

kurz & bündig	
Inhalt	Professionelles Arbeiten mit JUnit
Zusammenfassung	Sicherstellung der Qualität von JUnit Tests
Quellcode mit angeliefert	Nein

## Unter der Lupe

### Qualitätssicherung von JUnit-Tests

#### von klaus meffert

Der vierte und letzte Teil unserer Testing-Reihe widmet sich Möglichkeiten zur Sicherstellung und Verbesserung der Qualität von Software-Tests mit JUnit. Hauptaugenmerk wird auf die Vorstellung von frei erhältlichen Werkzeugen gelegt. Diese Werkzeuge ermöglichen es, die Qualität von JUnit-Tests zu beurteilen und zu verbessern.

#### Kasten

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

#### Ende Kasten

#### Einleitung

Entscheidend für die Beurteilung der Qualität von Software-Tests sind die Fragen: Wird das getestet, was getestet werden soll? Wird alles Getestete richtig getestet? Wird alles Getestete vollständig getestet, werden alle relevanten Perspektiven (Korrektheitsprüfung, Performanzprüfung, Bedienbarkeitsprüfung etc.) berücksichtigt? Entspricht die Testlogik den Formvorschriften? Ist die getestete Logik würdig, getestet zu werden? Eine erschöpfende oder eindeutige Antwort auf jede dieser berechtigten Fragen gibt es nicht. Aber mit Hilfe der im weiteren diskutierten Hilfsmittel lassen sich Indikatoren mit wenig Aufwand finden. So gibt es zum Beispiel ein Werkzeug, das durch kontrollierte Mutation von Quelltext unvollständige Testfälle entlarvt.

#### Gewollte Mutation von Quelltext

Die Idee ist simpel, der Aha-Effekt oft groß: Mit Hilfe eines Werkzeugs wird zuerst der eigene Quelltext modifiziert, unter der Prämisse, daß er danach noch kompilierbar und ausführbar ist. Danach werden alle verfügbaren Unit-Tests darauf losgelassen (durch Starten der Master-Testsuite). Anschließend läßt sich leicht feststellen, ob die ausgeführten Tests tatsächlich eine genügende Abdeckung bieten. Schlug nämlich für eine durch Mutation modifizierte Programmstelle kein Test fehl, gibt es zwei Möglichkeiten: entweder ein Test ist unvollständig und erkennt den Fehler nicht oder es ist bisher kein Test vorhanden. In beiden Fällen muß nachgearbeitet werden, sprich, es müssen Tests verändert oder hinzugefügt werden.

Das Werkzeug Jester [7] verfolgt eben diesen Ansatz der Quelltextmutation. Dazu werden in einem Ausgangs Quelltext Stellen identifiziert, die mutiert werden können, ohne Kompilierbarkeit und Ausführbarkeit des Quelltextes zu gefährden. Es handelt sich um solche Stellen, die einer von mehreren hinterlegten Regeln gehorchen. Normalerweise ist beispielsweise die Ersetzung einer ganzen Zahl durch eine andere ganze Zahl unkritisch bezüglich der Ablauffähigkeit eines Programms, ganz sicher aber unkritisch hinsichtlich dessen Kompilierbarkeit. Gelegentlich wird die Ablauffähigkeit zwar durch eine solche triviale Ersetzung gestört, aber dies sollte nur in ganz wenigen Fällen vorkommen. Die Bedienung von Jester ist recht einfach und soll hier nicht weiter beschrieben werden (siehe Jester's Homepage oder das JUnit-Buch [8] für weitere Beschreibungen).

## Programmanalyse mit JCSC

Für Unit Tests gelten prinzipiell erst einmal dieselben Programmierrichtlinien wie für produktiven Quelltext. Insbesondere trifft das auf Formatierungen, Modularität und Namenskonventionen zu. Im Open-Source Sektor bieten mehrere Werkzeuge die Möglichkeit, sowohl allgemeine Programmkonventionen zu überprüfen und deren Einhaltung zu fordern als dies auch speziell für Unit Tests zu tun. In jedem Fall kommen die vorgestellten Werkzeuge dem Ersteller von Unit Tests zugute.

JCSC [1] (*Java Coding Standard Checker*) überprüft Quelltext auf Verletzungen von gutem Stil. Das ermöglicht es dem Entwickler, seinen Quelltext hinsichtlich syntaktisch unsauberer Konstrukte zu beurteilen und Verletzungen von Formatierungsregeln zu erkennen. Überprüft werden insbesondere die Aspekte:

- Zeilenlänge (sollte maximal 80 Zeichen groß sein),
- Javadoc-Kommentare (sollten nicht fehlen),
- Modifizierer (wie *final* und *static*) sollten in Deklarationen in einheitlicher Reihenfolge verwendet werden sowie
- die Verwendung von komplexen Ausdrücken in Schleifen (innerhalb von `for(int i=0;i<size();i++)` ist der Aufruf der Methode `size()` ungünstig, er sollte durch einfaches Feld ersetzt werden).

Weiterhin ermittelt JCSC zu jeder analysierten Klasse Metriken: erstens die NCSS (*Non Commenting Source Statements*), sie gibt die Anzahl der funktionalen Code-Zeilen (Kommentare werden ignoriert) an; zweitens die CCN (*Cyclomatic Complexity Number*), welche die Anzahl der Ablaufflüsse berücksichtigt.

Vor der Verwendung von JCSC muss einerseits die Systemvariable `JCSC_HOME` auf das Verzeichnis gesetzt werden, in dem JCSC installiert wurde. Andererseits muss das Verzeichnis `JCSC_HOME\bin` dem Pfad (der `PATH`-Variable) hinzugefügt werden. JCSC kann in zwei Modi aufgerufen werden. Erstens direkt über eine Batchdatei: `jcsc.bat Java_Datei >jcsc_report.txt`. In diesem Modus wird die mit dem Parameter `Java_Datei` angegebene Klasse (Quelltext, nicht `.class`-Datei) analysiert. Die Umleitung in die Datei `jcsc_report.txt` ist sinnvoll, um unübersichtliche, nicht persistente Konsolenausgaben zu vermeiden.

Zweitens kann JCSC als Ant-Task aufgerufen werden. Vorteil ist, dass JCSC dann automatisch bei jedem Build gestartet werden kann. Weiterhin analysiert JCSC so auch alle `.java`-Dateien. Der Einbau in die `build.xml`-Datei funktioniert wie im Listing 1 gezeigt. Das Regelverzeichnis ist jenes, in dem die mitgelieferte Datei `jcsc.jcsc.xml` liegt. Das angegebene Zielverzeichnis (`destdir`) muss vorhanden sein. Dort werden Analyseergebnisse abgelegt. Das Ergebnis kann durch Aufruf der Datei `index.html` abgerufen werden. Für die Ant-Einbindung müssen die von JCSC benötigten Bibliotheken `jcsc.jar`, `gnu-regexp.jar`, `xercesImpl.jar` sowie `xml-apis.jar` in das `lib`-Verzeichnis von Ant kopiert werden.

Die Regeln von JCSC können konfiguriert werden. Dazu liefert JCSC ein Modul namens *Rules Editor UI* mit, das über eine graphische Oberfläche verfügt. Der Editor kann über die mitgelieferte Datei `ruleseditor.bat` im `bin`-Verzeichnis von JCSC gestartet werden. Er erlaubt vielfältige Einstellungen, die aufgrund der Tooltips selbsterklärend sind.

Tritt beim Ausführen von JCSC eine Ausnahme wie `StringIndexOutOfBoundsException` auf, dann liegt die Ursache höchstwahrscheinlich darin, dass JCSC Probleme mit einer Quelltextdatei hat. Welche Datei das ist, kann herausgefunden werden, indem der zeitlich zuletzt angelegte Report im Zielverzeichnis gesucht wird. Die generierten Reporte (aufrufbar über `index.html`) sind ähnlich strukturiert wie Javadoc, so dass sich der Java-Entwickler dort gut zurechtfinden kann.

## Das Tool Ashcroft

Ashcroft [6] legt Konventionen fest, gegen die ein Unit Test nicht verstoßen darf. Ashcroft verbietet beispielsweise den Dateizugriff (Lesen, Schreiben, Prüfen auf Vorhandensein) durch Unit Tests. Weiterhin sollte eine Testklasse nur eine Produktivklasse testen (Argumente hierzu sind unter [6] zu finden).

Die Verwendung von Ashcroft ist recht einfach. Wird Ant verwendet, muss in der *build.xml*-Datei der Abschnitt für die Unit Tests ergänzt werden, so dass er beispielsweise so aussieht wie in Listing 4. Die *ashcroft.jar*-Datei wird für jene Konfiguration im *lib*-Verzeichnis erwartet. Es ist weiterhin darauf zu achten, dass sich die Bibliothek *junit.jar* im *lib*-Verzeichnis von *ant* befindet. Die Option

```
<jvmarg value="-Dcom.thoughtworks.ashcroft.runtime.logging=egal.txt"/>
```

hatte in meinen Versuchen keinen Effekt, so dass sie wohl auch weggelassen werden kann. Ashcroft schrieb vielmehr in eine Datei namens *TEST-org.my.AllTests.txt* (wobei der Teil nach dem Präfix *TEST*- bis vor das Postfix *.txt* mit dem in der Eigenschaft *allTests* definierten Wert übereinstimmt).

Die Option `<formatter type="brief"/>` ist wichtig, um das Erstellen einer Reportdatei zu garantieren. Ashcroft kann weiterhin mit Maven, Eclipse sowie IntelliJ IDEA verwendet werden. Bisher scheinen die auf der Webseite von Ashcroft genannten Konventionen nicht alle geprüft zu werden! Jedenfalls eignet sich das Werkzeug erstens, um Dateizugriffe in Unit Tests zu identifizieren und zweitens kann es als Vorlage für Weiterentwicklungen von Konventionen dienen.

### Konventionen mit Checkstyle prüfen

Checkstyle [4] zielt darauf ab, Programmkonventionen für Quelltext zu überprüfen und Verstöße zu melden. Beispiele für solche Prüfregeln sind:

- Namenskonventionen für Variablen,
- Vorhandensein und Form von Javadoc-Kommentaren,
- Position von öffnenden und schließenden Klammern,
- Konventionen für Import-Statements,
- Länge einer Zeile sowie
- überflüssige *throws*-Deklarationen.

Die Prüfregeln können konfiguriert werden (später mehr dazu). Das ist insbesondere für die Berücksichtigung eigener Namenskonventionen sinnvoll (etwa wenn eine *private* Klassenvariable immer mit *m\_* anfangen soll). Checkstyle ist für viele Entwicklungsumgebungen verfügbar, darunter sind wohl alle populären vertreten. Ansonsten kann Checkstyle direkt in Ant eingebunden werden, so dass es bei jedem Build-Prozess Berücksichtigung findet (Stichwort: *Continuous Integration*). Ausserdem steht ein eigener Runner zur Verfügung, mit dem Checkstyle gestartet werden kann.

### Spezielle Konfiguration von Checkstyle für Unit Tests

Da jeder Entwickler oder jedes Entwicklerteam eigene Konventionen für die Benennung von Programmelementen (Variablen, Klassen etc.) sowie eine eigene Vorgehensweise beim Programmieren hat, empfiehlt sich die Anpassung der Prüfregeln auch für JUnit-Tests. Für Checkstyle sollte ein eigenes Ant-Target angelegt werden, das auf eine eigene Konfigurationsdatei für Prüfregeln zugreift. Die Konfigurationsdatei könnte etwa mit *unittest\_checks.xml* benannt sein und aus der mitgelieferten namens *sun\_checks.xml* kopiert werden. Dann ist es möglich, die Einhaltung von Programmkonventionen wesentlich spezifischer zu prüfen und *false positives* zu vermeiden (ein *false positive* ist in diesem Fall ein zu Unrecht gemeldeter Fehler bzw. eine Regelverletzung).

Um beispielsweise die zulässigen Methodennamen für Checkstyle zu konfigurieren, ist das Modul *MethodName* anzupassen, wie in Listing 3 dargestellt. Nun werden Methoden mit dem Namen *suite* oder solche,

die mit *test* anfangen, gefolgt von einem Großbuchstaben und weiteren Buchstaben, Ziffern oder dem Unterstrich, als gültig angesehen. Einige Module sollten bei Bedarf durch Auskommentieren oder Entfernen aus der Konfigurationsdatei deaktiviert werden, darunter

- *EmptyStatement*: Verbietet leere Anweisungen (gegeben durch ein einzelnes Semikolon).
- *PackageHtml*: Fordert, dass eine Datei namens *package.html* zu jedem Package existiert.
- *FinalParameters*: Verlangt, dass Parameter für Methoden und Konstruktoren final sind.
- *MagicNumber*: Prüft, dass für jede Zahl (ausser -1, 0, 1 und 2) eine Konstante definiert ist und die Zahl nur über den Konstantennamen verwendet wird.

## Das Analysewerkzeug PMD

PMD [5] enthält eine ganze Reihe von Regeln, gegen die ein Quelltext geprüft werden kann. PMD ist ein weit verbreitetes Werkzeug mit dem Zweck, potentielle Schwachstellen aufzudecken (s.u.). Neben seiner Eignung für die gewöhnliche Quelltextanalyse, besitzt PMD einige Prüfungen, die speziell auf JUnit bezogen sind. PMD ist besonders einfach zu verwenden, wenn es in eine Entwicklungsumgebung eingebunden wird. In der Version 3.6 werden Eclipse, NetBeans, JBuilder und IntelliJ IDEA sowie einige andere mehr unterstützt. Das jeweils für die eigene IDE benötigte Plugin muss übrigens gesondert heruntergeladen werden. Das Einbinden eines Plugins ist jeweils in der spezifischen Distribution erklärt. Für automatisierte Prüfungen liegt der Distribution eine Batch-Datei bei. Weil PMD, ähnlich wie Eclipse, das Einbinden von Modulen (Plugins) erlaubt und aufgrund der breiten Unterstützung von PMD in der Entwicklergemeinde genießt, kommen regelmäßig neue Erweiterungen hinzu.

Nach meiner Erfahrung sind die von PMD gemeldeten Warnungen meistens angebracht, so etwa:

- Die *suite()*-Methode eines Unit Tests muss öffentlich und statisch sein.
- Jede Testklasse sollte mindestens einen Testfall beinhalten.
- Zusicherungen sollten so verwendet werden, dass sie auch fehlschlagen können. *assertTrue(true)* wird niemals fehlschlagen.
- Die korrekte Schreibweise der Methoden *setUp()* und *tearDown()* wird geprüft. So werden etwa die Namen *setup* und *TearDown* als falsch erkannt.
- Eine Klasse mit dem Postfix *Test* sollte immer von der Klasse *junit.framework.TestCase* abgeleitet sein. Abhängig von der Namenskonvention kann das diskussionsbedürftig sein.
- *assertNull* anstatt *assertTrue*: Die Methode *assertTrue* als Alleskönner kann in missbräuchlich Weise auch anstatt *assertNull* verwendet werden.
- Allgemeine Regeln wie *Doppelte Importe sind unnötig* oder *Unbenutzte lokale Variablen können entfernt werden* runden das Angebot ab.

Der Aufruf von PMD über die Kommandozeile ist relativ einfach:

```
java -jar pmd-3.6.jar c:\dir_to\tests html rulesets/junit.xml >pmdreport.html
```

Für den Aufruf wechselt man am einfachsten vorher in das *lib*-Verzeichnis der PMD-Distribution. Das im Befehl angegebene Verzeichnis ist der Ablageort der Quelltexte für die Unit Tests. Der Parameter *html* kennzeichnet das Ausgabeformat (*xml* ist ebenso möglich). Der Parameter *rulesets/junit.xml* gibt den Namen des zu verwendenden Regelwerks an. Es handelt sich nicht um einen Dateinamen, sondern um einen Ressourcenqualifizierer. Die verfügbaren Regeln (bestimmt durch den zweiten Bestandteil nach *rulesets/*) können im Verzeichnis *docs\rules* der PMD-Distribution eingesehen werden. Der letzte Befehlsbestandteil leitet die Ausgabe in eine Datei um (wer will schon eine HTML-Ausgabe auf der Konsole sehen?).

Arbeiterleichternd wirkt sich die Anwendung der Regel *rulesets/unusedcode.xml* dann aus, wenn sie auf zu testenden Quelltext angewandt wird und hilft, unbenutzten (und somit entfernbar und nicht mehr zu testenden)

Code aufzuspüren. Wer eine der zahlreichen unterstützten IDE's verwendet, sollte lieber auf das entsprechende PMD-Plugin zurückgreifen. Zumal die Ergebnisse von PMD dann anklickbar sind und die IDE sofort in die entsprechende Zeile im Code navigiert (was im HTML-Report nicht so ist). Einige Prüfungen durch PMD können im Rahmen von Unit Tests unerwünscht sein. Um eine solche PMD-Prüfung auszuschalten, wird der entsprechenden Programmzeile der Kommentar `//NOPMD` hinten angestellt. Das empfiehlt sich etwa dann, wenn in Testfällen für leere `catch`-Blöcke keine Warnung generiert werden soll.

### Konventionen mit FindBugs prüfen

FindBugs [3] bietet ebenfalls einige für Unit Tests relevante Regeln an, darunter auch:

- Eine Testklasse muss mindestens eine Testmethode beinhalten.
- Eine implementierte `setUp()`-Methode muss `super.setUp()` aufrufen (analoge Regel zu `tearDown()`).
- `suite()`-Methoden sollten immer statisch sein (siehe auch PMD).
- Analyse des Programmflusses (wenn etwa innerhalb eines `if`-Blocks eine Variable `null` ist, weil dies in der `if`-Anweisung gefordert wurde, werden `NullPointerException` erkannt).
- Erkennen, wenn auf nicht initialisierte Felder in Konstruktoren zugegriffen wird.

FindBugs kann über die Kommandozeile als auch in Form eines Eclipse-Plugins verwendet werden. Über die Homepage von FindBugs kann das Werkzeug alternativ mittels Java Web Start ausgeführt werden. In meinem Fall funktionierte die Analyse dann allerdings nicht, weil keine Detektoren gefunden wurden. Am einfachsten wird die graphische Benutzeroberfläche von FindBugs mit Hilfe der Startdatei im Unterverzeichnis `bin` aufgerufen. Für Windows ist `findbugs.bat` zu starten, für andere Betriebssysteme steht die Datei `findbugs` im selben Verzeichnis zur Verfügung.

Nach dem Start von FindBugs präsentiert sich dem Benutzer eine Maske, die im Wesentlichen aus Menüzeile sowie drei Bereichen für die Angabe von zu analysierenden Klassen, den zugehörigen Quelltexten sowie externen Bibliotheken besteht. Die Bedienung der Findbugs-Oberfläche ist recht intuitiv (das genaue Vorgehen sowie weitergehende Betrachtungen sind in [8] beschrieben).

FindBugs analysiert eine riesige Anzahl an Fehlern. Es lohnt sich auf jeden Fall, das eigene Projekt diesem konkreten Test zu unterziehen. Selbst der erfahrene Entwickler wird gelegentlich von den konstruktiven Vorschlägen überrascht sein, die dieses Werkzeug unterbreitet. Beispielsweise empfiehlt FindBugs anstatt des Konstrukts `new Boolean(true)` die speichersparendere Variante `Boolean.valueOf(true)` zu verwenden.

Einige, von FindBugs als Fehler gemeldete Programmstellen können nur mit Hilfe einer Portion Kreativität so umgeformt werden, dass sie bei der nächsten Analyse nicht mehr aufgedeckt werden. Es bietet sich an, den FindBugs-Meldungen zunächst ein gewisses Vertrauen entgegenzubringen, in komplizierteren Fällen ein wenig zu probieren und erst nach erfolglosem Versuch davon auszugehen, dass in dem ein oder anderen Fall ein Hinweis oder eine Warnung unberechtigt ist.

Manche korrekte Programmstelle wird von FindBugs prinzipiell fälschlicherweise mit einer Warnung bedacht. Beispielsweise `DLS: Dead local store`. Die Beschreibung (Karteikarte `Einzelheiten` im Fehlerbericht von FindBugs) enthält hierzu auch Hinweise, die lesenswert sind. Im angegebenen Beispiel sind diese sogenannten *false positives* darauf zurückzuführen, dass FindBugs den Bytecode analysiert und dieser gelegentlich ein falsches Bild für die Analyse abgibt.

### Weitergehende Konfiguration

Die Standardeinstellungen von FindBugs reichen für einen ersten Analysedurchlauf aus. Wurden die gemeldeten Fehler ausgemerzt, kann FindBugs in einem Modus betrieben werden, der mehr Fehler zu finden verspricht (im Menü über `Einstellungen` → `Effort` → `Maximum`). Tauchen im Fehlerbericht einige Meldungen

auf, die nicht erwünscht sind, können diese durch Deaktivieren des zugehörigen Detektors beim nächsten Analyselauf ausgeblendet werden (Menü *Einstellungen* → *Detektoren konfigurieren*). Für jeden verfügbaren Detektor ist angegeben, wie zeitintensiv er ist. Das ist insbesondere für Projekte mit vielen Klassen ausschlaggebend. Um nur schwerwiegende Fehler zu berücksichtigen, wählt man den Menüpunkt *Ansicht* → *Filterwarnungen* → *Hohe Priorität*. Auch die Kategorie der Fehler kann über *Ansicht* → *Filterwarnungen* konfiguriert werden. FindBugs erlaubt es, einen Filter auf zu analysierende Klassen zu definieren. So können beispielsweise Klassen, die nicht mit dem Suffix *Test* enden, ignoriert werden. Dieses Feature steht allerdings momentan nicht über die GUI, sondern nur über die Kommandozeilenversion von FindBugs zur Verfügung.

### **Code Coverage, für Unit Tests**

Über Code Coverage bzw. Test Coverage wurde schon viel geschrieben. Es ist aber nicht so verbreitet, dass entgegen der klassischen Code Coverage Analyse festgestellt werden kann, ob alle Tests wie erwartet ausgeführt wurden. Beispielsweise wird eine Testmethode nicht ausgeführt, wenn ihr Name anstatt mit dem für JUnit 3.x vorgeschriebenen Präfix *test* mit dem falsch geschriebenen Präfix *Test* oder *tst* anfängt. Oder eine Reihe von Tests sind zwar korrekt vorhanden und auch als solche erkennbar, aber nicht über Testsuiten oder anderweitig in die Menge der auszuführenden Tests eingebunden. Letzteres kommt schon mal vor, da einige Tests, beispielsweise aufgrund ihrer längeren Laufzeit, in andere Testsuiten ausgegliedert werden. Werden Coverage Tools auf Testklassen anstatt auf Produktivcode losgelassen, decken sie nicht ausgeführte Tests sehr schnell auf.

### **Fazit**

Das Testen von Software ist kritisch genug, um diesem Aspekt der Software-Entwicklung besondere Aufmerksamkeit zu schenken. Damit dabei kein übermäßiger Aufwand entsteht, können zahlreiche, meist kostenlose Werkzeuge unterstützend verwendet werden. Sie finden unvollständige oder gar nicht ausgeführte Tests, Regelverletzungen und geben Hinweise, wie der eigene Code verbessert werden kann. In den täglichen Build-Prozess eingebunden, sind sie ein wertvolles Mittel zur Gewährleistung einer hohen Qualität von automatisierten Software-Tests. Es gibt wohl kein Software-Projekt, das die erstmalige werkzeuggestützte Untersuchung der Testfälle mit Bravour, also ohne nennenswerte Warnungen, übersteht! Es sollte aber nicht vergessen werden, dass Unit Tests durch weitere Testarten unterstützt werden sollten. So können Synchronisationsprobleme in Thread-Umgebungen mit dem Werkzeug JLint [2] entdeckt werden, ohne dass weitere Tests manuell zu erstellen sind.

Mit diesem Teil schließt die Artikelreihe zum Thema Unit Tests. Ich verbleibe mit der Hoffnung, Ihr Interesse für JUnit geweckt bzw. verstärkt zu haben und verweise für detailliertere Beschreibungen zu JUnit und Unit Tests auf das JUnit-Buch [8]. Regelmäßig neue Informationen bietet auch die zugehörige Webseite [9].

**Klaus Meffert** ist als Organisationsberater der Brandt & Partner GmbH tätig. Er arbeitet parallel an seiner Doktorarbeit zum selben Thema. Weiterhin engagiert er sich im Open-Source Bereich und als Fachautor.

### **Links & Literatur**

---

[1] JCSC: <http://jcsc.sourceforge.net>

[2] JLint: <http://artho.com/jlint>

[3] FindBugs: <http://findbugs.sourceforge.net>

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

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

[6] Ashcroft: <http://docs.codehaus.org/display/ASH/Home>

[7] Jester: <http://jester.sourceforge.net/>

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

[9] Die Seite rund um JUnit: <http://www.junit-buch.de/>

## Listing 1

### JCSC-Abschnitt in einer Ant-Builddatei

```
<taskdef name="jcsc"
  classname="rj.tools.jcsc.ant.JCSCTask"/>
<target name="jcscall">
  <jcsc rules="<Regelverzeichnis> "
    destdir="<Zielverzeichnis>"
    worstcount="20"
    copyassociatedfiles="true"
    failvalue="0.05"
    failseverity="5"
    jcshome="<Home-Verzeichnis von JCSC>"
    <fileset dir="${src}" includes="**/*.java"/>
  </jcsc>
</target>
```

Ende Listing

## Listing 2

### Checkstyle-Konfiguration mit Ant

```
<target name="checkstyle" description="egal">
  <checkstyle config="docs/sun_checks.xml"
    failureProperty="checkstyle.failure"
    failOnViolation="false">
    <formatter type="xml" tofile="checkst_rep.xml"/>
    <fileset dir="${src}" includes="**/*.java"/>
  </checkstyle>
  <style in="checkst_rep.xml" out="checkst_rep.html"
    style="dir/checkstyle-simple.xsl"/>
</target>
```

Ende Listing

## Listing 3

### Checkstyle-Konfiguration für JUnit

```
<module name="MethodName">
  <property name="format" value="^suite$|^test[A-Z][a-
    zA-Z0-9_]*$|^ [a-z][a-
    zA-Z0-9_]*$"/>
</module>
```

Ende Listing

## Listing 4

### Ausschnitt aus einer Ant-Datei für Ashcroft

```
...
<property name="lib" value="./lib"/>
<property name="allTests" value="org.my.AllTests" />
...
<target name="test" depends="compile">
<junit haltonfailure="no" fork="yes">
<formatter type="brief"/>
<jvmarg value="-Djava.security.manager=com.thoughtworks.ashcroft.runtime.JohnAshcroft"/>
<jvmarg value="-Dcom.thoughtworks.ashcroft.runtime.forgiving=false"/>
<jvmarg value="-Dcom.thoughtworks.ashcroft.runtime.logging=egal.txt"/><classpath>
...
<fileset dir="${lib}">
<include name="**/*.jar"/>
</fileset>
</classpath>
<test name="${allTests}"/>
</junit>
</target>
```

Ende Listing