OBJEKTspektrum

IT-Management und Software-Engineering

www.OBJEKTspektrum.de

- Haben wir das Richtige getestet?
- Immer kürzere Testphasen?

Die Autoren

Dr. Elmar Jürgens
Dr. Dennis Pagano

Erfahrungen mit Test-Gap-Analyse in der Praxis

Haben wir das Richtige getestet?

Bei langlebiger Software treten meist dort Fehler auf, wo viel geändert wurde. Testmanager versuchen daher, Änderungen besonders intensiv testen zu lassen. Unsere Studien zeigen jedoch, dass auch in gut strukturierten Testprozessen Änderungen oft ungewollt ungetestet bleiben. Test-Gap-Analyse zeigt ungetestete Änderungen auf, sodass diese noch rechtzeitig getestet werden können, und erlaubt somit eine wirksame Qualitätssicherung der Testprozesse. Nach einer Einführung in Test-Gap-Analyse stellen wir die Erfahrungen vor, die wir in den letzten Jahren im Einsatz bei Kunden und in der eigenen Entwicklung gesammelt haben, und zeigen, wie sich Test-Gap-Analyse in verschiedenen Testphasen einsetzen lässt.

In vielen Systemen machen manuelle Testfälle nach wie vor einen Großteil aller Tests aus. In einem großen System ist es alles andere als trivial, die manuellen Testfälle so auszuwählen, dass sie auch die Änderungen durchlaufen, die seit der letzten Testphase durchgeführt wurden und die vermutlich die meisten Fehler enthalten.

Wie gut werden Codeänderungen in der Praxis durch Tests wirklich abgedeckt?

Um besser zu verstehen, ob die Tests die Änderungen tatsächlich erreicht haben, haben wir eine wissenschaftliche Studie [Ede13] auf einem betrieblichen Informationssystem durchgeführt. Das untersuchte System umfasst ca. 340.000 Zeilen C#-Code. Wir haben die Studie über 14 Monate Entwicklung durchgeführt und dabei zwei aufeinanderfolgende Releases untersucht.

Durch statische Analysen haben wir ermittelt, welche Codebereiche für die beiden Releases neu entwickelt oder verändert worden sind. Für beide Releases wurden jeweils etwa 15 Prozent des Quelltextes modifiziert. Außerdem haben wir alle Testaktivitäten erhoben. Dafür haben wir die Testüberdeckung aller automatisierten und manuellen Tests über mehrere Monate aufgezeichnet.

Eine Auswertung der Kombination aus Änderungs- und Testdaten zeigte uns, dass *etwa die Hälfte der Änderungen ungetestet in Produktion gelangten* – obwohl der Testprozess sehr systematisch geplant und durchgeführt worden war.

Welche Folgen haben ungetestete Änderungen?

Um die Konsequenzen der ungetesteten Änderungen für die Anwender des Programms zu quantifizieren, haben wir retrospektiv alle Fehler analysiert, die in den Monaten nach den Releases aufgetreten sind. Dabei zeigte sich, dass die Fehlerwahrscheinlichkeit in geändertem, ungetestetem Code fünfmal höher war als in ungeändertem Code (und auch höher als in geändertem und getestetem Code).

Diese Studie führt uns vor Augen, dass Änderungen in der Praxis sehr häufig ungetestet in Produktion gelangen und dort den Großteil der Feldfehler verursachen. Sie zeigt uns damit aber auch einen konkreten Ansatzpunkt, um die Testqualität systematisch zu verbessern: wenn es uns gelingt, Änderungen zuverlässiger zu testen.

Warum rutscht Code durch den Test?

Die Menge an ungetestetem Code in Produktion hat uns offen gesagt überrascht, als wir diese Studie zum ersten Mal gemacht haben. Inzwischen haben wir vergleichbare Analysen in vielen Systemen, Programmiersprachen und Firmen durchgeführt und erhalten oft ein ähnliches Bild. Die Ursache für ungetestete Änderungen liegt jedoch – anders als man vielleicht vermuten könnte – nicht an mangelnder Disziplin oder am Einsatz der Tester; sondern vielmehr daran, dass es ohne geeignete Analysen sehr schwierig ist, geänderten Code in großen Systemen im Test zuverlässig zu erwischen.

Testmanager orientieren sich bei der Testauswahl häufig an den Änderungen, die im Issue-Tracker (Jira, TFS, Redmine, Bugzilla usw.) dokumentiert sind. Für fachlich motivierte Änderungen funktioniert das erfahrungsgemäß auch oft gut. Testfälle für manuelle Tests beschreiben typischerweise Interaktionssequenzen über die Nutzeroberfläche, um gewisse fachliche Abläufe zu testen. Enthält der Issue-Tracker Änderungen einer Fachlichkeit, werden die entsprechenden fachlichen Testfälle zur Durchführung ausgewählt.

Unsere Erfahrungen zeigen jedoch, dass Issue-Tracker aus zwei Gründen keine geeigneten Informationsquellen sind, um Änderungen lückenlos zu finden. Erstens gibt es häufig technisch motivierte Änderungen, wie beispielsweise Aufräumarbeiten oder Anpassungen an neue Versionen von Bibliotheken oder Schnittstellen zu Fremdsystemen. Bei derartigen Änderungen ist es für Tester nicht nachvollziehbar, welche fachlichen Testfälle durchgeführt werden müssten, um diese technischen Änderungen zu durchlaufen.

Zweitens, und noch gravierender ist jedoch, dass in vielen Fällen zentrale Änderungen am Issue-Tracker vorbeigehen, sei es aus Zeitdruck oder aus politischen Gründen. Dadurch sind die Daten im Issue-Tracker lückenhaft. Um Änderungen lückenlos zu finden, benötigen wir daher zuverlässige Informationen, welche Änderungen im Test durchlaufen wurden und welche nicht.

Was kann man machen, um die Lücken zu identifizieren?

Die Test-Gap-Analyse ist ein Ansatz, der statische und dynamische Analyseverfahren kombiniert, um geänderten, aber ungetesteten Code zu identifizieren. Sie umfasst folgende Schritte:

Eine *statische Analyse* vergleicht den aktuellen Stand des Quelltextes des *System under Test* mit dem Stand des letzten Releases, um neue und geänderte Codebereiche zu ermitteln. Dabei ist die Analyse intelligent genug, um unterschiedliche Arten von Änderungen voneinander zu unterscheiden. Refactorings, bei denen das Verhalten des Quelltextes nicht verändert wird (bspw. Änderung von Dokumentation, Umbenennungen von Methoden oder

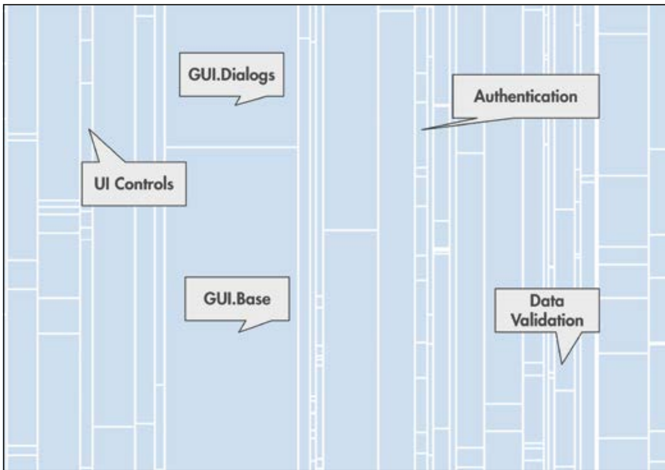


Abb. 1: Treemap mit den Komponenten des System under Test. Jedes Rechteck repräsentiert eine Komponente. Der Flächeninhalt korrespondiert mit der Größe der Komponente in Zeilen Quelltext. Exemplarisch ist die primäre Funktion einzelner Komponenten angegeben

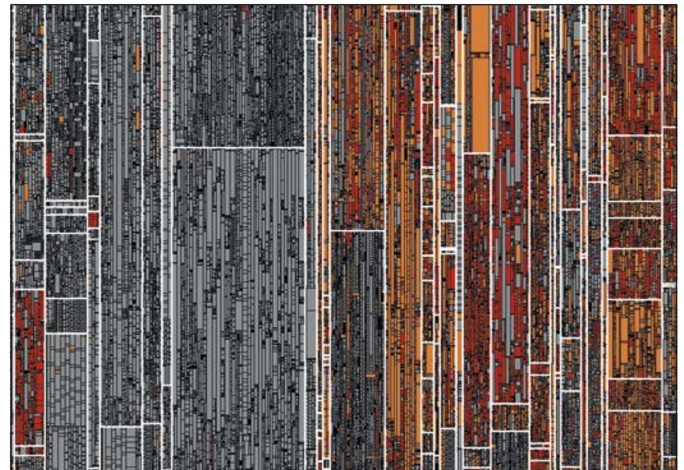


Abb. 2: Änderungen im System under Test seit dem letzten Release. Jedes kleine Rechteck stellt eine Methode im Quelltext dar. Unveränderte Methoden sind grau, neue Methoden rot und geänderte Methoden orange dargestellt

Verschiebungen von Code), können keine Fehler verursachen und daher herausgefiltert werden. Dadurch wird die Aufmerksamkeit auf die Änderungen gelenkt, durch die sich das Verhalten des Systems verändert hat. Die Änderungen eines der von uns analysierten Systeme sind in **Abbildung 2** dargestellt. **Abbildung 1** erläutert zusätzlich, wie das zugrunde liegende System aufgeteilt ist.

Ergänzend dazu wird mithilfe von *dynamischen Analysen* die Testüberdeckung ermittelt. Entscheidend ist dabei, dass alle durchgeführten Tests aufgezeichnet werden, also sowohl automatisierte als auch manuell durchgeführte Testfälle. Die durchlaufenen Methoden sind in **Abbildung 3** dargestellt.

Die Test-Gap-Analyse ermittelt dann durch die *Kombination* der Ergebnisse der statischen und dynamischen Analysen die ungetesteten Änderungen. **Abbildung 4** zeigt eine Treemap mit Ergebnissen einer Test-Gap-Analyse für dieses System. Die kleinen Rechtecke in den Komponenten repräsentieren hier die enthaltenen Methoden, ihr Flächeninhalt korrespondiert mit der Länge der Methode in Zeilen Quelltext. Die Farben der Rechtecke haben dabei die folgende Bedeutung:

- Graue Methoden wurden seit dem letzten Release nicht verändert.
- Grüne Methoden wurden verändert (oder neu programmiert) und kamen im Test zur Ausführung.
- Orange (und rote) Methoden wurden verändert (oder neu programmiert) und kamen im Test *nicht* zur Ausführung.

Man kann klar erkennen, dass im rechten Bereich der Treemap ganze Komponenten mit neuem oder verändertem Code im Test bisher nicht zur Ausführung kamen. Alle darin enthaltenen Fehler können daher nicht gefunden worden sein.

Wie kann Test-Gap-Analyse eingesetzt werden?

Nützlich wird die Test-Gap-Analyse dann, wenn sie kontinuierlich ausgeführt wird, beispielsweise jede Nacht, um morgens einen Überblick über die ausgeführten Tests und Änderungen bis zum letzten Abend zu geben. Hierfür werden Dashboards mit Informationen zu den Test-Gaps erstellt, wie in **Abbildung 5** gezeigt.

Die Dashboards erlauben den Test-Managern, rechtzeitig zu entscheiden, ob

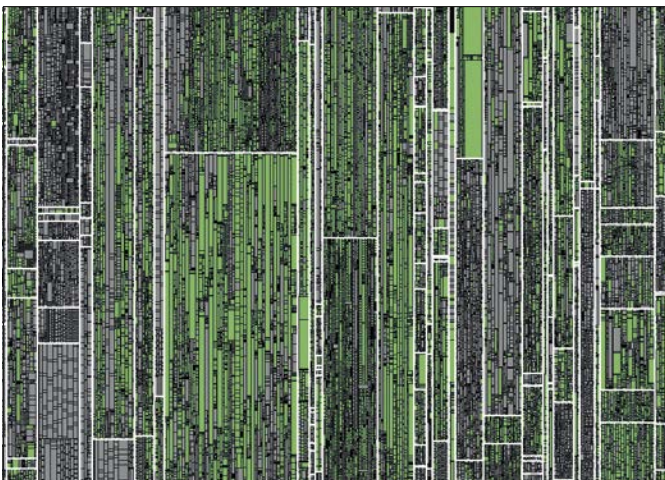


Abb. 3: Testabdeckung im System under Test am Ende der Testphase: Ungetestete Methoden sind grau dargestellt, im Test durchlaufene Methoden grün

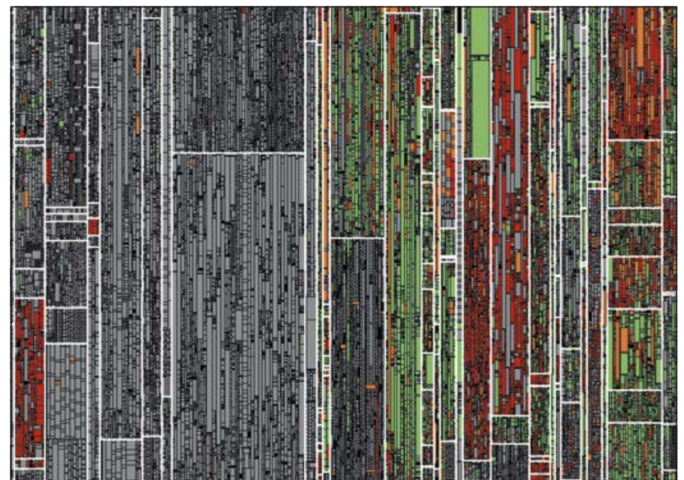


Abb. 4: Test-Gaps am Ende der Testphase. Unveränderte Methoden sind grau dargestellt. Geänderte Methoden, die getestet wurden, sind grün. Neue ungetestete Methoden sind rot, geänderte ungetestete Methoden orange dargestellt

weitere Testfälle notwendig sind, um die verbleibenden Änderungen noch während der Test-Phase zu durchlaufen. Ob das gelungen ist, kann am nächsten Tag in den neu berechneten Dashboards abgelesen werden.
Wenn mehrere Testumgebungen parallel betrieben werden, sollte für jede ein eigenes Dashboard eingerichtet werden, um die Test-Coverage gezielt zuordnen zu können. Zusätzlich gibt es ein Dashboard, in dem die Informationen aus allen Umgebungen zusammenlaufen. In **Abbildung 6** ist ein Beispiel mit drei verschiedenen Test-Umgebungen dargestellt:

- *Test:* In dieser Umgebung führen Tester ihre manuellen Testfälle durch.
- *Dev:* In dieser Umgebung werden die automatisierten Testfälle durchgeführt.
- *UAT:* In der User-Acceptance-Test-Umgebung führen Endanwender explorative Tests mit dem System under Test durch.
- *All:* Führt die Ergebnisse der drei Testumgebungen zusammen.

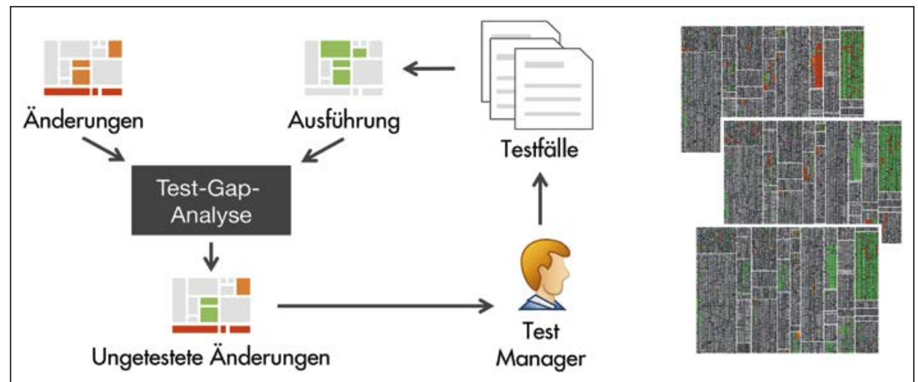


Abb. 5: Einsatz im Testprozess

Abb. 6: Einsatz von mehreren Dashboards für detaillierte Analyse

Für welche Projekte ist Test-Gap-Analyse einsetzbar?

Wir haben Test-Gap-Analyse bereits in den unterschiedlichsten Projekten eingesetzt: von betrieblichen Informationssystemen bis hin zu eingebetteter Software, von C/C++ über Java, C# und Python bis hin zu ABAP. Faktoren, die die Komplexität der Einführung beeinflussen, umfassen unter anderem:

- *Ausführungsumgebung:* Virtuelle Maschinen (z. B. Java, C#, ABAP) erleichtern die Erhebung von Test-Coverage-Daten.
- *Architektur:* Bei Server-basierten Anwendungen müssen die Test-Coverage-Daten auf weniger Maschinen erhoben werden als bei Fat-Client-Anwendungen.
- *Testprozess:* Definierte Testphasen erleichtern die Planung und Begleitung.

Was bringt Test-Gap-Analyse im Hot-Fix-Test?

Beim Test von Hot-Fixes steht meist nur sehr wenig Zeit zur Verfügung. Ziele im Hot-Fix-Test sind einerseits sicherzustellen, dass der behobene Fehler nicht mehr auftritt, und andererseits, dass dabei keine neuen Fehler eingebaut wurden. Für Letzteres sollte wenigstens sichergestellt werden, dass alle im Hot-Fix durchgeführten Änderungen durchlaufen wurden.

Hierfür wird in der Test-Gap-Analyse der Release-Stand als Referenzversion definiert und alle Änderungen werden ermittelt, die für das Hot-Fix (bspw. auf einem eigenen Branch) durchgeführt wurden, wie in **Abbildung 7** dargestellt.
Mithilfe der Test-Gap-Analyse wird dann ermittelt, ob im Fehlernachtest tatsächlich alle Änderungen durchlaufen worden sind. Im Beispiel in **Abbildung 8** zeigt sich, dass ein Teil der Methoden noch ungetestet ist. Unsere Erfahrungen haben gezeigt, dass sich gerade in Hot-Fix-Tests durch Test-Gap-Analyse leichtgewichtig und einfach die Sicherheit erhöhen lässt, durch die Änderungen keine neuen Fehler einzubauen.

Was bringt Test-Gap-Analyse im Release-Test?

Als Release-Test bezeichnen wir in diesem Artikel die Test-Phase vor einem größeren Release, in der typischerweise sowohl neu

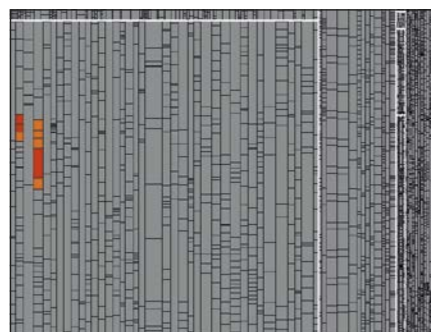


Abb. 7: Im Zuge eines Hot-Fix geänderte Methoden

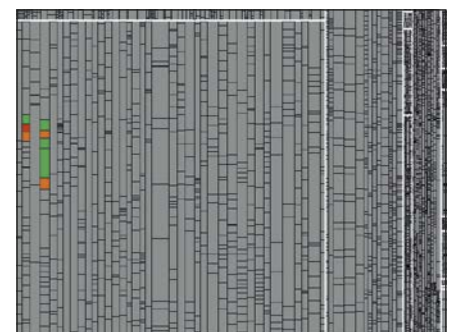


Abb. 8: Beim Fehlernachtest eines Hot-Fix getestete und ungetestete Methoden

Weitere Informationen

Wir haben unter www.testgap.io weiterführende Materialien zur Test-Gap-Analyse zusammengestellt, unter anderem Forschungsarbeiten, Blog-Einträge und Werkzeugunterstützung. Darüber hinaus freuen wir uns als Autoren auch per E-Mail über Fragen und Feedback (auch kritisches) zum Artikel oder zu Test-Gap-Analyse allgemein.

Kasten 1

implementierte Funktionalität überprüft wird als auch Regressionstests durchgeführt werden. Häufig kommen dabei unterschiedliche Arten von Tests zum Einsatz.
Unsere Erfahrung hat gezeigt, dass der Einsatz der Test-Gap-Analyse die Menge an Änderungen, die ungetestet in Produktion gelangen, deutlich reduziert.
In **Abbildung 9** ist eine Test-Gap-Treemap des gleichen Systems dargestellt, das auch in Abbildung 4 abgebildet ist. Während Abbildung 4 retrospektiv ermittelt wurde, ist Abbildung 9 ein Snapshot aus einer Iteration, in der Test-Gap-Analyse kontinuierlich eingesetzt wird. Dabei werden sowohl manuelle als auch automatisierte Tests betrachtet. Es ist klar zu erkennen, dass es deutlich weniger Test-Gaps gibt.
Unsere Beobachtung ist auch, dass in vielen Fällen einige Test-Gaps bewusst in Kauf genommen werden, beispielsweise weil der zugehörige Quelltext über die Benutzeroberfläche noch nicht erreichbar ist. Entscheidend ist jedoch, dass es sich hierbei um bewusste, fundierte Entscheidungen handelt, deren Auswirkungen abschätzbar sind.

Wo sind die Grenzen von Test-Gap-Analyse?

Wie jedes Analyseverfahren hat auch die Test-Gap-Analyse Grenzen. Ihre Kenntnis ist entscheidend, um sie sinnvoll einsetzen zu können.
Eine Grenze von Test-Gap-Analyse sind Änderungen, die auf Konfigurationsebene durchgeführt werden, ohne dass dabei Code verändert wird, da sie dadurch der Analyse verborgen bleiben.
Eine weitere Einschränkung von Test-Gap-Analyse ist die Aussagekraft von durchlaufenem Code. Test-Gap-Analyse betrachtet, welcher Code beim Test zur Ausführung gekommen ist. Wie gründlich bei der Ausführung getestet wurde, bleibt der Analyse verborgen. Dadurch ist es prinzipiell möglich, dass Fehler unentdeckt bleiben, obwohl der durchlaufene Code von der Analyse als „grün“ dargestellt wird. Dieser Effekt wird größer, je gröber die Messung der Code-Coverage ist.
Der Umkehrschluss gilt jedoch: Roter und orangener Code ist nicht durchlaufen worden. Enthaltene Fehler können daher nicht gefunden worden sein.
Unsere Erfahrung aus der Praxis zeigt, dass die Lücken, die durch den Einsatz von Test-Gap-Analyse aufgedeckt werden, meist so groß sind, dass substanzielle Erkenntnisse über Schwächen im Test-Prozess aufgedeckt werden. Bei diesen großen Lücken kommen die oben beschriebenen Grenzen nicht zum Tragen.

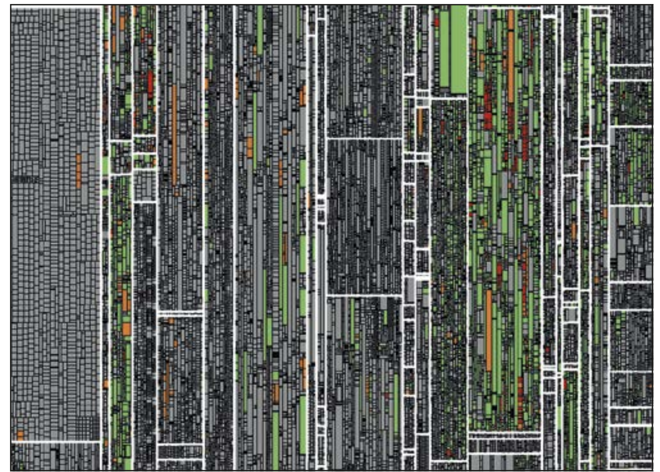


Abb. 9: Weniger Test-Gaps bei kontinuierlichem Einsatz

Ausblick

Ein weiteres spannendes Anwendungsfeld der beschriebenen Analysetechniken ist der Einsatz in einer Produktionsumgebung. Die aufgezeichneten Ausführungen sind dabei nicht mehr die ausgeführten Testfälle, sondern die Systeminteraktionen durch die Endanwender. Dadurch lässt sich ermitteln, welche der Features, die im letzten Release eingebaut wurden, eigentlich durch die Anwender verwendet werden. In unserer Erfahrung ergeben sich dabei oft Überraschungen.
In **Abbildung 10** ist die Nutzung eines betrieblichen Informationssystems in Anlehnung an einen Gantt-Chart dargestellt [Jür11]. Jede Zeile repräsentiert ein Fea-

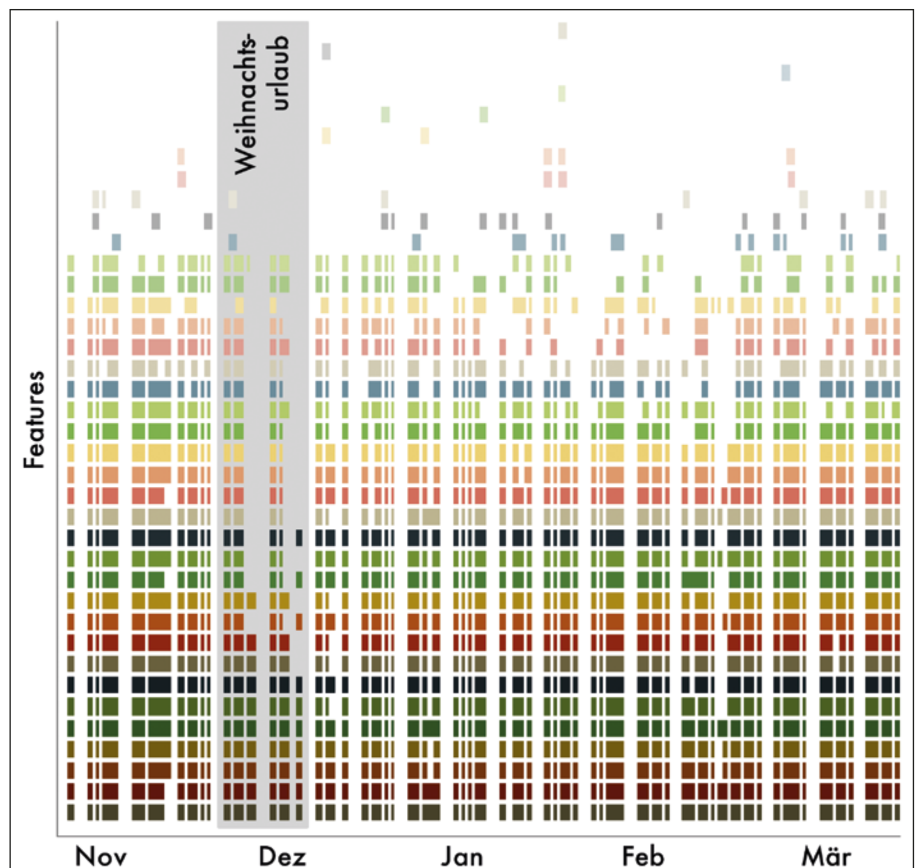


Abb. 10: Nutzung der Features eines betrieblichen Informationssystems in Produktion

Literatur & Links

[Ede13] S. Eder, B. Hauptmann, M. Junker, E. Jürgens, R. Vaas, K.-H. Prommer, Did we test our changes? Assessing alignment between tests and development in practice, in: Proc. of the 8. Int. Workshop on Automation of Software Test (AST'13), 2013, siehe auch: **https://www.cqse.eu/publications/2013-did-we-test-our-changes-assessing-alignment-between-tests-and-development-in-practice.pdf**

[Jür11] E. Jürgens, M. Feilkas, M. Herrmannsdoerfer, F. Deissenboeck, R. Vaas, K.-H. Prommer, Feature profiling for evolving systems, in: Proc. of the 19. IEEE Int. Conf. on Program Comprehension (ICPC'11), 2011, siehe auch: **https://www.cqse.eu/publications/2011-feature-profiling-for-evolving-systems.pdf**

ture der Anwendung. Auf der X-Achse ist der Messzeitraum dargestellt. An Wochenenden und in der Weihnachtszeit ist erwartungsgemäß weniger Nutzung als an den anderen betrachteten Tagen. In der Abbildung sind jedoch nur die Features dargestellt, die überhaupt verwendet wurden.

Die Analyse hat aber gezeigt, dass 28 Prozent der Features der Anwendung gar nicht verwendet wurden, was für alle beteiligten Stakeholder unerwartet war. In diesem Fall hat die Analyse zur Löschung von etwa einem Viertel des Quelltextes der Anwendung geführt, sodass sich in den folgenden Releases eine Reihe der Testaufwände einsparen oder nutzbringender einsetzen ließen[1]. ||

Die Autoren

Dr. Elmar Jürgens
(juergens@cqse.eu)

hat für seine Doktorarbeit über Qualitätsanalyse den Software-Engineering-Preis der Ernst Denert-Stiftung erhalten. Er ist Mitgründer der CQSE GmbH, spricht regelmäßig auf Fachkonferenzen und wurde zum Junior Fellow der Gesellschaft für Informatik ernannt.

Dr. Dennis Pagano
(pagano@cqse.eu)

hat in Software-Engineering promoviert und begleitet als Experte für Softwarequalität bei der CQSE GmbH viele Firmen beim Verbessern ihrer Qualitätssicherungs- und Testprozesse. Er ist aktiv an Forschung und Entwicklung im Bereich Test-Gap-Analyse beteiligt.

[1]) Für eine Nutzungsanalyse auf Feature-Ebene sind einige Techniken erforderlich, die über die in diesem Artikel beschriebenen Methoden hinausgehen. Sie sind in [Jür11] beschrieben.

Mit „Ticket Coverage“ verhindern, dass wichtige Features ungetestet bleiben

Immer kürzere Testphasen?

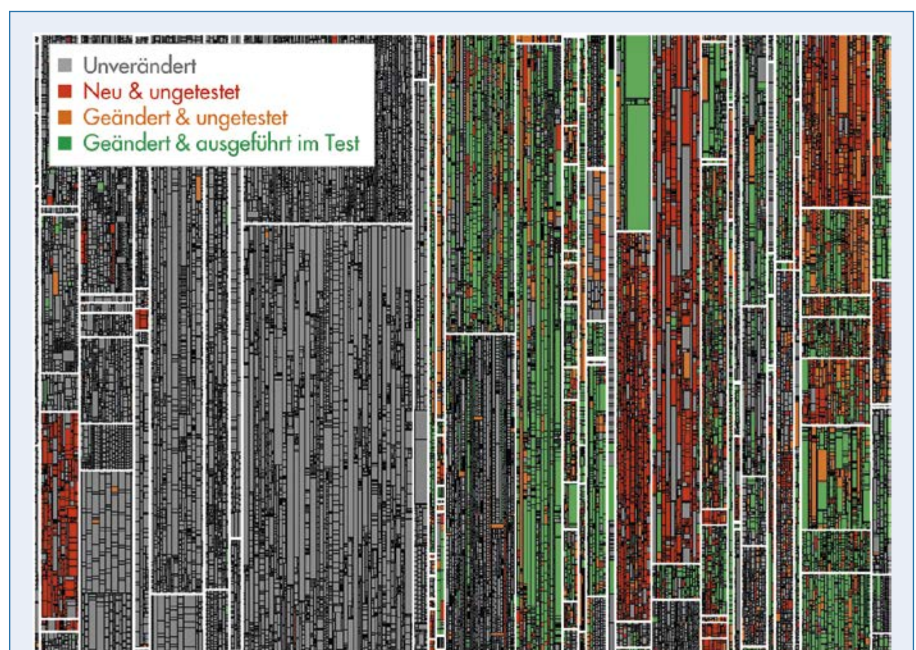
Viele Teams führen Tests parallel zur Entwicklung durch, da wegen kürzerer Releasezyklen immer weniger Zeit für dedizierte Testphasen zur Verfügung steht. Dadurch steigt die Gefahr, dass wichtige Funktionalität ungetestet in Produktion gelangt, weil man immer schwieriger überblicken kann, welche Tickets wie gründlich getestet wurden und wo aufgrund von zusätzlichen Änderungen erneut getestet werden müsste. In diesem Artikel stellen wir Ticket Coverage vor, die diejenigen Tickets identifiziert, deren zugehörige Codeänderungen nicht ausreichend getestet wurden. Wir gehen sowohl auf die Berechnung als auch auf empirische Studien zur Nützlichkeit von Ticket Coverage ein.

Da in der Praxis selten genug Ressourcen zur Verfügung stehen, um große Systeme vollständig zu testen, müssen sich Testmanager häufig auf einen – möglichst sinnvollen – Ausschnitt aller Tests beschränken. Ein zentrales Ergebnis der Forschung der letzten Jahre ist, dass die meisten Fehler typischerweise dort auftreten, wo in letzter Zeit (zum Beispiel seit dem letzten Release) viel geändert wurde (vgl. [Ede13], [Nag05]). Im Rahmen der Qualitätssicherung der Testaktivitäten sollte also überprüft werden, ob alle relevanten Änderungen getestet wurden.

Test-Gap-Analyse (**siehe Kasten 1**) identifiziert alle Änderungen seit einem Referenzzeitpunkt (zum Beispiel dem letzten Release), die nicht getestet wurden. Diese Betrachtung bewährt sich vor allem in Projekten, die (wie in **Abbildung 1** dargestellt) vor einem Release eine ausgedehnte Testphase durchführen, in der das gesamte Release getestet wird.

Agile Entwicklungsmethoden drängen jedoch seit Jahrzehnten auf kürzere Releasezyklen, um neue Funktionalität schneller zum Anwender zu bringen. In den letzten Jahren hat, in unserer Beobachtung, dieser Trend in den meisten Entwicklungsteams die Releasezyklen stark verkürzt. Mit der Verkürzung der Releasezyklen geht oft auch eine Verkürzung der Testphasen einher.

In vielen Fällen ist es gar nicht mehr möglich, vor einem Release eine ausgedehnte Testphase durchzuführen. Stattdessen müssen Tests iterationsbegleitend durchgeführt werden, wie in **Abbildung 2** dargestellt[1].



Hintergrund: Test-Gap-Analyse

Test-Gap-Analyse (vgl. [JüPa18]) kombiniert statische und dynamische Analyseverfahren, um geänderten, aber ungetesteten Code zu identifizieren. In obiger Abbildung ist ein typisches Ergebnis von Test-Gap-Analyse in Form einer Treemap dargestellt: Jedes weiß umrandete Rechteck repräsentiert dabei eine Komponente im System under Test. Jedes darin enthaltene kleinere, schwarz umrandete Rechteck entspricht einer Methode im Quelltext. Der Flächeninhalt eines Rechtecks korrespondiert mit der Größe der dazugehörigen Methode in Zeilen Code.

Methoden, die sich seit dem letzten Release nicht verändert haben, sind ausgegraut. Nur wenn die Änderungen grün dargestellt sind, wurden sie im Test durchlaufen. Die roten und orangenen Bereiche zeigen also Methoden, die seit dem letzten Release neu entwickelt oder verändert, aber noch nicht getestet wurden. In diesen Bereichen treten typischerweise die meisten Feldfehler auf.

Kasten 1

[1]) Um Tests iterationsbegleitend durchzuführen, können prinzipiell die unterschiedlichsten Testansätze eingesetzt werden. Von automatisierten (zum Beispiel 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 Testansätze für iterationsbegleitendes Testen sind außerhalb des Fokus dieses Artikels.

Abb. 1: Aufteilung von Entwicklungs- und Testaktivitäten bei wenigen großen Releases pro Jahr

Abb. 2: Aufteilung von Entwicklungs- und Testaktivitäten bei vielen kurzen Releasezyklen pro Jahr

Tickets im Fokus der Testplanung

Wir beobachten, dass in vielen Teams immer stärker einzelne *Tickets* im Fokus der Testplanung stehen, um Tests iterationsbegleitend steuern zu können. Testern werden dabei dedizierte Tickets zugewiesen, die sie zeitnah nach Abschluss der zugehörigen Entwicklungsarbeiten testen. Dadurch gelingt es, Tests verschränkt zur Entwicklung auszuführen, sodass dedizierte Testphasen entfallen oder kürzer ausfallen können.
Unter *Ticket* verstehen wir hierbei die Beschreibung einer geplanten Änderung am System, die in einem Ticket-System (beispielsweise JIRA, Redmine, TFS usw.) verwaltet wird. Je nach Art und Quelle der Änderung wird in Teams oft zwischen unterschiedlichen Ticket-Klassen unterschieden, wie User-Storys, Change-Requests, Bug-Reports oder allgemein Issues. Hier beziehen wir all diese 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 Testmanager sehr schwierig ist, zu beurteilen, wie gründlich Tickets im Rahmen der ausgeführten Tests getestet wurden:

- Als Test-Input werden oft die Beschreibungen der Tickets verwendet. Gerade im Vergleich zu strukturierten Testfällen sind diese allerdings oft deutlich informationsärmer, insbesondere in Hinsicht auf Sonderfälle.
- Da immer mehr Testarten gemeinsam (Unittests, skriptierte UI-Tests, manuelle Tests, Systemtests, explorative Tests usw.) und oft zeitgleich auf unterschiedlichen Teststufen (Smoke-Tests, Akzeptanztests usw.) zum Einsatz kommen, wird es immer schwieriger, den Überblick zu behalten, was eigentlich wo 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, den Überblick darüber zu behalten, welche Änderungen auf welchen Branches 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 *Ticket Coverage* vor. Ticket Coverage drückt aus, welcher Anteil des Codes, der im Zuge der Implementierung eines Tickets angefasst (hinzugefügt oder geändert) wurde, im Test zur Ausführung kam. Ähnlich wie Test-Gap-Analyse wird Ti-

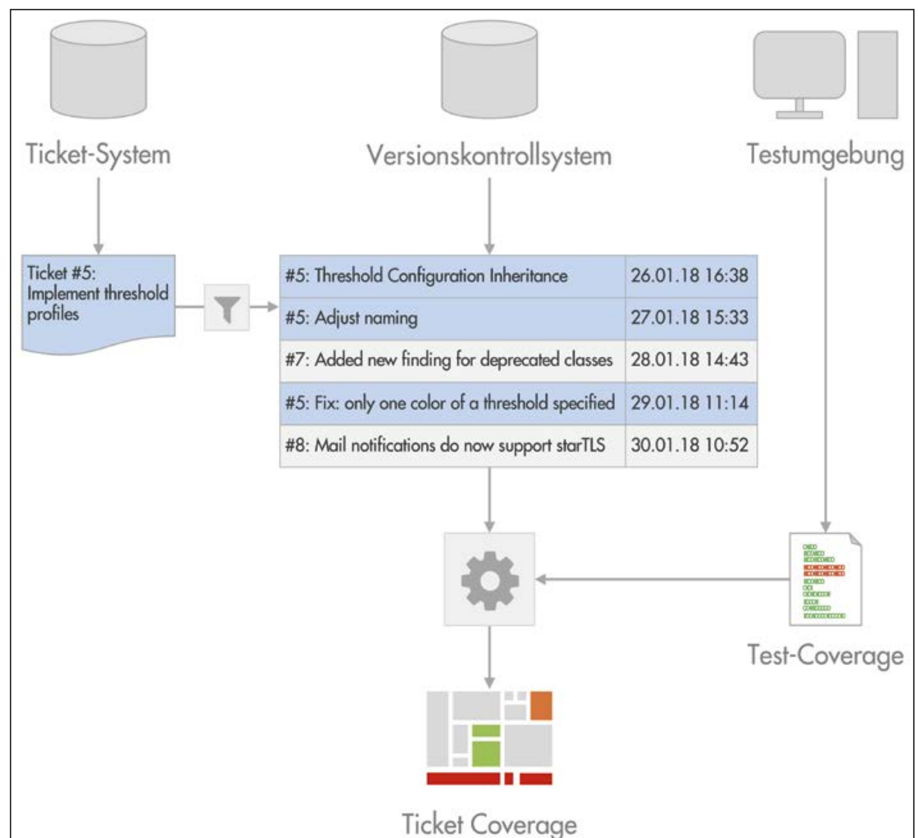


Abb. 3: Berechnung von Ticket Coverage

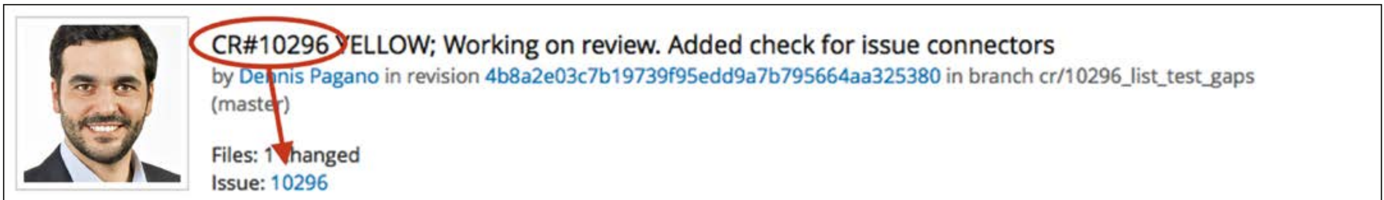


Abb. 4: Commit, bei dem die ID des Tickets angegeben wurde (10296), auf das sich die durchgeführten Änderungen beziehen

cket Coverage auf Basis von Änderungen und Ausführungsinformationen berechnet. Jedoch werden die Informationen nicht für das gesamte System, sondern für einzelne Tickets ausgewertet und dargestellt. Ticket Coverage wird in folgenden Schritten berechnet (**vgl. Abbildung 3**):

- Im ersten Schritt wird ermittelt, welche Methoden im Code im Rahmen eines Tickets angefasst wurden („Ticket-Code"). 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-System und der Versionshistorie kann dadurch für jedes Ticket die Menge der Methoden erhoben werden, die im Rahmen der Umsetzung des Tickets angefasst wurden.
- Im zweiten Schritt werden Test-Coverage-Daten aus verschiedenen Testläufen erhoben, um zu ermitteln, welche Methoden im Test zur Ausführung kamen.
- Im dritten Schritt wird durch Kombination von Ticket-Code und Test-Coverage-Daten ermittelt, welche der zum Ticket gehörenden Methoden im Test zur Ausführung kamen.

Ticket #	Subject	Ticket Coverage
TS-13701	Implement metric 'Nesting Depth' for Simulink	45%
TS-13714	External findings are not registered during first upload	100%
TS-13727	Manual test coverage upload during development	100%
TS-13730	Assert value is not null in ValueIndexBase.set... Methods	47%
TS-13731	Tool for transferring Finding Blacklists and Tasks (w/ Findings) between instances	100%
TS-13742	Show data timestamp in project analysis warning	100%
TS-13769	Show summary row in test perspective > issue view	71%
TS-13779	Get rid of Bootstrap Wells	100%
TS-13794	Extend ExternalAnalysisResult to include branch coverage information	51%
TS-13797	MetricTrendIndex is not rollbacked	100%

Abb. 5: Tickets aus unserer eigenen Entwicklung zusammen mit ihrer Ticket Coverage

Ticket Coverage lässt sich beispielsweise als Anteil der getesteten Methoden am gesamten Ticket-Code ausdrücken. In **Abbildung 5** ist beispielhaft eine Liste von Tickets aus unserer eigenen Entwicklung zusammen mit ihrer Ticket Coverage in Prozent abgebildet. Natürlich kann pro Ticket wiederum dargestellt werden, welche Methoden geändert wurden und welche davon ungetestet sind, um diese Testlücken im Ticket weiter zu untersuchen.

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 (vgl. [Dre17], [Rot17]) durchgeführt.
Als Studienobjekt haben wir das von uns entwickelte System *Teamscale* [TeamS] gewählt, weil wir hierfür die Ergebnisse der Studien besser beurteilen können als für fremden Code. Teamscale ist eine inkrementelle Software-Qualitätsanalyse-Suite, die von vielen namhaften Firmen eingesetzt wird. Der Quelltext umfasste zum Zeitpunkt der Studien ca. 650.000 Zeilen Java- beziehungsweise 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 mithilfe von Ticket Coverage relevante Testlücken in einem typischen Setting mit strukturierten manuellen Tests finden.

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 Plug-in Code	1	0,9%
	Methode für Unittests	2	1,8%
nicht relevant: 28 (25,5%)	einfacher Getter triviale	12	10,9%
	Methode	12	10,9%
	toString-Methode	4	3,6%
	keine Antwort	3	2,7%
	Σ	110	100%

Tabelle 1: Bewertung der Testlücken durch die Entwickler

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 1** dargestellt. Von den insgesamt 110 Testlücken wurden 20 (18,2 Prozent) 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 Codeänderungen. 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 [Rot17].

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 zum Beispiel mithilfe 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 [Dre17] 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 mithilfe der Ticket Coverage Testlücken in solchen explorativen Tests aufdecken können, die im Rahmen der Reviews durchgeführt werden. Hierfür haben wir vier Tickets zur näheren Analyse ausgewählt. Wir ha-

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

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

cr-9709/engine/com.teamscale.index/.../MethodHistoryService.java (revision 50dc29df...)

```
75
76      /** {@inheritDoc} */
77      @Override
78      public HttpResult process(HttpQuery query) throws ServiceException {
79          String target = query.getTarget();
80
81          long endTimestamp = determineEndTimestamp(query);
82
83          // Get the key of the method (for {@MethodInfoIndex}) in which we a
84          // interested.
85          String uniformPath = target;
86          OffsetBasedRegion region = new OffsetBasedRegion(
87                  query.getIntParameter("startOffset", 0),
88                  query.getIntParameter("endOffset", 0));
89
90          try {
91              List<MethodHistoryEntry> entries = createMethodHistoryEntries(
92                      uniformPath, region, endTimestamp);
93
94              return serializeObjectToHttp(entries, query);
95          } catch (ConQATException e) {
96              throw new InternalServiceException(e.getMessage(), e);
97          }
98      }
99
100     /**
101      * Uses the given method as base to go backwards through the history of
```

cr-9709/engine/com.teamscale.index/.../MethodHistoryService.java (revision EA:cr/970...)

```
75      @Override
76      public HttpResult process(HttpQuery query) throws ServiceException {
77          String uniformPath = query.getTarget();
78          long endTimestamp = determineEndTimestamp(query);
79
80          OffsetBasedRegion region = new OffsetBasedRegion(
81                  query.getIntParameter("startOffset", 0),
82                  query.getIntParameter("endOffset", 0));
83
84          try {
85              MethodInfo methodInfo = getMethodInfoForTimestamp(uniformPath,
86                      region, endTimestamp);
87              if (methodInfo == null) {
88                  throw new BadRequestException(
89                          "No method found for given region.");
90              }
91
92              List<MethodHistoryEntry> entries = new ArrayList<>();
93              addMethodHistoryEntries(uniformPath, region, endTimestamp,
94                      methodInfo, entries);
95
96              return serializeObjectToHttp(entries, query);
97          } catch (ConQATException e) {
98              throw new InternalServiceException(e.getMessage(), e);
99          }
100     }
101
```

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

ben 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[2] 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.
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 Prozent auf. Bei der anschließenden Bewertung dieser Lücken durch die Reviewer wurden 23 Methoden (76,6 Prozent) als testenswert identifiziert. Die Analyse der Ticket Coverage führte also auch im Fall der explorativen Tests zur Aufdeckung von relevanten Testlücken.

Interpretation
Wir haben in den wissenschaftlichen Studien Ticket Coverage für zwei unterschiedliche Testarten im Rahmen der Entwicklungs- und Testprozesse des 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: Ticketbeschreibungen reichen in vielen Fällen nicht aus, um einen vollständigen Test eines Tickets durchzuführen. Häufige Gründe hierfür sind Spezialfälle, technische Änderungen oder Wartbarkeitsverbesserungen, die vom Entwickler umgesetzt wurden, aber nicht im Ticket beschrieben sind. Ticket Coverage hilft, trotzdem zu erkennen, welche wichtigen Tickets nicht ausreichend getestet wurden. 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 gegebenenfalls 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 zutage fördert. Dies passt auch zu den Erfahrungen, die wir über unsere wissenschaftlichen Aktivitäten hinaus tagtäglich als Berater in Kundenprojekten gewinnen.

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 Prozent 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 bearbeitet. Wenn eine solche Methode beim Test eines Tickets durchlaufen wird, gilt sie gegebenenfalls auch für andere Tickets als getestet, es sei denn, es wird auf isolierten Feature-Branches gearbeitet. Es ist aber möglich, dass der Test anderer Tickets Fehler in dieser Methode aufdecken würde. Selbst eine Ticket Coverage von

[2]) Triviale Methoden sind beispielsweise reine Getter und Setter. Die Kriterien zur Klassifikation von Methoden als *zu trivial für den Test* wurden im Rahmen der Studie durch die beteiligten Entwickler validiert. Details finden sich in [Rot17].

100 Prozent 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.

Wie kann ich Ticket Coverage einsetzen?

Wir haben Ticket Coverage in den letzten Jahren bei vielen Firmen aus verschiedensten Domänen eingeführt. Fast immer haben wir dabei etwas andere Konstellationen aus Programmiersprachen, Technologien, Infrastruktur, Testwerkzeugen und Testansätzen vorgefunden. Unter **www.testgap.io** haben wir weiterführende Materialien zusammengestellt, unter anderem Blog-Einträge zur Einführung, Forschungsarbeiten und Werkzeugunterstützung. Darüber hinaus freuen wir uns auch per E-Mail über Fragen und Feedback (auch kritisches) zum Artikel oder zu Ticket Coverage allgemein.

Danksagung

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

Literatur & Links

[Dre17] F. Dreier, E. Jürgens, A. Göb, Test Accompanying Calculation of Test Gaps for Java Applications. Whitepaper, CQSE GmbH, 2017, siehe: **https://www.cqse.eu/publications/2017-test-accompanying-calculation-of-test-gaps-for-java-applications.pdf**

[Ede13] S. Eder, B. Hauptmann, M. Junker, E. Jürgens, R. Vaas, K.-H. Prommer, Did we test our changes? assessing alignment between tests and development in practice, in: Proc. of the 8. Int. Workshop on Automation of Software Test (AST'13), 2013, **siehe: https://www.cqse.eu/publications/2013-did-we-test-our-changes-assessing-alignment-between-tests-and-development-in-practice.pdf**

[JüPa18] E. Jürgens, D. Pagano, Erfahrungen mit Test-Gap-Analyse in der Praxis, in: OBJEKTspektrum, 2/2018

[Nag05] N. Nagappan, Th. Ball, Use of relative code churn measures to predict system defect density, in: Proc. of the 27. Int. Conf. on Software Engineering (ICSE) 2005

[Rot17] J. Rott, R. Niedermayr, E. Jürgens, D. Pagano, Ticket coverage: Putting test coverage into context, in: Proc. of the 8. Workshop on Emerging Trends in Software Metrics (WETSoM'17), 2017, siehe: **https://www.cqse.eu/publications/2017-ticket-coverage-putting-test-coverage-into-context.pdf**

[TeamS] Teamscale, eine inkrementelle Software-Qualitätsanalyse-Suite, die u. a. Ticket Coverage implementiert. Hier als Studienobjekt verwendet, siehe: **www.teamscale.com**

Die Autoren

Dr. Dennis Pagano
(pagano@cqse.eu)
hat in Software-Engineering promoviert und begleitet als Experte für Softwarequalität bei der CQSE GmbH viele Firmen beim Verbessern ihrer Qualitätssicherungs- und Testprozesse. Er ist aktiv an Forschung und Entwicklung im Bereich Test-Gap-Analyse beteiligt.

Dr. Elmar Jürgens
(juergens@cqse.eu)
hat für seine Doktorarbeit über Qualitätsanalyse den Software-Engineering-Preis der Ernst Denert-Stiftung erhalten. Er ist Mitgründer der CQSE GmbH, spricht regelmäßig auf Fachkonferenzen und wurde zum Junior Fellow der Gesellschaft für Informatik ernannt.