Immer kürzere Testphasen? Mit *Ticket Coverage* verhindern, dass wichtige Features ungetestet bleiben

Elmar Jürgens
CQSE GmbH
juergens@cqse.eu

Dennis Pagano
CQSE GmbH
pagano@cqse.eu

***Zusammenfassung*—In vielen Systemen werden die Release-Zyklen immer kürzer. Daher steht auch immer weniger Zeit für dedizierte Testphasen zur Verfügung. Viele Teams führen deshalb Tests parallel zur Entwicklung durch. Oft finden Entwicklung und Test dabei auf parallelen Branches und in verschiedenen Umgebungen statt.**

Dadurch wird es immer schwieriger im Blick zu behalten, welche Tickets (z.B. User Stories, Change Requests, Bug Reports, etc.) wie gründlich getestet wurden. Und nicht zuletzt, wo auf Grund von nachfolgenden Code-Änderungen nochmal getestet werden müsste. Dadurch steigt die Gefahr, dass wichtige Funktionalität ungetestet in Produktion gelangt.

In diesem Paper stellen wir *Ticket Coverage* als Maß von Test Coverage auf der Ebene von Tickets vor. Dadurch kann pro Ticket ermittelt werden, welche Bereiche nicht (oder nicht ausreichend) getestet wurden. Wir stellen die Ergebnisse von zwei empirischen Studien vor, in denen Ticket Coverage bei manuellen Systemtests und bei Entwicklertests zum Einsatz kam. In beiden Studien konnten dadurch relevante Testlücken identifiziert werden.

Qualitätssicherung von Tests

Bei großen Systemen stehen einem als Tester oder Testmanager selten genug Ressourcen zur Verfügung, um die gesamte Funktionalität vollständig zu testen. In der Praxis müssen wir uns in jeder Testphase daher notwendigerweise auf einen Ausschnitt aller möglichen Tests beschränken.

Die Frage, wie man diese auszuführenden Tests möglichst sinnvoll auswählt, beschäftigt seit Jahrzehnten ein eigenes Forschungsgebiet. Ein zentrales Ergebnis der Arbeiten der letzten Jahre ist, dass typischerweise in denjenigen Bereichen die meisten Fehler auftreten, in denen in letzter Zeit (z.B. seit dem letzten Release) am meisten geändert wurde [2], [4]. Daher sollte im Rahmen der Qualitätssicherung der Testaktivitäten überprüft werden, ob alle relevanten Änderungen auch getestet wurden. Diese Aufgabe wird erfahrungsgemäß um so wichtiger, je größer Systeme (und damit je unübersichtlicher Änderungen und Testaktivitäten) werden.

Um Tester und Testmanager bei dieser Überprüfung zu unterstützen, haben wir in den letzten Jahren die Test-Gap-Analyse entwickelt, die ermittelt, welche Änderungen noch ungetestet sind. Wir haben Test-Gap-Analyse u.a. auf dem QS-Tag 2016 vorgestellt [3]. Test-Gap-Analyse wird inzwischen von vielen Firmen eingesetzt.

Ein typisches Ergebnis von Test-Gap-Analyse ist in Abbildung 1 dargestellt. Jedes weiß umrandete Rechteck beschreibt eine Komponente im *System under Test*, jedes darin enthaltene kleinere schwarz umrandete Rechteck eine Methode. Der Flächeninhalt der Rechtecke korrespondiert mit der Größe der dazugehörigen Komponenten/Methoden in Lines of Code. Methoden, die sich seit dem letzten Release nicht verändert haben, werden grau dargestellt, veränderte farbig. Nur wenn die veränderten Methoden grün dargestellt werden, sind sie im Test durchlaufen worden. Die roten und orangenen Bereiche zeigen Methoden, die seit dem letzten Release neu entwickelt oder verändert, aber noch nicht getestet wurden.

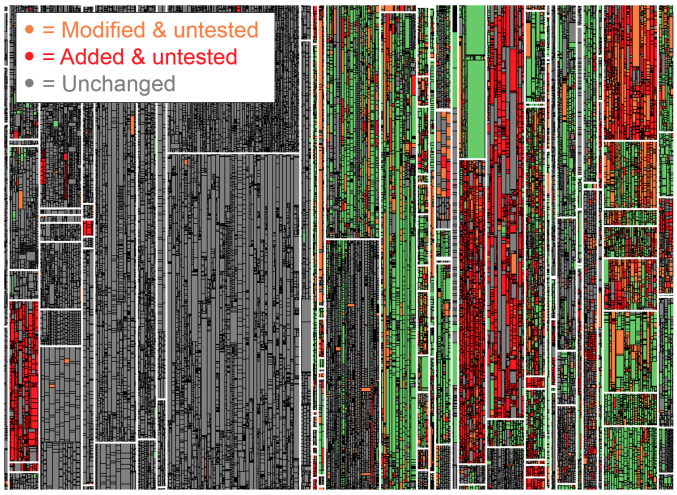

Abbildung 1. Getestete und ungetestete Änderungen

Anforderungen durch kürzere Release-Zyklen

Test-Gap-Analyse betrachtet alle Änderungen und alle Testaktivitäten, die seit einem Referenzzeitpunkt (z.B. dem letzten Release) durchgeführt wurden. Diese Betrachtung bewährt sich vor allem in Projekten, die (wie in Abbildung 2 dargestellt) vor einem Release eine ausgedehnte Testphase durchführen, in der das gesamte Release getestet wird.

Agile Entwicklungsmethoden drängen seit Jahrzehnten auf kürzere Releasezyklen, um neue Funktionalität schneller zum Anwender zu bringen. In den letzten Jahren hat, in unserer

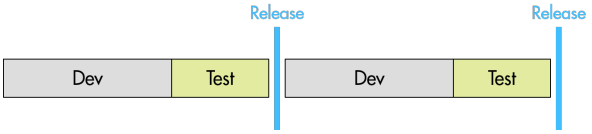

Abbildung 2. Aufteilung von Entwicklungsphasen (Dev) und Testphasen (Test) bei wenigen großen Releases pro Jahr.

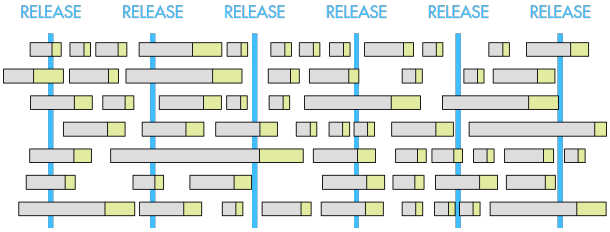

Abbildung 3. Aufteilung von Entwicklungs- und Testphasen bei vielen kleinen Releases pro Jahr.

Beobachtung, dieser Trend in den meisten Entwicklungsteams die Releasezyklen stark verkürzt. Mit der Verkürzung der Releasezyklen geht die Verkürzung der Testphasen einher.

Oft ist es nicht mehr möglich, vor einem Release eine ausgedehnte Testphase durchzuführen. Tests müssen stattdessen iterationsbegleitend durchgeführt werden, wie in Abbildung 3 dargestellt[1].

Tickets im Fokus der Testplanung

Wir beobachten, dass in vielen Teams, um Tests iterationsbegleitend steuern zu können, immer stärker einzelne *Tickets* im Fokus der Testplanung stehen. Testern werden dabei dedizierte Tickets zugewiesen, die sie zeitnah nach Abschluss der zugehörigen Entwicklungsarbeiten testen. Dadurch gelingt es, Tests verschränkt zur Entwicklung auszuführen, so dass dedizierte Testphasen entfallen oder kürzer ausfallen können.

Unter *Ticket* verstehen wir hierbei die Beschreibung einer geplanten Änderung am System, die in einem Change Management System[2] verwaltet wird. Je nach Art und Quelle der Änderung wird in Teams oft zwischen unterschiedlichen Ticket-Klassen unterschieden, wie bspw. User Stories, Change Requests, Bug Reports oder allgemein Issues. In diesem Paper beziehen wir all diese Ticket-Klassen im Begriff *Ticket* mit ein.

Problemstellung

Wir haben in den letzten Jahren mehrere Teams begleitet, die derartig vorgehen. Dabei haben wir die Beobachtung gemacht, dass es für Tester und Test Manager sehr schwierig ist zu beurteilen, wie gründlich Tickets im Rahmen der ausgeführten Tests getestet wurden:

[1]Um Tests iterationsbegleitend durchzuführen, können prinzipiell die unterschiedlichsten Test-Ansätze eingesetzt werden. Von automatisierten (z.B. Keyword-Driven), klassischen manuellen Tests die iterationsbegleitend ausgeführt werden bis hin zu explorativen Tests auf Basis der Beschreibung der User Story. Die Vor- und Nachteile der jeweiligen Test-Ansätze für iterationsbegleitendes Testen sind außerhalb des Fokus dieses Papers.

[2]z.B. Jira, Bugzilla, Redmine, TFS, ...

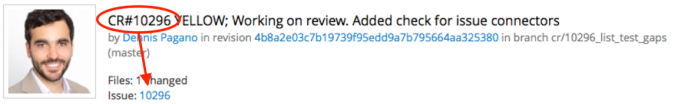

Abbildung 4. Commit, bei dem die ID des Tickets angegeben wurde (10296), auf das sich die durchgeführten Änderungen beziehen.

- Als Test-Input werden oft die Beschreibungen der Tickets verwendet. Die sind allerdings oft, gerade im Vergleich zu strukturierten Testfällen, deutlich informationsärmer, insbesondere in Hinsicht auf Sonderfälle.
- Da immer mehr Test-Arten (Unit-Tests, skriptierte UI-Tests, manuelle Tests, Systemtests, explorative Tests) und Test-Stufen (Smoke-Tests, Akzeptanztests, ...) zum Einsatz kommen, wird es immer schwieriger den Überblick zu behalten, was eigentlich wo getestet, und vor allem wie gründlich getestet wird.
- Die Entwicklung findet immer häufiger auf parallelen Branches statt. Oft fließen durch den Merge eines Feature Branches sehr plötzlich sehr große Mengen an Änderungen in den Release Branch ein. Da häufig unterschiedliche Branches in unterschiedlichen Teststufen zur Ausführung kommen, wird es immer schwieriger einen Überblick darüber zu behalten, welche Änderungen in welchen Branches schon getestet wurden.

Durch diese Faktoren steigt die Gefahr, dass auch zentrale, wichtige Tickets unzureichend getestet werden und zu Feldfehlern in kritischer Funktionalität führen. Testmanager und Tester brauchen daher ein Analysewerkzeug, um während der iterationsbegleitenden Tests einfach herausfinden zu können, welche Tickets nicht oder nicht ausreichend getestet wurden.

Lösungsansatz

Als Lösungsansatz schlagen wir in diesem Paper *Ticket Coverage* vor. Ticket Coverage drückt aus, welcher Anteil des Codes, der im Zuge der Implementierung eines Tickets angefasst (neu hinzugefügt oder geändert) wurde, im Test zur Ausführung kam.

Ticket-Coverage wird auf Basis von Test-Gap-Analyse berechnet. Im Unterschied zur Test-Gap-Analyse werden Informationen jedoch nicht für das gesamte System, sondern für einzelne Tickets ausgewertet und dargestellt.

Ticket Coverage wird in folgenden Schritten berechnet:

1) Im ersten Schritt wird ermittelt, welche Methoden im Code im Rahmen eines Tickets angefasst wurden. Wie in Abbildung 4 dargestellt, geben Entwickler bei Änderungen im Code an, auf welches Ticket sich eine Änderung bezieht. Durch Analyse der Daten im Ticket-Management-System und der Versionshistorie kann dadurch für jedes Ticket die Menge der Methoden erhoben werden, die im Rahmen der Umsetzung des Tickets angefasst wurden.
2) Im zweiten Schritt werden Test-Coverage-Daten aus verschiedenen Testläufen erhoben, um zu ermitteln, welche Methoden im Test zur Ausführung kamen.

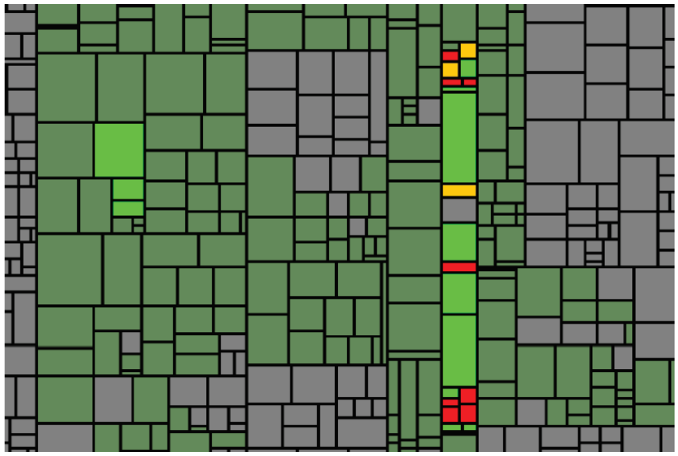

Abbildung 5. Ticket Coverage für ein Ticket: 12 geänderte Methoden wurden getestet (hellgrün dargestellt), 11 geänderte Methoden nicht (rot und orange dargestellt). Außerdem wurden im Test eine Reihe von Methoden durchlaufen, die nicht zum Ticket gehören (dunkelgrün dargestellt). Graue Methoden repräsentieren Code, der weder verändert, noch ausgeführt wurde.

3) Im dritten Schritt wird für jedes Ticket ermittelt, welche der zum Ticket zugehörigen Methoden im Test zur Ausführung kamen.

Ticket Coverage kann in einer Treemap visualisiert werden, wie in Abbildung 5 dargestellt. Ticket Coverage lässt sich auch als Anteil der getesteten Methoden an allen im Kontext des Ticket bearbeiteten Methoden ausdrücken. Im Beispiel aus Abbildung 5 ergibt sich daher eine Ticket Coverage von 52% (12 getestete Methoden von 23 angefassten Methoden).

Grenzen von Ticket Coverage

Wie jede statische oder dynamische Analysetechnik hat auch Ticket Coverage Grenzen. Die Kenntnis dieser Grenzen ist entscheidend, um Analyseergebnisse korrekt zu interpretieren. Dabei sollten die folgenden beiden Punkte Beachtung finden:

Coverage≠Getestet: Test Coverage misst, welcher Code im Test *ausgeführt* wurde, nicht wie gründlich er tatsächlich *getestet* wurde. Da Test Coverage auch für die Berechnung der Ticket Coverage herangezogen wird, gilt das auch hier. Konkret bedeutet 50% Ticket Coverage, dass jede zweite Methode des Tickets im Test zur Ausführung kam. Die Ticket Coverage erlaubt keine Aussage darüber, wie gründlich diese Methoden getestet wurden.

Seiteneffekte: Während sich einige Methoden eindeutig zu Tickets zuordnen lassen, werden andere Methoden im Kontext der Implementierung mehrerer Tickets angefasst. Wenn eine solche Methode beim Test eines Tickets durchlaufen wird, gilt sie ggf. auch für andere Tickets als getestet. Es ist aber möglich, dass der Test anderer Tickets Fehler in dieser Methode aufdecken würde. Selbst eine Ticket Coverage von 100% gibt daher keine Garantie, dass im betroffenen Code keine Fehler mehr enthalten sind.

Für beide Fälle gilt allerdings: Wird eine Methode als ungetestet erkannt, haben für sie überhaupt keine Tests stattgefunden. Keiner der möglicherweise enthaltenen Fehler kann daher gefunden worden sein. Wir sehen Ticket Coverage daher als Werkzeug, um Lücken im Test zu erkennen, nicht als Werkzeug um ohne weitere Analyse zu entscheiden, wann genug getestet wurde.

Was bringt Ticket Coverage in der Praxis?

Um besser einschätzen zu können, wie gut Ticket Coverage in der Praxis funktioniert und wo ihre Grenzen liegen, haben wir eine Reihe von wissenschaftlichen Studien [1], [5] durchgeführt.

Als Studienobjekt haben wir das von uns entwickelte System *Teamscale*[3] gewählt, weil wir hierfür die Ergebnisse der Studien besser beurteilen können, als für fremden Code. Teamscale ist eine inkrementelle Software-Qualitätsanalysesuite, die von vielen namhaften Firmen eingesetzt wird. Der Quelltext umfasste zum Zeitpunkt der Studien ca. 650.000 Zeilen Java- bzw. JavaScript-Code.

Studie: Finden wir relevante Testlücken im strukturierten Test?

Nicht jede Testlücke ist automatisch relevant und muss geschlossen werden. Ist etwa der zugehörige Code (noch) nicht erreichbar oder sind die betroffenen Codestellen trivial, kann es je nach Kontext durchaus sinnvoll sein, die Testlücken nicht zu schließen und stattdessen die Ressourcen anderweitig zu verwenden. Die zentrale Frage unserer ersten Studie war daher, ob wir mit Hilfe von Ticket Coverage relevante Testlücken in einem typischen Setting mit strukturierten manuellen Tests finden.

Um diese Frage zu beantworten, haben wir zunächst zufällig 54 Tickets ausgewählt, die in den letzten 20 Monaten bearbeitet wurden. Auf Basis der Ticketbeschreibung haben wir anschließend je eine strukturierte, manuelle Testfallbeschreibung erstellt und diese vom jeweiligen Entwickler des Tickets validieren lassen. Dann haben wir diesen Testfall unter Zuhilfenahme eines Profilers ausgeführt, der die Code Coverage aufgezeichnet hat. Abschließend haben wir die Ticket Coverage wie oben beschrieben berechnet. Eine im Zuge des Tickets geänderte Methode haben wir dann als Testlücke gewertet, wenn diese bei der Ausführung des Testfalls überhaupt nicht durchlaufen wurde.

Nach der Erhebung der Ticket Coverage haben wir die Entwickler mit den gefundenen Testlücken konfrontiert und sie gebeten, diese hinsichtlich ihrer Relevanz zu beurteilen. Die Ergebnisse sind in Tabelle I dargestellt. Von den insgesamt 110 Testlücken wurden 20 (18,2%) als kritisch eingestuft. In unserer Studie haben wir mit Ticket Coverage also tatsächlich relevante Testlücken gefunden.

Ein Großteil der als nicht relevant eingestuften Testlücken sind Refactorings, also Semantik-erhaltende Code-Änderungen. Viele dieser Refactorings können prinzipiell automatisch erkannt und von der Berechnung der Ticket Coverage ausgeschlossen werden, um die Analyse präziser zu machen. Weitere Details zu Forschungsfragen, Study Design und Ergebnissen hierzu finden sich in [5].

[3] www.teamscale.com

Tabelle I
BEWERTUNG DER TESTLÜCKEN DURCH DIE ENTWICKLER.

Bewertung	Erklärung	Häufigkeit	
kritisch: 20 (18,2%)	sollte ausgeführt werden	2	1,8%
	Testfall nicht ausreichend	18	16,4%
weniger kritisch: 56 (50,9%)	Refactoring oder nicht Ticket-relevant	54	49,1%
	Exception	1	0,9%
	überschriebene Methode	1	0,9%
nicht testbar: 3 (2,7%)	IDE plugin Code	1	0,9%
	Methode für Unit-Tests	2	1,8%
nicht relevant: 28 (25,5%)	einfacher Getter	12	10,9%
	triviale Methode	12	10,9%
	toString-Methode	4	3,6%
	keine Antwort	3	2,7%
	Σ	110	100%

Studie: Finden wir relevante Testlücken im explorativen Test?

Mit der ersten Studie haben wir eine Blackbox-Sicht auf das zu testende System eingenommen, wie sie ein Tester hat, der strukturierte manuelle Testfälle durchführt. In der zweiten Studie haben wir explorative Tests betrachtet, für die es typischerweise keine detaillierte Testfallbeschreibung gibt. Stattdessen wird bei explorativen Tests versucht, die entwickelte Funktionalität z.B. mit Hilfe der Beschreibung im zu testenden Ticket zu prüfen. Ein häufiges Problem hierbei ist jedoch, dass oft Sonderfälle durch den Entwickler umgesetzt wurden, von denen der Tester nichts weiß, da sie in der Beschreibung des Tickets nicht aufgeführt sind. In der zweiten Studie [1] haben wir daher untersucht, ob Ticket Coverage auch im explorativen Test relevante Testlücken aufzeigt.

Bei der Entwicklung von Teamscale kommt ein Peer-Review-Verfahren zum Einsatz. Der Reviewer führt dabei oft zunächst einen explorativen Test der umgesetzten User Story durch, um das beobachtete Verhalten zu bewerten. Uns hat in unserer Studie interessiert, ob wir mit Hilfe der Ticket Coverage Testlücken in solchen explorativen Tests aufdecken können, die im Rahmen der Reviews durchgeführt werden. Hierfür haben wir 4 Tickets zur näheren Analyse ausgewählt. Wir haben dann die jeweiligen Reviewer gebeten, explorative Tests durchzuführen und dabei die Ticket Coverage berechnet.

Im Vergleich zur ersten Studie haben wir hierbei Refactorings und triviale[4] Methoden automatisch ausgeschlossen. Außerdem haben wir Coverage zeilengenau berechnet (und nicht wie in der ersten Studie auf der Ebene von Methoden). Nach der Erhebung der Ticket Coverage haben wir dem jeweiligen Reviewer das Resultat des explorativen Tests, wie in Abbildung 6 gezeigt, präsentiert. Dargestellt ist pro Ticket eine Liste aller Methoden, die während der Entwicklung am Ticket verändert wurden oder neu hinzugekommen sind. Für jede Methode wird dargestellt, wie vollständig sie durch den explorativen Test abgedeckt wurde. Methoden, die als trivial erkannt wurden, sind ausgegraut. Diese Übersicht zeigt also aktuelle Testlücken für ein einzelnes Ticket feingranular auf.

[4]Triviale Methoden sind bspw. reine Getter und Setter. Die Kriterien zur Klassifikation von Methoden als *zu trivial für den Test* wurde im Rahmen der Studie durch die beteiligten Entwickler validiert. Details finden sich in [5].

Durch einen Klick auf eine einzelne Methode gelangt man zu einer weiteren Ansicht, auf der die Änderungen an der Methode zusammen mit der zeilengenauen Coverage zu sehen sind (vgl. Abbildung 7). Diese Darstellung erlaubt es, die Ursache der einzelnen Lücken zu identifizieren und somit zu beurteilen, ob es sich dabei um relevante Testlücken handelt.

Insgesamt waren 95 Methoden in den untersuchten Tickets geändert worden. Davon waren 30 nicht vollständig im Test durchlaufen worden, wiesen also einen Coverage-Wert von weniger als 100% auf. Bei der anschließenden Bewertung dieser Lücken durch die Reviewer wurden 23 Methoden (76,6%) als testenswert identifiziert. Die Analyse der Ticket Coverage führte also auch im Fall der explorativen Tests zur Aufdeckung von relevanten Testlücken.

Interpretation

Wir haben Ticket Coverage für zwei unterschiedliche Testarten im Rahmen der Entwicklungs- und Testprozesse des kommerziellen Qualitätsanalysewerkzeugs Teamscale eingesetzt. Zum einen im strukturierten, manuellen Test, wo wir die Sicht eines Testers eingenommen haben, zum anderen im explorativen Test, aus der Sicht eines Code Peer Reviewers.

In beiden Fällen sind wir zu ähnlichen Ergebnissen gekommen: Ticket Coverage hilft zu erkennen, welche wichtigen Tickets nicht ausreichend getestet werden. Der Fokus auf ein Ticket erlaubt es dabei, sich auf die Lücken zu konzentrieren, die zu kritischer Funktionalität gehören. Dies ermöglicht es, auch bei iterationsbegleitender Testdurchführung im Blick zu behalten, ob Änderungen an kritischer Fachlichkeit gründlich genug getestet wurden und rechtzeitig nachzusteuern.

Aufgrund der Ergebnisse unserer Studien halten wir es für wahrscheinlich, dass der Einsatz von Ticket Coverage auch bei anderen Testarten und Teststufen relevante Testlücken zu Tage fördert. Dies passt auch zu den Erfahrungen, die wir über unsere wissenschaftlichen Aktivitäten hinaus tagtäglich als Berater in Kundenprojekten gewinnen.

WIE KANN ICH TICKET COVERAGE EINSETZEN?

Die Messung von Ticket Coverage kann durch unsere inkrementelle Software-Qualitätsanalysesuite *Teamscale* durchgeführt werden. Teamscale unterstützt eine Vielzahl von Programmiersprachen (u.a. Java, C#, VB.NET, ABAP, Python, C/C++ uvm.), Versionskontrollsystemen (Git, SVN, TFS, SAP, uvm.) und Ticket-Systemen (Jira, TFS, Redmine, uvm.).

Wir haben diese Analysen in den letzten Jahren bei vielen Firmen aus verschiedensten Domänen eingeführt. Fast immer haben wir etwas andere Konstellationen aus Programmiersprachen, Technologien, Infrastruktur, Testwerkzeugen und Testansätzen vorgefunden. Inzwischen haben wir daher viel darüber gelernt, wie man diese Analysen an neue Umgebungen anpasst. Wir freuen uns auf einen persönlichen Austausch, um Ihren konkreten Fall zu betrachten.

Wir haben unter **www.testgap.io** weiterführende Materialien zu Test-Gap-Analyse allgemein und Ticket Coverage im Speziellen zusammengestellt, u.a. Forschungsarbeiten, Blog-Einträge und Werkzeugunterstützung. Darüber hinaus freuen

Affected methods (22)

Ticket Coverage: 86%

Class	Method	Line Coverage	Uncovered Lines	Change type
ProjectCreationService	retrieveProjectConfig	100%	0	changed
ProjectCreationService	fieldChangeRequiresReAnalysis	100%	0	added
ProjectService	elementUpdateQuery	100%	0	added
ProjectReanalysisService	process	100%	0	changed
ProjectCreator	refreshProject	100%	0	added
ProjectCreationService	nullOrToString	66%	1	added
ProjectCreationService	findConnectorByName	75%	1	added
ProjectCreationService	connectorRequiresReAnalysis	82%	3	added
ProjectCreationService	processPutRequest	84%	3	changed
ProjectCreationService	validateProjectConfiguration	80%	3	changed
ProjectCreationService	projectReAnalysisRequired	61%	5	added
ProjectCreationService	connectorsRequireReAnalysis	50%	7	added
ConfigOptionDescriptorBase	ConfigOptionDescriptorBase	100%	0	changed
NamingConventionConfiguration	NamingRegexOption	100%	0	changed
ProjectService	ProjectUpdateResult	100%	0	added

Abbildung 6. Detailansicht der Ticket Coverage, in der alle in der Umsetzung des Tickets bearbeiteten Methoden mit ihrer Coverage aufgelistet sind.

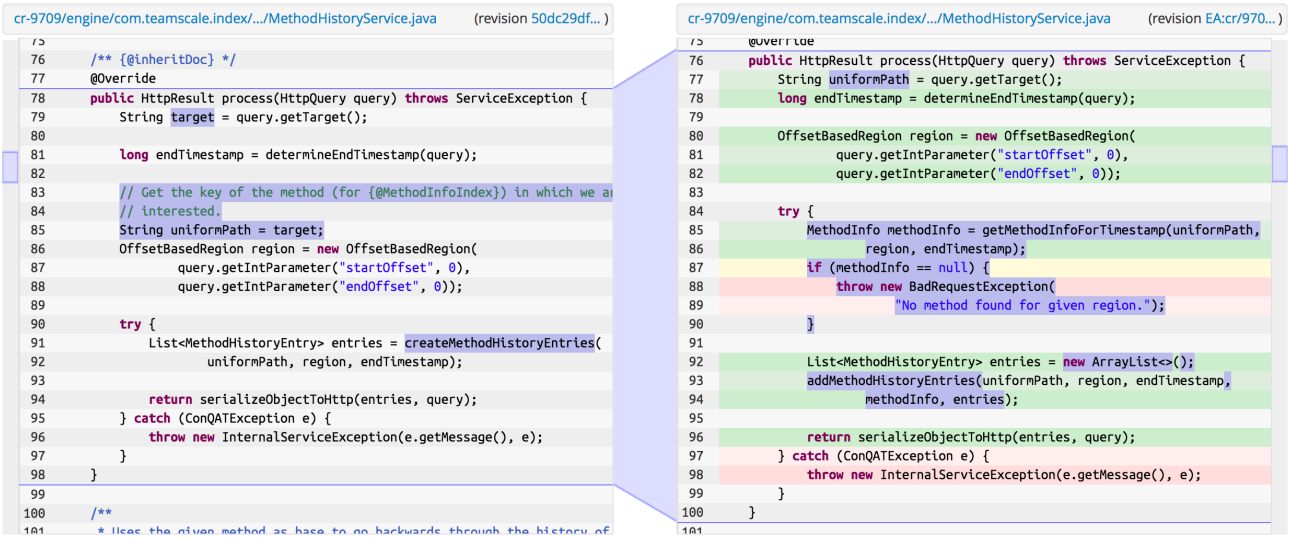

Abbildung 7. Detailansicht für eine einzelne Methode. Änderungen im Zuge des Tickets sind aufeinander abgebildet, Coverage ist farbig dargestellt.

wir uns auch per Email über Fragen und Feedback (auch kritisches) zum Artikel oder zu Ticket Coverage allgemein.

Danksagung

Dieses Paper baut auf Vorarbeiten und Ideen von Andreas Göb, Florian Dreier, Jakob Rott und Rainer Niedermayr auf, bei denen wir uns herzlich bedanken möchten.

Literatur

[1] Florian Dreier, Elmar Juergens, and Andreas Goeb. Test Accompanying Calculation of Test Gaps for Java Applications. Whitepaper, CQSE GmbH, 2017.

[2] Sebastian Eder, Benedikt Hauptmann, Maximilian Junker, Elmar Juergens, Rudolf Vaas, and Karl-Heinz Prommer. Did we test our changes? assessing alignment between tests and development in practice. In *Proceedings of the Eighth International Workshop on Automation of Software Test (AST'13)*, 2013.

[3] Elmar Juergens and Dennis Pagano. Haben wir das Richtige getestet? Erfahrungen mit Test-Gap-Analyse in der Praxis. In *QS-Tag*, 2016.

[4] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 284–292. IEEE, 2005.

[5] Jakob Rott, Rainer Niedermayr, Elmar Juergens, and Dennis Pagano. Ticket coverage: Putting test coverage into context. In *Proceedings of the 8th Workshop on Emerging Trends in Software Metrics (WETSoM'17)*, 2017.