Alles unter Kontrolle

Messung der Testabdeckung mit Open-Source-Tools

Tim Wellhausen

kontakt@tim-wellhausen.de
http://www.tim-wellhausen.de

29.06.2008

Zusammenfassung: Unit-Tests sind vielleicht nicht jedermanns Liebling - aus dem Projektalltag sind sie jedoch kaum noch wegzudenken. Schließlich können Software-Entwickler damit ihren Programmcode gründlich testen und somit Fehler schon früh im Projektverlauf entdecken. Trotzdem bedeutet eine große Anzahl von Unit-Tests nicht automatisch, dass diese Tests auch alle kritischen Code-Stellen überprüfen. Um herauszufinden, für welche Code-Abschnitte weitere Tests notwendig sind, bietet sich der Einsatz spezieller Tools an. Dieser Beitrag zeigt am Beispiel eines frei verfügbaren Tools, welche Aussagekraft die Testabdeckung hat und wie ein passendes Tool wertvolle Entscheidungshilfen geben kann.

Einführung

Unit-Tests helfen, den erstellten Programmcode schon während der Programmierung zu testen. Da Unit-Tests von den Entwicklern geschrieben werden, überprüfen die Entwickler somit ihren Code selbst auf Korrektheit, idealerweise, ohne ihn zuvor auf einem Testsystem ausrollen zu müssen.

Ein einheitliches Testkonzept für Unit-Tests gibt es eher selten. Vielmehr ist üblicherweise jeder Entwickler alleine dafür verantwortlich, passende Unit-Tests zu schreiben. In der Praxis werden Unit-Tests häufig erst nach dem eigentlichen Code geschrieben, und dann auch nur je nach verfügbarer Zeit. Zudem arbeiten oft mehrere Entwickler an einer Komponente; die Entwicklung einheitlicher Unit-Tests ist dann besonders schwierig – manche Code-Abschnitte werden mehrfach getestet, andere vielleicht gar nicht.

Für einen Projektleiter oder Architekten ist daher schwer zu entscheiden, wie verlässlich die vorhandenen Unit-Tests sind. Und auch das Projektmanagement hätte gerne eine Aussage darüber, welchen Wert diese Tests haben, deren Entwicklung zunächst viel Geld kostet, scheinbar ohne dass sie die Funktionalität des zu entwickelnden Systems voranbringen.

Somit stellt sich die Frage, wie sich die Qualität von Unit-Tests sicherstellen lässt und wie man die erreichte Qualität sowohl intern wie extern transparent kommunizieren kann. Eine mögliche Antwort darauf ist die Ermittlung der Test-Abdeckung. Daher wird im Folgenden beschrieben, was es mit der Test-Abdeckung auf sich hat und welchen Nutzen sie stiften, aber auch welche Probleme sie verursachen kann.

Was ist Test-Abdeckung?

Mit der Test-Abdeckung lässt sich messen, welche Bereiche des Programmcodes eines Systems beim Ausführen von Tests durchlaufen werden und welche nicht. Die Test-Abdeckung als Metrik gibt an, wie groß der Anteil des durchlaufenen Codes am gesamten Programmcode ist. Anhand der Testabdeckung lässt sich zudem anschaulich im Detail ermitteln, welche Code-Abschnitte aufgerufen worden sind. Da sich die Test-Abdeckung kaum von Hand ermitteln lässt, sind dazu automatisierte Tests nötig, also in der Regel Unit-Tests.

Die virtuelle Maschine von Java bietet mehrere Ansatzpunkte, um die Test-Abdeckung zu messen. Ein verbreiteter Ansatz manipuliert den vom Compiler erstellten Bytecode und fügt Anweisungen hinzu, die, wenn sie aufgerufen werden, die jeweilige Operation aufzeichnen. Manche Tools verwenden die Profiling-Schnittstelle, um über den Ablauf des Systems informiert zu werden. In jedem Fall werden nach dem Ende der Tests alle aufgezeichneten Vorgänge statistisch zusammengefasst.

Es lassen sich verschiedene Arten der Test-Abdeckung unterscheiden, von denen vor allem zwei Varianten verbreitet sind: die Messung der durchlaufenen Programmanweisungen sowie die Messung der durchlaufenen Alternativen bei Entscheidungszweigen (z.B. bei Programmsprüngen oder Schleifen). Bei ersterem wird untersucht, wie viele der Programmanweisungen eines Systems ausgeführt werden. Für eine bessere Anschaulichkeit lassen sich diese Anweisungen auf unterschiedlichen Ebenen zusammenfassen, z.B. auf Zeilen, Methoden, Klassen oder Paketen.

Bei der Messung der durchlaufenen Entscheidungszweige wird ermittelt, ob alle möglichen Alternativen durchlaufen worden sind. Bei einer `if`-Anweisung kann damit beispielsweise festgestellt werden, ob auch der `else`-Block aufgerufen wurde. Da bei einem

einzigen Aufruf einer Methode selten alle Blöcke einer `if`-Anweisung durchlaufen werden, werden alle Aufrufe statistisch zusammengefasst. Es geht also nicht darum, ob die verschiedenen Alternativen während der Ausführung eines Tests durchlaufen wurden, sondern während der Ausführung aller Tests.

Welchen Nutzen hat die Test-Abdeckung?

Die Ermittlung der Test-Abdeckung hilft in einem Projekt vor allem zwei Zielgruppen: Entwicklern und Projektleitern. Entwickler können die Test-Abdeckung dazu nutzen, Lücken in ihren Unit-Tests zu finden und durch neue Tests zu schließen. Dazu reicht in der Regel keine einfache Metrik aus, sondern die Entwickler müssen sich im Detail anschauen, welche Abschnitte des Programmcodes noch nicht durchlaufen worden sind, um maßgeschneiderte neue Tests zu erstellen.

Die Messung der Test-Abdeckung hilft aber nicht nur, Unit-Tests zu erstellen, sondern auch, vorhandenen Programmcode besser zu verstehen. Wenn es beispielsweise schwer zu erkennen ist, welcher Programmcode durch den Aufruf einer Methode durchlaufen wird, hilft es, die Test-Abdeckung für genau einen Unit-Test zu starten, der eben diese Methode aufruft. Anschließend wird ersichtlich, welche Teile des Systems durch den Methodenaufruf betroffen waren.

Zudem macht die Messung der Test-Abdeckung die vorhandenen Tests transparenter. Sie bietet damit denjenigen Entwicklern, die Unit-Tests genau nehmen, die Möglichkeit, ihren eventuell höheren Entwicklungsaufwand damit zu rechtfertigen.

Projektleiter wiederum können anhand der Metrik der Test-Abdeckung überprüfbare Vorgaben über die Erstellung der Unit-Tests aufstellen. Die reine Anzahl von Tests alleine ist selten ein gutes Qualitätsmerkmal. Die Vorgabe hingegen, bestimmte Code-Bereiche zu einem bestimmten Prozentsatz abzudecken, lässt bessere Rückschlüsse über die erstellten Tests zu.

Wichtiger noch als der absolute Prozentsatz der Test-Abdeckung ist seine Varianz im Laufe der Zeit. Wenn die Metrik über einen längeren Zeitraum ermittelt und aufgezeichnet wird, lassen sich an ihr Trends ablesen. Beispielsweise, ob auch in Zeiten eines großen Projektdrucks weiterhin auf Unit-Tests geachtet wird, oder ob diese Tests schrittweise ausgeschaltet werden, zum Beispiel, weil die Zeit fehlt, Änderungen an der Code-Basis nachzuziehen. Andersherum lässt sich damit natürlich auch der Fortschritt in der Abdeckung des Codes durch Tests nachvollziehen, falls diese Tests nicht von Beginn an geschrieben, sondern erst später eingeführt worden sind.

Weiterhin können Projektleiter die Test-Abdeckung als Kommunikationsmittel gegenüber dem Management verwenden. Zuerst können Projektleiter damit die Notwendigkeit für mehr Zeit und Budget für Unit-Tests vermitteln; später können sie anhand des Verlaufs der Metrik den Erfolg der Maßnahmen dokumentieren.

Welche Probleme bringt die Messung der Test-Abdeckung mit sich?

Wie bei praktisch jedem Mittel zur automatisierten Bestimmung von Code-Qualität ist auch die Test-Abdeckung mit Vorsicht zu genießen. Es ist verführerisch, die Qualität der Tests auf eine einzige, objektiv messbare Zahl zu reduzieren, zumal sie sich für Management-Präsentationen über den Projektfortschritt anbietet. Leider sagt die Test-Abdeckung nur etwas darüber aus, wie viel Programmcode durch Tests durchlaufen wurde, nicht jedoch, wie sinnvoll diese Tests sind.

Beispielsweise mag es sinnvoll sein, generierten Programmcode von Unit-Tests auszunehmen. Oder aufgrund von Zeitdruck soll nur die Geschäftslogik getestet werden, nicht jedoch die Logik zur Steuerung der Benutzungsoberfläche. Zudem können Unit-Tests einen großen Bereich des Codes durchlaufen, ohne aber die Ergebnisse der einzelnen Aufrufe des Systems auf ihre fachliche Korrektheit hin zu prüfen. Die Unit-Tests haben in einem solchen Fall nur wenig Aussagekraft. Die Gefahr liegt also vor allem darin, die Ergebnisse der Test-Abdeckung als alleinigen Maßstab anzusehen.

Zudem nimmt ab einem gewissen Wert der Nutzen einer immer höheren Test-Abdeckung im Verhältnis zum Aufwand stark ab. Spätestens, wenn versucht wird, bei allen Vorkommnisse von `if (log.isDebugEnabled()) log.debug(„...“);` beide Verzweigungsmöglichkeiten durch Tests abzudecken, wird über das Ziel hinausgeschossen.

Die Transparenz, die durch die Ermittlung der Test-Abdeckung erreicht wird, kann bei manchen Entwicklern zudem ein ungutes Gefühl hinterlassen. Auch eine geringe Test-Abdeckung ist vielleicht gut genug, wenn alle wichtigen Bereiche gründlich getestet werden. Diese Entscheidungskompetenz wird durch eine Metrik wie die der Test-Abdeckung hinterfragt. Daher sollten keine persönlichen Ziele an die Test-Abdeckung gekoppelt werden.

Ein konkretes Beispiel

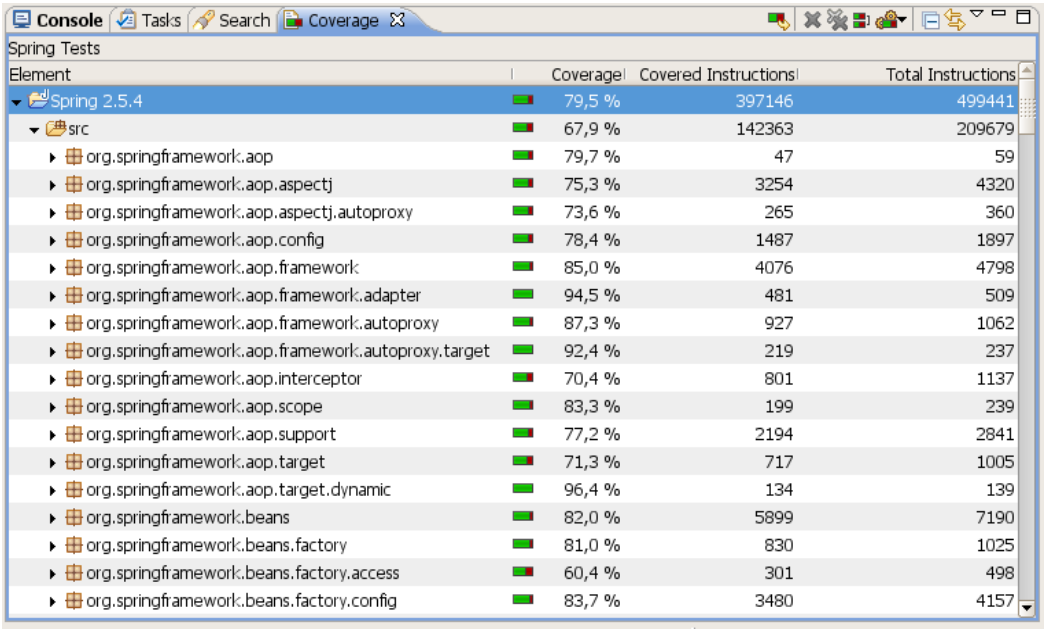
Im Java-Umfeld gibt es erfreulicherweise mehrere frei verfügbare Tools, mit denen sich die Test-Abdeckung ermitteln lässt. Jedes dieser Tools hat seine eigenen Stärken und Schwächen, sodass es sich lohnt, mehrere Tools auszuprobieren. Beispielhaft sei das Tool *EclEmma* herausgegriffen, dessen Stärke als Eclipse-Plug-In in der Integration in die Eclipse-Oberfläche liegt.[1]

Die Installation von EclEmma geschieht am Leichtesten über die Update-Mechanismen von Eclipse. Anschließend steht ein neuer Launch-Modus für die Messung der Test-Abdeckung zur Verfügung. Über diesen Modus lassen sich beliebige Launch-Konfigurationen starten. Man kann somit nicht nur die Abdeckung von Unit-Tests messen, sondern die Abdeckung eines beliebigen ausführbaren Programms. Während ein Programm ausgeführt wird, zeichnet EclEmma im Hintergrund den durchlaufenen Code auf und zeigt die Ergebnisse am Ende übersichtlich an.

Der aktuelle Quellcode des Spring-Frameworks dient als Beispiel dazu, wie die Ergebnisse von EclEmma interpretiert werden können. Dazu wird zunächst mit dem Quellcode von Spring und seinen Tests ein Eclipse-Projekt angelegt. Eine eigene Launch-Konfiguration startet dann alle Unit-Tests von Spring. Nach dem Abschluss der Tests werden die Ergebnisse in einer eigenen View anzeigt (s. Abb. 1).

In dieser Darstellung sieht man zunächst den Anteil der von den Tests durchlaufenen Anweisungen im Verhältnis zu allen Anweisungen im Programmcode, und zwar gruppiert auf Ebene der Java-Pakete. Man kann beispielsweise erkennen, dass der vollständige Quellcode von Spring (im Verzeichnis `src`) zu gut zwei Dritteln durch die Tests abgedeckt wird, die Abdeckung der AOP-Pakete jedoch im Schnitt bei etwa 80% liegt.

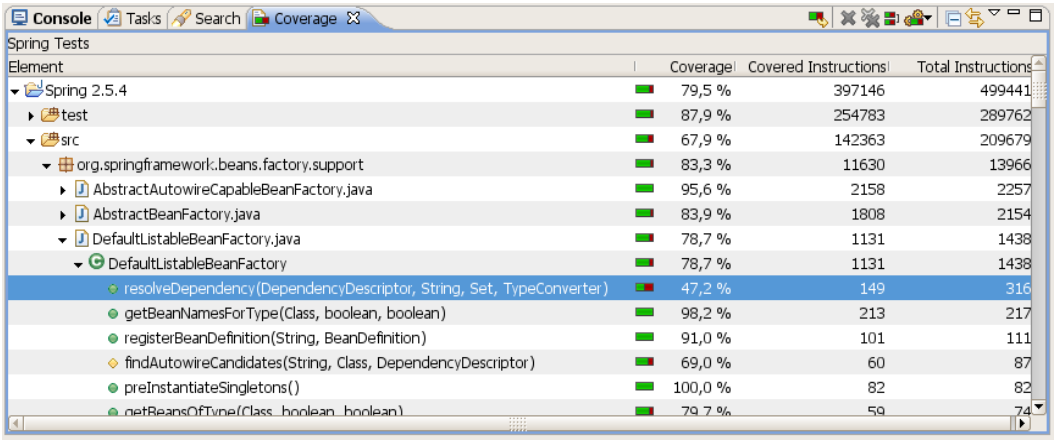
1 Die URL für dieses und andere Tools finden Sie am Ende des Artikels.



Element	Coverage	Covered Instructions	Total Instructions
Spring 2.5.4	79,5 %	397146	499441
src	67,9 %	142363	209679
org.springframework.aop	79,7 %	47	59
org.springframework.aop.aspectj	75,3 %	3254	4320
org.springframework.aop.aspectj.autoproxy	73,6 %	265	360
org.springframework.aop.config	78,4 %	1487	1897
org.springframework.aop.framework	85,0 %	4076	4798
org.springframework.aop.framework.adapter	94,5 %	481	509
org.springframework.aop.framework.autoproxy	87,3 %	927	1062
org.springframework.aop.framework.autoproxy.target	92,4 %	219	237
org.springframework.aop.interceptor	70,4 %	801	1137
org.springframework.aop.scope	83,3 %	199	239
org.springframework.aop.support	77,2 %	2194	2841
org.springframework.aop.target	71,3 %	717	1005
org.springframework.aop.target.dynamic	96,4 %	134	139
org.springframework.beans	82,0 %	5899	7190
org.springframework.beans.factory	81,0 %	830	1025
org.springframework.beans.factory.access	60,4 %	301	498
org.springframework.beans.factory.config	83,7 %	3480	4157

Abb. 1: Darstellung der Test-Abdeckung des Spring-Quellcodes

Ausgehend von der Test-Abdeckung vollständiger Pakete macht es natürlich Sinn, tiefer in den Code einzusteigen, und die Test-Abdeckung einzelner Klassen oder gar Methoden zu untersuchen. Abbildung 2 zeigt beispielsweise die Test-Abdeckung einzelner Methoden der Klasse `DefaultListableBeanFactory`.



Element	Coverage	Covered Instructions	Total Instructions
Spring 2.5.4	79,5 %	397146	499441
test	87,9 %	254783	289762
src	67,9 %	142363	209679
org.springframework.beans.factory.support	83,3 %	11630	13966
AbstractAutowireCapableBeanFactory.java	95,6 %	2158	2257
AbstractBeanFactory.java	83,9 %	1808	2154
DefaultListableBeanFactory.java	78,7 %	1131	1438
DefaultListableBeanFactory	78,7 %	1131	1438
resolveDependency(DependencyDescriptor, String, Set, TypeConverter)	47,2 %	149	316
getBeanNamesForType(Class, boolean, boolean)	98,2 %	213	217
registerBeanDefinition(String, BeanDefinition)	91,0 %	101	111
findAutowireCandidates(String, Class, DependencyDescriptor)	69,0 %	60	87
preInstantiateSingletons()	100,0 %	82	82
getBeansOfType(Class, boolean, boolean)	79,7 %	59	74

Abb. 2: Test-Abdeckung einzelner Klassen und Methoden

Wenn man sich die absoluten Werte genauer anschaut, ist die in diesen Beispielen verwendete Maßeinheit Instruktionen vielleicht nicht ganz so verständlich. Daher bietet EclEmma die Möglichkeit an, auf andere Maßeinheiten umzuschalten, und zwar auf Zeilen, Code-Blöcke, Methoden und Klassen. Bei Zeilen versucht das Tool zu erkennen, ob mindestens eine Anweisung einer Codezeile durchlaufen worden ist. Als Code-Block gelten alle Anweisungen zwischen zwei Programmsprüngen, die zwangsläufig aufeinander folgend ausgeführt werden. Und die Einstellung Methoden bzw. Klassen bietet einen schnellen Überblick, ob bestimmte Methoden oder Klassen überhaupt durchlaufen wurden.

Durch diese tabellarische Übersicht ist es also relativ einfach, Code-Abschnitt zu finden, die zu einem relativ niedrigen Teil von Tests abgedeckt sind. Um zu verstehen, warum das so ist, muss man den fraglichen Code in einen Editor laden. Dort werden die durchlaufe-

nen bzw. die nicht oder nur teilweise durchlaufenen Blöcke farblich hervorgehoben (s. Abb. 3).

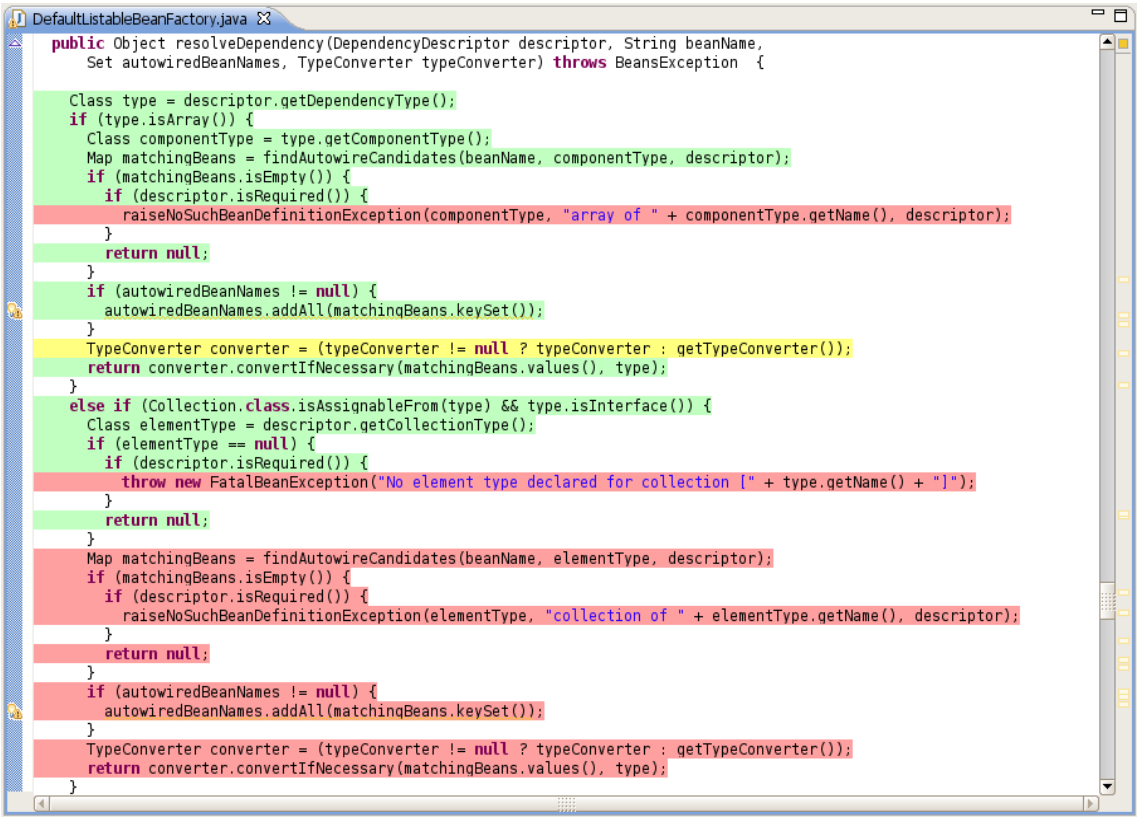


Abb. 3: Ergebnisse der Test-Abdeckung, im Quellcode markiert

Alle Code-Abschnitte, die von irgendeinem Test durchlaufen wurden, sind grün hinterlegt, alle nicht durchlaufenen rot. Man kann somit erkennen, dass beispielsweise im oberen Abschnitt die Methode `raiseNoSuchBeanDefinitionException` durch keinen Test aufgerufen worden ist. Ebenso, und das wiegt eventuell schwerwiegender, wurde der ganze untere Abschnitt der Methode nicht durchlaufen.

Es findet sich aber auch eine in gelb markierte Zeile. Gelb bedeutet hier, dass diese Zeile teilweise durchlaufen worden ist, aber nicht vollständig. In diesem Fall heißt es, dass die Abfrage `typeConverter != null` nur eine der beiden Möglichkeiten ergeben hat, also `true` oder `false`, und somit nur eine der beiden möglichen Zuweisungen stattgefunden hat.

Mit Hilfe dieser farblich markierten Darstellung des Programmcodes lassen sich mit ein wenig Übung sehr schnell diejenigen Programmstellen aufspüren, die bisher bei den Tests zu kurz gekommen sind. In den meisten Fällen finden sich aber auch viele Code-Abschnitte, für die Tests vielleicht weniger wichtig sind. Ob eine Test-Abdeckung von 100% erstrebenswert ist, ist eine Frage, die diskutiert wird (s. z.B. [6]), die im normalen Projektalltag aber nur eine geringe Rolle spielt.

Fazit

Die Ermittlung der Test-Abdeckung kann einen wertvollen Beitrag für die Qualitätssicherung eines Projekts leisten. Sie nutzt zum einen den Entwicklern, die ihre Tests damit ziel-

gerichtet schreiben können, und zum anderen der Projektleitung, die damit transparent prüfen kann, ob Vorgaben eingehalten werden.

Der Kennwert der Test-Abdeckung alleine ist jedoch nur bedingt aussagekräftig. Erst ein Review des durch die Tests durchlaufenen Programmcodes zeigt ein klareres Bild der Qualität der vorhandenen Tests. Ihren besonderen Nutzen zeigt die Messung der Test-Abdeckung vor allem dann, wenn sie in eine Build-Automatisierung eingegliedert ist und somit ein Vergleich der Werte im Verlauf der Zeit möglich ist.

Referenzen

[1] *EclEmma*, http://eclemma.org

[2] *Cobertura*, http://cobertura.sourceforge.net

[3] *CodeCover*, http://codecover.org

[4] *Coverlipse*, http://coverlipse.sourceforge.net

[5] Andrew Glover: *In pursuit of code quality: Don't be fooled by the coverage report*, http://www-128.ibm.com/developerworks/java/library/j-cq01316/?ca=dnw-704

[6] Toni Obermeit: *Lessons learned on the road to 100% code coverage*, http://homepage.mac.com/hey.you/lessons.html