Bewährte JUnit-Techniken aus der Praxis

Software erfolgreich testen mit JUnit

von klaus meffert

Bei der täglichen Erstellung von Testfällen wird der Entwickler oft vor Aufgaben gestellt, für deren Lösung Grundlageninformationen nicht ausreichend sind. Die Qualität von Testcode ist ohne weitergehende Unterstützung vom Geschick des Einzelnen abhängig. Dieser Artikel stellt zur Inspiration einige in der Praxis bewährte Techniken für die Erstellung von Testfällen vor und setzt die Artikelserie zum Thema JUnit fort.

Kasten

Der Text ist ein leicht modifizierter Auszug aus dem im Mai im Software & Support Verlag erschienenen Buch „JUnit Profi-Tipps – Software erfolgreich testen“.

Ende Kasten

Einleitung

JUnit ist die populärste Möglichkeit für Entwickler, die Qualität von Java-Quelltext durch Testfälle sicherzustellen. Im ersten Teil dieser Artikelserie wurde das neu erschienene Release JUnit 4 vorgestellt. Der vorige Teil widmete sich dem automatisierten Generieren von Unit Tests. In diesem Artikel werden Techniken vorgestellt, um Testfälle für bestimmte Aspekte realisieren zu können. Anhand des Abschnitts, der das Testen von Entwurfsmustern zum Gegenstand hat, wird deutlich, wie eng das Erstellen von Testfällen mit der eigentlichen Kodierung zusammenhängt. Letztendlich ist es entscheidend, einen Programmtext und eine dadurch ausgedrückte Softwarearchitektur zu entwerfen, der/die ohne unsaubere Eingriffe mit Hilfe von Testfällen verifiziert werden kann.

Kasten

JGAP ist ein Framework, das Genetische Algorithmen bereitstellt, um Probleme mit großem Suchraum lösen zu können. JGAP wurde konsequent unter Zuhilfenahme von JUnit entwickelt (mittlerweile existieren über 1000 JUnit-Testfälle). Beispielwendungen (siehe Artikel) sichern die allgemeine Funktionsfähigkeit ab und veranschaulichen die Funktionsweise des Programms. Weitere Informationen enthält die JGAP-Homepage [2].

Ende Kasten

Der Entwicklungsprozess

Zunächst ein Blick auf den Entwicklungsprozess, angefangen beim Kodieren (Programmieren) bis hin zum Erstellen des Releases („Deploy“). Die Abbildung 1 zeigt die wesentlichen Prozesse, die bei der Berücksichtigung von Unit Tests eine Rolle spielen. Integrations- und weitere Tests sind in der Abbildung (der Übersichtlichkeit halber und wegen der Ausrichtung dieses Artikels) nicht dargestellt. Im oberen Teil der Abbildung ist ein Block, bestehend aus Kodierung und Testerstellung, zu sehen. Je nach dem, ob der *Test First* oder der traditionelle Ansatz (*Code First*) gefahren wird, ist die eine oder die andere Tätigkeit primär. Danach folgt die Verifikation des Codes. Angedeutet sind zwei Möglichkeiten, eine davon ist natürlich der Rückgriff auf die erstellten Testfälle. Die andere ist die Auswertung des Programmverhaltens durch Starten einer Beispielanwendung, die zumindest von den Hauptfunktionen der Software unter Test Gebrauch macht. Jenes Verfahren wurde im Projekt JGAP [2] als Ergänzung zur ersten Möglichkeit mit Erfolg angewendet. Schließlich macht das auch Sinn, denn es wird selten eine alle Eventualitäten berücksichtigende Menge an Testfällen geben. Je nach dem, ob Fehler zutage getreten sind oder nicht, entscheidet sich das weitere Vorgehen. Entweder steht

nun die Fehlerbehebung an oder eine weitergehende Analyse. Nach einer eventuellen Fehleranalyse ist unbedingt mindestens ein Testfall zu erstellen, der den Fehler offenlegt („Expose a bug“). Im Erfolgsfall kann stattdessen mit Hilfe von Analysewerkzeugen (die im nächsten Artikel der Serie diskutiert werden) das Qualitätsbild ergänzt werden. War auch dieser Schritt erfolgreich, steht dem Release nichts mehr im Wege (wie gesagt, Integrationstests usw. außen vor gelassen). Haben sich Fehler, Mängel oder Verstöße gegen Programmierrichtlinien herausgestellt, ist der Rücksprung ganz an den Anfang angesagt. Ein neuer Zyklus beginnt.

Viele Praktiken zum Schreiben von Testfällen sind in der Literatur generell bekannt. Die folgenden Anmerkungen sollen das Bild abrunden und einen kompakten Überblick über die Vielschichtigkeit einiger Grundproblematiken vermitteln.

Testklassen

Prinzipiell gibt es vier Möglichkeiten der Ablage für Klassen, die Unit-Tests enthalten. Diese ergeben sich aus der Wahl, ob diese Klassen im selben Verzeichnis und/oder im selben Package wie die den Produktivcode enthaltenden Klassen liegen sollen. Ich empfehle aus der Erfahrung heraus, Testklassen im selben Package, aber in einem anderen Verzeichnis als den Produktivcode abzulegen. Das hat den Vorteil, dass man sehr einfach selektieren kann, ob in einem Release auch die Tests mitgegeben werden. Es macht oft keinen Sinn, Testcode ausserhalb der Entwicklungsabteilung mitzugeben. Befänden sich die Tests in einem anderen Package, wären sie während der Entwicklung nicht so leicht zu finden oder der zugehörigen Produktivklasse zuzuordnen.

Es empfiehlt sich in jedem Projekt, eine eigene abstrakte Oberklasse anzulegen, die *junit.framework.TestCase* erweitert. Diese abstrakte Klasse besitzt dann häufig benötigte Routinen (wie etwa den Zugriff auf private Variablen oder Methoden, vgl. die JUnit-Addons [1]). Sie ist die Basisklasse für alle eigenen Testfälle. Eine konkrete Implementierung findet sich exemplarisch im Open-Source Projekt JGAP [2] mit der abstrakten Testklasse *JGAPTestCase*. Für Gruppen von Tests können dann weitere abstrakte Oberklassen vorgesehen werden, die von der eben genannten Testklasse erben. Diese Oberklassen erlauben die Definition spezialisierter *setUp()*- und *tearDown()*-Methoden. Weiterhin können dort *protected*-Variablen oder Setter/Getter eingeführt werden, die von den konkreten Testklassen einfach verwendbar sind, etwa die Verbindung zu einer Datenbank. Tests werden logisch entweder dadurch gruppiert, dass sie in einem gemeinsamen Package liegen. Oder es werden Tests als einer Gruppe zugehörig angesehen, die eine vergleichbare Charakteristik haben. Beispiele für letzteres: Testen der Datenbankschicht, Performanztests, Parsen von Eingabedaten mit Hilfe derselben Utility-Klasse etc.

Testergebnisse werden bekanntlich mit Hilfe von Zusicherungen durch Verwendung von *assertXXX*-Methoden (etwa *assertEquals* oder *assertTrue*) überprüft. Es ist zu vermeiden, Ausgaben auf die Systemkonsole, etwa mittels *System.out.print(...)* vorzunehmen. Nur in ganz wenigen Fällen ist dies vorteilhaft! Stattdessen bieten JUnit-Methoden *assertXXX* sowie *fail* ausreichende Möglichkeiten, einen Meldungstext bei Fehlschlagen eines Testfalls auszugeben.

Um die zu einer produktiven Methode zugehörigen Testmethoden leicht finden zu können, sollte die Testmethode den Namen der zu testenden Produktivmethode enthalten. Werden mehrere Methoden gleichzeitig getestet, kann der Name der im Mittelpunkt stehenden Methoden genommen werden. Das Präfix von Testmethoden ist bei JUnit (noch) vorgegeben und lautet *test*. Ab der nächsten JUnit-Version erlauben Annotationen eine freie Namenswahl. Die erste Methode, die Methode *calcPrice* testet, könnte dann *testCalcPrice_0* genannt werden. Gibt es weitere Testmethoden für *calcPrice*, heissen diese *testCalcPrice_1, testCalcPrice_2* usw. Sind zwei Testmethoden für dieselbe Produktivmethode sehr ähnlich, kann für sie auch

derselbe Index (also etwa *_0*) verwendet werden und ein Unterindex hinzugenommen werden. Zwei ähnliche Testmethoden hießen dann beispielsweise *testCalcPrice_2_0* und *testCalcPrice_2_1*.

Pro Testmethode sollten nicht zu viele Zusicherungen ausgeführt werden, im theoretischen Idealfall genau eine. Denn schlägt eine Zusicherung fehl, werden die folgenden Zusicherungen in derselben Methode nicht ausgewertet werden können und der Entwickler für diese keine Rückmeldung erhält. Ist die Konstruktion von Testdaten und -objekten allerdings komplexer oder der Test relativ trivial, sind mehrere Zusicherungen pro Testmethode durchaus gerechtfertigt.

Testsuiten und Namenskonventionen

Wie gehabt, werden Testmethoden auf Testklasse (Testfälle) und Testklassen auf Testsuiten aufgeteilt. Pro Package sollte eine Testsuite existieren. Übergeordnete Packages besitzen weiterhin je eine Testsuite, die alle Testsuiten untergeordneter Packages inkludiert. Auf oberster Ebene gibt es dann eine Testsuite namens *AllTests*. Sie auszuführen, bedeutet alle Testfälle aller Pakete auszuführen, denn sie enthält eine Referenz auf alle untergeordneten Testsuiten. Die Namenskonvention, eine Testsuite mit dem Suffix *Tests* enden zu lassen, hat sich nicht nur eingebürgert. Sie erlaubt es auch, bei der Prüfung mit Hilfe von Tools wie Checkstyle [4] oder PMD [5] zu unterscheiden zwischen Testklassen und Produktivklassen. Testklassen wären dann solche, deren Namen mit *Test* (Testfall-Klasse) oder *Tests* (Testsuite-Klasse) endet. Häufig wird es weitere Klassen geben, die zur Unterstützung der Testdurchführung dienen. Entweder sind dies Utility-Klassen oder Implementierungen abstrakter Basisklassen (die sich aufgrund mangelnder Instanziierbarkeit nicht direkt testen lassen) oder Standardimplementierungen von Konfigurationen, Zufallsgeneratoren (siehe weiter unten) o.ä. All diese Klassen sollten entweder ebenfalls einer Namenskonvention folgen, die sein unterscheidbar macht von Klassen mit Produktivcode. Oder sie sind in einem gesonderten Package oder Verzeichnis anzulagern. So können Prüftools entsprechend einfach konfiguriert werden, um nicht zu berücksichtigende Klassen zu ignorieren. Weitere Details hierzu folgen im nächsten Teil der Artikelserie.

Testen auf Serialisierbarkeit

Eine Klasse kann durch Implementierung der Markerschnittstelle *java.io.Serializable* serialisierbar gemacht werden. Von einer Klasse referenzierte Typen können die Serialisierbarkeit jedoch gefährden. Ein Unit-Test schafft Gewissheit. Man kann zwei Tests unterschiedlicher Granularität durchführen. Erstens lässt sich sehr einfach prüfen, ob eine bestimmte Klasse die Schnittstelle *java.io.Serializable* implementiert, beispielsweise so:

```
java.io.Serializable pop = new Population();
```

Dies ist die erste Voraussetzung für die Serialisierbarkeit einer Klasse (von Externalisierbarkeit abgesehen). Zweitens überprüft man die Korrektheit der Operation, indem man versucht, ein Objekt der entsprechenden Klasse tatsächlich zu serialisieren. Deserialisiert man es anschließend wieder, lassen sich tatsächliche mit erwarteten Werten von Variablen überprüfen. Bei diesem Prozess auftretende Ausnahmen führen zum gewünschten Fehlschlag des Tests (vgl. *PopulationTest* in [2]).

Der endgültige Test, ob ein serialisiertes und danach wieder deserialisiertes Objekt dem ursprünglichen entspricht, ist durch ein `assertEquals(altesObjekt, de_serialisiertesObjekt)` möglich. Allerdings ist der damit durchgeführte Vergleich abhängig von einer entsprechend implementierten `equals(Object)`-Methode im zu vergleichenden Objekt!

Testen erwarteter Ausnahmen

Selbst in der ausserordentlich mitgliedsstarken JUnit-Mailingliste [3] werden zu scheinbar trivialen Fragen gelegentlich keine befriedigenden Antworten gefunden. Ein Beispiel ist das durchdachte Testen erwarteter Ausnahmen (Exceptions). Listing 1 zeigt, wie geprüft werden kann, ob die Ausnahme *IllegalArgumentException*

wie erwartete ausgelöst wird. Hierzu sind mehrere Anmerkungen zu machen: Die erwartete Auslösung der Ausnahme wird getestet, indem der Methodenaufruf *manager.save()*, der die Ausnahme schmeissen soll, in einen *try-catch* Block gekapselt wird. Es wird nur die spezielle Ausnahme *IllegalArgumentException* abgefangen. Alle anderen Ausnahmen werden durch die Klausel *throws Exception* der Testmethode selbst behandelt und führen zum gewünschten Fehlschlagen des Tests. Wird gar keine Ausnahme ausgelöst, führt die Ausführung der JUnit-Methode *fail()* zum Scheitern des Tests. Nun werden aber oft Werkzeuge eingesetzt, die prüfen, ob hinterlegte Regeln beim Programmieren eingehalten wurden. Daher steht innerhalb des Catch-Blocks ein einzelnes Semikolon. Es repräsentiert die leere Anweisung (aus Assemblerzeiten etwa als NOP – was für *NO Operation* steht – bekannt). Fehlte diese leere Anweisung, führt dies bei bekannten Syntaxprüfern wie CheckStyle [4] oder PMD [5] zu einer ungewollten und überflüssigen Warnung. Der knappe Kommentar hinter dem Semikolon erfüllt Dokumentationszwecke und sollte nicht fehlen.

Zufallsgeneratoren

Eine Geschäftslogik, die sich eines Zufallsgenerators bedient, kann mit Hilfe eines sogenannten Mock-Objekts getestet werden. Ein Mock-Objekt ist vergleichbar mit einem Stellvertreter, der klar definierte Rückgabewerte liefert. Es unterscheidet sich also in der Funktionalität des stellzuvertretenden Objekts, besitzt aber dieselbe Schnittstelle. Führt man nun einen Mock-Zufallsgenerator ein, kann man diesen derart gestalten, dass die von ihm zurückgegebenen Werte bei der Konstruktion vorbestimmt werden können. Nur durch Ausschalten des „Zufalls“ lassen sich von einem Zufallgenerator abhängige Logiken testen. Ein Beispiel hierfür ist in der Klasse *RandomGeneratorForTest* [2] zu finden, der in zahlreichen Tests verwendet wird.

Gleichheit von Objekten und primitiven Typen

Zum Vergleich zweier Objekte werden die dedizierten Methoden *equals(Object)* und *compareTo(Object)* verwendet. Wie oben bereit genannt, spielen sie auch bei der Verifikation der Serialisierbarkeit eines Objekts eine entscheidende Rolle. In den allermeisten Fällen sollte der Vergleich zweier Objekte o1 und o2 symmetrisch sein, also wenn `o1.equals(o2)` wahr ist, dann auch `o2.equals(o1)`. Existiert eine eigene Oberklasse für Testfälle (s.o.), dann kann dort eine Hilfsfunktion integriert werden, die diese Symmetrie automatisch prüft. Vergleiche hierzu auch die Klasse *ComparabilityTestCase* der JUnit-Addons [1].

Der Unterschied zwischen *assertEquals* und *assertSame* ist der folgende: *assertSame* ist nur für Objekte sinnvoll, nicht für primitive Typen. Denn damit wird geprüft, ob zwei Objektinstanzen dasselbe Objekt repräsentieren (als schriebe man `obj1 == obj2`). *assertEquals* gibt es in mehreren Ausprägungen, sowohl für Objekte allgemein als auch für Strings als auch für primitive Typen. Für Objekte ruft *assertEquals* deren *equals*-Methode auf, weshalb der in diesem Artikel erwähnte Test auf Serialisierbarkeit auf eine korrekte Implementierung von *equals* angewiesen ist. Für primitive Typen ist die Sache insbesondere bei Fließkommazahlen komplizierter. Da deren interne Darstellung diskret ist und nicht alle Werte des (kontinuierlichen) Zahlenstrahls abbilden kann, kann eine sogenannte Delta-Umgebung (in der Mathematik oft als Epsilon-Umgebung bezeichnet) angegeben werden. Liegen dann zwei Zahlenwerte sehr dicht beieinander, nämlich so dicht, dass der eine nicht weiter als der Deltawert vom anderen entfernt ist, dann gelten sie als gleich.

Eine fehlgeschlagene Zusicherung mittels *assertXXX(...)* führt nicht nur zu einem Abbruch des entsprechenden Testfalls, sondern auch zu einer Fehlermeldung. Je aussagekräftiger diese Fehlermeldung ist, desto einfacher ist der Fehler zu finden. Deshalb sollte immer der *assert*-Befehl verwendet werden, der seiner Bestimmung nach am geeignetsten ist, auch wenn die Umsetzung mit einem anderen *assert*-Befehl ebenfalls möglich wäre. Ein einfaches Beispiel: Eine Methode *myObj.hasChanged()* liefert mit einem booleschen Wert (also *true* oder *false*) zurück, ob sich das Objekt *myObj* geändert hat. Bei erwartetem Rückgabewert *true* ist die folgende Schreibweise möglich: `assertEquals(true, myObj.hasChanged())`. Ebenso möglich ist:

`assertTrue(myObj.hasChanged())`. Kürzer ist die zweite Schreibweise. Die erste Variante gibt jedenfalls den besseren Meldungstext aus (sinngemäß: „erwartet: <true>, erhalten: <false>“) im Gegensatz zur zweiten (kein Meldungstext). Prüftools (wie Checkstyle [4]) mäkeln allerdings die erste Variante an. Da bei einem fehlgeschlagenen Test aber der Quelltext der Testmethode sowieso inspiziert werden muss, offenbart das *assertTrue* seine Bedeutung von alleine, ohne weitere Meldung, und ist daher vorzuziehen.

Testen von Entwurfsmustern

Ein Entwurfsmuster (siehe [6] für weitere Beschreibungen und Beispiele) ist eine bewährte Lösungsvorschrift für ein häufig auftretendes Entwurfsproblem. Es handelt sich um eine Lösungsvorschrift und nicht um eine konkrete Lösung, weil mit Entwurfsmustern (oft wird auch der kürzere Begriff *Muster* verwendet) eine Klasse von Problemen gelöst werden kann. Entwurfsmuster dokumentieren weiterhin die Entwurfsabsicht des Entwicklers und helfen so beim Verständnis eines Programms. Ein Entwurfsmuster stellt demnach keine vollends ausprogrammierte Lösung dar. Vielmehr handelt es sich um eine Menge von unveränderlichen und veränderlichen (kontextabhängigen) Bestandteilen. Unveränderliche Bestandteile müssen an das jeweilige Problem nicht angepasst werden. Beispiele für solche Teile sind insbesondere abstrakte Klassen und Schnittstellen, aber auch einzelne Methoden oder Anweisungen. Veränderliche Bestandteile müssen an das konkrete Problem angepasst werden. Der Kontext ist durch das Problem definiert. Das Testen solcher veränderlicher Teile kann einem Grundprinzip folgen, denn die Absicht, die hinter einem bestimmten Musterteil steckt, sollte immer gleich sein.

Das Entwurfsmuster *Singleton* ist der populärste Vertreter der Erzeugungsmuster. Es wird immer dann eingesetzt, wenn von einem Objekt genau eine Instanz existieren soll. Eine Abwandlung ist das *Fewton*, das, wörtlich übersetzt, einige wenige Instanzen zulässt. Ein *Singleton*-Test auf Anwendungsebene gestaltet sich recht einfach, wie in Listing 2 gezeigt. Tools wie PMD [5] prüfen im Ansatz, ob für eine Klasse das *Singleton* angebracht wäre. PMD etwa schlägt vor, aus einer (konkreten) Klasse ein *Singleton* zu machen, wenn diese nur statische Methoden enthält. Der Test der Thread-sicheren Variante von *Singleton* ist wesentlich komplizierter und wird in [7] weitergehend behandelt. Das Muster *Factory Method* kann auf ähnlich einfache Weise getestet werden, auch hierzu wird auf [7] verwiesen.

Iterator ist ein häufiges verwendetes Muster zum sequentiellen Durchlaufen aller Elemente einer Liste. Das Muster *Iterator* abstrahiert das Traversieren über die Listenelemente, indem es die beiden Methoden *hasNext()* und *next()* einführt. Die erste Methode meldet *true* zurück, wenn das letzte Element noch nicht erreicht wurde. In diesem Fall ermittelt die zweite Methode das nächste Element, welches als *java.lang.Object* zurückgegeben wird und deshalb bei Bedarf mit einem *Type Cast* versehen werden muss. Die korrekte Funktionsweise einer *Iterator*-Implementierung kann einfach geprüft werden, indem das Ergebnis von *hasNext()* und *next()* mit einer parallel durchlaufenen Schleife über die zugrunde liegende Liste verglichen wird. Um das Beispiel zu verkürzen, wird in Listing 3 der Test für die in der Klasse *java.util.Vector* integrierte *Iterator*-Implementierung durchgeführt. In diesem Listing werden zwei Tests durchgeführt. Erstens wird geprüft, ob die Listenelemente in der erwarteten Reihenfolge iteriert werden und zweitens, ob alle Elemente berücksichtigt werden. Im zweiten Fall kann es vorkommen, dass entweder zu wenige oder zu viele Elemente traversiert werden sollen. Für jede der beiden Möglichkeiten wird im Fehlerfall eine unterschiedliche Fehlermeldung ausgegeben.

Fazit

Auch die Programmierung von Testfällen will gelernt sein. Entartete Testlogik ist zwar besser als gar keine, verzögert und behindert ein Projekt auf Dauer aber mehr als es ihm nützt. Insbesondere in Projekten mit mehreren Entwicklern ist es wichtig, dass der verantwortliche Entwickler ein Auge auf die anderen Entwickler hat, um unsauber geschriebene Testfälle zu vermeiden. Insbesondere diese führen zu einer trügerischen

Sicherheit, wenn eine Aussage über die Güte eines Softwarepaketes getroffen werden soll. Der folgende Teil dieser Artikelserie beschäftigt sich daher mit der werkzeuggestützten Ermittlung der Güte von Testfällen. Es wird gezeigt, wie der Einsatz von Werkzeugen zum Aufspüren von unsauberem oder potentiell verbesserungswürdigem Testcode führt. Der angenehme Nebeneffekt einer automatisierten Prüfung ist die Möglichkeit, diese jederzeit und ohne Aufwand durchführbare Ausführung, die wesentlich mehr Spaß bringt als die manuelle Nachkontrolle. Daraus wird deutlich, insbesondere für größere Projekte und solche mit mehreren Entwicklern: Es gilt bei der Implementierung von Logik jeder Art nicht nur, auf eine saubere und verständliche Implementierung zu achten, sondern möglichst auch noch so zu programmieren, dass für richtig befundene und durch Prüftools verifizierte Richtlinien nicht verletzt werden!

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-Addons: http://junit-addons.sourceforge.net/

[2] JGAP: http://jgap.sourceforge.net/

[3] JUnit-Mailingliste: http://groups.yahoo.com/group/junit/

[4] CheckStyle: http://checkstyle.sourceforge.net/

[5] PMD: http://pmd.sourceforge.net/

[6] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software. Addison-Wesley.

[7] Klaus Meffert: JUnit Profi-Tipps. Bewährte Techniken und Antipatterns, Software & Support Verlag (erscheint im Mai 2006).

Listing 1

Testen von Exceptions

```
public void testSave_0()
   throws Exception {
  try {
    manager.save("Datensatz 1");
    fail();
  catch (IllegalArgumentException iex) {
   ; // erwartetes Verhalten
  }
}
```

Ende Listing

Listing 2

Testen des Musters Singleton

```
public void testSingleton() {
  MyClass myObj1 = MyClass.getInstance();
  MyClass myObj2 = MyClass.getInstance();
  assertSame(myObj1, myObj2);
}
```

Ende Listing

Listing 3

Testen des Musters Iterator

```
import java.util.*;
...
public void testIterator() {
  List l = new Vector();
  l.add("Eintrag 1");
  l.add("Eintrag 2");
  l.add("Eintrag 3");
  Iterator it = l.iterator();
```

```
  int i = 0;
  while (it.hasNext()) {
    if (i >= l.size())
      fail("hasNext fehlerhaft: kein Abbruch");
    String ist = (String)it.next();
    String soll = (String)l.get(i);
    i++;
    assertEquals(soll, ist);
  }
  if (i < l.size())
    fail("hasNext fehlerhaft: vorzeitiger Abbruch");
}
```

Ende Listing

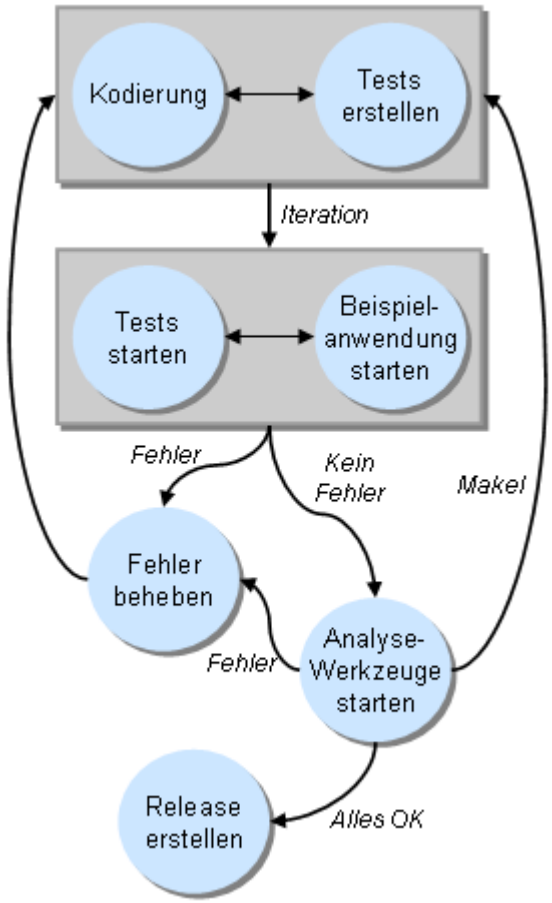

Abb. 1: Von der Software-Erstellung zum Release