Generieren geht über Kodieren

von Klaus Meffert

Das Erstellen von Unit Tests ist wichtig, denn sie ermöglichen es, schnell einen Eindruck über die Fehlerhaftigkeit des getesteten Programmcodes zu vermitteln. Grundlegende Tätigkeiten bei der Testfallerstellung können automatisiert werden und verschaffen dem Entwickler so Zeit für andere Aufgaben. Dieser Artikel gibt einen Einblick in die Möglichkeiten, die vorhandene Werkzeuge hierfür bieten. Der Text ist ein Extrakt aus dem im Mai erscheinenden Buch *JUnit Profi-Tipps – Bewährte Techniken und Antipatterns* und die Fortsetzung des vorigen Artikels [6].

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

Im Rahmen des *Test Driven Development* (*TDD*) ist es selbstverständlich, Testfälle vor dem eigentlichen Coding zu erstellen. Ein anderer Ansatz ist die Erstellung von Testfällen nach dem Coding. Zwischen diesen Vorgehensweisen gibt es Abstufungen, bis hin zum Weglassen von Unit Tests. Besonders in kommerziellen Softwareprojekten ist der Zeitfaktor für die Testerstellung ein politisch entscheidender. Denn der Zeitplan ist oft so eng gefasst, dass in erster Linie Resultate zählen. Unit Tests stellen einen Zusatzaufwand dar, der nicht von jedem Interessenvertreter als direkter Vorteil angesehen wird. Der Entwickler andererseits möchte seine Unit Tests so effizient wie möglich implementieren und dabei auf unterstützende Werkzeuge zugreifen.

Es hätte etwas von Magie, wenn sich Programme von selbst schrieben. Etwas realistischer erscheint die Aufgabe, Unit Tests für bestehenden Code durch ein Werkzeug erstellen zu lassen. Nichtsdestotrotz kann diese Aufgabe nicht beliebig automatisiert werden. Ganz verloren ist der auf Unterstützung hoffende Entwickler aber nicht. Zumindest in Grundzügen helfen zahlreiche Werkzeuge – viele davon aus dem Open Source-Sektor – bei der Erstellung von rudimentären Testfällen. Einige solcher dankenswerten Möglichkeiten werden in diesem Artikel diskutiert. Insbesondere besitzen manche Entwicklungsumgebungen eingebaute Funktionen zur Generierung von Testfällen. Es kann sich mitunter lohnen, für die Testfallerstellung eine andere Entwicklungsumgebung als die sonst verwendete einzusetzen.

Grundprinzip

Die meisten Werkzeuge, die Testfälle automatisch erstellen, arbeiten nach demselben Grundprinzip. Zuerst werden die Klassen manuell vorgegeben, für die Testfälle zu erstellen sind. Entweder wird eine Liste von Klassennamen angegeben. Oder es erfolgt eine Auswahl über einen Package-Namensraum. Anschließend werden diese Klassen vom Werkzeug daraufhin analysiert, ob sie für die Testfallerstellung berücksichtigt werden sollen. Manche Werkzeuge bieten beispielsweise die Möglichkeit, innere Klassen auszuschließen. Für jede relevante Klasse werden dann die enthaltenen Methoden untersucht. Für eine private Methode etwa wird im Normalfall kein Testfall generiert. Für alle geeigneten Methoden werden automatisch Testfallmethoden erzeugt. Es handelt sich hierbei erst einmal um Methodenrümpfe ohne Logik. Abhängig vom verwendeten Werkzeug wird dann die Logik der Originalmethode ausgewertet und hierfür eine Testlogik generiert. Das Generat, so wird der erzeugte Code auch genannt, sollte dann vom Entwickler erweitert oder angepasst werden. Die Generatoren sind nämlich nicht in der Lage, jederzeit ein akzeptables Ergebnis zu produzieren. Vielmehr sollte man das Ergebnis einer automatischen Testfallerstellung als Grundlage ansehen. Jedenfalls ist das Schreiben von Testfällen oft schneller möglich, wenn bereits ein Rahmen existiert. Im Weiteren widmen wir uns mit

JUnitDoclet einerseits einem populären Werkzeug, das ausschließlich der Generation von Testfällen dient. Andererseits fließen mit Eclipse, NetBeans und JBuilder drei populäre Entwicklungsumgebungen in die Betrachtung ein. Alle vier Softwarepakete eignen sich jedenfalls als Unterstützung bei der Arbeit mit JUnit [1].

JUnitDoclet

JUnitDoclet [2] ist in erster Linie ein Code Generator, wenn auch ein etwas unkonventioneller. Er verwendet das Javadoc-Tool, um Informationen über die Struktur einer Klasse zu gewinnen. Ein Javadoc-Kommentar ist ein Kommentar, der in einem durch die Zeichen /** und */ begrenzten Kommentarblock steht. Spezielle Javadoc-Tags beginnen innerhalb dieses Kommentars mit dem at-Symbol @.

JUnitDoclet verwendet die mit Java mitgelieferte Javadoc-Verarbeitung nun, um einen Quelltext auswerten und daraus Testfälle generieren zu können. Das Ergebnis ist ein automatisch erzeugtes Skelett von Testsuiten und Testfällen. Dem Entwickler wird die lästige Routinearbeit abgenommen, Testklassen anzulegen, mit der grundlegenden Infrastruktur zu versorgen und pro Geschäftsmethode mindestens eine Testmethode implementieren zu müssen. Weiterhin kann mit JUnitDoclet sichergestellt werden, dass sämtliche automatisch erstellten Testfälle auch in Testsuiten berücksichtigt werden.

Bei der automatisierten Testfallerstellung kann mitunter nichtkompilierbarer Programmcode entstehen. Daran wird deutlich, dass ein vollständig automatisch erzeugter Testfall im Allgemeinen und nach aktuellem Stand der Technologie nicht möglich ist. Jedoch erfährt der Entwickler eine enorme Arbeitserleichterung, bekommt er den essentiellen Rahmen doch generiert und muss sich nicht mehr um derartige „Trivialitäten“ kümmern. Listing 1 zeigt den Aufruf von JUnitDoclet über die Kommandozeile (berücksichtigt sind die wichtigsten Optionen). Wie dort zu sehen, wird JUnitDoclet über das Javadoc-Hilfsprogramm (für Windows: *javadoc.exe*) aufgerufen. Es befindet sich im *bin*-Verzeichnis unterhalb des Java-Installationsverzeichnisses. Die o.g. Optionen sind im einzelnen:

- *-docletpath*: Pfad, in dem sich die Datei *JUnitDoclet.jar* befindet.
- *-doclet*: Hiermit wird *javadoc.exe* mitgeteilt, wo sich die JUnitDoclet-Klasse befindet. Dieser Wert ist für JUnitDoclet (in der vorliegenden Version) immer gleich dem Klassennamen *com.objectfab.tools.junitdoclet.JUnitDoclet*.
- *-sourcepath*: Pfad, in dem sich die Quellen befinden.
- *-d*: Verzeichnis, in dem die generierten Testklassen abgelegt werden. Das Verzeichnis muss existieren.
- *-buildall*: Wird dieser Parameter angegeben, werden alle Tests neu erstellt, auch wenn dazu kein Bedarf besteht.
- *<ohne Optionsname>*: Am Befehlsende wird eine Liste von Packages angegeben, für die Testfälle automatisch erstellt werden sollen. Jede öffentliche (*public*) Klasse, die direkt in einem der angegebenen Packages liegt, wird berücksichtigt, mit Ausnahme von abstrakten Klassen und Schnittstellen.

Das Ergebnis der Ausführung wird für das obige Beispiel im Unterverzeichnis *junit* (vergleiche Parameter *-d*) abgelegt und ergibt sich aus folgenden Schritten:

- Für jede nichtabstrakte nichtinnere öffentliche Klasse, die keine Schnittstelle und keine *Exception*-Klasse ist und deren Name mit der sie enthaltenden Datei übereinstimmt, wird eine Testklasse angelegt. Klassen, die zusätzlich zur (Haupt-)Klasse deklariert wurden, werden nicht berücksichtigt, ebenso wenig innere Klassen. Der Name der Testklasse lautet so wie die ursprüngliche Klasse, versehen mit dem Postfix *Test*.
- Für jede öffentliche Methode innerhalb einer berücksichtigten Klasse wird in der neu generierten Testklasse eine Testmethode (Testfall) ohne Inhalt angelegt. JUnitDoclet berücksichtigt auch Methoden ohne Rückgabewert.

- Für jede Testklasse werden die JUnit-Methoden *setUp()* und *tearDown()* automatisch mit generiert, die dann allerdings keine wesentliche Logik beinhalten.
- JUnitDoclet generiert eine Testsuite, die alle generierten Testklassen enthält. Der Name der Suite wird aus dem letzten Package-Bestandteil durch Großschreibung des ersten Buchstabens gebildet, indem die Zeichenkette „Suite“ angehängt wird. Für das Paket *org.myPackage* lautet der Name der generierten Testsuite also *MyPackageSuite*.
- Testklassen werden nur neu generiert, wenn sich eine Änderung ergeben würde, wenn sich also zumindest eine Methodensignatur der zu testenden Klasse geändert hat, eine Methode hinzugefügt oder eine entfernt wurde.

Da JUnitDoclet ein Code Generator ist und in den generierten Unit Tests manuell weitere Implementierungen vorgenommen werden müssen, muss sichergestellt sein, dass diese manuellen Änderungen bei Neugenerierung von Tests nicht gelöscht werden. Bei Code Generatoren hat sich ein Mechanismus eingebürgert, der *Protected Region* (zu Deutsch: geschützte Region) heisst und genau dafür gedacht ist, manuelle Änderungen innerhalb von generiertem Code zu bewahren. JUnitDoclet verwendet dazu einfache Kommentare der folgenden Form für Methodenrümpfe bzw. analoge Kommentare innerhalb von Klassen.

```
// JUnitDoclet begin method X
...
// JUnitDoclet end method X
```

Alle Anweisungen, die sich innerhalb der beiden eben genannten Kommentare befinden, werden bei der Neugenerierung konserviert und nicht gelöscht. Alles, was nicht innerhalb der beiden Kommentare steht, ist nach einer Neugenerierung ersetzt worden.

Wie reagiert JUnitDoclet bei der Neugenerierung, wenn sich eine Klasse geändert hat? Im Falle von neu hinzugefügten Methoden wird pro Methode einfach eine neue (leere) Testmethode hinzugefügt. Wird eine Methode geändert, hat das nur Auswirkungen auf die Testfallgenerierung, wenn es sich um eine relevante Änderung handelt. Eine Änderung auf Methodenebene ist für JUnitDoclet nur relevant, wenn der Name der Methode geändert wurde oder die Sichtbarkeit nicht mehr *public* ist. Änderungen an der Parameterliste oder dem Rückgabewert einer Methode wirken sich nicht aus. Der Grund ist, dass das Gerüst einer Testmethode losgelöst davon ist. Weil aber der Name einer Testmethode mit der korrespondierenden Geschäftsmethode verbunden ist, wirkt sich eine Namensänderung aus. Bei der Neugenerierung wird die Testmethode mit dem alten Namen entfernt und zwar inklusive des Rumpfes, der den geschützten Bereich (*Protected Region*) enthält! Weiterhin wird eine neue Testmethode angelegt, die dem neuen Namen der Geschäftsmethode entspricht. Um eventuell vorhandene Logik im geschützten Bereich nicht für immer im Nirwana verschwinden zu lassen, hat JUnitDoclet eine Sicherung eingebaut. Die eben genannte Logik wird nicht restlos gelöscht, sondern in eine spezielle Methode *testVault()* verschoben. In dieser Methode werden alle als obsolet angesehenen Logiken aus geschützten Bereichen gesammelt, bis der Entwickler sie manuell von dort gelöscht hat.

Ein Beispiel: Es existiert eine Testmethode *testBerechnePreis()*, in die manuell Testlogik eingefügt wurde. Die für unser Beispiel wesentlichen Teile der von JUnitDoclet generierten Testklasse sehen nun so aus wie in Listing 2 gezeigt.

Nun benennen wir die Methode *berechnePreis()* in *berechneKosten()* um. Listing 3 zeigt das Generat nach erneutem Ausführen von JUnitDoclet. Jetzt befindet sich in *testBerechneKosten()* keine Testlogik mehr! Mitunter wäre im Beispiel dieselbe Logik zu verwenden wie zuvor. In diesem Fall hätte der Entwickler die Möglichkeit, aus der „Haldenmethode“ *testVault()* die oben fett gedruckte Passage zu übernehmen, er müsste sie nicht neu implementieren!

Wird eine Klasse komplett gelöscht, entfernt JUnitDoclet nicht die dazu generierte Testklasse. Dies muss aus Sicherheitsgründen der Entwickler von sich aus tun. Nicht optimal hingegen ist das Verhalten des Tools,

wenn eine Klasse in eine *Exception*-Klasse umgewandelt wird, für die ja keine Testfälle generiert werden. Dann werden beim Generierungsprozess keine Warnungen, Informationen oder Fehler ausgegeben; die Generierung ignoriert die Klasse einfach. Wurden in der Klasse gleichzeitig Methoden umbenannt oder neu eingefügt, hätte der Entwickler womöglich ein zusätzliches Interesse zu erfahren, dass eine generierte Testklasse nun nicht mehr aktuell ist.

Weitere Optionen von JUnitDoclet

Wem das Standardverhalten von JUnitDoclet nicht ausreicht, der kann als zusätzliche Option beim Aufruf des Werkzeugs eigene Strategieklassen übergeben, die sich um das Erstellen der Testfälle, das Benennen der Testklassen und Testmethoden sowie um das Lesen und Schreiben von Dateien kümmern. Als Vorlage für die Erstellung eigener Strategieklassen dienen die Standardimplementierungen im Package *com.objectfab.tools.junitdoclet*. Die Klassen lauten:

- *DefaultTestingStrategy*: Ermittelt, für welche Packages, Klassen und Methoden Tests zu erstellen sind und stellt das Generat bereit (unter Zuhilfenahme von Vorlagen und Properties).
- *DefaultNamingStrategy*: Erzeugt Namen für generierte Klassen und Methoden und prüft, ob ein gegebener Name eine Testklasse o.ä. repräsentiert.
- *DefaultWritingStrategy*: Liest und schreibt eine Testklasse.

NetBeans

Eine in der Entwicklergemeinde beliebte Entwicklungsumgebung ist NetBeans [3]. Die Verantwortlichen von NetBeans legen übrigens größten Wert auf die Sicherstellung der Qualität ihres Produkts mit Hilfe von Unit Tests (siehe *http://www.netbeans.org/community/guidelines/q-evangelism.html*). NetBeans ist kostenlos erhältlich und liefert die Möglichkeit der automatischen Testfallerstellung von Hause aus mit. Die gebotenen Optionen sind umfangreicher als die von JUnitDoclet, aber nicht flexibler (vergleiche die Strategieklassen, die für JUnitDoclet beliebig implementiert werden können). Ausserdem ist die prinzipiell graphische Darstellung anwenderfreundlich (siehe Abbildung 1).

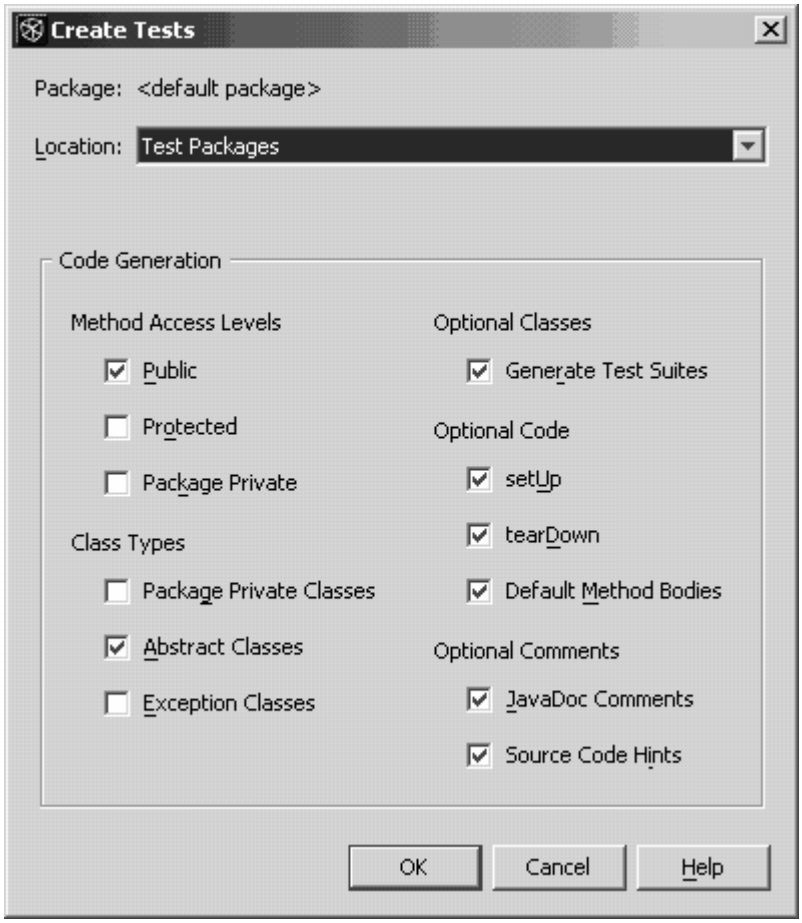

Abbildung 1: NetBeans-Optionen zum Generieren von Unit Tests

Um einen Unit Test für eine Klasse zu erstellen, ruft man das Kontextmenü der Klasse per Rechtsklick auf. Unter dem Untermenü *Tools* befindet sich der Eintrag *Create JUnit-Test*. Das Ergebnis des Prozesses sind gemäß den Optionen erstellte Testklassen samt Testmethoden sowie pro Package eine Testsuite. Weiterhin wird im obersten Package (ohne Namen) eine Testsuite namens *RootSuite* angelegt, die alle generierten Testsuiten enthält. Für Methoden mit Rückgabewert fügt NetBeans sowohl einen Aufruf des Getters als auch eine *assertEquals*-Anweisung ein. Dies ist natürlich nur als Vorlage zu betrachten und muss in den meisten Fällen manuell angepasst werden. Als störend können die in jede Testmethode eingefügten Konsolenausgaben mittels *System.out(...)* aufgefasst werden, die auf jeden Fall entfernt werden sollten. Werden abstrakte Klassen bei der Generierung von Tests berücksichtigt, sind die dafür generierten Testmethoden manuell zu bearbeiten. Denn NetBeans erzeugt ungeachtet dessen, ob eine Klasse abstrakt ist, eine Anweisung zur Instantiierung der Klasse durch Aufruf von deren Standardkonstruktor mittels *new*-Operator, etwa `MyAbstractClass instance = new MyAbstractClass();`. Was passiert, wenn eine Methode umbenannt oder gelöscht wird? Die Antwort lautet: NetBeans behandelt diesen Fall nicht. In Konsequenz bedeutet das, obsolete Testfälle bleiben im Generat erhalten, bis sie manuell gelöscht werden. NetBeans gibt dem Entwickler keinerlei Informationen hierzu. Glücklicherweise erinnert der Compiler durch entsprechende Fehler an diesen Missstand.

Eclipse

Eclipse [4] bietet die Möglichkeit, eine Testklasse für eine bestehende Produktivklasse anzulegen. Im Kontextmenü zum Projekt (Sicht *Package Explorer*) wählt man *New* → *JUnit Test Case*. Eclipse fragt dann eine Reihe von Parametern ab, darunter auch, für welche Klasse und welche Methoden daraus Testfälle erstellt werden sollen. Das Ergebnis ist allerdings mager: Der erzeugte Quelltext enthält nur leere Methoden. Für JUnitDoclet gibt es ein Eclipse-Plugin. Die Distribution findet sich auf der JUnitDoclet-Homepage [2] und wird einfach in das Plugins-Verzeichnis von Eclipse entpackt. Nach dem Neustart von Eclipse muss das Javadoc-Tool für JUnitDoclet angegeben werden (Menü *Window* →*Preferences* → *JUnitDoclet*). Nun kann im Kontextmenü

einer Klasse in der Package-Explorer Sicht der Eintrag *JUnitDoclet*, anschließend das Untermenü *Create JUnit Classes* gewählt werden.

JBuilder

JBuilder [5] ist eine ebenfalls weit verbreitete Entwicklungsumgebung, die als Testversion kostenlos erhältlich ist. JBuilder soll zwar aus dem Portfolio von Borland ausgegliedert und durch eine Art „Eclipse-Version“ ersetzt werden, wird aber noch von vielen Entwicklern verwendet und ist recht leistungsfähig. Auch in JBuilder ist von Hause aus eine automatische Testfallgenerierung integriert. Sie kann einfach über das Hauptmenü *File* → *New* → *Test* (im Baum links) aufgerufen werden (für die mir nicht vorliegende deutsche Version wahrscheinlich: *Datei* → *Neu* → *Test*). Nun bietet JBuilder die Möglichkeit, zwischen *Test Case* und *Test Suite* zu wählen. Für *Test Case* wird eine Klasse abgefragt, die zu testen ist. Zu der abgefragten Klasse kann der Benutzer zusätzlich die zu berücksichtigenden Methoden einschränken, für die ein Testfallgerüst zu erstellen ist. Für eine Testsuite können die einzubeziehenden Testklassen sowie ein TestRunner ausgewählt werden. Weiterhin können optional *Fixtures* zu einem Testfall angegeben werden. Bereitgestellte *Fixtures* sind beispielsweise die Klassen *JdbcFixture* sowie *JndiFixture* im Package *com.borland.jbuilder.unittest*. Eine Fixture dient zur einfachen (De-)Initialisierung einer bestimmten Ressource, etwa einer Datenbankverbindung (*JdbcFixture*). Die speziellen Methoden *setUp()* und *tearDown()* legt JBuilder automatisch an. In *setUp()* wird freundlicherweise bereits eine Objektinstanz der zu testenden Klasse erzeugt, die dann in den Testfällen verwendet wird. Eine Ausnahme bilden Testmethoden für Konstruktoren. In diesen Testmethoden fügt JBuilder eine Anweisung für die Erstellung einer neuen Instanz ein. Darauf folgt ein Kommentar, der zum Ausimplementieren des Tests auffordert. Er lautet bei JBuilder */**@todo fill in the test code*/*.
Es wirkt sich erleichternd aus, dass alle im Konstruktor benötigten Eingabeparameter in der Testmethode zumindest deklariert und mit dem Wert *null* initialisiert werden. Das erspart einiges an lästiger Schreibarbeit. Für Testmethoden, die sich auf normale Geschäftsmethoden (nicht auf Konstruktoren) beziehen, wird statt einer Konstruktionsanweisung ein Codeblock eingefügt, der vom Rückgabewert der unter Test stehenden Methode abhängt. Besitzt diese Methode einen Rückgabewert, sieht der automatisch erstellte Testfall etwa so aus wie in Listing 4. *genotype* ist dabei die Instanzvariable vom Typ der Klasse, die getestet wird. Der Entwickler muss nun noch das erwartete Ergebnis (Variable *expectedReturn*) eintragen sowie den ersten Parameter von *assertEquals* durch eine Fehlermeldung ersetzten. Die Angabe der Fehlermeldung kann auch ersatzlos gestrichen werden. Für zu testende Methoden ohne Rückgabewerte gibt es kein *actualResult*. Einen solchen Testfall zeigt Listing 5. Der Vergleich mit einem erwarteten Ergebnis muss nun nachimplementiert werden. Anstelle dessen kann ein *Refactoring* des Codes in Erwägung gezogen werden.

Fazit

Populäre Entwicklungsumgebungen bieten von Hause aus die Möglichkeit, Testfälle automatisch zu generieren. Zumindest zur Erstellung von Gerüsten eignen sie sich, um die Fertigstellung der Testfälle muss sich der Entwickler kümmern. JUnitDoclet unterstützt *Protected Regions* und stellt eine gute Basis für die Ausimplementierung generierter Testfälle bereit. Für Eclipse sind zahlreiche Plugins erhältlich, die bei der Generation von Testfällen behilflich sind. JBuilder und NetBeans unterstützen JUnit von Hause aus durch Assistenten, die ein Rahmenprogramm für Testfälle erstellen. Triviale Konstrukte, wie der Test von Settern und Gettern, sowie Code-Grundgerüste können damit zufriedenstellend realisiert werden. Unterstützung beim Test von komplexerer Logik – und das ist der überwiegende Teil aller Testlogik – muss manuell ausimplementiert werden. Immerhin erleichtern JUnitDoclet & Co die Arbeit mit JUnit, indem dem Entwickler lästige Routineaufgaben abgenommen werden. Die testgetriebene Entwicklung ist nicht kompatibel mit dem beschriebenen Verfahren, weil dort der Code erst nach den Testfällen geschrieben wird.

Klaus Meffert arbeitet als Organisationsberater der Brandt & Partner GmbH in der Softwareentwicklung. Parallel dazu verwirklicht er seine Doktorarbeit zum selben Thema. Weiterhin engagiert er sich im Open Source-Bereich und als Fachautor.

Links & Literatur

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

[2] JUnitDoclet-Homepage: www.junitdoclet.org

[3] NetBeans JUnit Module: junit.netbeans.org

[4] Eclipse-Homepage: www.eclipse.org

[5] JBuilder-Homepage: www.borland.com/de/products/jbuilder/

[6] Klaus Meffert: JUnit 4 – Was lange währt…, in *Java Magazin* 04.2006

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

[8] Andreas Braig und Steffen Gemkow: Testen ohne wenn und aber, in *Java Magazin* 01.2003

[9] Eberhardt Wolff und Andreas Braig: Pragmatisch Programmieren: Code-Generierung entzaubert, in *Java Magazin* 07.2003

Listing 1: Aufruf von JUnitDoclet über die Kommandozeile

```
import static org.junit.Assert.*;
}
javadoc
 -docletpath ./lib/JUnitDoclet.jar
 -doclet com.objectfab.tools.junitdoclet.JunitDoclet
 -sourcepath ./src
 -d ./junit
 -buildall
 org.jgap org.jgap.audit
```

Listing 2: Von JUnitDoclet generierte Testmethoden

```
public void testBerechnePreis() throws Exception {
  // JUnitDoclet begin method berechnePreis
  double preis = testKlasse.berechnePreis();
  // JUnitDoclet end method berechnePreis
}
public void testVault() throws Exception {
  // JUnitDoclet begin method testcase.testVault
  // JUnitDoclet end method testcase.testVault
}
```

Listing 3: Code-Konservierung mit JUnitDoclet

```
public void testBerechneKosten() throws Exception {
  // JUnitDoclet begin method berechneKosten
  // JUnitDoclet end method berechneKosten
}
public void testVault() throws Exception {
  // JUnitDoclet begin method testcase.testVault
// JUnitDoclet begin method berechnePreis
  double preis = testKlasse.berechnePreis();
// JUnitDoclet end method berechnePreis
  // JUnitDoclet end method testcase.testVault
}
```

Listing 4: Von JBuilder generierter Testfall

```
public void testGetChromosomes() {
 Chromosome[] expectedReturn = null;
 Chromosome[] actualReturn =genotype.getChromosomes();
 assertEquals("return value", expectedReturn,
              actualReturn);
 /**@todo fill in the test code*/
}
```

Listing 5: Von JBuilder generierter Test für Methoden ohne Rückgabewert

```
public void testEvolve1() {
  int a_numberOfEvolutions = 0;
  genotype.evolve(a_numberOfEvolutions);
  /**@todo fill in the test code*/
}
```