Haben wir das Richtige getestet? Erfahrungen mit Test-Gap-Analyse in der Praxis

Elmar Jürgens
CQSE GmbH
juergens@cqse.eu

Dennis Pagano
CQSE GmbH
pagano@cqse.eu

***Zusammenfassung*—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 Code-Änderungen auch in gut strukturierten Testprozessen oft ungewollt ungetestet bleiben.**

Test-Gap-Analyse zeigt ungetestete Änderungen auf und erlaubt Testern dadurch, sie noch rechtzeitig zu testen. Damit erlaubt Test-Gap-Analyse eine wirksame Qualitätssicherung der eigenen Test-Prozesse.

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 gehen dabei vor allem darauf ein, wie sich Test-Gap-Analyse in verschiedenen Test-Phasen einsetzen lässt.

Wie gut werden Code-Änderungen in der Praxis durch Tests wirklich abgedeckt?

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 vermutlich die meisten Fehler enthalten.

Um besser zu verstehen, ob die Tests die Änderungen tatsächlich erreicht haben, haben wir eine wissenschaftliche Studie [1] 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 Code-Bereiche für die beiden Releases neu entwickelt oder verändert worden sind. Für beide Releases wurde jeweils etwa 15% 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, etc.) 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

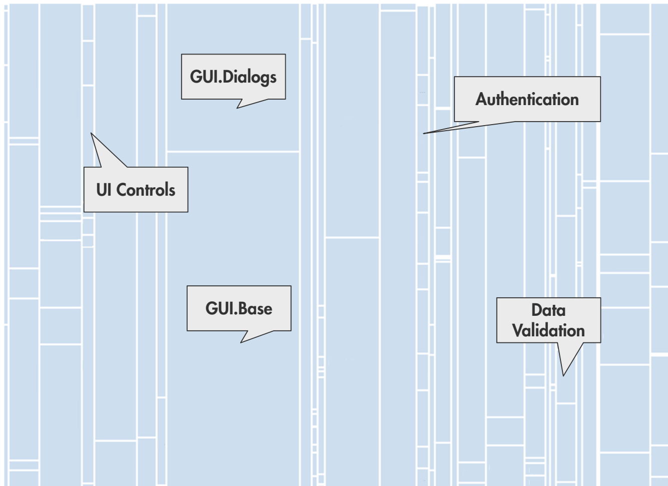

Abbildung 1. Treemap mit den Komponenten des System under Test. Jedes Rechteck repräsentiert eine Komponente. Der Flächeninhalt korrespondiert mit der Größe der Komponente in Zeilen Quelltext. Exemplarisch ist die primäre Funktion einzelner Komponenten angegeben.

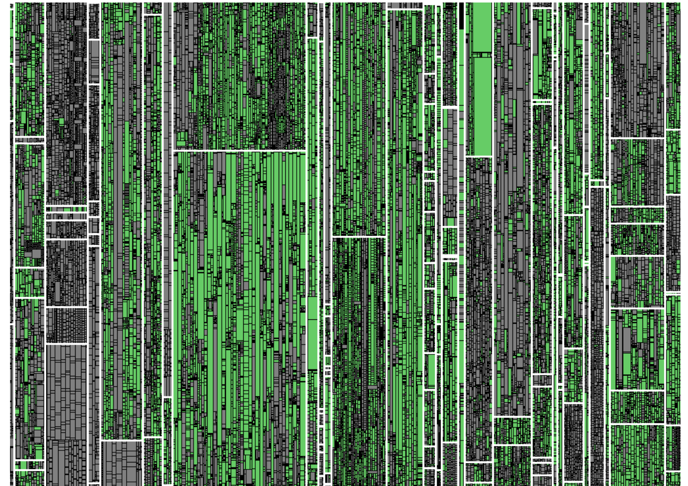

Abbildung 3. Testabdeckung im System under Test am Ende der Testphase: Ungetestete Methoden sind grau dargestellt, im Test durchlaufene Methoden grün.

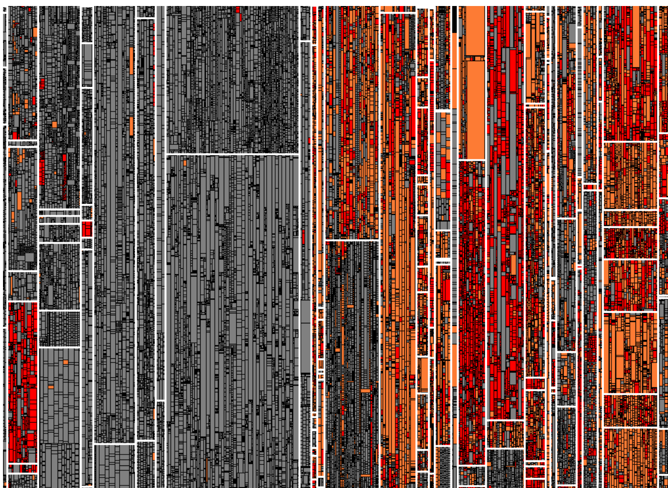

Abbildung 2. Änderungen im System under Test seit dem letzten Release: Jedes kleine Rechteck stellt eine Methode im Quelltext dar. Unveränderte Methoden sind grau, neue Methoden rot und geänderte Methoden orange dargestellt.

lückenlos zu finden, benötigen wir daher zuverlässige Informationen, welche Änderungen im Test durchlaufen wurden und welche nicht.

WAS KANN MAN MACHEN?

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

Statische Analyse. Eine statische Analyse vergleicht den aktuellen Stand des Quelltextes des *System under Test* mit dem Stand des letzten Releases, um neue und geänderte Code-Bereiche 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 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 zugrundeliegende System aufgeteilt ist.

Dynamische Analyse. Ergänzend dazu wird mit Hilfe 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.

Kombination. 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.

Abbildung 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.

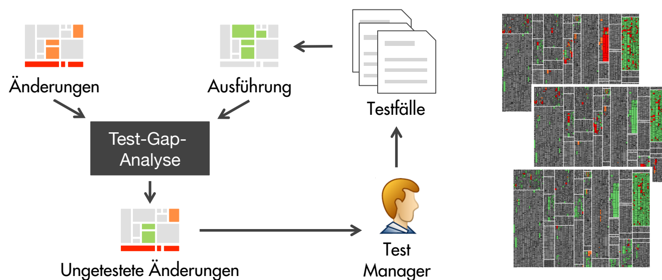

Abbildung 5. Einsatz im Testprozess.

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 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.

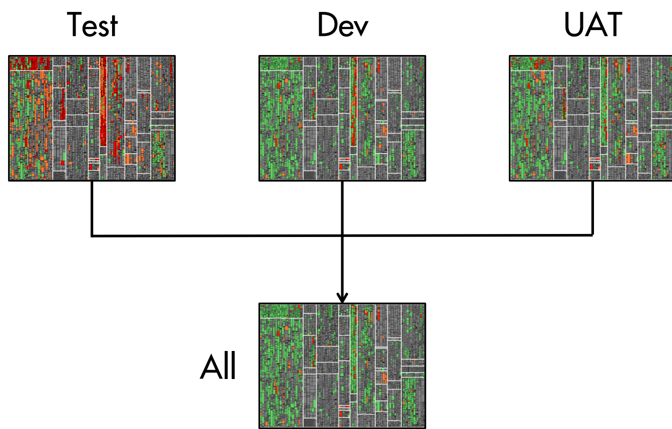

Abbildung 6. Einsatz von mehreren Dashboards für detaillierte Analyse.

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 Hotfix-Test?

Beim Test von Hot-Fixes steht meist nur sehr wenig Zeit zur Verfügung. Ziele im Hotfix-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 ermittelt, die für das Hot-Fix (bspw. auf einem eigenen Branch) durchgeführt wurden, wie in Abbildung 7 dargestellt.

Mit Hilfe 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

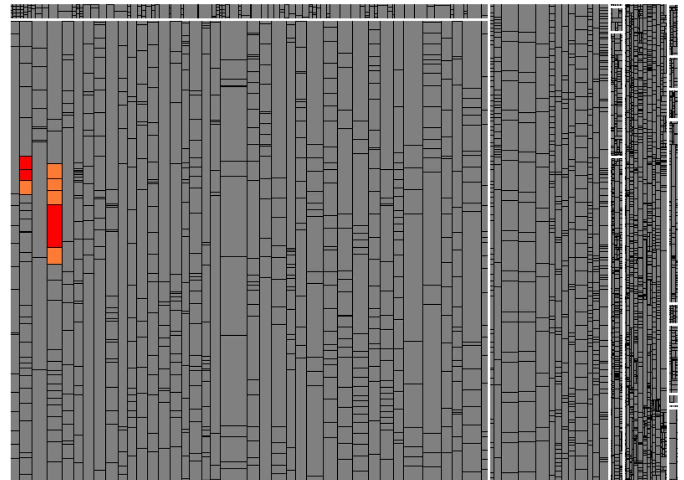

Abbildung 7. Im Zuge eines Hot-Fix geänderte Methoden

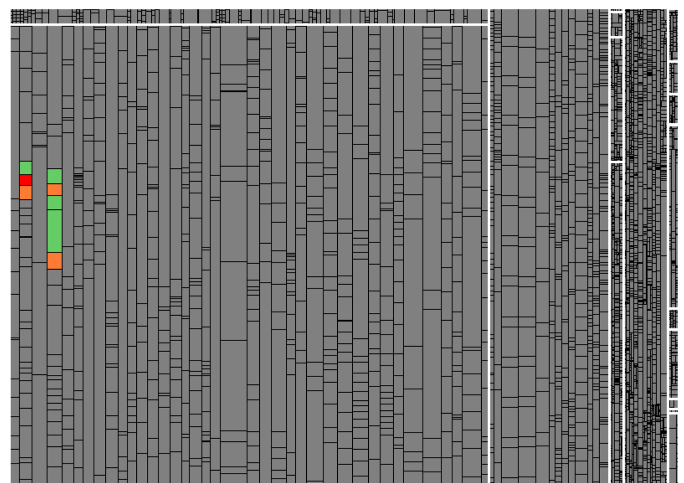

Abbildung 8. Beim Fehlernachtest eines Hot-Fix getestete und ungetestete Methoden

sowohl neu implementierte Funktionalität überprüft, 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, bspw. weil der zugehörige Quelltext über die Benutzeroberfläche noch nicht erreichbar ist. Entscheidend ist jedoch, dass es sich

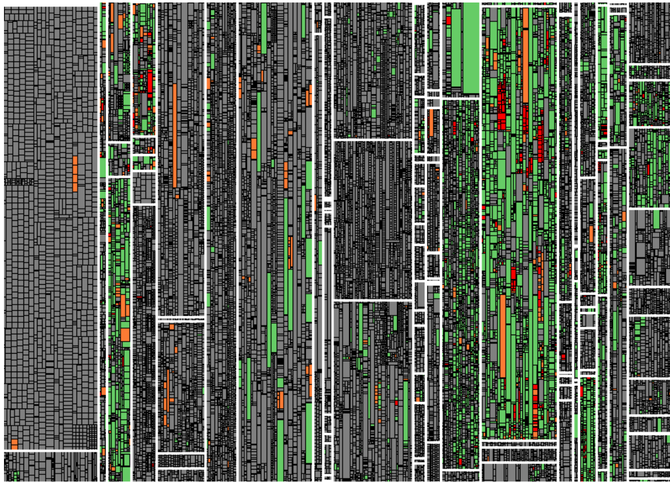

Abbildung 9. Weniger Test-Gaps bei kontinuierlichem Einsatz.

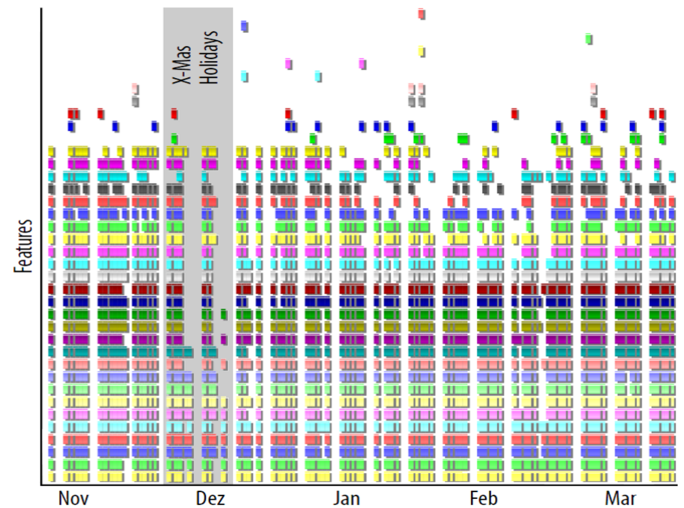

Abbildung 10. Nutzung der Features eines betrieblichen Informationssystems in Produktion

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 substantielle Erkenntnisse über Schwächen im Test-Prozess aufgedeckt werden. Bei diesen großen Lücken kommen die oben beschriebenen Grenzen nicht zum Tragen.

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 [2]. Jede Zeile repräsentiert ein Feature der Anwendung. Auf der X-Achse ist der Messzeitraum dargestellt. An Wochenenden und in der Weihnachtszeit ist erwartungsgemäß weniger Nutzung, als in den anderen betrachteten Tagen. In der Abbildung sind jedoch nur die Features dargestellt, die überhaupt verwendet wurden. Die Analyse hat aber gezeigt, dass 28% 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, so dass sich in den folgenden Releases eine Reihe der Testaufwände einsparen oder nutzbringender einsetzen ließen[1].

Weitere Informationen

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

Literatur

[1] 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.

[2] Elmar Juergens, Martin Feilkas, Markus Herrmannsdoerfer, Florian Deissenboeck, Rudolf Vaas, and K Prommer. Feature profiling for evolving systems. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 171–180. IEEE, 2011.

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