUnit-Erweiterungen, seien es Frameworks oder Bibliotheken, erstellt werden können.

---

Neben Unit Tests werden häufig die vier weiteren Testarten Funktionstest, Integrationstest, Anwendertest sowie Last- oder Performanztest genannt und unterschieden. Unit Tests, Funktions- sowie Lasttests grenzen sich von den anderen zwei Testarten zumindest dadurch ab, dass sie vom Entwickler erstellt werden. Gelegentlich werden Integrationstests auch den programmatischen Tests zugeordnet. Jedoch ist damit oft stattdessen die Durchführung von Anwender-ähnlichen Tests aus Sicht der Integration neuer oder veränderter Komponenten in

1. Siehe *http://www.artima.com/weblogs/viewpost.jsp?thread=126923*
2. Beispielsweise *http://beust.com/weblog/archives/000319.html*

eine bestehende Umgebung durch die Testabteilung gemeint. Nicht selten ist die Entwicklungsabteilung beteiligt. Funktionstests lassen sich prinzipiell ebenfalls mit JUnit abbilden. Sie gelten als grobkörniger gegenüber klassischen Unit Tests, die kleinere Programmstücke (Module) ins Visier nehmen und als Vorstufe für Funktionstests geeignet sind. Anwendertests versuchen, die Geschäftsprozesse und Abläufe zu examinieren, die ein Anwender eines Moduls typischerweise ausführt. Hierunter fällt auch das Sicherstellen der Funktionstüchtigkeit und Erwartungskonformität der Benutzeroberfläche. Insbesondere die gegenüber dem Kunden der auszuliefernden Software verantwortlichen Abteilungen legen viel Wert auf diese Art von Tests. Diese Abteilungen prüfen die Güte der auf einen ersten Blick erkennbaren Funktionen und Oberflächenelemente hin auf ihre Funktionalität und Bedienbarkeit.

Im Rahmen von durch Menschen durchgeführte Tests können Lastsituationen nur schwer erzeugt werden. Aber gerade die Performanz einer Anwendung ist kritisch für den Produktivbetrieb. Daher werden Lasttests nicht nur von anderen Tests unterschieden, sondern sie werden zudem durch zahlreiche Werkzeuge, wie Profiler, unterstützt. Auch zum Thema Lasttests enthält dieses Buch einige Hinweise.

Wie im folgenden Kapitel vertieft, gibt es eine spezielle Herangehensweise bei der Softwareentwicklung, die an die erste Stelle die Erstellung von Unit Tests setzt. Diese so genannte testgetriebene Entwicklung basiert darauf, zuerst Unit Tests zu schreiben, und zwar noch vor der eigentlichen Programmierung! Dieses radikale Prinzip lässt sich in der kommerziellen Softwareentwicklung, in der eine Vielfalt an Projekten unterschiedlicher Größe, Bürokratie und Besetzung existiert, offiziell meist nicht umsetzen. Es sei angemerkt, dass mancher Entwickler gar der Meinung ist, testgetriebene Entwicklung und Unit Tests hätten nicht viel miteinander zu tun. Ich bin anderer Meinung. Einige Rezepte dieses Buchs sind vielleicht für den, der die testgetriebene Entwicklung konsequent lebt, nicht geeignet. Die zahlreichen übrigen, hoffentlich nützlichen Informationen sollten jedoch dafür entschädigen.

## 1.2 Nutzen von Unit Tests

Der Einstieg in JUnit und Unit Tests gestaltet sich recht einfach. Hat man erst einmal ein paar Tests erstellt, ist das Prinzip klar, dann geht die Umsetzung weiterer Tests fast wie von selbst von der Hand und bereitet vielen Entwicklern schon nach kurzer Zeit viel Freude. Wie bei der Entwicklung der eigentlichen Geschäftslogik auch, kann bei der Erstellung eines Testfalles das gewünschte

Ziel auf beliebig viele Arten erreicht werden. Die meisten der möglichen Wege sind nicht optimal, sie sind teilweise sogar kontraproduktiv. Zumindest trifft dies auf die Erstellung von Geschäftslogik zu. Im Rahmen von Unit Tests ist ein Test dann hilfreich, wenn er eine Methode auf deren korrekte Funktionsweise hin testet und Fehler in der Logik aufdeckt. Im Laufe der Evolution des Programmcodes muss oft auch der zugehörige Testfall angepasst werden. Je nachdem wie übersichtlich, effizient und dokumentiert er gestaltet wurde, ist dies einmal leichter möglich, ein anderes Mal mit mehr Schwierigkeiten verbunden. Insbesondere kommt diese Problematik dann ins Spiel, wenn ein anderer Entwickler einen „fremden" Testfall anpassen muss.

Deshalb ist es wichtig, für die Erstellung von Testfällen mindestens ebenso hohe Ansprüche an deren Qualität zu stellen, als dies für Geschäftslogik in alltäglicher Weise der Fall sein sollte. Eine so hohe Qualität für Testfälle ist deshalb berechtigt, weil nur automatisierte Tests eine ausreichende Sicherheit bieten, sowohl bestehenden als auch geänderten Programmtext auf seine Richtigkeit hin zu verifizieren. Das Gegenbeispiel sind Programme, die nicht automatisiert testbar sind. Für sie müssen – streng genommen nach jeder winzigen Änderung des Codes mit anschließender Neukompilierung – umfangreiche, zeitaufwändige und oft unbefriedigende manuelle Integrationstests durchgeführt werden. In der Praxis versanden Integrationstests bereits nach einer kurzen Dauer, denn die Tester (die Entwickler selbst oder Personen der internen Qualitätssicherung und teils auch Kunden des Produktes) können sich auf Dauer angenehmere Arbeiten vorstellen als das stupide Ausführen derselben Aktionen. Bereits relativ kleine Softwaresysteme besitzen eine so hohe Komplexität, gemessen an der Menge möglicher Programmabläufe, dass ein vollständiger manueller Test auf Dauer undenkbar ist.

Um dem Entwickler eine Möglichkeit an die Hand zu geben, seine Testfälle sauber, effizient und korrekt erstellen zu können, wurde dieses Buch geschrieben. Es soll sich von anderen Büchern, die bisher zum Thema Softwaretests erschienen sind, dadurch unterscheiden, dass hier zum einen in konzentrierter und leicht auffindbarer Rezeptform Praxistipps vermittelt werden. Zum anderen wurde versucht, die Rezepte praxisorientiert, vielfältig und inspirierend zu gestalten. Referenzen auf seit langem bestehende Open Source-Projekte verweisen auf Parallelen aus der Praxis. So genannte Antipatterns, das sind Muster, die auf einen konzeptionellen Fehler oder eine Unsauberkeit bei der Implementierung hinweisen, ergänzen die Praxistipps. Sie zeigen häufig gemachte Fehler und liefern über ein korrespondierendes Rezept gleich einen Lösungsvorschlag mit. Die Einbeziehung von JUnit 4 macht das Buch zu einem aktuellen Nachschlagewerk.

## 1.3 Aufbau des Buchs

Das Buch ist in drei Teile gegliedert. Der erste Teil vermittelt Grundlagen und setzt den typischen, durch Tests unterstützen Entwicklungsprozess in Beziehung mit den Inhalten dieses Buchs.

Der zweite Teil ist der Hauptteil. Er enthält viele Rezepte und Antipatterns, die thematisch gruppiert sind. Zwischen den Rezepten existieren Querverweise, um technisch oder thematisch ähnliche Rezepte als Zusatzlektüre näher zu bringen.

Zum Ende hin schließt das Buch mit einer Nachbetrachtung. Weiterhin werden populäre Werkzeuge und Frameworks aufgelistet, meist aus dem Open Source-Bereich. Referenzprojekte sind in der Liste ebenfalls enthalten. Einige Literaturhinweise sind als Anregung für weitere Beschäftigung mit dem Thema und verwandten Gebieten gedacht. Verweise auf die Liste der Programme sind im Text durch eckige Klammern gekennzeichnet. So bedeutet [JGAP], dass unter diesem Kürzel im Literaturverzeichnis ein Eintrag zu finden ist.

Für das Verständnis der ein oder anderen Schilderung sind Kenntnisse objektorientierter Programmierung sowie von Java und JUnit erforderlich. Dieses Buch kann aus Platzgründen keine Einleitung in diese Themen geben, mit Ausnahme des nächsten Kapitels. Weitere Informationen enthält die Webseite zum Buch *www.junit-buch.de*.

## 1.4 Die Icons in diesem Buch

Steht für eine Meinung oder subjektive Anmerkung des Autors, die kontrovers diskutiert werden kann.

Fasst einen Abschnitt in Stichpunkten zusammen, um einen schnellen Überblick zu geben.

Die Rezepte sind fortlaufend nummeriert und stellen eigenständige Tipps dar oder geben Hinweise, wie Tests mit JUnit und verwandten Werkzeugen gestaltet werden können.

Antipatterns sind ebenfalls fortlaufend nummeriert. Sie schildern zu vermeidende Situationen und zeigen bessere Lösungswege.

Veranschaulicht ein Beispiel oder Negativbeispiel mit Referenz auf ein Software-Projekt und zeigt Möglichkeiten für weitergehende Recherchen.

Gibt einen Hinweis oder eine Erklärung zu einem Schlagwort.

# 2 Grundlagen

In diesem Grundlagenabschnitt werden einige häufig verwendete Begriffe aus dem Bereich der Unit Tests beschrieben. Danach folgt eine Betrachtung der *assert*-Methoden sowohl aus JUnit 3.x (bisherige JUnit-Version, die wohl die meisten verwendet haben) und JUnit 4 (erst 2006 erschienen). Anschließend werden verschiedene Zeitpunkte identifiziert, zu denen der Entwickler typischerweise Testfälle erstellen kann. Am Ende des Abschnitts wird ein möglicher Softwareentwicklungsprozess dargestellt und mit Bezug auf das Buch erläutert.

## 2.1 Grundlegende Begriffe

Dieser Abschnitt beschreibt die im Buch häufiger verwendeten Begriffe Testkörper und Testklasse sowie Unit Test. Er ergänzt die in der Einleitung angerissenen Schilderungen. Nicht fehlen dürfen auch ein paar Anmerkungen zu JUnit und zum Thema Annotationen, die universell einsetzbar sind und extensiv von JUnit 4 genutzt werden.

### 2.1.1 Testkörper und Testklasse

Die Begriffe Testkörper und Testklasse stehen zueinander in Beziehung. Die folgende Abbildung verdeutlicht den Zusammenhang.

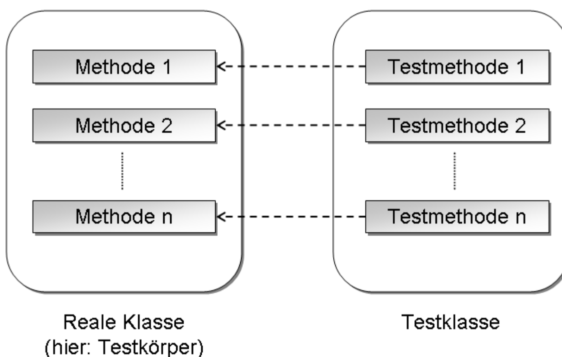


Abb. 2.1: Testkörper und Testklasse

Als Testkörper wird in diesem Buch eine reale Klasse bezeichnet. Eine reale Klasse ist eine solche, die letztendlich den zu testenden Produktivcode enthält. Der Produktivcode wird oft auch als Geschäftslogik bezeichnet. Pro Methode des Testkörpers existiert normalerweise eine Methode in einer Testklasse, welche die erstgenannte Methode testet.

Eine reale Klasse, oder auch Produktivklasse, beinhaltet mindestens eine nichtabstrakte Methode. Die Klassen, die keine oder nur abstrakte Methoden enthalten, interessieren uns im Allgemeinen nicht. Denn was soll hier getestet werden? Lediglich Eigenschaften wie Serialisierbarkeit bleibt für diese Gruppe von Klassen zum Test übrig (siehe auch *Rezept 39: Testen der Serialisierbarkeit einer Klasse*). Methoden sind in einer Klasse übrigens auch zu testen, wenn sie von einer anderen Klasse geerbt wurden. Denn es muss sichergestellt werden, dass geerbte Methoden nicht überschrieben wurden oder wenn, dass sie dann das erwartete Verhalten zeigen.

Der Test einer Methode befindet sich in einer anderen Methode, die sich wiederum in einer anderen Klasse, der so genannten Testklasse befindet. Gelegentlich ruft eine Testmethode mehr als eine Produktivmethode (Methode einer Produktivklasse) auf. Entscheidend ist aber, welche Methoden sie testet. Ein Test findet statt, indem eine Produktivklasse in einen definierten Zustand versetzt wird (etwa durch Konstruktion mit bestimmten Parametern), eine Reihe von Methoden aufgerufen wird und danach das Ergebnis dieser Aufrufe gegen einen Erwartungswert geprüft wird. Anstatt einer Produktivklasse können auch mehrere involviert sein. Der Erwartungswert ist das vom Entwickler als korrekt angesehene Ergebnis. Weicht das tatsächliche Ergebnis davon ab, schlägt der Test fehl und deutet so auf einen Programmfehler hin. Es kann natürlich auch sein, dass der Test falsch implementiert wurde und kein Programmfehler vorliegt. Das Risiko für letzteres kann durch mehrere unabhängige Tests derselben Logik reduziert werden.

### 2.1.2 Testfall

Ein Testfall ist eine Testmethode, wie sie eben beschrieben wurde. Im Rahmen von JUnit liegt eine scheinbare Doppeldeutigkeit dieses Begriffs vor. Denn jede JUnit-Testklasse erbt im Normalfall letztendlich von der JUnit-Kernklasse *junit.framework.TestCase*. Der Klassenname kann mit „Testfall" übersetzt werden. Nun stellt sich die Frage: Warum ist denn eine Klasse ein Testfall, wenn eine Testmethode innerhalb dieser Klasse eigentlich der Testfall sein soll? Die

Frage lässt sich durch Betrachten der JUnit zugrunde liegenden Philosophie auflösen.

Jeder Testfall (im Sinne von „Testmethode“) soll unabhängig von allen anderen Testfällen sein. Daher wird die Testklasse, also die Klasse, innerhalb derer sich eine Testmethode befindet, für jede Testmethode neu instantiiert (siehe auch das *Rezept 9: (De-)Initialisierung in Testklassen*, in dem weitergehende Informationen vermittelt werden).

Weil nun eine Testklasse quasi pro Testfall einmal instantiiert wird, erbt sie von *junit.framework.TestCase*. Der Begriff Testfall meint aber in seiner korrekten Bedeutung eine Testmethode, keine Testklasse!

### 2.1.3 Unit Test

Ein so genannter Unit Test behandelt, bei oberflächlicher Betrachtung, das Testen einer Einheit (engl.: *unit* = Einheit). Hier kommt unweigerlich die Frage auf, was eine Einheit in diesem Kontext sein könnte. Eine Einheit kann zunächst als ein überschaubares, in sich abgeschlossenes Stück Programmcode definiert werden. Im Rahmen eines automatisierten Tests sollte dieser Code zudem unabhängig von anderem Code – bzw. unabhängig von anderen Einheiten – getestet werden können.

Innerhalb objektorientierter Sprachen liegt es nahe, eine Klasse mit einer Einheit gleichzusetzen. Eine Klasse ist zumindest auf den ersten Blick eine in sich geschlossene Einheit geeigneter Größe, die unabhängig von anderen Klassen getestet werden kann. Außerdem widmet sich eine Testklasse im lehrbuchhaften Fall exakt einer zu testenden Klasse. Genauer hingesehen, ist eine Klasse eine logische Einheit, die in den meisten Fällen keineswegs unabhängig von anderen Klassen ist. Alleine eine Vererbungsbeziehung oder eine Referenz (Variablendeklaration, Eingabeparameter in einer Methode) auf eine andere Klasse genügt, um die Unabhängigkeit schwinden zu sehen. Im Übrigen sollte eine Klasse gar nicht vollkommen unabhängig von anderen Klassen sein, ansonsten wäre das mit der objektorientierten Programmierung eingeführte tragende Prinzip der Vererbung konterkariert. Da jede Klasse implizit von der Klasse *java.lang.Object* erbt, ist die Frage prinzipiell sowieso nur, ob eine Klasse von bereits getesteten Klassen abhängt oder von noch zu testenden Klassen. Im Falle der Java-Masterklasse *java.lang.Object* jedenfalls ist von uneingeschränkter Funktionstüchtigkeit auszugehen.

Rein akademisch betrachtet, gibt es also gemäß der obigen Aussagen keine wirklichen Unit Tests. Denn quasi jede Klasse besitzt Abhängigkeiten zu anderen Klassen. Jedoch handelt JUnit von in der Praxis umsetzbaren Testfällen, nicht von akademischen Abhandlungen. Demnach sollte der Begriff Unit Test nicht auf die Waagschale geworfen werden. Im Rahmen dieses Buches wird auch künftig der Begriff Unit Test verwendet, denn er hat sich weit verbreitet und ist mit JUnit, schon alleine wegen des Namens des Frameworks (nämlich JUnit), stark verknüpft.

Unit Tests ebnen den Weg zu Integrationstests, denn sie stellen die Funktionstüchtigkeit eines kleinen, isoliert betrachteten Programmstücks sicher, bevor im Integrationstest eine Menge voneinander abhängiger Komponenten (meist manuell) im Zusammenwirken getestet wird. Das Testen im Kleinen dient also als Grundlage für das Testen im Großen. Beide Ansätze ergänzen einander und nur die Berücksichtigung aller Perspektiven kann letztendlich eine nahezu fehlerfreie Software hervorbringen. Als Testen im Großen wird häufig der Black-Box- oder Systemtest bezeichnet. Diese Art von Test wird meist nicht von Entwicklern, sondern von mit der Qualitätssicherung betrauten Personen oder von Key Usern durchgeführt.

Aus anderem Blickwinkel betrachtet erscheint die Hauptfunktion von Unit Tests nicht das Testen von Geschäftslogik zu sein. Denn es ist nicht selten, dass Unit Tests vielmehr als Mittel zur Dokumentation sowohl auf Entwurfs- als auch auf funktionaler Ebene angesehen werden. Im Rahmen der weiter unten diskutierten testgetriebenen Entwicklung leisten sie noch mehr: Sie treiben die Mikroarchitektur der Software auf Methoden- und Klassenebene und teils auch auf Ebene der Beziehungen zwischen Klassen voran.

Um Unit Tests wie vorgesehen nutzen zu können, ist es vor allem wichtig, dass sie eine so kurze Laufzeit wie möglich haben, einfach ausführbar sind und dem Entwickler eine klare Rückmeldung im Fehlerfalle geben; die genannte Laufzeit entscheidet schließlich über die Praktikabilität dieses mächtigen Werkzeugs.

Wer einmal mit Unit Tests gearbeitet hat, wird besonders in Zeiten häufig wechselnder Anforderungen (Stichwort: Pflichtenheft) wahrscheinlich froh gewesen sein, nach einer anforderungsbedingten Programmänderung Seiteneffekte erkannt und gebannt zu haben. In einer Kultur des Kosten- und Zeitdrucks sowie steigender Abhängigkeiten zwischen verschiedenen Softwaresystemen sind termingerecht fixierte Pflichtenhefte in Softwareprojekten eher die Ausnahme als die Regel.

### 2.1.4 JUnit kompakt

Auch wenn dieses Buch Kenntnisse über JUnit voraussetzt, soll in Kürze die Verwendung des Frameworks beschrieben werden.

Zuerst ist es notwendig, die Datei *junit.jar* (über [JUnit] erhältlich) in das eigene Projekt einzubinden.

Testfälle müssen für JUnit bis vor Version 4 in Klassen, die von *junit.framework.TestCase* abgeleitet sind, abgelegt werden. Ein Testfall wird als solcher daran erkannt, dass er mit dem Präfix *test* anfängt und die Methode als public deklariert ist. Ein Testfall muss so erstellt werden, dass er auch fehlschlagen kann. Dafür gibt es Zusicherungen, die weiter unten in diesem Kapitel vorgestellt werden. Vorweg: Eine Zusicherung wird von JUnit als statische Methode angeboten. Es gibt mehrere solcher Methoden, beispielsweise die mit dem Namen *assertEquals*.

Eine exemplarische Testklasse mit einem Testfall, welcher eine Zusicherung enthält, könnte so aussehen:

```
import junit.framework.TestCase;
public class MeinTestFall extends TestCase {
  public void testIrgendwas() {
    MeineKlasse myObj = new MeineKlasse();
    assertEquals("1.07a", myObj.getVersion());
  }
}
```

Listing 2.1: Testfallerstellung mit JUnit

Der Testfall aus Listing 2.1 prüft, ob die erzeugte Instanz der Klasse *MeineKlasse* beim Aufruf deren Methode *getVersion()* den Wert *1.07a* zurückliefert. Ausnahmen (Exceptions) können ebenso geprüft werden, siehe *Rezept 32: Testen von Ausnahmen*. Wie Tests ausgeführt werden, beschreibt *Rezept 19: Eine allumfassende Testsuite namens AllTests*. Ein graphischer Testrunner zeigt einen erfolgreichen Test in grüner Farbe sowie einem grünen Balken, wenn kein Test fehlschlug. Daher wird das Ziel bei der Arbeit mit Unit Tests oft benannt mit *Make the bar green* („den Balken grün machen“).

## 2.1.5 assert- und fail-Methoden

Kernstück von JUnit sind die Methoden zum Sicherstellen einer Bedingung (*assert*) sowie zum definierten Fehlschlagenlassen eines Tests (*fail*). Die eben genannte Bedingung wird oft auch als Zusicherung (engl.: *assertion*) bezeichnet.

Da die *assert*-Methoden alle mit dem Präfix *assert* beginnen, werden sie in ihrer Gesamtheit gerne durch *assertXXX* abgekürzt. Alle *assertXXX*-Methoden liegen innerhalb der Klasse *junit.framework.Assert*. Die Klasse besitzt einen nichtöffentlichen Konstruktor. Das bedeutet, die Klasse ist nicht dazu gedacht, instantiiert zu werden. Alle *assertXXX*-Methoden sind öffentlich, statisch und besitzen keinen Rückgabewert. Sie können also direkt aufgerufen werden, ohne die Klasse *junit.framework.Assert* instantiieren zu müssen. Schlägt eine *assert*-Methode fehl, erzeugt sie eine Ausnahme.

### JUnit Version 3.x

Die während der Erstellung des Buches über lange Strecken aktuelle (produktive) Version war JUnit 3.8.1 bzw. 3.8.2. In dieser Version stellt JUnit insgesamt 32 verschiedene *assert*-Methoden zur Verfügung, wovon *assertEquals* mit 20 verschiedenen Signaturen am stärksten vertreten ist:

| Methodenname | Anzahl unterschiedlicher Signaturen |
| --- | --- |
| assertEquals | 20 |
| assertFalse | 2 |
| assertNotNull | 2 |
| assertNotSame | 2 |
| assertNull | 2 |
| assertSame | 2 |
| assertTrue | 2 |

Tabelle 2.1: *assert*-Methoden in JUnit 3.8.1 und 3.8.2

Dass *assertEquals* am häufigsten vertreten ist, lässt sich einfach begründen: Nur diese Methode akzeptiert auch primitive Typen (*int*, *boolean*, *double* usw.) als Argument, alle anderen Methoden akzeptieren sinnvollerweise nur Objekte als Eingabeparameter. Der Vergleich zweier primitiver Typen auf Identität mit *assertSame(…)* ist überflüssig (siehe auch *Rezept 28: Testen von Typen auf Gleichheit*).

Neben den *assertXXX*-Methoden existieren einige wenige *fail*-Methoden. Von diesen sind nur zwei öffentlich. Sie heißen beide *fail* und lassen einen Test definiert fehlschlagen. Eine Variante führt zum Abbruch ohne Meldung, die andere mit Meldung.

### JUnit Version 4.x

JUnit 4 wurde erst vor kurzem in einem öffentlichen Release vorgestellt. Das *Kapitel 11* enthält dazu einige tiefergehende Informationen. Die statischen *assert*-Methoden liegen hier in der Klasse *org.junit.Assert*, die ursprüngliche *Assert*-Klasse wird ebenfalls mitgeliefert. Nach aktuellem Stand hat sich die Statistik nur für *assertEquals* geändert. Es existieren für JUnit 4 nur zehn Methoden unterschiedlicher Signatur mit diesem Namen.

Wie ist diese Reduktion zu erklären? Da sich JUnit 4 auf die neue Java Version, Java 5, bezieht, können die neuen Eigenschaften dieser Plattform verwendet werden. Dazu gehört unter anderem das so genannte *Autoboxing* (siehe [Java 5]). Es ermöglicht die implizite Umwandlung primitiver Typen in Objekte. So wird – ohne Eingreifen des Entwicklers – ein primitiver Typ *int* in einen Objekttyp *Integer* umgewandelt. Somit werden lediglich für *double* und *float* eigene *assertEquals*-Methoden angeboten, so dass für *assertEquals* zehn Varianten weniger notwendig sind. Die beiden zuletzt genannten Zahlentypen werden Chip-intern nicht kontinuierlich auf dem Zahlenstrahl abgebildet. Deshalb ist es bei solchen Zahlentypen für Vergleiche wichtig, eine so genannte Epsilon-Umgebung (Delta) um den erwarteten Wert angeben zu können. Die *fail*-Methoden sind in JUnit 4 unverändert geblieben.

## 2.1.6 Annotationen

Mit Java 5 wurden Annotationen als vollwertige Sprachkonstrukte eingeführt. Sie erlauben es, Angaben über Daten, genauer: über Java-Programmkonstrukte, zu machen. Angaben über Daten, oder wie man populärer formuliert, Daten über Daten werden auch Metadaten genannt. Weil Annotationen sich eignen, Initialisierungen und Aufräumarbeiten elegant umzusetzen und weil JUnit 4 diese intensiv nutzt, soll hierzu ein kleiner Exkurs stattfinden.

Annotationen basieren auf Annotationstypen, das sind Definitionen, die den Anwendungsbereich einer Annotation sowie deren Parameter festlegen. Annotationstypen repräsentieren quasi einen Bauplan für Annotationen. Annotationen selbst sind dann konkrete Anwendungen dieser Typen. Verwendet man eine

Annotation, werden im Zuge dessen Parameterwerte angegeben, die im zugrunde liegenden Annotationstyp definiert sind. Das Verhältnis zwischen Annotationstyp und Annotation ist vergleichbar mit dem zwischen Klasse und Objekt.

Java 5 selbst stellt bereits einige vorgefertigte Annotationstypen bereit, die der Entwickler direkt benutzen kann. Hier einige der von Java 5 bereitgestellten Annotationstypen samt kurzer Erklärung:

- *java.lang.Override*: Kennzeichnet eine Methode, die eine andere Methode überschreiben soll. Verschreibt man sich im Methodennamen, kann der Compiler dies erkennen und dem Entwickler Meldung darüber erstatten.
- *java.lang.Deprecated*: Hat dieselbe Bedeutung wie das Javadoc-Äquivalent, kennzeichnet also eine veraltete, nach Möglichkeit nicht mehr zu verwendende Methode.
- *java.lang.Target*: Gibt für einen Annotationstypen den Gültigkeitsbereich an, also zu welchem Programmelement die zugehörige Annotation angebracht werden darf. Zu den möglichen Werten gehören u.a. *TYPE*, *FIELD*, *METHOD* sowie *CONSTRUCTOR*.

Betrachtet man sich den Annotationstyp *java.lang.Deprecated*, dessen Wirkungsweise ja auch mit Javadoc-Kommentaren abgebildet werden kann, ist die Frage berechtigt, warum dann Annotationstypen und Annotationen eingeführt wurden? Die Antwort ist einfach: Javadoc-Kommentare sind keine Programmkonstrukte und lassen sich nicht vom Compiler prüfen. Weiterhin gehorchen sie keiner formalen Syntax, im Gegensatz zu Annotationen. Annotationen lassen sich also wesentlich strenger prüfen als Javadoc-Kommentare.

## 2.2 Zeitpunkt der Erstellung von Testfällen

Ergänzend zu der folgenden Darstellung des Themas bietet *Rezept 3: Zeitpunkt der Testfallerstellung* weitere Anregungen.

### 2.2.1 Konventionelle Herangehensweise

Die hier als konventionell bezeichnete Herangehensweise beim Erstellen von Testfällen stellt an die erste Position die Programmierung von Geschäftslogik. Erst nachdem ein als funktionierend angesehener Code existiert, werden hierfür Testfälle erstellt. Dieses Vorgehen zieht insbesondere in der kommerziellen Softwareentwicklung nicht selten seine Motivation daraus, dass von höherer Stelle die Losung ausgegeben wird, vorrangig produktiven Code zu entwickeln. Es ist schließlich für

manche (oft fachfremde) Personen nicht nachvollziehbar, dass nur ein möglichst automatisiert getestetes, meist komplexes Softwareprodukt auf Dauer relativ fehlerfrei und effizient weiterentwickelt und gewartet werden kann. Außerdem fehlt oft die Einsicht, dass Unit Tests im Nachhinein nur schwer in Masse hinzugefügt werden können, insbesondere dann, wenn Zeitdruck herrscht.

Der erfahrene Entwickler wird allerdings häufig Geschäftsmethoden erstellen können, die er im Voraus weitgehend gedanklich abbilden kann. Er wird sich dann selbst zugestehen, den einen oder anderen Testfall erst nachträglich anzufertigen. Dieses Vorgehen mag dazu führen, dass gelegentlich vergessen wird, einen Testfall anzulegen. Das *Kapitel 7* zeigt, wie solche Ausreißer eingefangen werden können.

### 2.2.2 Testgetriebene Entwicklung

Die testgetriebene Entwicklung (engl.: *Test Driven Development* oder kurz: *TDD*) ist eine agile Methode, die vor der Erstellung von eigentlichem Programmcode die Erstellung eines Testfalls dafür fordert. Dieses erstrangige Arbeiten mit Tests hat erwiesenermaßen Einfluss auf das Design des Programms an sich. Demnach wird die testgetriebene Entwicklung häufig weniger als Testprinzip bezeichnet denn vielmehr als Entwurfsparadigma.

Anstatt des Begriffs der testgetriebenen Entwicklung wird wegen der Vorschrift, zuerst Unit Tests zu implementieren, von einigen auch der des *Test-First Programming* verwendet.

Als agil wird dieses Konzept bezeichnet, weil es dem Entwickler ein hohes Maß an Flexibilität garantieren soll, wenn er mit sich ändernden Produktanforderungen konfrontiert wird. Eine übergeordnete agile Methode, welche die testgetriebene Entwicklung beinhaltet, wird mit dem Schlagwort *Extreme Programming* (oft mit *XP* abgekürzt) bezeichnet. *Extreme Programming* fordert weiterhin eine vollständige Testabdeckung, das Beschränken auf das Wesentliche, tägliche Kurzmeetings sowie *Pair Programming* (zwei Entwickler an einem Rechner, einer entwickelt vorrangig, der andere überlegt oder diskutiert und kritisiert konstruktiv).

Wie bereits in der Einleitung erwähnt, hat sich die testgetriebene Entwicklung nicht durchgesetzt; weder in kommerziellen Projekten, noch auf breiter Basis bei Hobbyprogrammierern. Ein Grund neben der politischen Problematik in vielen Softwarehäusern mag der zunächst steigende Aufwand sein, der auf den Entwickler zukommt, bevor er eine vermeintlich produktive Zeile Quelltext vor sich sieht. Die verbreiteten menschlichen Schwächen Bequemlichkeit und mangelnde Disziplin tragen ihr Übriges dazu bei. Insbesondere das Pair-Program-

ming wird oft kritisch betrachtet. Viele Entwickler arbeiten lieber alleine als zusammen mit einem Kollegen an einem einzigen Computer.

### 2.2.3 Temporäre Testfälle

Besonders in der Phase der Fehlersuche sind oft schnelle Lösungen gefordert, um dem Zeitdruck und dem Zwang des Entwicklers entgegenzuwirken, einen möglicherweise produktionsverhindernden[1] Fehler finden zu müssen. Unter Zeitdruck kommt es gelegentlich vor, dass Entwicklungsrichtlinien, gute Vorsätze und Best Practices vergessen oder ignoriert werden. Bei der Suche nach einem Fehler ringen sich manche Entwickler dennoch durch – sofern die Art des Fehlers dies zulässt – Testfälle zum Aufspüren des Fehlers zu erstellen. Manche Testfälle werden dann leider so nachlässig erstellt, dass sie während der Implementierung in Gedanken schon wieder für die Entfernung nach erfolgreicher Fehlersuche vorgesehen werden. Der *Abschnitt Testen privater Methoden und Felder* in *Kapitel 5* kann unter diesem Gesichtspunkt diskutiert werden. Beispielsweise ist das Erstellen von Testfällen für private Methoden mit einer grundsätzlichen Problematik behaftet: Idealerweise werden private Methoden nicht getestet, wie dort aufgeführt ist. Muss dies aus den gerade beschriebenen Gründen dennoch geschehen, enthält der eben genannte Abschnitt dazu einige nützliche Hinweise. Auch das *Antipattern 6: System.out und System.err in Tests* ist dem Gesichtspunkt der Zwischenzeitlichkeit zuzuordnen.

---

**Unit Test**

- Die Definition des Begriffs Unit Test ist nicht eindeutig, sie muss zweckmäßig vorgenommen werden.
- Unit Tests können vor (*Test-First, XP*) oder nach der eigentlichen Programmlogik erstellt werden sowie dauerhaft oder temporär.
- Ein Unit Test testet im Idealfall eine Methode.
- Wichtiges Prinzip bei Unit Tests ist deren schnelle und einfache Ausführbarkeit sowie die Reproduzierbarkeit des Ergebnisses.
- Nach jeder abgeschlossenen Programmänderung sind alle Unit Tests auszuführen, um durch die Änderung evtl. auftretende Seiteneffekte zu erkennen.

---

1 Produktionsverhindernd ist ein Fehler, der ein produktiv eingesetztes System derart signifikant beeinträchtigt, dass für das Geschäft des Kunden erhebliche Teilbereiche des Systems nicht arbeitsfähig zur Verfügung stehen.

## 2.3 Der Software-Entwicklungsprozess

Der Vorgang der Softwareentwicklung ist ein sehr komplexer, der in größeren Projekten leicht die Komplexität des Autobaus übertreffen kann. Schließlich gilt es, aus einer unendlichen Menge möglicher Implementierungen gerade die eine zu finden, die allen Anforderungen so gut als möglich gerecht wird. Zu diesen Anforderungen zählen insbesondere die in gewisser Weise statischen Eigenschaften

- Fehlerfreiheit,
- Bedienbarkeit (Oberfläche, Programmfluss),
- Benutzbarkeit (Ablaufgeschwindigkeit, Programmfluss) sowie
- Wartbarkeit.

Diese Eigenschaften können einmalig festgestellt werden. Wird die Software ohne Änderungen und immer auf demselben System und mit denselben Anbindungen an andere Systeme betrieben, ändern sich die obigen Eigenschaften nicht. Ändert sich jedoch das Einsatzgebiet der Software oder werden Änderungen im Quelltext vorgenommen, kommen weitere dynamische Eigenschaften zum Tragen, wie

- Erweiterbarkeit (bei Änderungen),
- Stabilität (nach Änderungen),
- Portierbarkeit auf andere Plattformen und
- Konfigurierbarkeit (hinsichtlich anderer Umgebungen).

Einige dieser komplexen Anforderungen lassen sich durch automatisierte Tests gewährleisten, das trifft insbesondere auf Fehlerfreiheit und Stabilität zu. Nicht nur unter Zugriff auf das Konzept des *Extreme Programming* helfen Testfälle aber auch dabei, die Qualität der Softwarearchitektur zu verbessern. Das wirkt sich direkt auf die Erweiterbarkeit des Programms aus. Abbildung 2.2 gibt einen Überblick über einen Software-Entwicklungsprozess, wie er typisch ist bei Verwendung automatisierter Tests.

Die Illustration hebt deutlich die den Gesamtprozess bestimmenden Teilprozesse hervor. Der oberste rechteckige Block repräsentiert die Hauptarbeit des Programmierens, bezeichnet durch die Prozesse *Kodierung* und *Testerstellung*. Hierin enthalten ist natürlich einerseits die Implementierung von Produktivcode und andererseits auch die Erstellung von Unit Tests, GUI Tests, Performanztests usw.

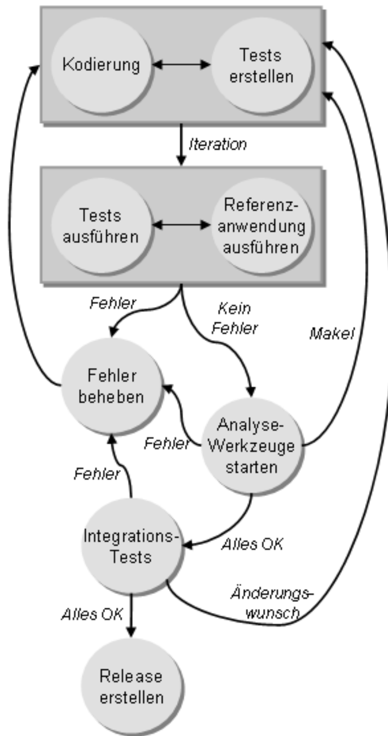


Abb. 2.2: Möglicher Software-Entwicklungsprozess

Nach einem Programmierzyklus gilt es, die programmierte Logik zu verifizieren, also zu testen. Hierfür werden alle relevanten Unit Tests einfach (!) ausgeführt. Es ist wirklich einfach, ganz im Gegensatz zu manuell durchzuführenden Tests, die im Endeffekt ein ganz enormes Vielfaches an Zeit und Nerven erfordern! Das kann für viele größere Softwareprojekte festgestellt werden.

Neben der Verifikation durch Unit Tests bietet es sich an, eine Referenzanwendung (oder auch mehrere) zu erstellen, die ohne Eingriff des Entwicklers ablaufen kann. Sie verwendet die hauptsächlichen Funktionen des Softwarepakets und gibt ein Ergebnis aus (etwa auf der Konsole oder in eine Datei oder als Bild, je nach Anwendungsfall). Die Referenzanwendung kann quasi als kleiner *Proof Of Concept* (Nachweis der Funktionstüchtigkeit) angesehen werden. Die Laufzeit dieser

Beispielanwendung sollte so kurz wie möglich sein, da eine häufige Ausführung nur von Vorteil ist. Der geschulte Entwickler sieht anhand des entsprechend ausgegebenen Ergebnisses, ob die Anwendung wie gewünscht gearbeitet hat. Mit einer solchen Beispielanwendung lassen sich malade Effekte identifizieren, die ein Unit Test nicht oder nur schwer zu Tage fördern könnte. Ein Beispiel aus der Praxis: Im Rahmen des Projektes [JGAP] wurde eine solche Testanwendung erstellt und regelmäßig verwendet, sowohl nach täglichen Entwicklungen als auch vor Releases. Die Anwendung hatte die Aufgabe, in mehreren Durchläufen (Iterationen) eine Berechnung durchzuführen. Einmal stellte sich heraus, dass ungewöhnlich viele Iterationen notwendig waren, um in die Nähe des erwarteten Ergebnisses zu kommen. Daraufhin konnte ein Programmfehler gefunden und ein Unit Test dafür erstellt werden, der den Fehler „bloßstellt".

Das so genannte Bloßstellen von Fehlern ist eine wesentliche Aufgabe im Rahmen der Testfallerstellung. Im Englischen wird das mit *expose a bug* bezeichnet. Die Fehlerbehebung schließt sinnvollerweise direkt an die Kodierung und Testfallerstellung an. Äussert sich ein Fehler, wird hierfür also zunächst ein Testfall (es können aber auch mehrere sein) erstellt, der fehlschlägt. Nun wird die fehlerhafte Stelle modifiziert und der neu erstellte Test muss nun durchlaufen. Weiterhin sollten natürlich alle anderen Tests grün sein! Denn Seiteneffekte sollen durch eine Programmänderung nicht entstehen (zu vermeiden ist ja „ein Fehler behoben, ein neuer entsteht"). Deshalb führt von der Fehlerbehebung nur ein Weg zurück, und zwar ganz an den Anfang!

Haben Unit Tests und Beispielanwendung keine Fehler erkennen lassen, ist es vor der Erstellung eines Releases an der Zeit, den Programmtext sowie das ausführbare Programm einer speziellen Analyse zu unterziehen. Auf dem Markt existieren einige (oft kostenlose) Werkzeuge, die solche Analysen durchführen. Diese schließen folgende Aspekte ein:

- Programmkonventionen,
- Ablaufanalyse,
- Lastverhalten
- und viele mehr.

Dieses Buch möchte Sie, lieber Leser, bei der Entwicklung Ihrer Software unterstützen. Sie finden in dieser Lektüre zahlreiche Anknüpfpunkte zu den skizzierten Prozessen, die in der nachstehenden Tabelle exemplarisch angegeben sind, um eine Idee zu vermitteln.

| Aspekt | Stelle im Buch |
|---|---|
| Programmanalyse | Kapitel 6 |
| Erstellung von Unit Tests | Kapitel 3, Kapitel 5 |
| Testen von Entwurfsmustern | Kapitel 9 |
| Ausführen von Unit Tests | Rezept 14: Pro Package eine umfassende Suite |
| Release-Erstellung | Rezept 14: Pro Package eine umfassende Suite;<br>Rezept 20: Ablage von Tests, Testdaten und Geschäftslogik;<br>Rezept 21: Tests für lizenzierte Bibliotheken auslagern und weitere |
| Fehlerbehebung | Kurzer Anriss in diesem Kapitel |
| Integrationstests | Kapitel 1, dieses Kapitel sowie Kapitel 10 |

Tabelle 2.2: Unterstützung beim Software-Entwicklungsprozess

Weitere nützliche Kapitel, Rezepte und Antipatterns sind über die thematische Gliederung des Buches zugänglich. Vielleicht stoßen Sie beim Blättern durch das Buch auf den ein oder anderen nützlichen Tipp, der Sie schon jetzt anregt oder Ihnen später wieder ins Gedächtnis kommt.

**Der Software-Entwicklungsprozess**

- Automatisierte Softwaretests sollten im Software-Entwicklungsprozess jedes Projektes eine wichtige Rolle spielen.
- Nach jeder Programmänderung sind bei Bedarf Testfälle zu modifizieren oder weitere anzufertigen.
- Es kann nie zu viele Testfälle geben. Mir ist jedenfalls kein Projekt bekannt, bei dem dies zuträfe.
- Beispielanwendungen ergänzen Unit Tests hervorragend.
- Die Verwendung von Analysewerkzeugen ist ein wertvolles Hilfsmittel zur Qualitätssicherung.
- Ein Integrationstest macht erst Sinn, nachdem Fehler und Makel soweit wie möglich ausgeschlossen wurden.